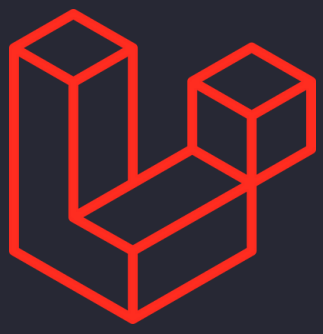
Dr. Gludovátz Attila

LARAVEL KERETRENDSZER A GYAKORLATBAN

(V10 ÉS V11)



Eötvös Loránd Tudományegyetem
Informatikai Kar

2024

Előszó

Aki a webfejlesztést választja hivatásául, annak egy folyamatos versenyfutás lesz az élete. Kis túlzással akár elmondhatjuk, hogy naponta jönnek ki új keretrendszerek, illetve a legnépszerűbb keretrendszerek újdonságai, friss verziói. Ez a könyv is átesett a készítése során az alapjául szolgáló keretrendszer egy fő verzióváltásán. De nem is ez a lényeg, hogy éppen mindent, a napi szintű újításokat tudjuk-e arról a nyelvről, keretrendszerről, amellyel dolgozunk, hanem az, hogy az alapjairól és a mélységeiről egy biztos tudással rendelkezzünk! Ha pedig egy újítás érkezik hozzá, azt könnyedén tudjuk beépíteni abba a tudásanyagba, tapasztalati tőkébe, amelyet már addig megalkottunk.

Ez a dokumentum sem vállalja azt, hogy minden információ, amely leírásra került benne, akár 1-2 év múlva is ugyanúgy érvényes lesz a Laravel keretrendszer későbbi verzióiban, mint eddig. Mivel mindig történnek változások, változtatások benne, elég csak a 10 és 11 közötti különbségeket áttekinteni. Azt viszont vállalja, hogy ha átlátjuk, megértjük a benne lévő eljárásokat, és alkalmazni tudjuk az eszközöket, problémakereső és -megoldó technikákat, akkor a folyamatosan felmerülő újabb és újabb kihívásoknak sokkal könnyebben fogunk tudni megfelelni. Így érdemes tehát elolvasni, tanulni, tapasztalatot szerezni ebből a könyvből! Ehhez kívánok hasznos és kellemes időtöltést!

– Dr. Gludovátz Attila

Köszönetnyilvánítás

Mindenekelőtt a családomnak vagyok hálás, akik mindvégig támogattak, biztattak azért, hogy elérjem ezt a célomat is.

Az Eötvös Loránd Tudományegyetemnek a pályázati lehetőséget és a támogatást köszönöm, enélkül nem valósulhatott volna meg ez a könyv.

A hallgatóimnak a folyamatos visszajelzésekért, hibák feltárásáért szeretnék köszönetet mondani. Az írás ideje alatt több mint 80-an voltak, akik ellenőrizték azt, hogy amit leírtam, az helyes-e, valóban úgy működik-e, ahogy azt lejegyeztem.

A lektornak a javításokért, szakmai észrevételekért, tanácsokért vagyok leginkább hálás.

Tartalom

01 - 05

Tartalom

06 - 10

Tartalom

11 - 15

Tartalomjegyzék

1. Bevezetés (Introduction)

A fejezet segít rögtön tisztázni, hogy kinek is szól a könyv, és miért a Laravel keretrendszerre esett a választásom, amikor belevágtam a feladatba. Utána áttekintem, hogy miről fogunk tanulni konkrétan és tanácsokkal látom el az Olvasót, ha elakadna. Végül röviden bemutatom a saját és a könyvem lektorának kapcsolatát a webfejlesztéssel és a Laravel keretrendszerrel,

1.1. Kinek szól a könyv?

A könyvet leginkább azoknak ajánlom, akik már rendelkeznek legalább egy minimális tudással HTML, CSS, JavaScript, PHP nyelvek és adatbázisok témakörében. De, ha mindez nem lenne meg, akkor is érdemes lehet ezt használni, legfeljebb egy kicsit több utánajárást kell végezni az adott terület mélyebb megértéséhez. Mindenképpen fontos a Laravel keretrendszer megismerése felé mutatott elköteleződés. Bátran merem állítani, hogy ahogy haladunk majd előre a könyvben, úgy fog egyre jobban megtetszeni ennek a kiváló keretrendszernek az alkalmazása.

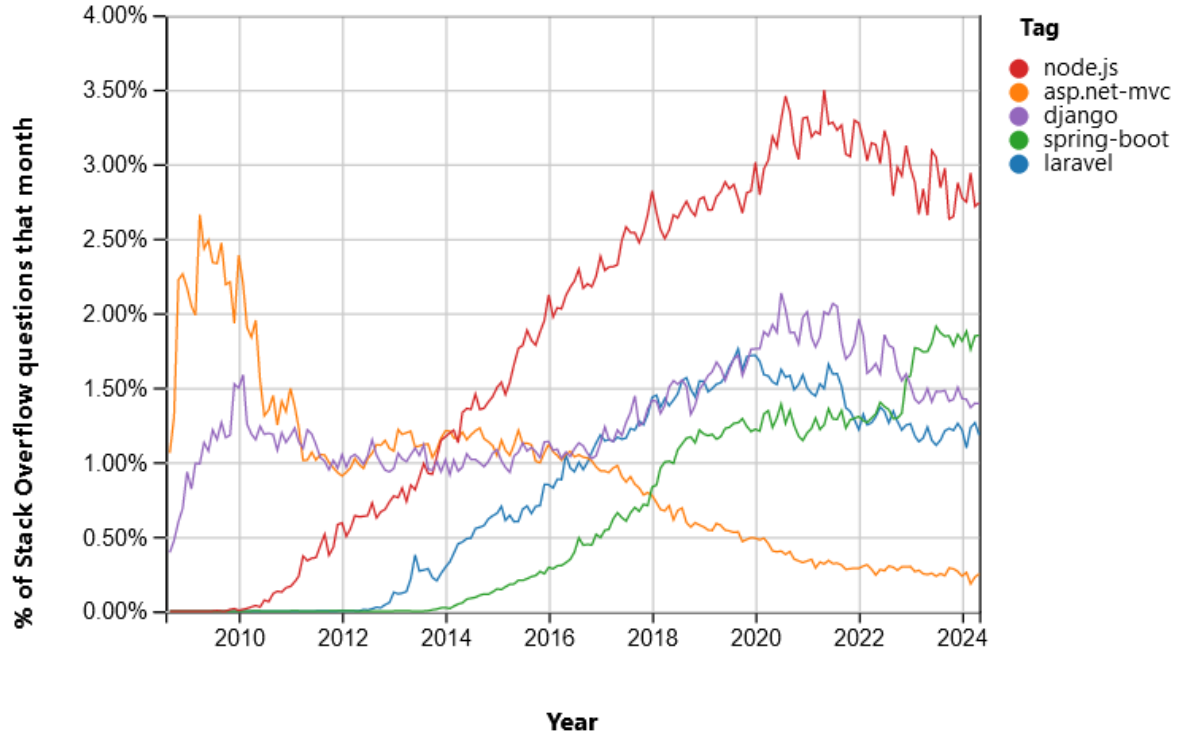
A könyv használatát javaslom egy informatikai (legyen szó akár programtervező informatikus, műszaki informatikus vagy éppen gazdaságinformatikus) egyetemi képzés webes irányvonalának, tantárgyi ágának a 3. félévében (tehát például 1. féléves HTML, CSS és valamilyen CSS keretrendszer; a 2. féléves JavaScript, PHP megismerése után).

A könyv különböző változatait, részeit az elmúlt néhány év során már közel 100 hallgatónak oktattam, így a tapasztalatokat folyamatosan leszűrhettem, amelyek szintén a tananyag előnyére váltak. Az esetlegesen előforduló hibákat, pontatlanságokat, vagy a verziókülönbségekből adódó eltéréseket is mindig jelzem, ha korábbi verziókban másképpen volt valami.

1.2. Miért a Laravel keretrendszerre esett a választás?

Gyanútlan olvasóim biztos sokféle rangsorolásba belefutottak már és próbálnának minél népszerűbb rendszerek, nyelvek megtanulása felé elindulni. Ez már csak azért is van így, mivel a népszerűség biztosan nem a véletlen műve: valószínűleg közrejátszik benne a nyílt forráskód, a bővíthetőség, az ingyenesség, a jól dokumentáltság, a hordozhatóság, a modularitás, az aktív fejlődés és ez a felsorolás még biztosan nem is teljes. Ha már népszerűség, akkor én két tényezőt szeretnék vizsgálni. A Laravel alapvetően backend oldali webfejlesztésben segít, így érdemes az *„almát-almával, körtét-körtével”* összehasonlítás mentén haladni. Ebben az utóbbi 10-15 évben segítséget nyújt számunkra a StackOverflow, amely megmutatja, hogy adott témakörben hónapról-hónapra a feltett kérdések hány százaléka tartozott adott területhez. Emiatt először összehasonlítottam a legnagyobb és legnépszerűbb nem PHP nyelven írt MVC[1] tervezési mintát követő keretrendszerekkel a Laravel-t (1–1. ábra).

[1] Model-View-Controller



1–1. ábra: Különböző programozási nyelvek MVC alapú keretrendszereinek összehasonlítása a StackOverflow által (Forrás: [1])

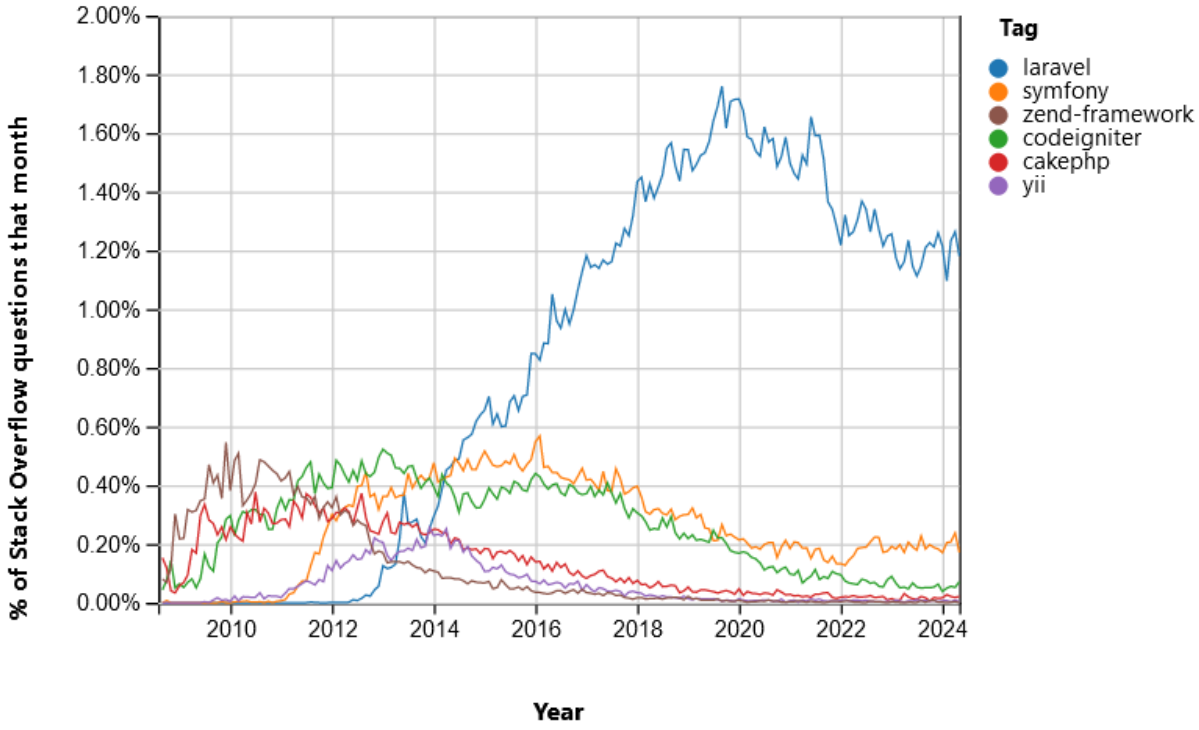
Leszűrhetjük az ábrából, hogy a leginkább alkalmazott backend oldali webes keretrendszerek között a Laravel nagyjából a középmezőnyt képviseli (a Java alapú Spring-gel és a Python alapú Django keretrendszerrel együtt), megelőzve a leginkább C# nyelvhez köthető Microsoft ASP.NET MVC projekteket. Viszont el van maradva a legnépszerűbb programozási nyelv, vagyis a JavaScript futtatókörnyezetétől, a Node.js-től, amelynek segítségével könnyedén hozhatók létre MVC tervezési mintára épülő backend oldali projektek. A fentiek miatt tehát még nem feltétlenül lenne érdemes a Laravel-t választani, viszont a PHP napjainkban még mindig elég népszerű – persze ez a CMS[2]-ek, vagyis a tartalomkezelő rendszerek elterjedtsége miatt is igaz. A mai weboldalak motorját leggyakrabban a WordPress, Drupal, Joomla tartalomkezelő rendszerek adják (mind PHP nyelven írt). Ha pedig egy cég nem is vágyik saját weboldalra és tartalmakra, webshop szolgáltatást biztosan szeretne nyújtani a vevőinek, felhasználóinak, hogy a saját termékeiket és/vagy szolgáltatásaikat elérhetővé tegye a számukra, ebben az esetben is nyerő választás lehet egy PHP nyelven írt CMS kiegészítő vagy webshop rendszer. Így tehát picit csalnék a statisztikával, ha kijelenteném, hogy mindenképpen PHP nyelven lenne érdemes fejleszteni a jövőben. De az sem lenne fair, ha a nyelveket versenyeztetném meg egymással StackOverflow-s kérdésfeltevés számában. Hiszen például a Python biztosan megelőzné, de ez vajon bizonyítaná azt, hogy a Python-os Django *„jobb, népszerűbb vagy esetleg használhatóbb”* a Laravel-nél? Nem annyira… a Python népszerűségét leginkább másik két tényező növelné véleményem szerint: először is nagyon jó tanuló programozási nyelv, másodszor pedig az adattudomány területén rendkívül elterjedt, mivel adatelemzésre talán ez alkalmazható a

[2] Content Management System

legjobban. Vigyázzunk tehát azzal, ha valaki statisztikákkal akar elámítani minket annak kapcsán, hogy adott nyelv vagy keretrendszer mennyire népszerű.

A fentiekkel azért amellett próbáltam érvelni, hogy érdemes a PHP nyelvet választani backend oldali webfejlesztéshez, viszont mind a tartalomkezelő rendszerek, mind a webshopok csak hobbi fejlesztőknek szólnak leginkább. Ebből a körből mindenképpen érdemes fejlesztőként továbblépni, és egy teljeskörű keretrendszert választani ahhoz, hogy megmaradjon a gyors, hatékony építkezés lehetősége, a teljesen egyedi, minden igényt kielégítő megoldások mellett.

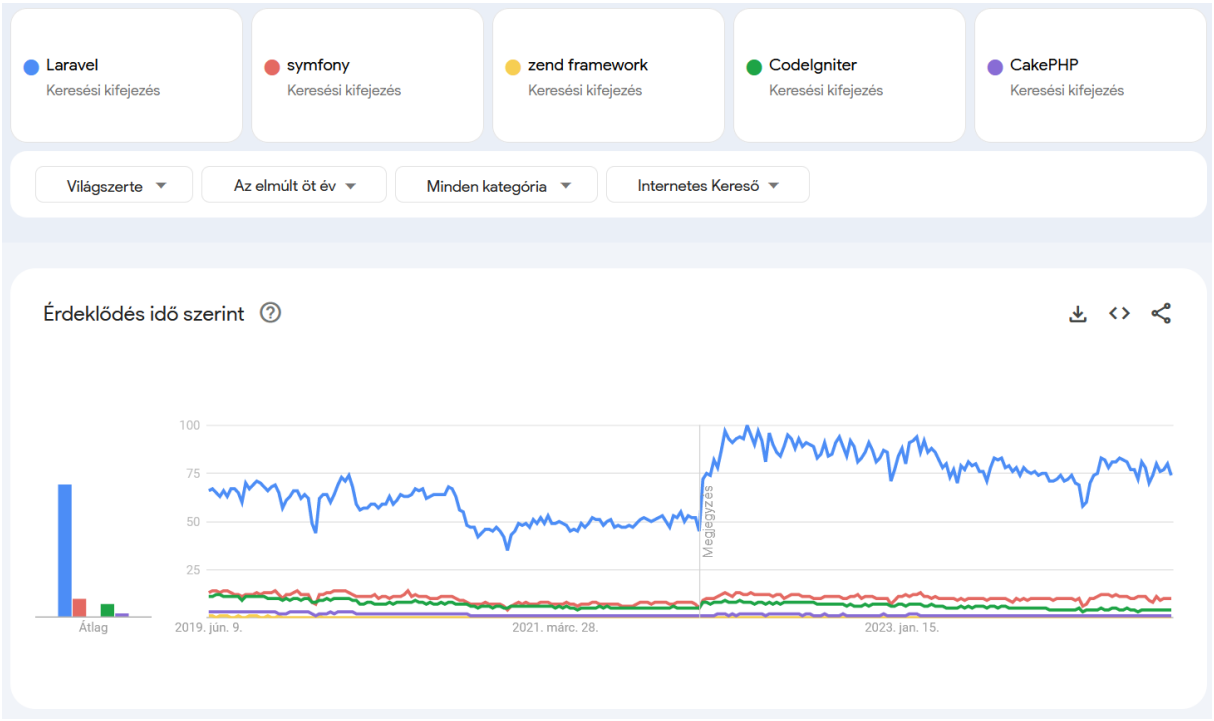
Ha viszont már letettük a voksunkat a PHP alapú keretrendszer mellett, akkor nézzük meg, hogy milyen alternatíváink vannak, és ezekről mit mutat nekünk a StackOverflow (1–2. ábra).



1–2. ábra: PHP nyelven írt MVC alapú keretrendszerek összehasonlítása a StackOverflow által (Forrás: [2])

Az összehasonlításhoz alapul vett további PHP keretrendszerek: Symfony (https://symfony.com/), Zend Framework (https://framework.zend.com/), CodeIgniter (https://codeigniter.com/), CakePHP (https://cakephp.org/), Yii Framework (https://www.yiiframework.com/). Mindegyikről elmondható, hogy ingyenesen használható, részletes dokumentációval rendelkeznek.

A Laravel ezek közül kiemelkedik a StackOverflow alapján nagyjából 2014 óta folyamatosan. De fordulhatunk tanácsért a Google Trends-hez is (1–3. ábra), és megnézhetjük, hogy az elmúlt 5 évben világszinten ezekre a PHP keretrendszerekre mennyien kerestek rá napi szinten (itt is egyértelműen kiemelkedik a Laravel, 2021 után még inkább, mint előtte).



1–3. ábra: PHP nyelven írt MVC alapú keretrendszerek összehasonlítása a Google Trends által (Forrás: [3])

Ha ez még nem lenne elég, akkor vannak további tényszerű érveim is a Laravel használata mellett:

- Open Source megoldás, tehát nyílt forráskódú, aminek az előnye abban rejlik majd, hogy a keretrendszer teljes magját meg lehet tekinteni. Számos fejlesztő készít hozzá kiegészítő modulokat, kiterjesztéseket. A szakmai közössége meglehetősen nagy, ahogy jeleztem is az imént: rengeteg keresést (StackOverflow, Google, ChatGPT) produkálnak vele kapcsolatban a programozók.
- A fenti trend ábrákból is kivehető, hogy a Laravel fejlődése és népszerűsége töretlen. Ez egy Laravel-lel ismerkedő számára azért fontos információ, mert ha ez a keretrendszer egyre elterjedtebb, akkor a szoftverfejlesztő cégek piacán is valószínűleg egyre többen fogják ezt használni, így a könyv által megszerzett tudás egyre értékesebb, tehát jobb állásokra, pozíciókra tudunk majd pályázni a cégeknél.
- A Laravel a webes környezetben gyakran alkalmazott MVC tervezési mintára épül. Ennek a részleteit a későbbiekben megismerjük és mindenki számára nyilvánvaló lehet a működése.
- A Laravel letisztult weboldallal rendelkezik (https://laravel.com/), ahonnan elérhető a minden részletre és verzióra kiterjedő dokumentációja (https://laravel.com/docs/10.x a 10. verziónál illetve https://laravel.com/docs/11.x a 11. verziónál – ebből is látszik, hogy a linkek jobb esetben mindig működnek, pusztán a verziószámot kell módosítani), valamint a számunkra is fontos Laracasts online oktatási weboldal (https://laracasts.com/) sok ingyenes és fizetős tartalommal (*de mindenkit biztatok az előfizetésre, mert nagyon megéri!*), Jeffrey Way az egyik legjobb oktató, aki rendkívül jól magyarázza a bonyolult dolgokat is, amiket érdemes tudni a Laravel keretrendszerről. A fejlődés pedig elengedhetetlen, úgyhogy mindenképpen javaslom Povilas Korop munkásságát napi szinten nyomon követni a Laravel Daily (https://laraveldaily.com/) weboldalon és Youtube-csatornán (https://www.youtube.com/c/LaravelDaily).

1.3. Miről fogunk tanulni konkrétan?

Erre a rövid válasz az, hogy a Laravel keretrendszerről gyakorlatiasan. A kicsit hosszabb válaszban pedig mindenképpen fontos megjegyezni, hogy egy ilyen keretrendszer teljeskörű ismertetése nem lehetséges egyetlen ilyen könyv formájában. A Laravel már annyi mindent támogat, hogy egyszerűen nem lehetséges mindenre időt és helyet biztosítani, de az alapokról egy nagyon biztos tudással fogunk rendelkezni, ha a könyv elolvasása után nekiáll az olvasó saját projektjeinek.

A fejezeteket és tartalmaikat megpróbálom úgy felépíteni, hogy egy már rutinosabb HTML / CSS / JavaScript / PHP fejlesztő aránylag gördülékenyen tudjon majd haladni a témakörök feldolgozásával. A feldolgozásra kerülő témakörök:

- Fejlesztőkörnyezet összeállítása (Development environment)
- Projektek telepítésének módjai (Installation)
- Verziókövetés, verziókezelés (Version control)
- MVC (Model-View-Controller) tervezési minta a Laravel-ben
- Útvonalválasztás weben és az API (Application Programming Interface) felületen
- Nézetek, szerkezetek, komponensek (Views)
- Különböző adatbázis-kapcsolatok (Database connections: MySQL, SQLite, Microsoft SQL Server)
- Adatbázis-kezelés: adatdefiníciós, -lekérdező és -manipulációs műveletek, adatkapcsolatok (Database access and management)
- Létrehozó, lekérdező, frissítő és törlő adatbázis műveletek (CRUD = Create, Read, Update, Delete)
- REST API használata és tesztelése
- Eltérő projektkörnyezetek definiálása, például fejlesztésre, tesztelésre, éles működésre (Different environments)
- Működés, vagyis az üzleti logika és a felhasználói felület integrálása (Integration of business logic and user interface)
- Felhasználói adatérvényesítés, vagyis a kliens és szerver oldali validáció (Client and server side validation)
- Köztes rétegek (Middleware)
- Felhasználói hitelesítés (Authentication): funkcionalitások és felületek integrálása kezdő készletek, csomagok segítségével (Starter kits: Breeze, Jetstream, Fortify, Sanctum)
- Felhasználói engedélyeztetés (Authorization)
- A rendszer magjának építőkövei: gyűjtemények, szolgáltatások, értesítések, események (The core elements of the system: collections, service container, façades, events, observers)
- Kódminőség ellenőrzése, javítása (Software Code quality assurance)
- Webalkalmazás publikálása nyilvános tárhelyre és egy választott felhőszolgáltatásra (Deployment to a public server and a Cloud service – Microsoft Azure)
- Hasznos útravaló a jövőre: függőségek, többnyelvűsítés, kiberbiztonság (Helpful hints for the future related to the Dependency management, Localization, Cyber Security)
- Hivatkozások gyűjteménye lesz segítségünkre a fejlődésben (Useful weblinks)

Mindegyik témakörnél az automatikus tesztelésre is nagy hangsúlyt fektetek.

1.4. Hol találhatok segítséget?

A könyv a Laravel legfrissebb verziójáról szól, de az alapok, amiket belőle megtanulhatunk, azok általában érvényesek a későbbiekre vonatkozóan is és esetleg csak finomításokra van szükség, hogy ha egy majdani újabb verziószámú keretrendszerrel dolgozunk tovább. A könyvben található kódokat nyilvános GitHub repository-kban mindig az adott fejezetnél az olvasó rendelkezésére bocsátom, így meg lehet nézni, hogy milyen változásokat eszközöltem az aktuális projektjeimben, amiket a fejezetben felvázoltam. Az ELTE-s felhasználóm GitHub fiókja itt érhető el a publikus tartalmakkal együtt: https://github.com/gla-elte

Ezen kívül ajánlom még azt a gyűjteményt, amelyet a 17. fejezetben összeállítottam. Ezek szintén érdekesek és értékesek lehetnek a Laravel-lel ismerkedők számára.

1.5. A szerző és a Laravel

Bár én az ipari folyamatok és rendszerek integrációjából doktoráltam 2019-ben, de már a 2000-es évek közepe óta élénken érdeklődök a webprogramozás iránt is. Azóta számos sikeres projektet valósítottam meg itthon és külföldön egyaránt. A Laravel keretrendszerrel 2019-ben kezdtem el foglalkozni és egyre inkább megtetszett. Ez főleg a Laracasts sorozatainak köszönhető, amelyekből sokat tanultam. A könyv elkészítése során felhasználtam több mint 10 éves szoftvertervezési és -fejlesztési tantárgyaknál összegyűjtött oktatási tapasztalataimat is.

2019. nyarán határoztam el, hogy készítek egy olyan könyvet, amely alapján az egyetemi hallgatóim, valamint a további érdeklődő olvasók is képesek könnyedén elsajátítani a Laravel keretrendszer használatát (ekkor az 5.7-es verzió volt a legújabb). 2019. decemberében döntöttem a könyv folytatása mellett, ekkor dolgoztam fel az 5.8-as verziójú Laravel újdonságait, 2020 januárjában pedig folytattam a munkát a 6-os, majd a 7-es verzió részleteivel, 2021-ben a 8-as verziót használtuk már, míg 2022-től a 9-est. Az évek során megismert elméleti és gyakorlati tudásra alapoztam, de jó néhány helyen a saját tapasztalataimmal is kiegészítem a leírásokat és tanácsokat adok a problémák megoldásához. A könyv tehát 2019. óta létezik, ha nem is ebben az írásos formában.

2022. január 24-én elindítottam a saját blogomat (https://attila.gludovatz.hu/blog), amelyen időről-időre új bejegyzéseket készítettem. A blogom témája főleg a Laravel keretrendszer volt. Maga a keretrendszer és az általam indított blog is számos hallgatóm tetszését elnyerte, így ők is ezt a fejlesztési, fejlődési irányvonalat választották szakdolgozatuk, egyéb projektjeik elkészítéséhez. A témához kapcsolódó szorgalmasabb hallgatóim blogbejegyzéseket is írtak nálam, amelyek főleg a Laravel backend-ként, kvázi háttérben lévő kiszolgálóként használják a Laravel-t valamilyen frontend oldali JavaScript keretrendszer mögött. Korábban és jelenleg is, több általam témavezetett, szakdolgozó hallgató helyezkedett el a szakmában sikeresen, főleg, mint Laravel-es webfejlesztő.

2023-ban megjelent a Laravel 10-es verziója, amely egy megfelelő pontnak tűnt ahhoz, hogy belekezdjek ennek a könyvnek az elkészítésébe, és összefoglaljam benne tapasztalataimat, iránymutatásaimat. Ezek az ismeretek szintén ennek a dokumentumnak a részét képezik. Ahogy az a verziók korábbi felsorolásából látszódik is, a Laravel nagyon gyorsan fejlődik, folyamatosan jönnek az újabbnál újabb beépített

funkcionalitások hozzá, viszont a biztos alapok nem változnak. Ha ezeket jól elsajátítjuk, megértjük, használjuk és betartjuk a *„játékszabályokat" (névkonvenciókat),* akkor egy biztos tudásnak leszünk a birtokában.

A több éves tapasztalatom birtokában, és számos sikeres projekt kivitelezése után vágtam bele ennek a könyvnek az elkészítésébe. Végül a Laravel keretrendszer megismeréséhez szükséges legfontosabb témaköröket összegyűjtöttem, rendszereztem, hogy egy kerek egész alakuljon ki belőle ebben a könyvben.

Az írás során gyakran előfordul majd, hogy többesszám első személyben fogalmazok, ugyanis az Olvasó és én egy **csapat** leszünk a feladatok végrehajtása során. Én igyekszem jól és hatékonyan vezetni, az Olvasónak pedig érdemes követnie az iránymutatásaimat.

1.6. A lektorok és a Laravel

Jelen dokumentumot a több mint 80 hallgatómon kívül, akik folyamatosan ellenőrizték és kipróbálták a benne lévő programkódokat, az fws online Zrt. webfejlesztő cég vezető fejlesztője, Fekete Zsolt lektorálták.

1.7. Új keretrendszer verzió: Laravel 11 (kiadás éve: 2024)

A könyvem legnagyobb részét 2023-ban írtam. Ekkor még a Laravel keretrendszer 10-es verziója volt a legfrissebb. 2024 első negyedévére (márciusra) ígérik a Laravel keretrendszer 11-es, stabil, végleges verziójának kijövetelét. 2024 februárjában a 11-es fejlesztői verziójának – 10-es verzióhoz képest – megjelenő újdonságait fogom bemutatni elméleti és gyakorlati szempontból egyaránt.

Ahogy említettem, a Laravel fejlődését az 5.7-es verziója óta követem. Kijelenthető, hogy egy stabil keretrendszert építenek a fejlesztői, amelyet az újabb verziók mind-mind finomhangoltak vagy kiegészítettek adott módokon, illetve területeken. Akit érdekel a keretrendszer verzióinak története és újdonságai, annak ezt a leírást javaslom (https://benjamincrozat.com/laravel-versions). Aki nem szeretne nagyon részletesen elmélyedni a verziókban, annak talán érdemes az utóbbi néhány évet szemügyre venni. Azt mindenképpen fontos kiemelni, hogy a Laravel együtt fejlődik az alapját képező nyelvvel (PHP) és a környezetével (kliens oldali keretrendszerek, adatbázis-kezelő rendszerek és a hozzájuk tartozó driver-ek, tesztelési keretrendszerek stb.).

Új eszközök a 10-es alap verzióhoz képest

A Laravel keretrendszer fejlesztői és vezetőjük (Taylor Otwell) a 11-es verzió újdonságainál a következőkre koncentrálnak (számos eszköz ezek közül már létezett a Laravel 10-es verziójában is, azonban csak a későbbi, magasabb alverzió számoknál jelentek meg, és a Laravel 11-ben még nagyobb hangsúlyt fektetnek ezek beépítésére, használatára):

- **Folio:** segíti az útvonalválasztást úgy, hogy egyből az adott oldal kódjához irányítja a felhasználói kérést és jeleníti meg neki például az egyszerű HTML kódú fájlt.

- **Volt (Livewire):** egy olyan eszköz, amelynek használatával egy fájlba írhatjuk az üzleti logikát és a megjelenítést, hasonlóan, mint a más, népszerű kliens oldali keretrendszereknél. Bár sok Laravel (backend) oldali fejlesztő kétségekkel fogadja ezt az újítást, de az idő és a tapasztalatok majd megmutatják, hogy mennyire lesz népszerű ez az új eszköz.
- **Prompts:** egy PHP csomag, amellyel szép (kinézet), felhasználóbarát kezelőfelületet tudunk biztosítani a console-os / terminal-os környezetben. A felület az űrlapoknál – a böngészős felületen megszokott – helyőrzőket (placeholder) és érvényesítési (validation) eszközöket is nyújtja számunkra.
- **Reverb:** egy új WebSocket szerver a Laravel alkalmazások számára, amely valós-idejű kommunikációt tesz lehetővé a kliensek és a szerver között.
- **Pest:** egy tesztelési keretrendszer, amelynek használata már a 10-es verziónál elkezdett népszerűvé válni, de a 11-esnél még több újítást ígérnek a használatával kapcsolatban. A Laravel 10-ig a PHPUnit-ot és a Dusk-ot használhattuk tesztelésre.
- **Herd:** egy villámgyors, natív Laravel és PHP fejlesztőkörnyezet macOS rendszerekre. Minden olyan eszközt tartalmaz, amire a Laravel fejlesztés során szükség lehet, beleértve a PHP fordítót és az Nginx webszervert is. (Megjegyzés: én ezt nem fogom használni, de talán valakinek hasznos lehet, ha tud róla, hogy van ilyen.)

Ezekre az eszközökre is ki fogok térni a könyvem fejezeteinek bemutatása során, amikor az őket érintő területeket prezentálom.

1–1. újdonság: Eszközök, csomagok

Ezeken kívül, a könyvben én mindig kiemelem majd azokat a részeket (ugyanilyen piros hátterű formátumban), ahol, amikor és ahogyan a Laravel 11 újdonságot hoz a 10-es verzióhoz képest. De látni fogjuk, hogy lesznek olyan területek is, például *„10.1. A köztes réteg (Middleware"* fejezet tárgyalása során, amikor kimondottan hasznos lesz az, hogy ismerjük a 10-es verzió miatt a köztes réteg elemeit és működését, emiatt jobban átláthatjuk majd azt, amit a 11-es verziójú keretrendszer utána egy kicsit elrejt a fejlesztők elől. Ez az *„elrejtés"* sem véletlen, mivel a Laravel fejlesztőinek meglátása az, hogy már-már túl nagyra nőtt a Laravel projektek mérete, és a projekt tartalmaz – a véleményük szerint – olyan fájlokat is, amelyekre a jövőben nem feltétlenül van szükség látható helyen. Tehát egyfajta karcsúsítást, méretcsökkentést (*„slim skeleton"* elv) akarnak elérni, ami azért sok helyen nagy változást jelent a könyvtárstruktúrák magjában és az eddig esetleg már megismert fájlok tartalmában. A fejlesztők *„slimmer application skeleton"* ötlete szerint, ha egy új fejlesztő kezdi el megismerni a Laravel keretrendszert, akkor könnyebben fog eligazodni benne, mivel kevesebb forráskóddal (fájllal) kell foglalkoznia…

2. Kezdő lépések az induláshoz (Getting started: first steps)

Elsőként meg kell vizsgálni, hogy mi szükséges a fejlesztéshez, milyen fejlesztőkörnyezetet érdemes használni. Ezután telepítjük az első Laravel alkalmazásunkat és megnézzük, hogy a projekt struktúrája hogyan követi az MVC tervezési mintát. A fejezet zárásaként feltöltjük az első projektünket a GitHub verziókövető rendszerébe.

2.1. Fejlesztés előkövetelményei

A szoftverfejlesztéshez egy jól kiválasztott és beállított munkakörnyezetre van szükségünk. Én egy jól bevált, kipróbált környezetet fogok itt definiálni, de ízléstől és lehetőségektől függően el lehet ettől térni.

2.1.1. Futtatókörnyezet

Operációs rendszernek megfelelő választás a Windows 10 vagy 11-es verziója, de aki már ismeri és megszokta valamelyik Linux disztribúció alkalmazását, annak ott sem fog problémát okozni a környezet beállítása. Én Windows 10-et használok, így az anyagok ismertetése közben, ha szükséges, akkor ennek beállítására, finomhangolására teszek javaslatot.

Szükségünk lesz (vagy lehet) még a webfejlesztők *„szentháromságára”*, vagyis egy webszerverre, tipikusan megfelelő választás erre az Apache, egy adatbázis-kezelő szerverre, például MySQL vagy MariaDB, illetve magára a PHP fordítóra. Ez a három dolog külön-külön is beszerezhető, de vannak előre elkészített programcsomagok, amelyek mindezeket, sőt, még egy picit többet is tartalmaznak. Egy jól bevált szoftvercsomag, mint például a XAMPP szerver (https://www.apachefriends.org/download.html) vagy a WampServer (https://www.wampserver.com/en/) tartalmazzák a fejlesztéshez szükséges elemeket. Arra figyeljünk, hogy ezekből van 32 bites (x86) és 64 bites (x64) verzió is, úgyhogy az operációs rendszerünknek megfelelő típusút kell választani. Én a XAMPP-ot javaslom, mivel egyszerű a használata, továbbá, ha frissíteni kell valamelyik elemének a verzióját, az is aránylag könnyedén megtehető. Adatbázis terén pedig a „kisebb” feladatok elvégzéséhez még mindig ingyenes MySQL használható, nagyobb igény esetén ez már fizetős lenne (mivel felvásárolta az Oracle vállalat), míg a MariaDB a MySQL továbbra is nyílt forráskódú és ingyenes verziójú *„folytatása”*, a korábbi MySQL fejlesztők által fejlesztett rendszer.

Adatbázis-kezelő rendszer szempontból a Laravel 10-es keretrendszer alapértelmezetten a MySQL-t használja a projekt létrehozása után, míg a 11-es keretrendszer már az SQLite-ot használja (minimálisan elvárt verzió a 3.35.0+ a Laravel 11 esetében). Ez utóbbi miatt, a fájl alapú SQLite adatbázis-kezelőre is szükségünk lesz a jövőben, de ezt mindenki egyéni „ízlésére”, prioritásaira bízom. Néhány javaslatot azért teszek a menedzselő programokra: ha egy kicsi asztali alkalmazást szeretnénk telepíteni, akkor a DB Browser for SQLite-ot javaslom (https://sqlitebrowser.org/), de néha az is elég, ha a Visual Studio Code-ban egy kiterjesztést (https://marketplace.visualstudio.com/items?itemName=qwtel.sqlite-viewer) telepítünk, és már tudjuk is vele menedzselni az SQLite adatbázisunkat. Hallgatóim kedvence pedig a DBeaver (https://dbeaver.io/), amely szintén egy asztali alkalmazás, rendkívül univerzális, tehát rengeteg adatbázis-kezelő rendszer típushoz tudunk csatlakozni és menedzselni őket. Talán az univerzitása miatt szeretik ezt annyian.

Szükségünk lesz továbbá a **Composer** alkalmazásra, amely egy csomagkezelő. A webes világban nagyon sokszor használunk úgynevezett 3. féltől származó (3rd party) eszközkészletet, csomagot vagy ún. osztálykönyvtárat. A Laravel-hez majd ezzel a Composer-rel tudjuk beszerezni a szükséges kiegészítőket. (https://getcomposer.org/download/ - Composer-Setup.exe -vel telepíthető Windows rendszeren).

A Laravel főleg szerver oldalon működik, viszont a kliens oldali megjelenítésére is oda kell figyelnünk majd. Ehhez szükség lesz **Node.js**-re és egyéb kiegészítőire, például az **npm** csomagkezelő alkalmazásra, amely a projektjeink kliens oldali függőségeit, csomagjait kezeli (https://nodejs.org/en/download/ - Windows installer).

Ahhoz pedig, hogy a projektjeinket, programkódjainkat megfelelő verziókövetéssel tudjuk ellátni, használjuk a **Git**-et (https://git-scm.com/download/win - standalone installer). Ez számos előnnyel jár (változások támogatása, csoportmunka megkönnyítése stb.), továbbá a használata egy szoftverfejlesztő munkakör esetén már alapvetően szükséges.

Amíg ezeket a sorokat olvastad, addig remélhetőleg el is indítottad ennek a komplett fejlesztőkörnyezetnek az összerakását, vagyis az elemeinek a telepítését.

Ha sikeresek voltak a telepítések, akkor a Windows 10/11 esetén már nem is feltétlenül van szükség újraindításra, hanem anélkül fogja érzékelni, hogy a szoftverek működnek, meghívhatók és használhatók. Ezeket a következő parancsokkal tudjuk megtenni a parancssorban (többféle módon előhívható, először talán a legegyszerűbb, ha a Start menüben beírjuk: `cmd` és ENTER) vagy a PowerShell terminal-ban is megfelelő:

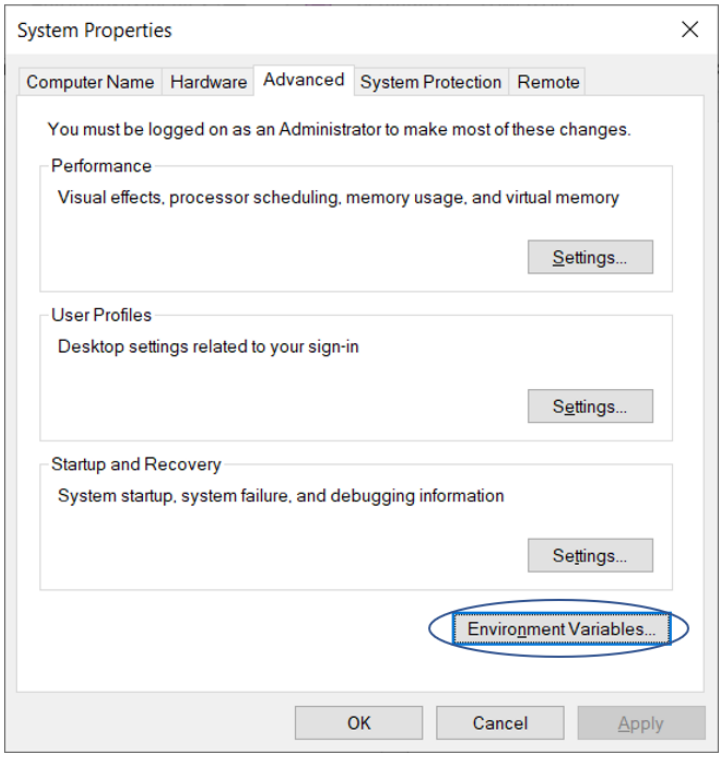
```
php -v

node -v

npm -v

composer about

git -v
```

Ha minden rendben van, akkor hasonló eredményt kell látnunk, mint amit a 2–1. ábra mutat (esetleg különböző verziószámokkal):

```
PS C:\xampp\htdocs> php -v
PHP 8.2.4 (cli) (built: Mar 14 2023 17:54:25) (ZTS Visual C++ 2019 x64)
Copyright (c) The PHP Group
Zend Engine v4.2.4, Copyright (c) Zend Technologies
PS C:\xampp\htdocs> node -v
v18.17.1
PS C:\xampp\htdocs> npm -v
9.8.1
PS C:\xampp\htdocs> composer about
Composer - Dependency Manager for PHP - version 2.6.2
Composer is a dependency manager tracking local dependencies of your projects and libraries.
See https://getcomposer.org/ for more information.
PS C:\xampp\htdocs> git --version
git version 2.39.2.windows.1
```

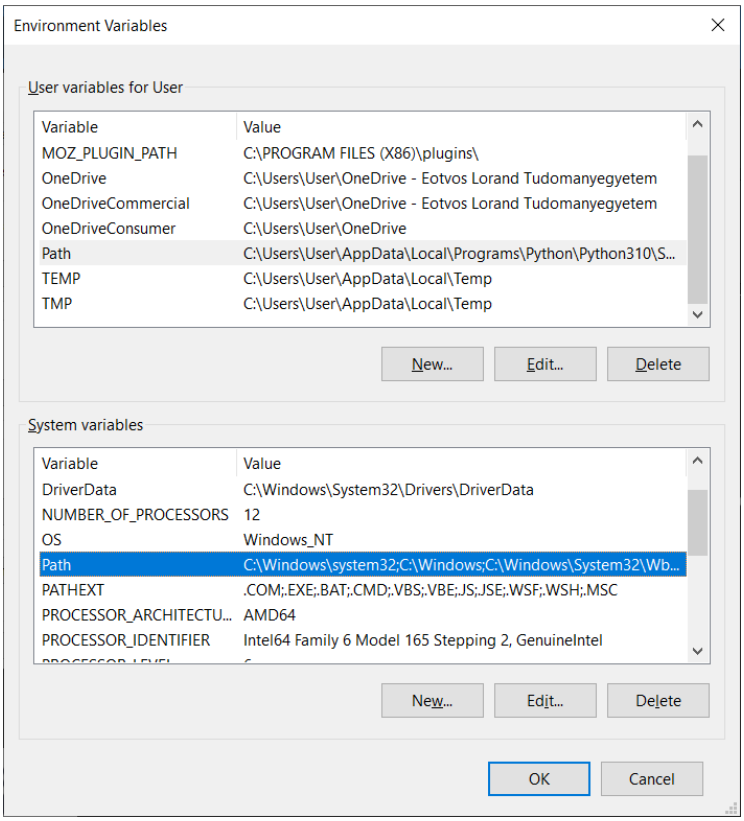
2–1. ábra: Futtatókörnyezet alkalmazásai elérhetők és működnek (Windows PowerShell terminal-ban)

Ha mégis olyan hibaüzenetet kapnánk arról, hogy valamelyik parancsot nem ismeri fel a Windows, akkor ezt is orvosolhatjuk a következők szerint (de lehet, hogy egy egyszerű újraindítás is segíthet). A Windows-ban a futtatható állományok útvonala, amelyeket bárhonnan elérhetünk, azok elérési útvonala bekerül az úgynevezett *„Környezeti Változók"* vagy angolul *„Environment Variables"* (2–2. ábra) gomb megnyomása után a rendszerszintű vagy felhasználói szintű *PATH* elemei közé. Legegyszerűbben a Start menüvel hívható ez elő ez a kis ablak az angol vagy magyar név beírásával:



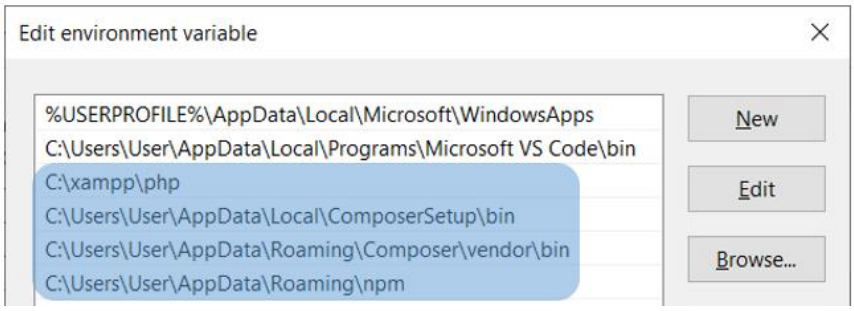
2–2. ábra: Rendszer beállítások ablak a környezeti változók eléréséhez

Majd a felhasználói- (fenti lista) vagy a rendszerszintű (lenti lista) változók közül kiválasztjuk a PATH-ot és szerkesztjük (2–3. ábra).



2–3. ábra: Felhasználói- és rendszerszintű környezeti változók

Ekkor ellenőrizhetjük, hogy az esetlegesen nem működő alkalmazás(ok) futtatható állományát tartalmazó mappának az elérési útja szerepel-e a listában (2–4. ábra).



2–4. ábra: A PATH környezeti változó elemei (kiemelten a Laravel fejlesztéshez elengedhetetlen alkalmazások mappái)

Itt például kiemelten látható a PHP fordító, a Composer és az npm csomagkezelők elérési útja. Ha valamelyik alkalmazás verziószámának lekérésével problémánk volt, akkor hozzunk létre ide a listába új elemet és tallózzuk be azt a mappát, ahol a futtatható .exe kiterjesztésű állomány benne van. Utána már biztosan menniük kell a fenti verziószámot lekérő parancsoknak.

Remélhetőleg a saját gépén fejleszt majd mindenki, úgyhogy élek azzal a feltételezéssel, hogy rendszergazdák vagyunk alapértelmezetten. Ha ez így van, akkor a későbbiekben több (jogosultsági hiányosságból adódó) problémát sikeresen elkerülhetünk akár a konkrét felmerülésük nélkül is.

2.1.2. Kódszerkesztő, adatbázis menedzser

Kódszerkesztésre használjunk **Visual Studio Code** (továbbiakban röviden: **VSCode**) programot (https://code.visualstudio.com/download - Windows 10-re). De előfordulhat, hogy valakinek jobban kézre áll valamilyen másik alkalmazás, például **a PHPStorm** (korlátozottan ingyenes) vagy a **Notepad++**, akkor használja nyugodtan azt. Én a VSCode-ot fogom alkalmazni a példák bemutatása során, és gyakran ennek a terminal-ját, mivel elég egyszerűen előhívható a Terminal menü -> New Terminal menüpont segítségével.

Adatbázis-kezelő rendszer menedzselése szempontjából a XAMPP vagy a WampServer által alapértelmezetten nyújtott **phpMyAdmin** webes (böngészőben futó) menedzselő alkalmazás is elegendő lehet, viszont én mindenképpen javaslom a **MySQL Workbench** (https://www.mysql.com/products/workbench/) alkalmazást, mert használata kényelmes és a webes phpMyAdmin megoldásnál egy kicsit stabilabb is, főleg, ha nagyobb lekérdezéseket szeretnénk futtatni a jövőben.

Most már csak néhány lépésre vagyunk attól, hogy megkezdhessük a Laravel használatát és fejlesztést, úgyhogy folytassuk is a munkát a telepítéssel.

2.2. Első Laravel projektünk telepítése

A Windows alapon működő fejlesztőkörnyezet miatt a legegyszerűbb talán a XAMPP szerver használata, mivel ez biztosítja nekünk a PHP nyelvi fordítót és a MySQL adatbázisszervert is (sőt még az Apache webszervert is, de majd meglátjuk, hogy erre nem feltétlenül lesz szükségünk a fejlesztés során). Én az általam legegyszerűbbnek tekintett utat mutatom itt be, de a 2.2.4. alfejezetben adok tanácsot, illetve iránymutatást arra vonatkozóan, hogy ha valaki a Docker virtualizációs platformon szeretné kialakítani a fejlesztőkörnyezetét és benne a Laravel-t *„konténerizáltan”* futtatni.

2.2.1. Laravel 10 projekt létrehozása

Ha valaki menet közben elbizonytalanodna, vagy alapból más utat választana, akkor mindenképpen javaslom a Laravel dokumentáció ide vonatkozó témakörét áttekinteni: https://laravel.com/docs/10.x/installation

Új Laravel projektek létrehozásához mindig valamilyen terminal-t fogunk használni, ami Windows felhasználóként elsőre szokatlan lehet, de nagyon gyorsan bele fogunk szokni és a jövőben már nem is tudnánk másképp elképzelni a hasonló feladatok végrehajtását.

Az első (legfrissebb verziójú) projektünk létrehozását a Composer csomagkezelővel fogjuk tudni létrehozni.

```
composer create-project laravel/laravel my-first-site
```

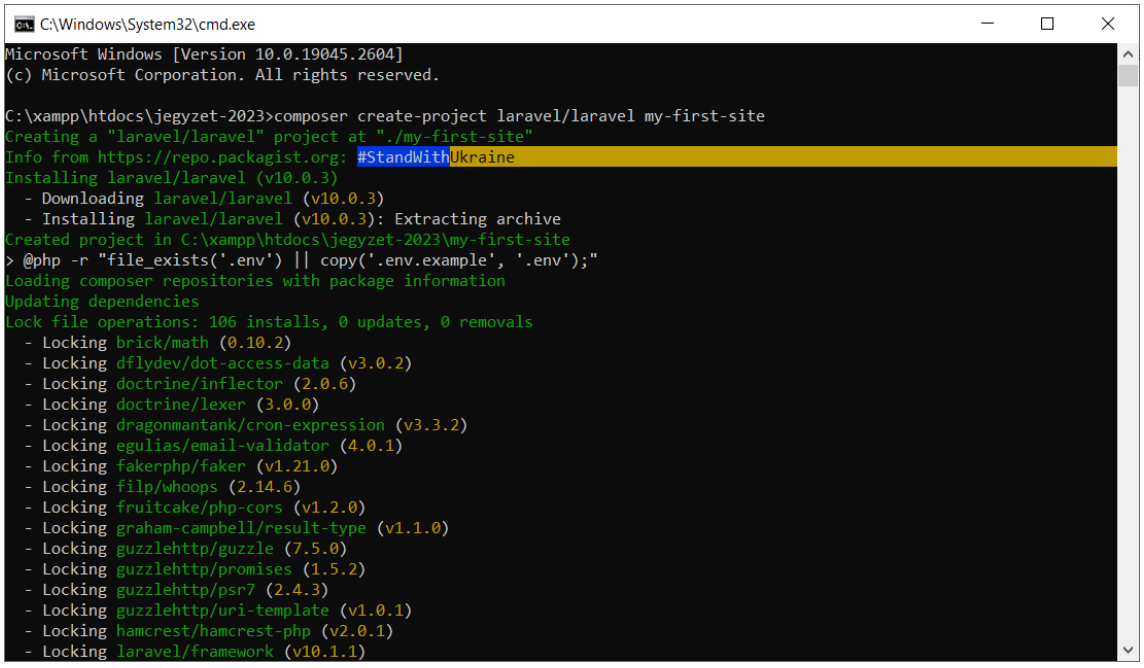
Az utasítás kiadásának hatására a rendszer letölti a szükséges csomagokat és utána telepíti is őket (2–5. ábra).

Tipp: a PHP verziónk alapvetően meghatározza, hogy milyen Laravel verziót fogunk tudni telepíteni. Így, ha egy régebbi PHP verziónk van, esetleg csak a 8-as vagy 9-es Laravel fog

települni a gépünkre, amelynek a működése még eléggé más lehet az itt bemutatott megoldásokhoz képest. A telepítés után a Laravel verzió ellenőrzését a következő paranccsal tudjuk megtenni:

```
php artisan --version
```

Lehetséges hiba, figyelmeztetés megoldása: előfordulhat, hogy a telepítés során minden csomagnál sárga háttérszínű figyelmeztetést kapunk. Ez azt jelzi nekünk, hogy hiányzik a PHP-nak egy kiterjesztése, amivel a csomagokat tudja kicsomagolni, ez a *„zip"* kiterjesztés. Ekkor állítsuk le a futást **Ctrl + c** billentyű-kombinációval. Majd nyissuk meg a **C:\xampp\php\php.ini** fájlt szerkesztésre. Keressük meg az **extension=zip** sort, aminek az elején egy pontosvessző (;) található. Ez gyakorlatilag megjegyzésbe teszi a kiterjesztés alkalmazását, így vegyük ki a pontosvesszőt a sor elejéről majd mentsük el a fájlt. Ezután térjünk vissza a terminal-hoz és indítsuk újra a projekt létrehozását (futtassuk újra a parancsot, egy felfelé mutató nyíllal könnyel előhozható újra a parancs). Ha minden jól megy, akkor a következő ábrához (2–5. ábra) hasonló képet kell látnunk a csomagok hibamentes letöltése és telepítése során.



2–5. ábra: Első Laravel projekt telepítése (Megjegyzés: ezt a példa projekteket egy külön erre a célra létrehozott „jegyzet-2023" mappába telepítem a C:\xampp\htdocs mappán belül)

A jövőben, mivel sokszor fogunk Laravel projekteket létrehozni, ezért érdemes lehet a Laravel saját telepítő segédjét is telepíteni a gépünkre. Ezt így tehetjük meg:

```
composer global require laravel/installer
```

Utána pedig már sokkal könnyebben megjegyezhető parancsot is használhatnánk a projektünk létrehozásához (ezt most még ne hajtsuk végre, mivel ez Laravel 11-es projektet hozna létre nekünk):

```
laravel new my-first-site
```

Tipp: előfordulhat, hogy valamilyen oknál fogva (például verziókülönbségekből adódó probléma miatt) egy korábbi verziójú Laravel projektre van szükségünk, ilyenkor mindenképpen a Composer-es telepítést válasszuk és adjuk meg a konkrét verziószámot, amire szükségünk van (én itt példaként az 9.0-et adtam meg):

```
composer create-project laravel/laravel=9.0 my-first-old-site
```

Ha túl régi verziószámú Laravel projektet szeretnénk telepíteni, akkor előfordulhat, hogy a PHP verziónk ezt nem támogatja, úgyhogy akkor abban is vissza kellene lépni, ami okozhat egyéb problémákat számunkra a jövőre nézve…

Mivel a dokumentum kiadásakor már a 11-es a legfrissebb Laravel verzió, de számos példa még a 10-esben is bemutatásra kerül, ezért fontos ismét kiemelni, hogy Laravel 10-es projektet így lehet létrehozni a composer segítségével:

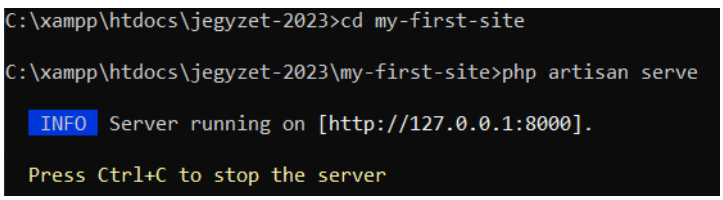
```
composer create-project laravel/laravel=10.0 my-first-site
```

Ezzel telepítésre kerül a Laravel alkalmazás és csomagjai a rendszerünkbe. Utána érdemes elnavigálnunk abba a mappába, ahol létrejött a web alkalmazás projektünk. Tipikusan ezeket a projekteket egy helyre érdemes *„gyűjteni”,* én a projekteket a **c:\xampp\htdocs** mappába teszem. Ekkor az az előnyünk megvan a XAMPP szerver használata esetén, hogy az Apache webszerver is ki tudja szolgálni a webalkalmazásainkat, de máshova is telepíthetjük majd a projektjeinket (viszont, ha máshova telepítjük őket, akkor a XAMPP-ban lévő Apache alapértelmezetten nem tudja majd kiszolgálni őket). Ha mégis máshova telepítenénk a projektjeinket, akkor az a lényeg, hogy azon a helyen legyen majd a mappákra és fájlokra írási, olvasási és végrehajtási jogunk.

A parancssorban a mappák között a `cd` parancs segítségével tudunk navigálni, tehát a `cd c:\xampp\htdocs` paranccsal az imént említett mappába fogunk eljutni. Itt hoztuk létre a legelső Laravel projektünket, ami a **my-first-site** nevet kapta. Ha a terminal-ban a *„prompt”* (sor elején lévő aktuális mappánk megjelölése) a c:\xampp\htdocs elérési utat mutatja, akkor elég csak a `cd my-first-site` parancsot kiadni a mappába való belépéshez (mappából való kilépést a `cd ..` parancs kiadásával tudjuk megtenni).

Most következhet a projekt elindítása és a kiszolgálás futtatása: írjuk be a terminal-ba a következő parancsot:

```
php artisan serve
```

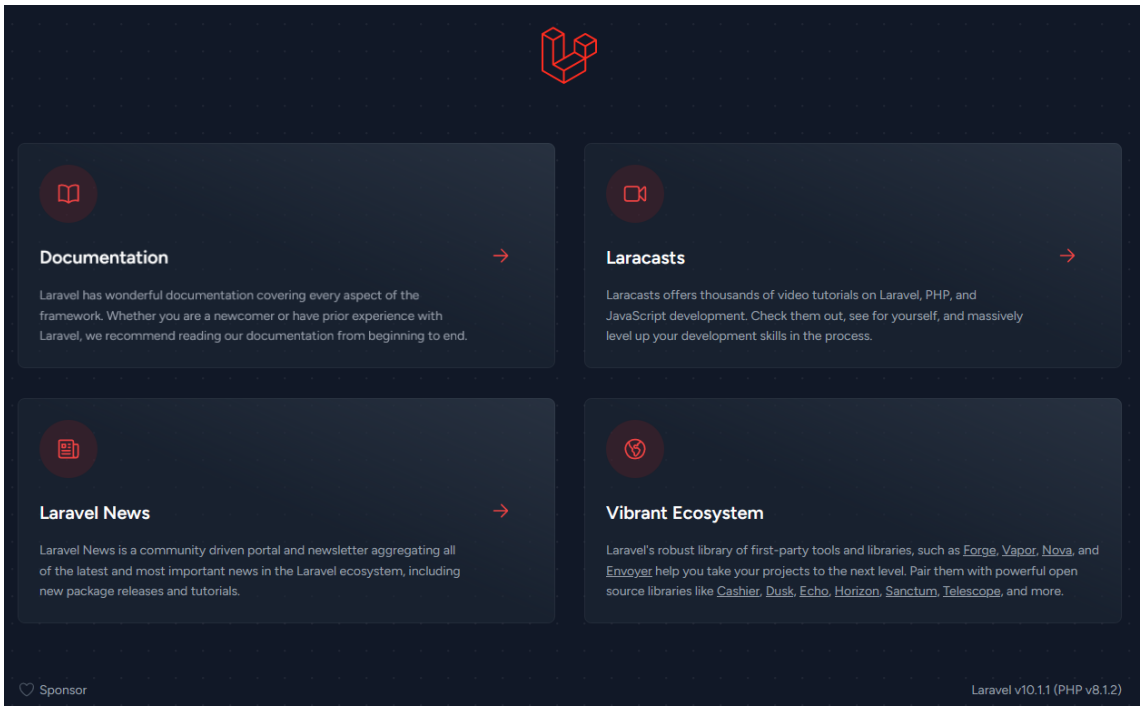


2–6. ábra: Első Laravel webalkalmazásunk kiszolgálásának indítása

Tipp: Artisan-nak hívják a CLI (Command-Line Interface) eszközt, amellyel a terminal-ból tudjuk vezérelni az alkalmazásunkat. A `php artisan` parancs kiadásával terjedelmes listát kapunk a kiadható parancsokról, amelyekhez van egy-egy rövid leírás, de ha valamelyik parancsról több információt, paraméterezési, beállítási lehetőséget szeretnénk megtudni, akkor használhatjuk a `-help` kapcsolót. Az **artisan** fájl a projektünk gyökerében található.

A Laravel keretrendszer verziószámát a `php artisan --version` paranccsal tudjuk lekérni.

Ennek hatására elindul az alkalmazás kiszolgálása úgy, hogy az Apache webszerverre nincs is szükség, mivel a PHP az 5-ös verzió óta biztosít számunkra egy fejlesztési webszervert, ami alapértelmezetten a http://localhost:8000 vagy a http://127.0.0.1:8000 IP címen és porton fogadja a kéréseket a böngészőben. Így most már meg is nyithatjuk a két link bármelyikével az alkalmazásunkat.



2–7. ábra: Első Laravel webalkalmazásunk kezdő oldala

Az alkalmazás kiszolgálását megfigyelhetjük a terminal-ban, hogy milyen kérések érkeztek a böngészőn keresztül és milyen gyorsan kerültek kiszolgálásra. Az alkalmazás futtatását a terminal-ban a **Ctrl + c** billentyűkombinációval tudjuk befejezni. Ha utána szeretnénk megnyitni vagy frissíteni a böngészőnkben a webcímet, akkor már a webhely nem lesz elérhető.

A böngészőválasztást mindenkinek az egyéni ízlésére bízom. Én leginkább Firefox-párti vagyok, de webfejlesztőként kötelező más böngészőkben (Google Chrome-ban, Microsoft Edge-ben és akár az Apple Safari-jában) is kipróbálni a piacra szánt alkalmazásunkat.

Verzióváltás (Upgrade) Laravel 10-ről 11-re

A verzióváltás lehetséges és nem is okoz annyi kellemetlenséget. Először érdemes felfrissíteni a szerver oldali függőségeket egy `composer update` parancs kiadásával a terminal-ban. Amikor minden felfrissült, akkor utána a projekt gyökerében lévő **composer.json** fájlt nyissuk meg, és a következő csomagok verzióit állítsuk be (legalább) ezekre:

- "laravel/framework": "^11.0",
- "laravel/sanctum": "^4.0",
- "nunomaduro/collision": "^8.0"

Ha most újra futtatjuk a `composer update` parancsot, akkor a Laravel keretrendszer verziója felfrissül a 11-esre.

Előfordulhat, hogy vannak a projektünkben további csomagok, amelyek frissítése ütközik a Laravel keretrendszer fő verzióváltásával. Erről a terminal-ban kapunk tájékoztatást. Ha a hibasorokból kiszűrtük, hogy melyik a problémás csomag, akkor arra érdemes rákeresni a https://packagist.org/ oldalon. Ott pedig látható lesz, hogy melyik az érintett csomag legfrissebb verziója, milyen egyéb csomagoktól függ ő, és tőle milyen csomagok függenek. Általában az egy jó megoldás szokott lenni, hogy ha a **composer.json** fájlunkban a legfrissebb csomagverziót adjuk meg neki, majd utána futtatjuk újra a `composer update` parancsot.

További információt erről a fő verzióváltásról a hivatalos dokumentációban találunk: https://laravel.com/docs/11.x/upgrade

2–1. újdonság: Laravel verzióváltás 10-ről 11-re, új függőségi projekt verziókkal

2.2.2. A projekt fájl- és könyvtárstruktúrája, felépítése

Először a projekt mappánk gyökerében lévő fontosabb fájlokat tekintjük át röviden, de a későbbiek során újra és újra találkozni fogunk velük egy-egy adott téma feldolgozása során:

- **.editorconfig:** ez mondja meg a szövegszerkesztőnek (például VSCode), hogy milyen formátumot alkalmazzon a szerkesztés során (ezt a fájlt, akár törölni is lehetne).
- **.env:** (environment), számos beállítást meg tudunk adni itt (alapadatok a projektről, adatbázis elérési információk, üzenetküldés paraméterei stb.). *Megjegyzés:* ezt a fájlt sose fogjuk feltölteni a verziókövető rendszerünkbe (például a GitHub-ra), mivel minden projektben résztvevő fejlesztő a saját környezetére vonatkozóan adhatja meg a beállításokat.
    - Különböző **.env** fájlok lehetnek a lokális, tesztkörnyezetre (**testing**) és az éles (**production**) környezetre is.

- .gitignore: ez meghatározza, hogy miket ne töltsön fel a rendszer soha egy `git push` során. A verziókezeléssel már ebben a fejezetben fogunk találkozni, de a későbbiekben is mindvégig használni fogjuk.
- .gitattributes: ez meghatározza, hogy melyik fájltípusra milyen előírást kövessen a rendszer
- artisan: amikor beírjuk a `php artisan` parancsot a terminal-ba, akkor gyakorlatilag ezt a fájlt hívjuk meg és dolgozza fel a parancsokat, majd végzi el a feladatot és adja vissza az eredményt. Az `artisan` parancsok rettentően megkönnyítik a munkánkat a fejlesztés során, mivel például adott fájlok kezdeti szerkezetét utasításokkal tudjuk meghatározni, de ezen kívül az adatbázis struktúráját is utasításokkal tudjuk majd menedzselni, verziókövetni stb.
- composer.json: a webes projektünk szerver oldali csomagjait, függőségeit tartalmazza.
    - composer.lock: ez is a verziófelügyelethez szükségeltetik, ezt sose módosítsuk, mivel a composer.json fájl alapján generálódik és frissül. A szerver oldali csomagok további függőségeit, részleteit tartalmazza.
- package.json: kliens oldalhoz szükséges csomagokat, függőségeket tartalmazza.
    - package-lock.json: a logika nagyon hasonló a composer.lock fájlhoz, tehát a kliens oldali csomagok további függőségeit is tartalmazza.
- phpunit.xml: a webes projekt tesztelésének körülményeit tartalmazza. A PHPUnit tesztelési eszközt a későbbiekben számos fejezetben fogjuk alkalmazni, konkrétan a phpunit.xml beállításával pedig a 9.3. alfejezetben fogunk találkozni.
- vite.config.js: a kliens oldali fájlok (például: CSS, JavaScript) elérését lehet itt definiálni, amely megkönnyíti például publikálás (deployment) után az ilyen fájlok elérhetőségét útvonalválasztás szempontjából. A Vite eszközzel a 4.3. alfejezetben fogunk részletesebben találkozni.

A fájlokkal továbbra is foglalkozunk, de már a projekt mappáin belül.

- app: az egyik legfontosabb mappa, ahol az alkalmazásunk igazán él a fejlesztéseink által.
    - A mappa gyökerében találhatók a *Model*-ek a Models mappában. A Model-View-Controller tervezési minta részeit a következő, 2.3. alfejezetben fogjuk megismerni.
    - A Http / Controllers mappában találhatók a *Controller*-eink.
    - Ami még kicsit bonyolultnak tűnhet, a köztes rétegeink, *Middleware*-jeink is itt vannak. A nevük beszédes, próbáljuk meg kitalálni, hogy melyik miért felel majd. Ezekkel részletesebben a 10.1. alfejezetben találkozhatunk.
    - A Kernel.php fájlban lehet áttekinteni azokat a Middleware-eket, amelyeket beregisztráltunk (vagyis alapértelmezetten beregisztrálásra került) eddig a rendszerbe. De azért ez még bonyolultabb dolog, szóval ne ijedjünk meg ha nem értünk még mindent ebből. A Laravel 11 pedig ki is iktatja már ezt a fájlt a rendszerből, de a működését nem felejthetjük el vagy tekinthetünk el tőle, mert a logikája megmarad, ha nem is pontosan ugyanúgy kell kezelni, mint a Laravel 10-ben.
    - A Providers mappában található a szolgáltatások támogatására és beregisztrálására (bekötésére) alkalmas fájlok és osztályok. Ez is magasabb szintű tudást igényel, de a

későbbiekben majd ezt is jobban megismerjük. Legelőször az útvonalak kezelésénél (3.5. alfejezet) találkozunk vele, aztán pedig majd a felhasználói hitelesítés (9. fejezet) során.

- **bootstrap:** az app fájl beindítja a Laravel keretrendszert, de ez a mappa tartalmazza a cache mappát is, amely a teljesítmény optimalizálás céljából létrejövő fájlokat tartalmazza (főleg az útvonalak és a szolgáltatások gyorsítótárazott fájljait). Laravel 10-ben ezt a mappát nagyon ritkán szerkesztettük, de a Laravel 11-nél ez a mappa és elemei kulcsszerephez jutnak a működés során. Ennek részleteit például a 10.1.3. alfejezetben is láthatjuk.
- **config:** itt találhatók meg azok a beállítási fájlok (mindegyik egy-egy fontos funkcióért paraméterezéséért felel, tömbök segítségével, kulcs-érték párok megadásával). Az itt található fájlokkal az alkalmazásaink működése testre szabható. A fájlok alapértelmezett értékeket tartalmaznak, amelyeket az **.env** környezeti fájljainkkal felül tudunk definiálni.
- **database** adatbázis struktúra létrehozásáért és menedzseléséért, adatok generálásának biztosításáért felel. A mappa elemei:
    - **factories:** adatgyárakat tartalmazó mappa. Tesztelés szempontjából rendkívül fontos mappa és fájljai. A témát részleteiben a 6.4.2. alfejezetben járjuk körül.
    - **migrations:** itt találhatók a migrációs fájlok, amelyek az egyes adattáblák létrehozásának sémáit tartalmazzák kezdetben, de a későbbiekben az adattáblák változtatásait is itt végezzük el és nem manuálisan az adatbázis menedzser alkalmazásban. A témát részleteiben az 5.2. alfejezetben ismerhetjük meg.
    - **seeders:** kezdeti vagy tesztelési adatfeltöltésért felelős fájlokat, osztályokat tartalmazó mappa. A témát részleteiben a 6.5. alfejezetben járjuk körül.
    - **database.sqlite** fájl alapértelmezetten az SQLite adatbázist tartalmazza, amelyben adatokat tudunk tárolni egyszerűen. A Laravel 11-ben már ez az alapértelmezetten adattárolásra használt fájl. Az adatbáziskezelőt először az 5.1.2. alfejezetben ismerjük meg részleteiben.
- **lang:** azokat a nyelvi mappákat tartalmazza, amelyeken szeretnénk elérhetővé tenni az alkalmazásunkat. Az almappák fájlokba szervezve tartalmazzák, kvázi szótárként, kulcs-érték párokkal az adott nyelvnek megfelelő fordításokat. A témát részletesen a 15.3. alfejezetben ismerhetjük meg.
- **node_modules:** a (főleg) kliens oldali függőségi csomagokat tartalmazó mappa. Addig érintőlegesen többször, de a 15.2.3.2. alfejezetben részleteiben is foglalkozunk ezzel a mappával.
- **public:** a felhasználók számára nyilvánosan (publikusan) elérhető mappa. Az itt található **index.php** fájl az alkalmazáshoz érkező felhasználói kéréseknek az indulási, belépési pontja. Korábban ide kerültek be a JavaScript és CSS fájlok optimalizált kódjai, de ezt a Vite eszköz használata módosította. A publikusan elérhető kép fájlok is ebben a mappában találhatók meg.
- **resources:** itt találhatóak a nézeteink, komponenseink, de itt vannak a JavaScript, SASS, CSS fájljaink is. A nézeteket és komponenseket PHP nyelven, a Blade sablonnyelv segítségével készíthetjük el, és dolgozza fel aztán a keretrendszer. A további kliens oldali erőforrás fájloknak a feldolgozását Vite eszköz készíti elő a kliens oldali felhasználásra. Az erőforrásokkal a 4. fejezetben foglalkozunk először részletesen, de aztán a későbbiekben is még számos alkalommal előkerülnek.

- routes: a mappa fájljaiban az alkalmazás elérési útvonalait tudjuk regisztrálni. Ezeket a felhasználók különböző felületeken keresztül érhetik el (weben keresztül – web.php fájlban létrehozott útvonalakon –, vagy például más alkalmazások által – api.php fájlban regisztrált útvonalakon). De saját artisan parancsokat is tudunk definiálni a console.php fájlban, így a terminal-on keresztül is elérhetővé válik az alkalmazásunk, akár teljesen egyedi saját utasítások segítségével. Az útvonalválasztással majdnem mindegyik fejezetben foglalkozunk, legelőször a következő főfejezetben.
- storage: a webes alkalmazás publikus vagy privát fájljait tárolhatjuk itt, továbbá a naplózásért felelős logs / laravel.log fájlt is itt érhetjük el. Amit még ebben a mappában tárolunk, az a gyorsítótárazást (cache-elést) és a felhasználói munkamenetek kezelését segítő almappák és fájlok.
- tests: a teszteseteket tartalmazza a mappa. A fejezetekben lévő témák feldolgozása során mindig kiemelt figyelmet fordítunk majd arra, hogy teszteljük is a funkcionalitásokat, felületeket, adatokat stb. Legelőször a 3.7.2. alfejezetben fogunk találkozni a tesztelés témájával részleteiben, ekkor az almappa- és fájlszerkezetével kapcsolatos ismereteket is tovább finomítjuk.
- vendor: a szerver oldali függőségi csomagokat tartalmazó mappa. Addig érintőlegesen sokszor, de a 15.2.2.2. alfejezetben részleteiben is foglalkozunk ezzel a mappával.

A Laravel 10 könyvtár- (mappa-) és fájlstruktúrája bővebb, de ha megismerjük a részleteit, akkor meglehetősen átgondolt a kialakítása.

Ezt a struktúrát a Laravel 11 keretrendszer leszűkíti, így több mindent elrejt majd a rendszert felhasználó programozó elől, de a háttérben fontos tudni, hogy az elrejtett részek funkcionalitásai továbbra is ilyen-olyan módon ott vannak a rendszer magjában vagy háttérműködésében.

2.2.3. Laravel 11 projekt létrehozása

A könyv írásakor történt meg a fő verzióváltás a Laravel-ben, így a Laravel 10, a fejlesztői 11-es és a stabil 11-es változat telepítését is végrehajtottam.

2.2.3.1. Laravel 11 fejlesztői (dev) változat

A 11-es verziójú, fejlesztői változatban létező Laravel projektet egy composer paranccsal lehet létrehozni.

A sikeres létrehozáshoz minimum 8.2-es PHP-ra van szükségünk.

```
composer create-project --prefer-dist laravel/laravel laravel-dev dev-master
```

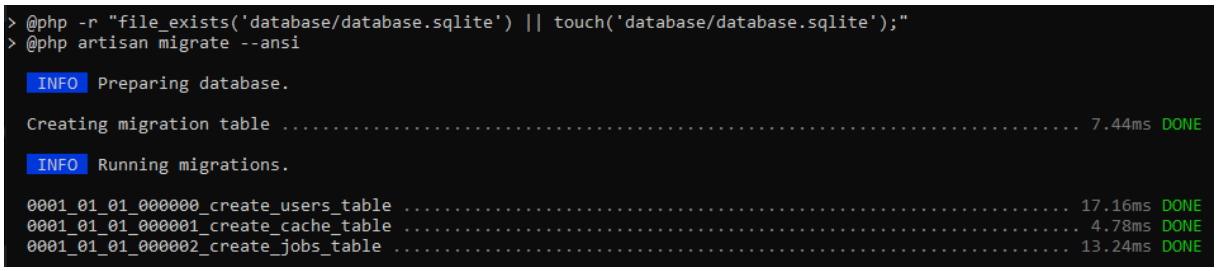
A parancsban lévő paraméterek és magyarázatuk:

- laravel/laravel: ez a csomag tartalmazza a Laravel keretrendszer alapértelmezett fájljait, könyvtárait;
- laravel-dev: a projekt neve, ami gyakorlatilag az újonnan létrejövő könyvtár neve is lesz, ez természetesen módosítható, megváltoztatható;
- dev-master: ez jelképezi 2024. februárjában az újonnan érkező Laravel 11-es verziójú keretrendszert.

Alternatív módon így is lehet telepíteni, ha már a laravel installer alkalmazását telepítettük a rendszerünkben:

```
laravel new projectname --dev
```

Ahogy említettem korábban, a Laravel 11 alapértelmezetten SQLite adatbázist használ, amit már a telepítési parancs lefutásának végén megfigyelhetünk:

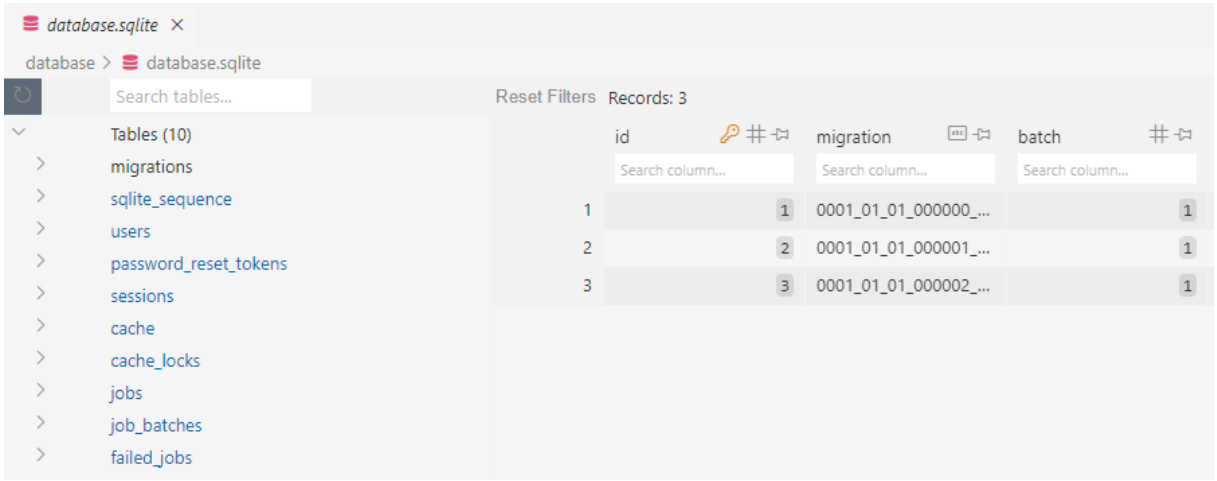


2–8. ábra: Részlet a Laravel 11-es verziójának telepítési folyamatából

1. létrehozza a projekt **database** mappájába a **database.sqlite** fájlt,
2. előkészíti az adatbázist (létrehozza benne a **migrations** táblát),
3. az adatbázisba aztán migrál is néhány alapértelmezetten létrejövő adattáblát (**users**, **cache**, **jobs** stb.).

Az SQLite adatbázis fájlhoz történő kapcsolódásról az 5.1.2. alfejezetben olvashatunk részletesebben, míg a migrációs fájlokról és gyakorlati alkalmazásukról az 5.2.1. alfejezetben.

A létrejövő adatbázisfájl (**database.sqlite**) struktúrája, táblái, adatai a VSCode kiterjesztésével böngészve így néz ki:



2–9. ábra: Adatbázis (database.sqlite) alapértelmezetten létrejövő struktúrája és adatai (részlet)

A Laravel 11-es első alkalmazás fejlődését ebben a GitHub repo-ban lehet majd nyomon követni: https://github.com/gla-elte/l11-first-app

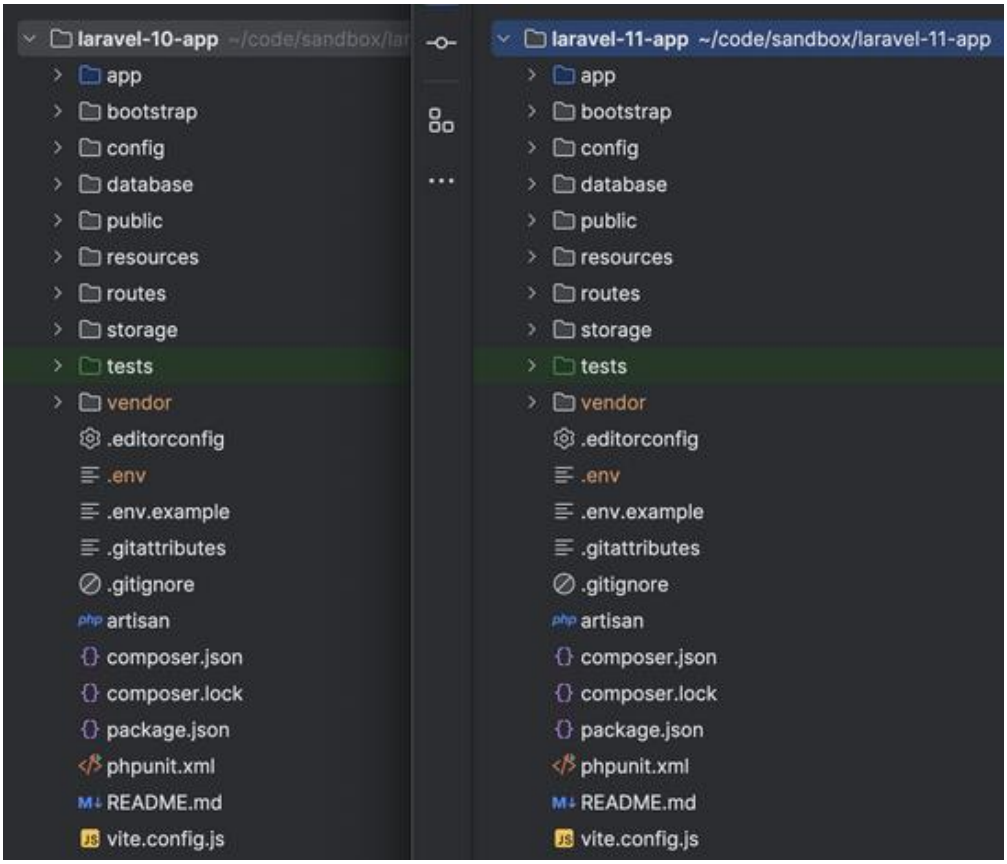
11

Megújult struktúra és tartalom: könyvtár- és fájlszerkezet (*„slimmer application skeleton”* ötlet).

A Laravel keretrendszer legfelső szintű könyvtárstruktúrája ugyanúgy néz ki, mint a 10-es verziónál volt (lásd a következő ábrát). Azonban az *„ördög a részletekben rejlik”*, és majd számos esetben azt fogjuk tapasztalni, hogy a mappából hiányoznak olyan fájlok (számszerűsítve ~69 darab), amelyeket korábban már megszokhattunk esetleg, vagy ez a könyv éppen taglalja azok lényegét és célját. Ettől a struktúra karcsúbb lesz, de azért nem árt (sőt!) ismerni azt, hogy mi is rejlik a háttérben. Olyan fájlok tűntek el többnyire például az app mappából, amelyekhez a legritkább esetben nyúltunk hozzá például a Proivders mappában vagy a Http Middleware fájljaihoz.

A könyvtárak/fájlok elrejtésének oka leginkább az volt, hogy akik újonnan érkeztek a Laravel keretrendszerben fejleszteni más környezetekből, azok ne ijedjenek meg, ne vesszenek el a túl soknak tűnő könyvtárak és fájlok halmazában, hanem így egy kicsit talán könnyebben átláthassák a Laravel keretrendszer szerkezetét.

2–2. újdonság: Megújult könyvtár- és fájlstruktúra



2–10. ábra: Laravel 10-es (balra) és 11-es (jobbra) verziójú projektjeinek struktúrája

2.2.3.2. Laravel 11 stabil változat

A stabil változat telepítéséhez érdemes már használni a Laravel telepítő csomagot. Ezt korábban frissíteni nem tudtuk, ha esetleg régebbről rendelkezésre állt már, akkor először törölni kellett majd utána újra telepítettük:

```
composer global remove laravel/installer
```

```
composer global require laravel/installer
```

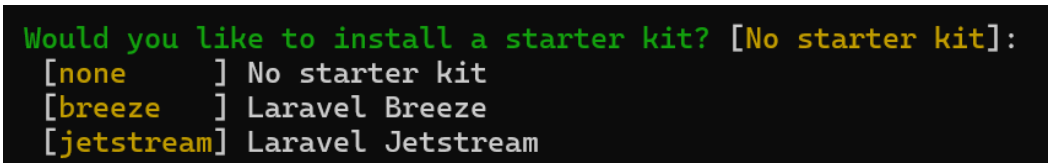
Ez a két parancs (global nélkül) egyébként más csomagok törlésére és telepítésére is alkalmazható a composer segítségével. 2024-ben már működik ennek a csomagnak a frissítése ezzel az utasítással:

```
composer global update laravel/installer
```

Utána az új projekt létrehozása következhet:

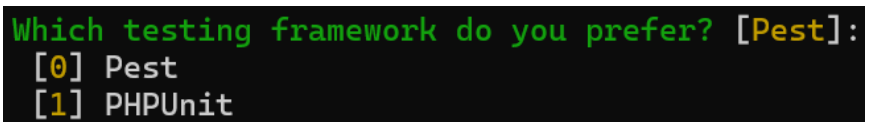
```
laravel new l11-my-first-site
```

Kezdetben néhány kérdést feltesz nekünk a rendszer, mivel például a felhasználói hitelesítés kezdő készletét már az alkalmazás legelején érdemes eldönteni, hogy melyiket szeretnénk majd használni: **Laravel Breeze** vagy **Laravel Jetstream** (a felhasználói hitelesítésről bővebben a 1. fejezetben lesz szó). Ha úgy döntünk, hogy nem szeretnénk még a felhasználókkal és a hitelesítésükkel foglalkozni, akkor a **none** választ adjuk a kérdésre.



2–11. ábra: Felhasználói hitelesítési kezdőcsomag hozzáadásának lehetősége a projekt telepítésekor

Ezután a tesztelési keretrendszerre kérdez rá a telepítő, amelynél a **Pest** és a **PHPUnit** között tudunk választani. Kezdetben válasszuk a PHPUnit-ot, vagyis az 1-es számot, mivel PHP fejlesztések során talán ezzel van nagyobb tapasztalatunk, aztán a későbbiekben majd megismerkedünk a Pest tesztelési keretrendszerrel is.



2–12. ábra: Tesztelési keretrendszer kiválasztása a projekt telepítésekor

Legvégül a Git repository inicializálását kérdezi meg, amelyre válaszolhatunk igennel (yes), így a későbbiekben már egy inicializációs parancsot *„megspóroltunk”* magunknak.

A telepítés zárásakor láthatjuk, hogy a **database / database.sqlite** fájlba már be is kerültek a kezdeti adatbázis tábláink, de a rendszer rákérdez, hogy melyik adatbáziskezelő rendszert akarjuk a Laravel projektünk mögött használni. A Laravel 11 stabil verziója is a fejlesztői verziónál alkalmazott **SQLite**-ot

szeretné használni alapértelmezetten, amelyet egy *enter* megnyomásával mi is megerősíthetünk (ez a későbbiekben bármikor módosítható).

2–13. ábra: Adatbáziskezelő rendszer kiválasztása a projekt telepítésének végén

Így elkészült a stabil Laravel 11-es verziójú webes alkalmazásunk projektje. Beléphetünk a mappába:

```
cd l11-my-first-site
```

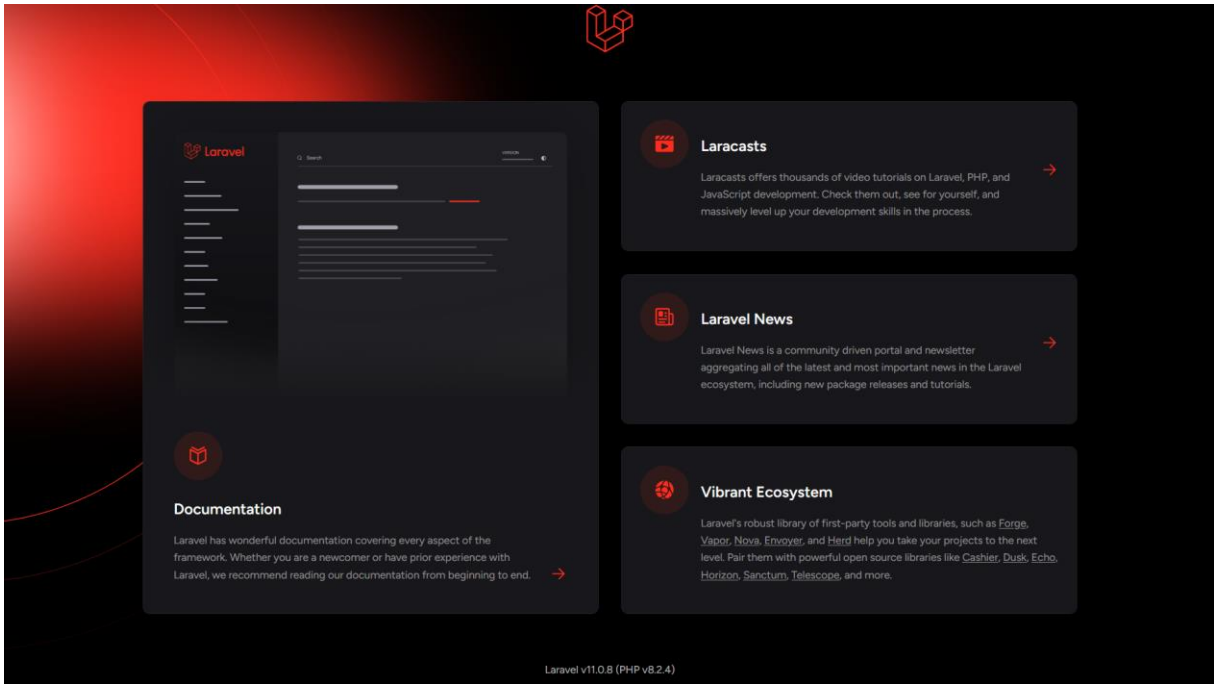
Megnyithatjuk a mappát VSCode-ban:

```
code .
```

Ott pedig a terminal-ban elindíthatjuk az alkalmazásunk futtatását:

```
php artisan serve
```

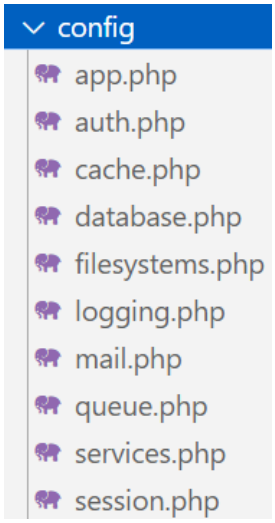
A böngészőben megnyithatjuk az alkalmazásunkat és ezt az alkalmazást kell látnunk:



2–14. ábra: Stabil verziójú Laravel 11 webalkalmazás első futtatása a böngészőben

A weboldal alján látható a Laravel és a PHP verziószámunk, továbbá észrevehetjük, hogy teljesen megújult a **welcome** nézet oldal felülete, stílusa.

A stabil Laravel 11-es verziójú webalkalmazás ezen a GitHub linken elérhető. A stabil változatba végül alapértelmezetten ezek a fájlok kerültek be a **config** mappába:



2–15. ábra: A config mappa alapértelmezetten publikált beállítási fájljai a stabil Laravel 11-es verzióban

2.2.4. Telepítési alternatíva: Docker virtualizációs platform és a Laravel

Előfordulhat, hogy valaki nem a saját környezetében, hanem *„virtualizáltan", „egy konténerben elszeparáltan"* szeretné létrehozni, telepíteni és működtetni az alkalmazásainak, projektjeinek a futtatókörnyezetet, hanem a Docker virtualizációs platformot akarja erre a célra használni. Erre is van lehetőség, ugyanakkor a fejlesztőkörnyezet összerakás ebben az esetben egy picit bonyolultabb, mint ahogy azt a korábbiakban felvázoltam, de előfordulhat, hogy ennek ellenére valaki ezt az irányvonalat szeretné követni.

Habár ennek a könyvnek nem témája a Docker maga, viszont szeretnék ehhez is segítséget nyújtani és itt felhívni a figyelmet a folyamatosan frissülő blog oldalamra. Számos tanulási célú anyag elérhető rajta, többek között a Docker futtatásával és a Laravel *„konténerizálásával"* kapcsolatban is:

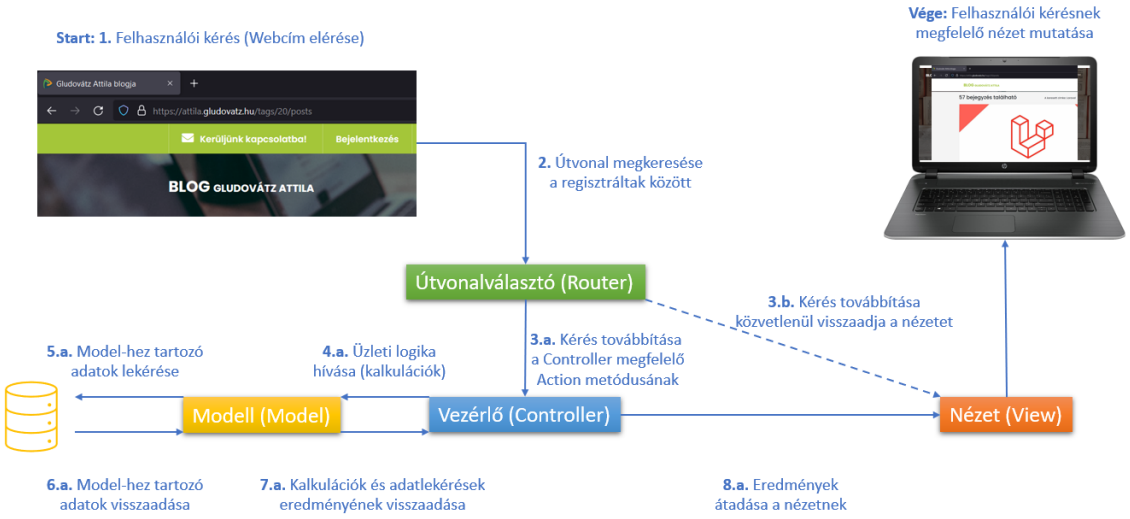
- Docker - 1. rész: Alapok, telepítés
- Docker - 2. rész: Első használatba vétel
- Docker - 3. rész: Egyszerűbb adattárolás és adatkötés
- Docker - 4. rész: Együttműködő konténerek alkalmazása
- Docker - 5. rész: Együttműködő konténerek Docker Compose használatával
- Docker - 6. rész: Legjobb gyakorlatok és technikák
- Laravel Sail - konténerizált projekt

2.3. MVC programtervezési minta és a Laravel

A **Model-View-Controller** egy szoftvertervezési minta, amely a webes alkalmazásoknál nagyon elterjedt. Ahogy a neve is mutatja, három eleme van, amelyek elkülönülnek egymástól és eltérő dologokért felelősek.

2.3.1. Felhasználói kérés kiszolgálása az MVC-ben: általános folyamatábra

Egy webes alkalmazásnál nagyon fontos, hogy megértsük, milyen úton-módon történik meg a felhasználói kéréseknek a kiszolgálása az alkalmazás (és a kapcsolódó elemei, vagyis a Model-View-Controller) által. Ennek megértéséhez összeállítottam egy folyamatábrát (2–16. ábra).



2–16. ábra: Folyamatábra egy felhasználói kérés-kiszolgálás végig követéséhez az MVC architektúrában

A folyamatábrát bal fentről érdemes nézni, mert a felvázolt esetben a felhasználó le szeretné kérni a blog oldalam bizonyos címkéhez (tag-hez) tartozó blogbejegyzéseit. Ezt nagyon egyszerűen meg tudja tenni, akár manuálisan beüti a böngészője címsorába az általa ismert URL-t, vagy a weboldalamon rákattint egy hivatkozásra (1. lépés). Innentől a felhasználó feladatai véget érnek és átveszi az alkalmazásom a tőle érkező kérést, amit megpróbál tökéletesen kiszolgálni.

Az alkalmazás első fontos szereplője, az **Útvonalválasztó (Router)** a kért webcím alap URL utáni részét a webalkalmazáson belüli regisztrált *útvonalak* között elkezdi keresni, ez a 2. lépés. Az Útvonalválasztó az, akinél a programozó regisztrálja az útvonalakat, hogy mely URL-eken akarja fogadni a felhasználóit. Biztos mindenki futott már bele olyan *„404 - Not found"* hibakódba, amit akkor kap, ha rossz URL-t írt be a böngészőbe. Ez a Laravel esetén akkor fordulhat elő, ha olyan URL-t szeretne elérni a felhasználó, amit a programozó nem regisztrált be korábban. Az útvonalválasztóra tehát egy amolyan hotel recepcióként tekinthetünk: a leendő vendégek (felhasználók) bemennek a hotelbe és le akarják foglalni az adott számú szobát, ha az létezik (és még szabad is...), akkor elirányítja oda a vendéget a recepciós, a felhasználó pedig örül ennek, mert megkapja a szobáját.

A 3. lépéstől kettéválik a felhasználói kérés kiszolgálása. Létezik egy a hosszabb ág és egy b rövidebb ág. A b ággal kezdem, mert ez az egyszerűsített kiszolgálás és kezdetben majd mi is ilyeneket fogunk csinálni, hogy aztán majd később az a ág bejárását is rutinosan végre tudjuk majd hajtani.

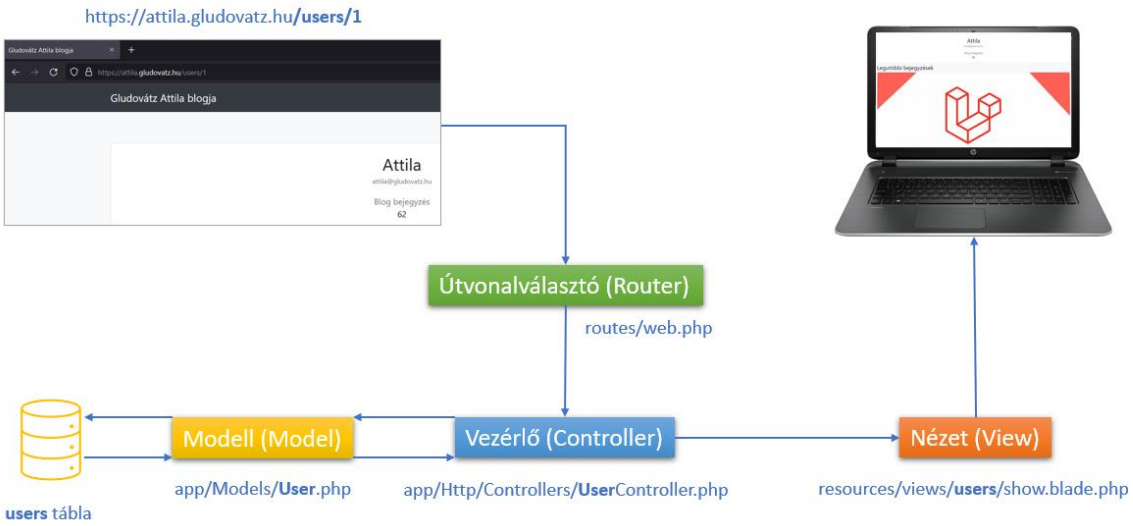
A 3.b. lépés alapján, ha létezik a felhasználó által kért útvonal az alkalmazásban, akkor egyszerűen megadjuk az útvonalhoz tartozó nézetet, amit aztán visszaküld a felhasználó böngészőjének megjelenítésre. Ez az eset (ág) tipikusan akkor szokott előfordulni, amikor egy útvonalhoz hozzá van rendelve egy *statikus* Nézetünk (View), amely nem tartalmaz kalkulációkat vagy adatbázisból kinyert adatokat. Ilyen statikus nézet lehet például egy felhasználót köszöntő kezdőoldal, ami például egyszerűen a HTML és CSS segítségével megjelenít neki egy ilyen statikus weboldalt.

A 3.a. lépéssel kezdődő a ágon akkor haladunk végig, ha a felhasználó bizonyos *dinamikus* tartalmakhoz szeretne hozzáférni az alkalmazásunkban. Az MVC esetén az útvonalválasztóból leggyakrabban a Vezérlő (Controller) felé megy tovább a felhasználói kérés és annak kiszolgálása. Ő az, aki majd meg tudja kérni a Model osztályokat (4.a. lépés), hogy például számítsanak ki valamit (a szállodás terminológiánál maradva, a foglalni kívánt időszakra a foglalás végösszegét), vagy (5.a. lépés) kérjenek le adatot az adatbázisból (például a szoba foglaltságát). Amit utána visszakap adathalmazt (6.a. lépés) és/vagy kalkulációt (7.a. lépés) a Modell (Model) elemtől, azt utána tovább tudja adni a megfelelő Nézetnek (View) (8.a. lépés). Így ezen az ágon is visszajuthatunk a felhasználó böngészőjéhez, amely megjeleníti a kérés kiszolgálásának eredményét. *(Megjegyzés: Előfordulhat ennél az ágnál, hogy az 5.a. és 6.a. lépések kimaradnak, ha nincsen szükség a kalkulációk elvégzéséhez az adatbázisból származó adatokra, de ez azért elég ritka.)*

Bár az iménti a ágleírás tartalmazza, azért kiemelem külön is, hogy hogyan működhet és milyen szerepe van a Model és View elemeknek. A Model elemek biztosítják a hozzáférést az adatbázisban lévő adatokhoz és ők végeznek el számításokat, amelyek miatt is őket tekintjük az alkalmazás üzleti logikai rétegének. A View elemek fogják megjeleníteni a felhasználóknak a kérés eredményét (az adatokkal és/vagy a kalkulációkkal együtt) a böngészőjükben, ők képviselik tehát a megjelenítési réteget, amelyen keresztül a felhasználó interakcióba tud lépni a webes alkalmazásunkkal.

2.3.2. Felhasználói kérés kiszolgálása az MVC-alapú Laravel keretrendszerben

Az általános folyamatábra áttekintése után itt bemutatom, hogy a Laravel keretrendszerben hogyan történik meg a felhasználói kérések kiszolgálása és melyik MVC elem játszik szerepet benne (2–17. ábra).



2–17. ábra: Folyamatábra egy látogatói kérés kiszolgálásáról a Laravel keretrendszerben

Vegyünk egy Laravel keretrendszerre épülő weboldalt, például a saját blogomat. Az egyszerű látogató felkeres egy webcímet és a böngészőjébe beír egy URL-t. Legyen ez a következő: https://attila.gludovatz.hu/users/1 A látogató így meg tudja tekinteni az 1-es számú regisztrált felhasználót (és hogy milyen blogbejegyzéseket írt az oldalra).

A **users/1** útvonal létezik a regisztrált útvonalak között, vagyis létre van hozva a **routes / web.php** fájlban. Így a kérés kiszolgálása itt nem akadhat meg, hanem tovább küldésre kerül a felhasználókat kezelő vezérlő felé, ami a Laravel könyvtár- és fájlstruktúrájában az **app / Http / Controllers / UserController.php** helyen érhető el. A **UserController** megfelelő (**show()** nevű) metódusában a **User** Model osztály (**app / Models / User.php**) segítségével lekérjük az adatbázisban lévő **users** tábla 1-es **id**-val (azonosítóval) rendelkező sorát. Ezután a **UserController** az adatbázisból visszakapott konkrét felhasználói adatsort (a kiegészítő blogbejegyzéseket tartalmazó információkkal együtt) továbbítja a **resources / views / users / show.blade.php** nézetnek, amely elvégzi a megjelenítés összeállítását és visszaküldi a látogató böngészőjének a tartalmakat a kinézet stílusával együtt. A folyamatábrán a fájlok elérésének címét írtam ki, a vastagon szedett részek változhatnak a különböző lekérések kapcsán, például, ha a blogbejegyzéseket akarnánk lekérni akkor a **User**-t ki kellene cserélni **Post**-ra és a **users**-t **posts**-ra, viszont a nem vastagon szedett címrészek változatlanul használhatók maradnának.

Ha relációs adatbázisunk van (táblák sorokkal és oszlopokkal) a webalkalmazás mögött, akkor érdemes úgy elnevezni a Model fájljainkat, amilyen nevet akarunk adni az adattáblának az adatbázisban. Például **User.php** Model fájl a user**s** nevű adattáblához adja majd a hozzáférést (az **s** betűt azért tettem vastaggá, mert az angol elnevezésben a Model osztály többesszámát használja az adattábla létrehozásához), így pusztán a névazonosság (névkonvenció) biztosítja az összeköttetést a Model fájl és az adattábla között. (Persze adhatunk más nevet a Model fájlunknak, mint az adattáblának, de ki akarna pluszban akár csak egy-két sort is programozni, ha nem muszáj ezt megtennie.)

A View fájlok tipikusan HTML(szerű) fájlok, amelyek a Blade nevű sablont használ(hat)ják. Emiatt minden nézet fájlnak a neve **.blade.php**-re végződik, például: **show.blade.php** fájl.

A Laravel MVC legfőbb elemeinek, összetevőinek helye a mappastruktúrában:

- Kezdetben az *útvonalainkat* tartalmazó fájl a routes mappában lévő web.php lesz (de a routes mappában a legtöbb fájlba útvonalakat tudunk elhelyezni, és így regisztrálni tudjuk őket a fájlokon belül. A fájlok nevei az útvonalak elérési jellegéből adódó csoportosítást segítik, például a web.php a weben keresztüli eléréshez biztosítja az útvonalakat, az api.php-ban az API[3]-n keresztül való elérés útvonalait tudjuk regisztrálni).
- *Model fájlok:* az app / Models mappában vannak ezek a fájlok (korábbi verziókban – a 6-os előtt – egyszerűen az app mappában voltak, de most így már beszédesebb, hogy egy kitüntetett helyük van az app / Models mappában).
- *View fájlok:* a resources / views mappában vannak (mappa szerint érdemes ezen belül is rendezni őket, például a felhasználók megjelenítését a users mappába, a blogbejegyzések megjelenítéseit, nézet fájljait a posts mappába helyezzük el, logikusan.
- *Controller fájlok:* app / Http / Controllers mappában vannak a vezérlő fájlok. Talán a nézetekhez képest itt ritkább esetben (alkalmazás méretétől és struktúrájától függ) van szükség arra, hogy még további könyvtárakba szervezzük a fájlokat, de rögtön hozok is egy ellenpéldát erre, a felhasználói hitelesítés megvalósításakor a rendszer az app / Http / Controllers / Auth mappába helyezi el a hozzá tartozó Controller-eket.

Az MVC tervezési mintára épülő keretrendszerek egyik legnagyobb erőssége az, hogy *egyszerűen csak működik,* de a háttérben minden a névkonvencióra épül. Ez a számunkra annyit jelent, hogy ha betartjuk a névadási szabályokat, akkor az MVC-re épülő keretrendszerek (amilyen a Laravel is), segítenek nekünk mindenben, például a funkcionalitások létrehozását is sokkal egyszerűbbé teszik a számunkra.

2.4. Laravel projektünk verziókövetése a GitHub segítségével

A verziókövetés témakörének ismerete manapság már alapvető elvárás egy programozótól. Olyan sok előnyt nyújt a számunkra ez a terület, hogy nem is lehetne az összeset felsorolni (alább azért majd megpróbálom). De én azt hangsúlyoznám ki, hogy azért használjuk ezt a verziókövetést a könyv feldolgozása során, hogy az általam végigvezetett projekteken folyamatosan láthassátok, mikor, milyen változtatásokat hajtottam végre. Így ez is egyfajta változáskövető naplóként szolgál majd nekünk és segíti az előrehaladást mindenki számára.

Ha valaki a fejlesztőkörnyezet kiépítésénél már telepítette a Git-et a gépére, akkor ezzel a kezdeti elvárás már teljesült is. Az első Laravel webalkalmazásunkat (my-first-site) is létrehoztuk már, úgyhogy nincs más hátra a kezdő lépések végrehajtásából, mint az, hogy *„beindítsuk"* a verziókövetést a projektünknél.

2.4.1. Verziókezelés alapjai

Egy verziókezelő rendszer követi a változásokat egy fájlon vagy fájlok csoportjain, így képesek vagyunk általa a következőkre:

- Fájl vagy az egész projekt visszaállítása hiba esetén (kvázi biztonsági mentés).

[3] Application Programming Interface

- A felelősség végigkövetése: látszódik, hogy a szoftver melyik részét ki írta, ki módosította. Meg lehet nézni, hogy ki módosított utoljára (ki hibázott).
- Változások végigkövetése kommentek segítségével.
- Változtatásokat a projekt adminisztrátora engedélyezheti (legalábbis GitHub esetén).
- Fejlesztési ágak, branch-ek létrehozása, így könnyebb új funkcionalitásokat fejleszteni az alkalmazáshoz, amelyeket aztán sikeres futtatások esetén össze lehet fűzni.

A verziókövetés alapfogalmai és műveletei:

- **Repository**: tárhely, amelyen a projekt van.
- **Add**: hozzáadjuk a változásokat a repository-hoz, a hozzáadás után a fájl / fájlok már feltölthetők a távoli repository-ba. Ekkor még a fájlok változásai (különbségei: **diff** = *difference*) visszavonhatók, eldobhatók (**Revert**).
- **Commit**: változtatások érvényre juttatása, gyakorlatilag egy változáscsomag, amely több fájlt / mappát is érinthet. Névadás segíthet a változáscsomag jellemzésében, későbbi nyomon követésénél, azonosításánál.
- **Push**: saját gépünkön lévő munkánk feltöltése távoli repository-ra.
- **Pull**: szinkronizáció, távoli repository fájljainak letöltése. Ha valamilyen új dologba kezdünk bele, előbb mindig frissítsük az alkalmazásunkat a repository-ban lévő legfrissebb verzióval.
- **Clone**: teljes repository lemásolása saját gépre.
- **Branch**: fejlesztés ágazata (új projekt esetén a kezdeti ág neve alapértelmezetten master volt régen, most már main a neve). Így a meglévő kódból (fő ágból) kiindulva új ág hozható létre, amelyen a fejlesztést végre lehet hajtani. A *„régi"* kód a fő ágon még karbantartható és kezelhető, az új ág pedig majd összefűzésre (**Merge**) kerülhet a fő ágban, ha már a kívánalmaknak megfelelően működik és hibamentes is.
- Az utóbbi összefűzéssel kapcsolatos a **Conflict** fogalma. Ha konfliktus, lényegi eltérés van a fő ág és az újonnan fejlesztett ágak fájl módosításai között és a verziókezelő rendszer nem tudja eldönteni, hogy érvényre jusson-e a változás (vagy melyik jusson érvényre), akkor ezt a konfliktust kezelni kell. Tipikusan akkor van szükség a konfliktuskezelésre, ha többen dolgoznak ugyanazon a fájlon és másképp módosítják: ekkor a legelső, leggyorsabb fejlesztő változtatása érvényre jut, de a következő(k)é már nem, hiszen a kód változtatása már érvényre jutott az előző fejlesztő által. Ilyenkor konfliktus keletkezik, amelyet többnyire manuálisan kell lekezelni, megoldani.
- **Tag**: verziószámot és szöveges leírását adhatunk hozzá az elvégzett programmódosításhoz, illetve annak érvényre juttatásához.

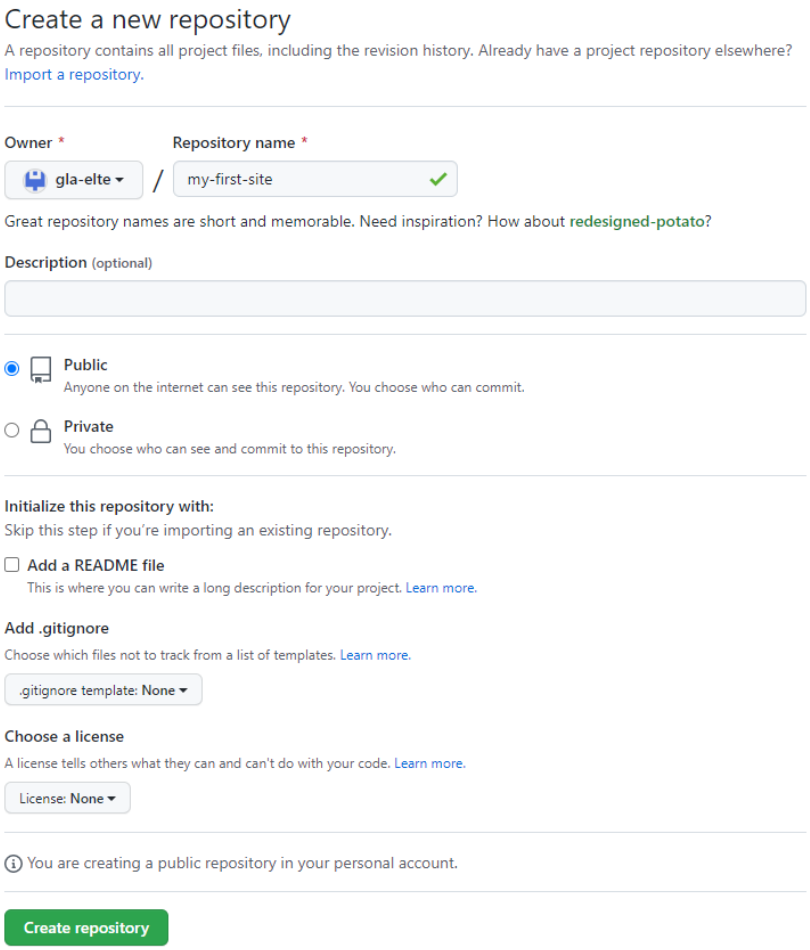
2.4.2. Verziókezelés GitHub segítségével

GitHub-os regisztráció és bejelentkezés után létre tudjuk hozni ott a saját *„repo"-nkat* (egy közkedvelt rövidítése a repository szónak). A repo-mnak ugyanazt a nevet fogom adni, mint amit a webalkalmazás telepítésekor adtam a projektnek: **my-first-site**. Majd kattinthatunk a **Create repository** gombra alul (2–18. ábra).

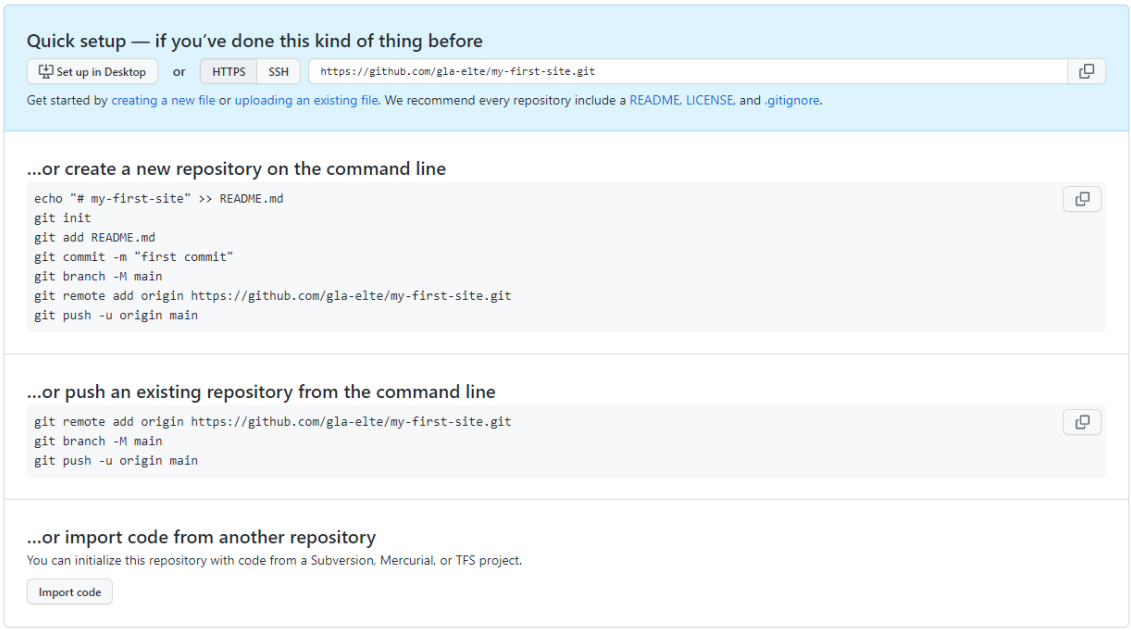
Természetesen adhatunk a projektünkhöz leírást, **README.md** fájlt (a Laravel alapértelmezetten hozzáad majd egyet), vagy akár priváttá is tehetjük, ha nem akarjuk, hogy a külvilágból elérjék, én azonban most

publikusan hagyom és majd linkelem is az alfejezet végén a saját repo-mat. Így, ha valaki a későbbiek során elakadna, mindig látni fogja, hogy mit is csináltunk az adott részben az aktuális projektünkben.

A repository létrehozása után több lehetőségünk is van arra, hogy hozzákössük a saját gépünkön lévő projekt mappánkhoz a repo-t (2–19. ábra). Ebben a leírás segít is bennünket.



2–18. ábra: Első Laravel repository létrehozása a GitHub segítségével



2–19. ábra: Üres GitHub repository feltöltésének lehetőségei

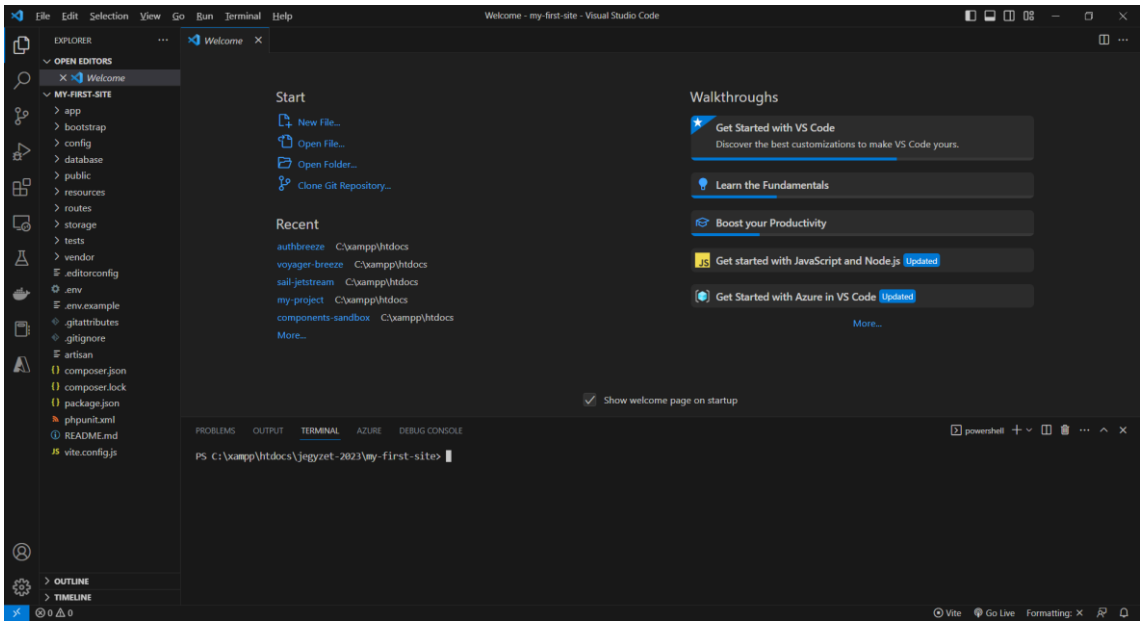
2.4.3. Verziókezelés a Visual Studio Code-ban

Mivel a saját gépen lévő projekt mappánk még nem tekinthető repo-nak, ezért először ezt alakítsuk ki. Ha még nyitva van az a terminal-unk, amivel létrehoztuk korábban a projektünket, akkor érdemes általa megnyitni a VSCode-ot, ugyanis ennek a terminal-ját sokkal kényelmesebb lesz használni most és majd a jövőben is (ez persze ízlés dolga). A VSCode megnyitása az adott mappával a legegyszerűbb így:

```
code .
```

A megnyitott mappát érdemes egy kicsit böngészni a VSCode-ban, és felfedezni bennük a „2.3.2. Felhasználói kérés kiszolgálása az MVC-alapú Laravel keretrendszerben" alfejezet végén taglalt könyvtárakat és fájlokat (app, routes, resources és almappáik, fájljaik).

A VSCode-ban új terminal-t a Terminal -> New Terminal menüponttal tudunk előhozni. Ez alapértelmezetten egy PowerShell (PS) típusú terminal lesz, de ha a „**+**" jel melletti listát lenyitom, akkor további más típusú terminal-okat is megnyithatnék. Példaként egyet említenék meg, ami mindenkinél megvan, az a *„Git Bash"* típusú terminal, mert a Git-et telepítettük már korábban. A mostani Git-es feladatainkhoz viszont nem feltétlenül kell az, tökéletesen megfelelő hozzá a PS-es terminal is.



2–20. ábra: Projektünk megnyitása utáni VSCode kezdőképernyő (bal oldalt a könyvtárstruktúra, alul a megnyitott terminal, középre kerülnek majd a megnyitott kódfájlok)

Mindenekelőtt állítsuk be a git-es felhasználónevünket és e-mail címünket a terminal-ban, ha még sohasem használtunk volna git-et:

```
git config --global user.name "Your Name"
```

Az idézőjelek közti részt cseréljük ki a saját felhasználónevünkre, majd utána ugyanígy az e-mail beállításánál is:

```
git config --global user.email "youremail@yourdomain.com"
```

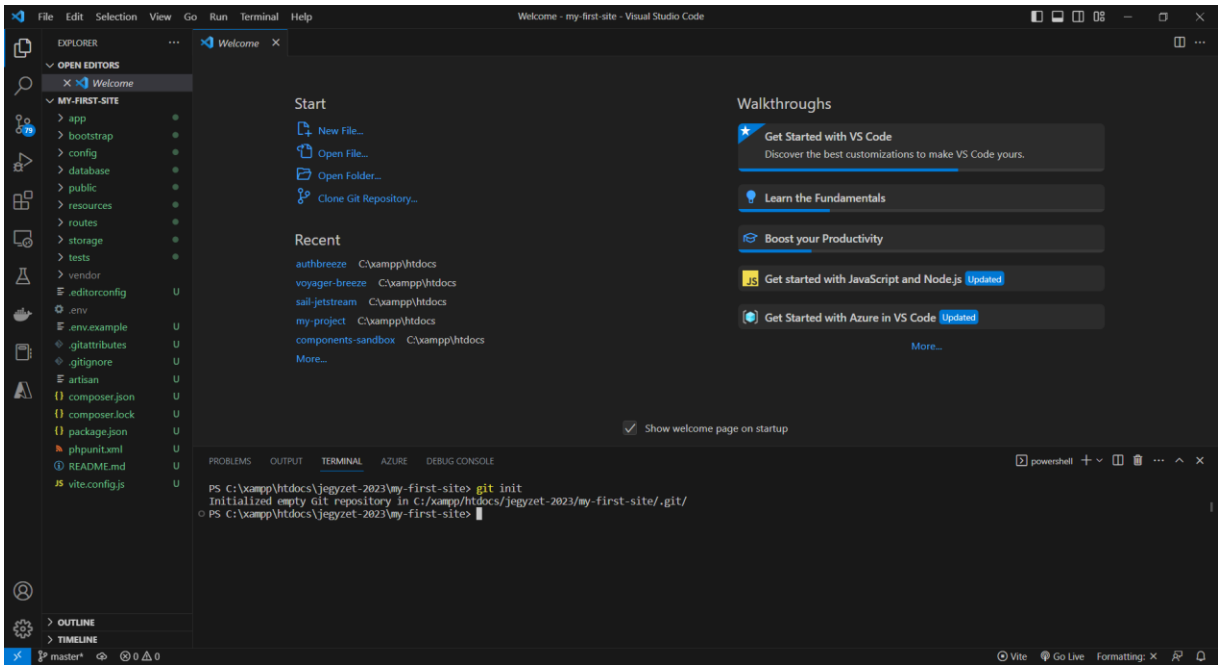
Ellenőrizni is tudjuk, hogy sikerült-e végrehajtanunk a beállításainkat:

```
git config --list
```

Ezután vágjunk akkor bele a helyi projektünk és a távoli (GitHub-on lévő) repo-k összekötésébe. Elsőként inicializáljuk a VSCode terminal-jában a projektünket repo-ként, kvázi jelezzük a rendszernek, hogy ez egy Git-es verziókezelővel nyomon követett alkalmazás lesz:

```
git init
```

Ennek a parancsnak a kiadásával egy üres Git repo-nk jött létre helyben a projekt mappánkban. Továbbá ez egy rejtett **.git** nevű mappát hoz létre a projekt könyvtárában, amely a verziókezeléshez szükséges fájlokat tartalmazza (Windows alatt Vezérlőpultban kell beállítani a rejtett állományok megjelenítését, hogy látszódjon a rejtett – ponttal kezdődő – mappa).



2–21. ábra: Az inicializáló parancs kiadása után megváltozik a mappák/fájlok színezése

Tipikus hibák lehetnek, ha nem sikerült az inicializálás:

1. Nem települt fel rendesen a Git, emiatt azt írja rá a rendszer, hogy ismeretlen parancs... ekkor ellenőrizzük le újra azokat a dolgokat, amelyeket az előfeltételek teljesülése során megtettünk a Git kapcsán.
2. Nem a megfelelő mappában adtuk ki a parancsot. Ez a mappa, ahogy korábban már említettem is: **C:\xampp\htdocs\my-first-site**. *„Ha nem ott lennénk”* a prompt alapján, akkor navigáljunk el oda a `cd` parancs(ok) kiadásával, vagy pedig nyissunk meg egy új VSCode ablakot, az Open Folder menüpontot válasszuk ki a File menüben, tallózzuk be a **my-first-site** webalkalmazásunk gyökér mappáját, és kattintsunk a megnyitásra. Ekkor a VSCode-ban bal oldalt látszódik is a projektünk mappa és fájlszerkezete (ha mégsem látszódik, akkor válasszuk ki a függőleges menüből a felső Explorer nevű  ikont).

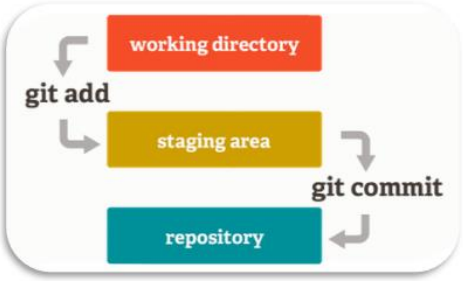
Látható (2–21. ábra), hogy nálam megtörtént az inicializáció és a mappák, fájlok színe meg is változott. A legtöbb mappa – bár én sajnos színtévesztő vagyok –, zöld színű lett, ami arra utal, hogy ezeket még nem adtuk hozzá a helyi munka könyvtárunkhoz, ezt azonban mindjárt pótoljuk. De mielőtt megtennénk, felhívnám a figyelmet a könyvtár- és fájllistában két elemre: a **vendor** mappára és az **.env** fájlra. Ezeket nem fogjuk feltölteni (szinkronizálni) a GitHub-ra, majd a későbbiekben rátérek, hogy miért is nem. Azt pedig, hogy mit nem akarunk feltölteni a GitHub-ra, a **.gitignore** fájlban tudjuk megtenni soronként felsorolva (ha rákattintunk a fájlra és megnézzük, akkor láthatjuk, hogy a **vendor** mappa és az **.env** fájl benne is van a listában).

A `git status` parancs kiadásával láthatjuk is, hogy még egy `commit`-ot sem hajtottunk végre, és még semmilyen fájlt nem *„verziókövetünk”*. Tegyük most ezt meg, adjuk hozzá a fájlokat és mappákat a helyi munkakönyvtárunkhoz, vagyis *„verziókövessük őket”:* ezt megtehetnénk egyesével a fájlokkal vagy

mappákkal és a jövőben legtöbbször leginkább arra lesz szükség, hogy egyesével vagy kisebb csomagban adjunk hozzá fájlokat / mappákat a helyi munkakönyvtárunkhoz, azonban most ezt, a kezdeti inicializációs lépés után adjunk hozzá minden mappát és fájlt (amik nem szerepelnek a **.gitignore** fájlban):

```
git add .
```

Itt a sorvégi pont fontos karakter, ez fog minden mappát és fájlt hozzáadni a helyi munkakönyvtárunkhoz. A fájljaink és mappáink így a *„staging area"-ba* kerülnek (2–22. ábra). Ha újra lekérnénk a `git status`-t, akkor láthatjuk az eredményt az előző futtatáshoz képest. Továbbá a VSCode-ban a fájlok neve melletti **U** (vagyis **U**ntracked, nem követett) betű is megváltozik és **A** (vagyis **A**dded, hozzáadott) lesz.



2–22. ábra: Git parancsok kódfájljainkra és mappáinkra gyakorolt hatásai

Következhet a `git commit` parancs kiadása, azonban ezt lássuk el egy `-m` kapcsolóval is, amelynek segítéségével nevet (*„üzenetet"*, megjegyzést) adhatunk a commit-nak:

```
git commit -m "first commit"
```

Ezután a GitHub ajánlásának megfelelően a **branch**-et (fejlesztési ágat) nevezzük át a jelenlegi **master**-ről **main**-re (mielőtt kiadnánk a parancsot, figyeljük meg a VSCode-ban bal alul a fejlesztési ág nevét, ami **master**, utána ez fog megváltozni **main**-re):

```
git branch -M main
```

Összeköthetjük ezekután a *„staging area"*-nkat a távoli GitHub repository-val (nyilván itt az URL-ben ki kell cserélni a felhasználónevet a saját repo-tok címében):

```
git remote add origin https://github.com/<felhasználónév>/my-first-site.git
```

2.4.4. Kitérő: GitHub-os felhasználói hitelesítés

2021. augusztusa óta a GitHub-os felhasználói hitelesítéshez nem feltétlenül elég, ha megadjuk neki parancssorból a felhasználónevünket, e-mail címünket, majd egy felugró kis ablakban a jelszavunkat. Egy hozzáférési token segítségével tudjuk azonosítani magunkat (és a gépünket) a GitHub felé, amikor szinkronizálni akarunk valamit a GitHub segítségével. A token megszerzéséhez a böngészőnkben jelentkezzünk be a GitHub oldalára és kövessük végig ezt a folyamatot:

1. Jobb felül kattintsunk a felhasználói profilképünkre és válasszuk ki a lenyíló listából a *„Settings"* menüpontot.
2. A bal oldali menüben válasszuk ki a (legalsó) *„<> Developer Settings"* menüpontot.

3. A bal oldali menüben válasszuk ki a *„Personal access tokens"* lenyíló listából a *„Tokens (classic)"* menüpontot.
4. Jobb oldalon kattintsunk rá a *„Generate new token"* gombra, majd a lenyíló listából válasszuk a *„Generate new token (classic)"* menüpontot.
5. Kérni fogja a jelszavunkat, amit adjunk meg neki.
6. Eljutunk így a *„New personal access token (classic)"* lapig, ahol egy űrlapot kell kitölteni.
    a. A *„Note"* (megjegyzés) mezőt elvileg nem kötelező megadni, de ha nem írjuk be, akkor a végén reklamálni fog ezért, úgyhogy írjuk be, hogy hol, miért és mire fogjuk használni ezt a hozzáférést.
    b. Az *„Expiration"* (lejárat) kötelező mező, meg kell adni, hogy mennyi ideig szeretnénk használni ezt a hozzáférési token-t. Alapértelmezetten 30 nap van kiválasztva, de ha huzamosabb ideig szeretnénk használni, akkor módosítsuk ezt, például a *„No expiration"* opcióra, bár a GitHub ezt nem javasolja a biztonsági kockázat miatt.
    c. A következő hosszabb szekcióban ki kell választani, hogy milyen funkcionalitásokat szeretnénk engedélyeztetni ennek a token-nek a birtokában. Például ezzel a token-nel szeretnénk a repo-inkhoz hozzáférni. Ezért azt javaslom, hogy válasszuk ki a *„repo"* csoport legelső elemét (így a többi eleme is kiválasztásra kerül).
    d. Végül az űrlap alján kattintsunk a *„Generate token"* gombra.
7. Ha az űrlap kitöltésénél minden rendben volt, akkor megkapjuk a token-t, ami egy hosszú, véletlenszerűen generált karaktersorozat. Erre szükségünk lesz a további lépések során, úgyhogy másoljuk ki a vágólapunkra.
8. Nyissuk meg a *„Vezérlőpult"*-ot (Control Panel) és ha esetleg nem látszik minden vezérlőpult elem, akkor jobb felül a *„Megtekintés a következő szerint:"* listát tegyük át a *„Nagy ikonok"* opcióra.
9. Válasszuk ki a *„Hitelesítőadat-kezelő"* (Credential Manager) menüpontot.
10. Válasszuk ki a *„Windows rendszerbeli hitelesítő adatok"* (Windows Credentials) lapfület.
11. Az *„Általános hitelesítő adatok"* (Generic Credentials) szekció fejléc szélén kattintsunk rá az *„Általános hitelesítő adat hozzáadás"* (Add a generic credential) linkre.
12. Az új ablakban megnyílik egy űrlap, ahova a hitelesítési adatokat meg kell adnunk:
    a. Internet vagy hálózati cím (Internet or network address): git:https://github.com
    b. Felhasználónév (User name): a saját felhasználónevünk
    c. Jelszó (Password): amit a vágólapra kimásoltunk hosszú karaktersorozat (token)
    d. Majd OK
13. Ezután nyitnunk kell egy új terminal-t a VSCode-ban.

2.4.5. Visszatérés: a terminal-os működtetéshez

Az előző folyamat végén következhet a `git push` utasítás kiadása, amivel feltöltjük a GitHub-ra a webalkalmazásunkat. A későbbiekben elég kiadni a `git push` utasítást, csak legelőször kell ilyen *„hosszan megfogalmazni".*

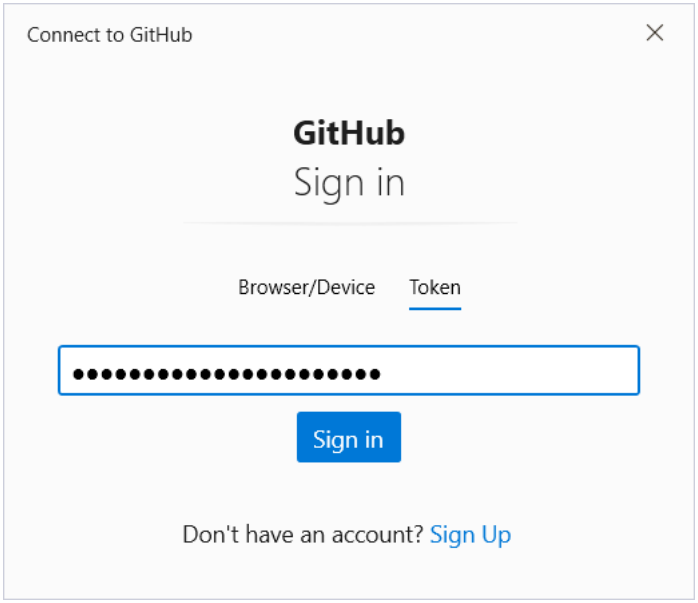
```
git push -u origin main
```

Ha hibamentesen végigfut a folyamat, akkor elégedettek lehetünk (2–27. ábra), de ez így még nem teljesen jó nálam. Az alább kiírt 401-es (Unauthorized) HTTP állapotkód (hiba) szerint még mindig bizalmatlan velünk szemben a GitHub, ezért újra bejelentkeztet majd.

```
PS C:\xampp\htdocs\jegyzet-2023\my-first-site> git push -u origin main
fatal: Response status code does not indicate success: 401 (Unauthorized).
```

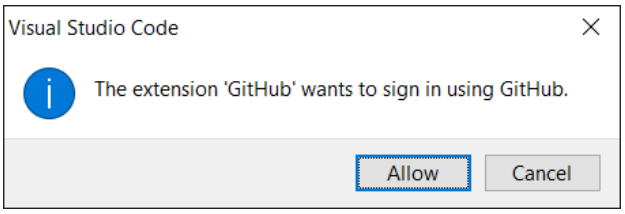
2–23. ábra: Lehetséges 401-es HTTP hibakód a terminal-ban

Ha a token-es bejelentkezést választjuk, akkor a bejelentkezésnél válasszuk a *„Token"* lapfület és illesszük be ide is a vágólapunkon lévő token-t, majd *„Sign in"* (2–24. ábra).

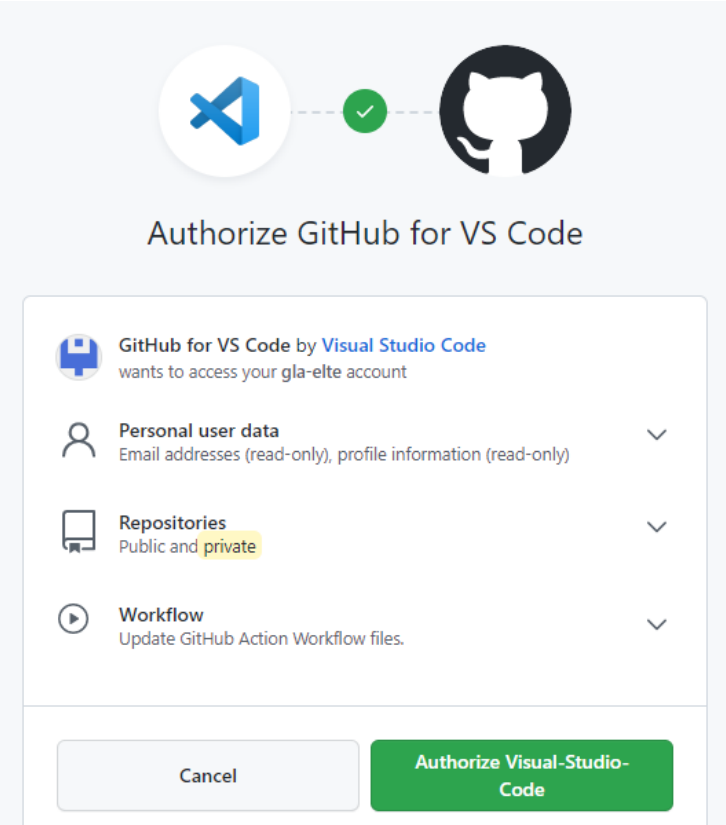


2–24. ábra: Bejelentkezés a GitHub-ra Token segítségével

Megjegyzés: nálam a VSCode-ban van egy GitHub kiterjesztés (extension) is, ami szintén hozzáférést kér a GitHub-hoz. Az *„Allow"* gomb megnyomásával (2–25. ábra) a böngészőben is tudom engedélyeztetni ezt az *„Authorize Visual-Studio Code"* gomb megnyomásával (2–26. ábra).



2–25. ábra: VSCode kiterjesztés engedélyeztetésre rákérdezés



2–26. ábra: VSCode és GitHub összekötésének engedélyezése

Ezután már minden bizonnyal megfelelően le kell futnia a git push utasítás folyamatának (2–27. ábra).

```
PS C:\xampp\htdocs\jegyzet-2023\my-first-site> git push -u origin main
fatal: Response status code does not indicate success: 401 (Unauthorized).
Enumerating objects: 102, done.
Counting objects: 100% (102/102), done.
Delta compression using up to 12 threads
Compressing objects: 100% (84/84), done.
Writing objects: 100% (102/102), 71.06 KiB | 3.95 MiB/s, done.
Total 102 (delta 6), reused 0 (delta 0), pack-reused 0
remote: Resolving deltas: 100% (6/6), done.
To https://github.com/gla-elte/my-first-site.git
 * [new branch]      main -> main
branch 'main' set up to track 'origin/main'.
PS C:\xampp\htdocs\jegyzet-2023\my-first-site>
```

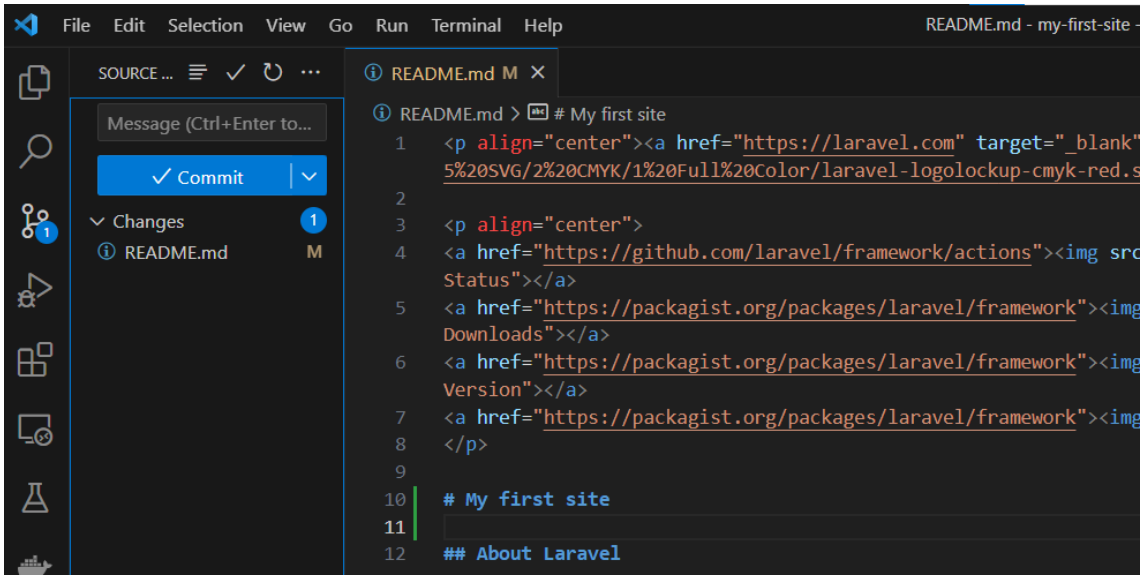
2–27. ábra: „git push” utasítás sikeres lefutása

Sikeresen feltöltöttük a GitHub-ra a Laravel projektünket! Az eredmény ezen a címen látható (felhasználónév cserélendő): https://github.com/<felhasználónév>/my-first-site (az én konkrét repo-m itt érhető el).

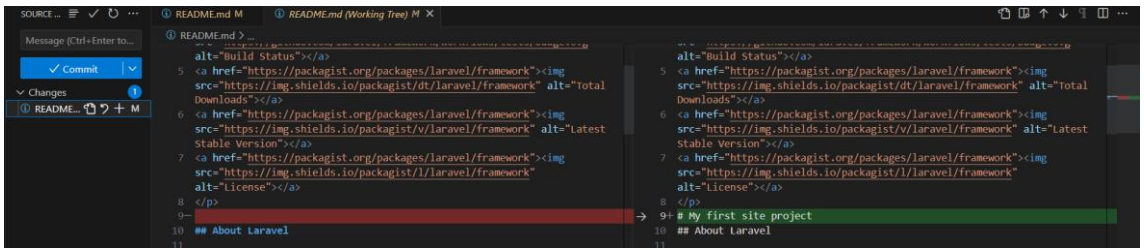
2.4.6. Verziókövetés támogatása a VSCode-ban grafikus módon

A VSCode is segítségünkre van annak kapcsán, hogy Git-tel kövessük az alkalmazásunk verzióit. Mutatok egy példát:

1. Nyissuk meg a VSCode-ban a **README.md** fájlt (ez az a fájl, aminek a tartalma látszódik a GitHub projektünk könyvtár és fájl listája alatt is). Úgynevezett GitHub Markup nyelven van benne tartalom. Egy kódkészletet használva tudunk formázott szöveget létrehozni egyszerű szöveges tartalmakból. Például a ## (dupla kettőskereszttel) kezdődő szövegrészek címsorok lesznek a GitHub-on a feldolgozás után.
2. Másoljuk le a 10. sort (## About Laravel) és illesszük be elé újra, de írjuk át erre: # My first site szöveget és legyen előtte-utána is egy-egy üres sor (2–28. ábra).
3. Ezt a VSCode mutatja nekünk a sorok száma mellett színesen kiemelve, hogy ez új sor. A bal oldali függőleges menüben pedig kiválasztottam Source Control menüpontot, ami jelzi, hogy jelenleg 1 fájl változott a legutóbb GitHub-ra push-olt változathoz képest (2–28. ábra). Mivel a *„Changes"* lenyíló lista alatt van most a fájlunk, ezért ez még a *„working directory"-ban* van. Futtathatnám újra a terminal-ban a `git add .` vagy jelen esetben a `git add readme.md` parancsot mivel csak ez az egy fájl változott és átkerülne a módosítás a *„staging area"-ba,* de megtehetjük ezt úgy is, hogy a fájl mellett a *„+"*jelre kattintunk, ha fölé visszük az egeret, és így kevesebbet kell gépelni. Ha pedig rákattintunk a Changes alatt a fájl nevére, akkor megnyitja a fájlt a VSCode egy osztott ablakban és kiszínezi nekünk, hogy mi változott a fájlban (2–29. ábra).



2–28. ábra: Változások grafikus jelzése a VSCode-ban

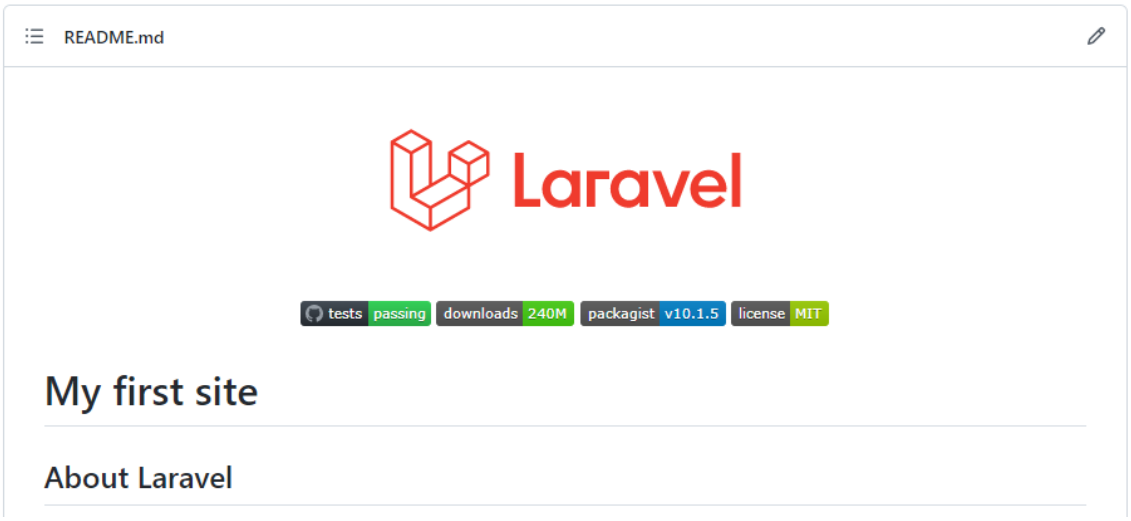


2–29. ábra: Változás megtekintése

4. Kattintsunk a **+** jelre a fájl neve mellett. Kis idő után most már nem a *„Changes"* szekció alatt lesz a fájlunk neve, hanem a *„Staged Changes"* részben. Ismét írhatnánk a terminal-ba a `git commit`

-m "readme.md módosítása" parancsot, de megtehetjük mindezt úgy is, hogy a *„Staged Changes”* feletti *„Message”* szövegdobozba írjuk be a változtatáshoz tartozó üzenetet, majd a felette lévő ✓ Commit gombot. Ekkor megjelenik egy gomb ugyanott a következő szöveggel: *„Sync Changes 1”*. Ezt megnyomva, gyakorlatilag a háttérben egy git push utasítás hajtódik végre (esetleg rákérdez, hogy biztos akarjuk-e push-olni a változtatást és rányomhatunk, hogy OK és hogy többet ne jelenjen meg ez az ablak).

5. Újra megnézhetjük a frissített GitHub projektünk weboldalát és a könyvtár és fájl lista alatt most már az új *„My first site”* címsor szöveg is megjelenik (2–30. ábra).



2–30. ábra: README.md fájl módosított tartalma a GitHub oldalon

A fájllistában a **readme.md** fájl mellett látható, hogy melyik commit-tel módosult utoljára, és akár részletesen tudjuk böngészni az adott commit-ok érintett módosításait, fájljait is, valamint egy fájl történetét is meg tudjuk nézni, hogy milyen commit-ek milyen változásokat okoztak bennük. Egyelőre nekünk most elég ennyire ismernünk a GitHub-ot és a verziókövetést. A későbbiekben, ahol releváns, úgyis visszatérünk még a finomságaira.

Én alapból szeretem a kódszerkesztőmet sötét témával használni (*„így kevésbé látszanak a hibák”* – egy kis programozói humor). Témát váltani a VSCode-ban egyszerűen a File -> Preferences -> Theme -> Color Theme menüponttal, vagy még egyszerűbben a Ctrl + k és utána a Ctrl + t billentyűkombinációkkal lehetséges. Viszont a VSCode-ban a jobb láthatóság (a könyv kinyomtatásakor a kevesebb nyomtatófesték elhasználása miatt) áttérek egy világosabb témára a későbbiekben. Az általam választott téma neve: **Quiet Light**.

Még két megjegyzés az alfejezet végére, hogy ha a VSCode-tól több segítséget várunk el a Git-es verziókövetés segítésére, akkor a Gitlens kiterjesztést érdemes lehet telepíteni. (VSCode-ban nagyon hasonlóan működnek a kiterjesztések, mint a Firefox-ban vagy a Chrome-ban. Kipróbálhatjuk őket, és ha mégsem tetszenek, akkor lehet őket törölni is).

2.5. Összegzés

A fejezet során megismerkedtünk a fejlesztés előkövetelményeivel, magával a futtató- és a fejlesztőkörnyezettel.

Az MVC tervezési mintát nem csak a Laravel kapcsán érdemes ismerni, mivel számos egyéb keretrendszer is erre épül, emiatt mindenképpen hasznos volt, hogy áttekintettük a működését, egy felhasználói kérés kiszolgálásának folyamatát és az egyes szereplők feladatait. Ezután azonosítottuk a Laravel keretrendszer könyvtárstruktúrájában az MVC tervezési minta legfontosabb szereplőit.

Végül áttekintettük a Git verziókezelő rendszer használatának alapvetéseit, és a VSCode segítségével az első Laravel projektünket feltöltöttük a GitHub felületére, ahol mostantól kezdve mindig nyomon tudjuk követni a projektjeink változásait, együtt tudunk dolgozni társainkkal és például külön fejlesztési ágakat is létrehozhatunk a funkcionalitásainknak.

3. Útvonalválasztás (Routing)

Amikor a felhasználó beírja a böngészőbe a weboldal címét, akkor gyakorlatilag egy útvonalat kér le az alkalmazásunktól. Ez az alkalmazásunk belépési pontja, innen indul minden. A fejezet során áttekintjük az útvonalválasztás alapjait. Az adatok átadását útvonalakon keresztül a nézetnek, vezérlőnek. Továbbá olyan generikus útvonalakat is létrehozunk, amelyek illeszkednek a kapott útvonal sablonra és aszerint, adatvezérelten működnek utána. A fejezet végén a tesztelés témakörét is röviden bemutatom, illetve még azt, hogy a Laravel-ben hogyan is működik ez hatékonyan.

Az útvonalválasztás (routing) és az útvonalak regisztrációja elég sokat változott az idők és az újabb Laravel keretrendszer verziók publikálása során, úgyhogy ezekre a változásokra is rávilágítok majd, ahogy haladunk előre.

3.1. Alapok

A Laravel dokumentációja ebben a témában is sok hasznos dolgot mutat be példákon keresztül, érdemes vetni rá egy pillantást, amikor az útvonalakat vesszük górcső alá. Főleg, mivel tapasztalataim szerint a legutóbbi verziókban ez változott talán a legtöbbször, hogy hogyan is kell őket kezelni. Úgyhogy érdemes mindig figyelni, ha éppen valamilyen problémára keresünk megoldást, hogy milyen Laravel keretrendszer verzióhoz írnak, javasolnak megoldásokat a különböző fórumokon, mert könnyedén előfordulhat, hogy például, ami a Laravel 7-ben még működött, az a 10-ben már nem fog, és fordítva.

A Laravel-lel való ismerkedést érdemes az útvonalválasztóval, útvonalak regisztrálásával kezdeni, mert gyorsan, látványos dolgokat tudunk elérni vele. Kezdetben a legfőbb fájl, amivel foglalkozunk: a **routes** mappában a **web.php**. Nyissuk meg!

Ami legelőször feltűnhet a PHP fájlban, hogy egy többsoros megjegyzés fogad minket, ami el is magyarázza röviden, hogy mi ez a fájl, mire használható: útvonalakat tudunk itt regisztrálni a web-es elérésekhez. Amit még rögtön észrevehetünk, hogy objektumorientált PHP-t használ a keretrendszer, tehát vannak osztályaink, metódusaink, amelyeket aztán újra felhasználás céljából meg tudunk hívni más osztályokból is.

Tipp: mivel most már fejleszteni fogjuk a Laravel-t a VSCode-ban, érdemes telepíteni hozzá olyan kiegészítőket , amelyek támogatják, segítik a fejlesztést. Én a *„Laravel Extension Pack"*-et javaslom, ami igazából egy kiterjesztés csomag. Ez tartalmaz több olyan hasznos kiegészítést a VSCode-hoz, ami a jövőben a segítségünkre lesz.

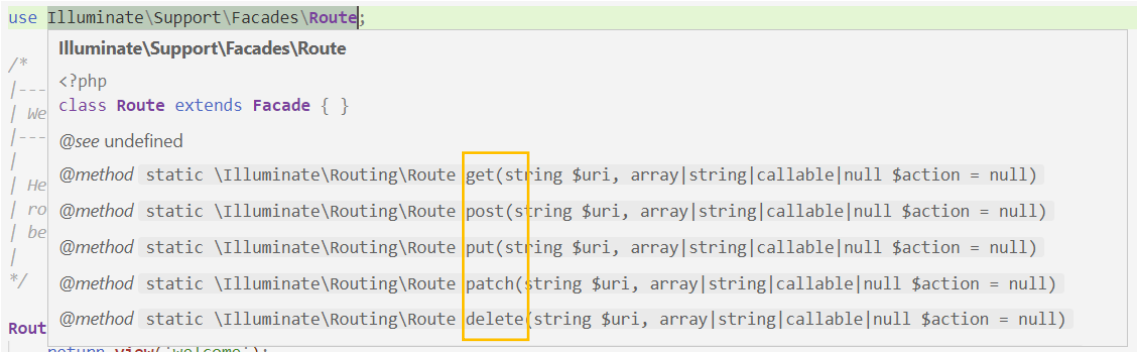
Amikor először elindítottuk az alkalmazásunk kiszolgálását a PHP fejlesztési webszerver segítségével (2.2. alfejezetben), a `php artisan serve` paranccsal, akkor a 127.0.0.1:8000 IP címen és port számon volt elérhető az alkalmazásunk nyitó oldala. Ez a kezdőoldal a **/** (perjel) útvonalat jelenti az alkalmazásban, amit láthatunk is ebben az egy utasításban.

```
Route::get('/', function () {
    return view('welcome');
```

```
});
```

3–1. kódrészlet: A kezdőoldal útvonala a web.php fájlban

Az látható még itt, hogy a **Route** nevű osztályt és annak a **get()** nevű statikus metódusát használjuk. A **Route** osztályt a Laravel keretrendszer magja adja nekünk, amely itt a fájl tetején importálásra került a **use** kulcsszó után az osztálynévvel. Ha fölé visszük az egeret az itt megjelölt **Route** osztálynak, akkor már látható is egy részlet abból, hogy milyen statikus metódusokat kínál még nekünk (3–1. ábra).



3–1. ábra: Route osztály statikus metódusai (részlet)

Ez csak egy részlet volt, de ha rákattintunk a **Route**-ra és megnyomjuk az F12 billentyűt, akkor rögtön el is irányít minket a VSCode erre az osztályra. Többek között ez is egy nagy előnye annak, hogy nyílt forráskódú a keretrendszer, mivel mindennek a végére tudunk járni, meg tudjuk nézni, hogy mi hogyan működik a rendszer magjában. Módosítani persze ne módosítsuk a mag kódját, meg van annak is a módja, hogy hogyan lehet felülírni valamit, amit a rendszer nyújt nekünk, de mi még nem tartunk ezen a ponton.

Térjünk vissza az 3–1. kódrészletünkre és gondoljuk át, hogy mi lehet az a **get()** metódus. A WEB-en dolgozunk, ezért fontos, hogy ismerjük a HTTP (HyperText Transfer Protocol) protokollt, mivel ezt használjuk. A protokoll a felhasználói kéréseknél meghatároz egy metóduslistát, ami gyakorlatilag definiálja, hogy hogyan, milyen módokon lehet kommunikálni a protokoll szabályai szerint. A metóduslistáról itt van egy részletesebb leírás, de a 3–1. ábra mutat is nekünk ezek közül néhányat: get, post, put, patch, delete. Leggyakrabban útvonalat szeretnének lekérni a felhasználók, amihez a HTTP GET metódusát használják, úgyhogy itt is ez húzódik meg a háttérben, amikor a Laravel **Route** osztályának a **get()** metódusát hívjuk és adjuk vissza az eredményt a felhasználóknak.

A **get** metóduson belül két paraméterünk van, vesszővel elválasztva egymástól. Az első paraméter volt a kezdőoldal **/** jele, míg a második paraméter ebben az esetben egy *„névtelen”* függvényhívás, amely egyetlen **return** utasítással rendelkezik. A return utasítás egy view-val, vagyis nézettel fog visszatérni a felhasználói kérésre válaszként. A **view** metódus itt egy Laravel-es segédfüggvény (helper), ennek paraméteréül kell adni a visszaadandó nézet nevét. Amit ide beírunk, akkor azt a keretrendszer a mappaszerkezetben a **resources / views** mappában fogja keresni, és ha megtalálja, visszaadja a felhasználó böngészőjének eredményül. Ez mind abból adódik, hogy a Laravel-ben vannak névkonvenciók, szabályok, amelyeket, ha betartunk, akkor helyesen fog működni az alkalmazás. A **welcome** nézetet fogja visszaadni a rendszer az útvonal lekérésénél, ami a **resources / views** mappában van, mégpedig ez a **welcome.blade.php** fájl (a VSCode segít amúgy az útvonalnál, mert Ctrl billentyű lenyomása közben, ha

rákattintunk a welcome szóra a **view()** segédmetóduson belül, akkor rögtön oda navigál minket a nézet fájlra – ha létezik). Ennek a **.php** kiterjesztésű fájlnak nagyobbik része egyszerű HTML és CSS kód, kiegészítve Blade specifikus elemekkel. A Blade sablon motorról (template engine) fogunk tanulni a 4. fejezetben, egyelőre legyen elég annyit tudni, hogy minden általunk definiálni kívánt nézet fájlhoz (welcome, about, contact, vagy bármi egyéb nevet találunk ki) írjuk oda *„kiterjesztésként”,* hogy **.blade.php**, és ekkor működni fog az alkalmazásunk.

Ha még nem tettük volna meg, indítsuk el az alkalmazásunk kiszolgálását a `php artisan serve` paranccsal.

Ha a **welcome.blade.php** fájlban módosítjuk a kódot (134. sorban – később akár változhat is, hogy melyik sor is tartalmazza ezt az oldalon jobb alul megjelenő feliratot – írjuk át a *„Laravel”*-t *„My first Laravel App”*-ra), majd mentsük el, és frissítsük a böngészőben a http://127.0.0.1:8000/ weboldalunkat. Eredményül nagyon hasonló dolgot kapunk, mint amit a korábbi, 2–7. ábra is mutat, de a jobb alsó sarokban már megváltozott az alkalmazásunk neve a módosításunk hatására.

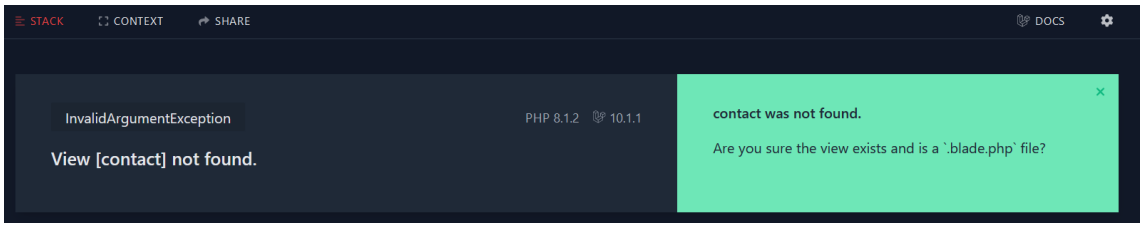
Definiáljunk most mi magunk egy új útvonalat. Az egyszerűség kedvéért lemásolhatjuk a már meglévő utasítást és azt a beillesztés után már módosíthatjuk.

```php
Route::get('/contact', function () {
    return view('contact');
});
```

3–2. kódrészlet: Új kapcsolati útvonal regisztrálása

Itt egy kapcsolati oldalt szeretnénk elérni majd, ha a böngészőbe beírjuk ezt: http://127.0.0.1:8000/contact

Ha ezt megtesszük, akkor egy Laravel keretrendszerre jellemző hibaoldalt fogunk kapni, ami nagy segítségünkre tud lenni most is és a jövőben is, mivel eléggé beszédes információkat oszt meg velünk a hibáról (3–2. ábra).



3–2. ábra: Hiba, a nézet nem található

Bal felül írja, hogy a **contact** nevű nézet nem található, jobb felül pedig rá is kérdez, hogy biztosan létezik-e a nézet fájl és **.blade.php** a kiterjesztése. Mivel még nem hoztuk létre ezt a fájlt, így biztosan ennek a hiánya okozza a problémát. A későbbiekben, amikor nem ennyire egyértelmű a probléma megoldása, akkor a fenti *„CONTEXT”* menüpont is a segítségünkre lehet, amely mutatja a felhasználói kérés fejlécét, törzsét, az útvonalat és az alkalmazás verzióit. Ezek szintén hasznos információk lehetnek a hibák okának kinyomozásában és megoldásában.

Most hozzuk létre a **resources / views** mappában a **contact.blade.php** fájlt, hogy ez ne okozhasson problémát. A tartalmát is adjuk meg, de csak nagyon egyszerűen jelezzük, hogy mi is ez:

```
<h1>Ez egy kapcsolati oldal.</h1>
```

3–3. kódrészlet: Kapcsolati nézet oldal tartalma

Most, ha ráfrissítünk a korábbi hibát jelző oldalra, már visszakapjuk az iménti egyszerű kiíratást arról, hogy ez egy kapcsolati oldal.

Vegyük észre, hogy amiket most megvalósítottunk, az a korábbi 2.3.1. MVC-t tárgyaló fejezetben a 2–16. ábra szerinti 3.b. ágat mutatja, amikor az útvonaltól közvetlenül a nézetet visszaadva történik meg a felhasználói kérésnek a kiszolgálása.

Amikor ilyen egyszerűsített ágat jár be a felhasználói kérés, akkor lehetőségünk van arra, hogy egyszerűsítsük az útvonalainkat, például a **contact**-ot így (az első paraméter az útvonal, a második pedig a nézet neve):

```
Route::view('/contact', 'contact');
```

3–4. kódrészlet: Contact útvonal regisztrálása egyszerűsített formában

Az útvonalunk és a nézetünk már megvan, viszont mivel manuálisan írtuk be az útvonalat a böngészőbe, esetleg az is előfordulhatott volna, hogy elírjuk a címet: http://127.0.0.1:8000/contac (lehagyjuk a szó végi t betűt). Ekkor *„404 / NOT FOUND"* hibajelzést kapunk, ami azt jelenti, hogy az útvonal nem található (a regisztrált útvonalak között). A Laravel keretrendszer biztosít a számunkra szép, egyszerű, letisztult nézeteket a leggyakoribb hibakódokhoz. Ezeket a nézeteket felül is lehet majd definiálni és saját hibakód oldalakat is létrehozhatunk, például úgy, hogy magyarul írjuk ki a hibaüzenetet a felhasználónak (lásd majd a 3.3. alfejezetet).

Új alapértelmezetten létező útvonal: /up

A Laravel az alkalmazásunk helyes működését (*„egészségi állapotát"*) ellenőrző új útvonalat alapértelmezetten, a projekt létrehozáskor már tartalmazza.

A Laravel 11 alapértelmezetten web-ről (böngészőből) érkező felhasználói kérések kiszolgálására van felkészülve. A más forrásokból érkező kérések feldolgozását a 3.6. alfejezetben fogjuk megismerni először.

3–1. újdonság: Projekt helyes működését ellenőrző útvonal

3.1.1. Útvonalakhoz kapcsolódó artisan parancsok

Az útvonalakhoz kapcsolódó artisan parancsok lekérhetők a következő utasítással:

```
php artisan route –help
```

Három fontos utasítás tartozik hozzá, amelyekkel tudjuk kezelni az útvonalakat:

1. A legfontosabb a route:list, amellyel a regisztrált útvonalakat tudjuk böngészni, szűrni, rendezni őket.

2. A route:cache és a route:clear utasítások az útvonalak gyorsítótárazását támogatják (az első eltárol és így gyorsítja az alkalmazás útvonal regisztrációt, a második törli az útvonalakat a gyorsítótárból). *Fontos!* Ne használjuk a gyorsítótárazást (cache) addig, ameddig fejlesztjük az oldalt, mivel sok bosszúságot tud okozni, hogy nem történik változás, amíg újra nem *„gyorsítótárazzuk"* például az útvonalakat, ha felviszünk egy új útvonalat a **web.php**-ban. Gyorsítótárat akkor használjunk, ha elkészülés közelébe ért a webes alkalmazásunk és már csak a teljesítményen szeretnénk javítani.

A route:list utasítással egy kvázi táblázatban kapjuk meg a regisztrált útvonalakat. Az oszlopokban megkapjuk az elérés HTTP metódusát (GET, POST stb.), magát az elérési útvonalat és hogy milyen Controller/metódus veszi át az útvonal alapján a felhasználói kérés kiszolgálását. Ez az utasítás sokszor segítségünkre lesz még a munkánk során, főleg akkor, ha a webalkalmazásunk jó sok útvonallal rendelkezik már, akkor érdemes szűrni az utasítással az útvonalak között. A szűréshez információt kaphatunk a következő utasítással:

```
php artisan route:list –help
```

Néhány példa az útvonalak szűrésére:

- php artisan route:list --only-vendor
  - Csak a Laravel keretrendszerhez tartozó alapértelmezett útvonalakat adja vissza.
- php artisan route:list --method=post
  - Csak a POST metódus szerinti (adatküldési) útvonalakat adja vissza.
- php artisan route:list --path=post
  - A fejezet során később ismertetett *„post"*-ot tartalmazó útvonalakat adja vissza.

Az alfejezetben végrehajtott programkód-módosítások ebben a GitHub commit-ben érhetők el.

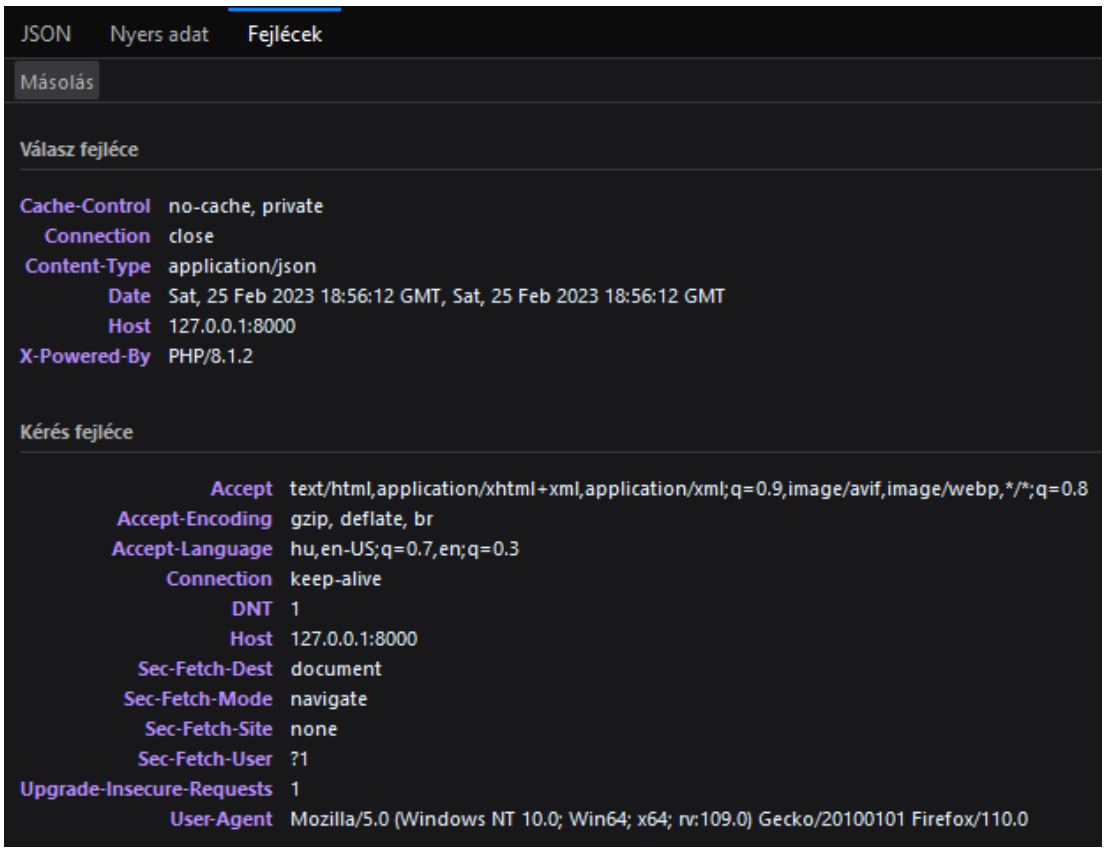
3.2. Adatküldés a nézetnek

Azt már tisztáztuk, hogy az MVC tervezési minta milyen ágakon képes kiszolgálni a felhasználói kéréseket. Ebben az alfejezetben azt nézzük meg, hogy a kérések kiszolgálása során hogyan tudunk adatot visszaküldeni a felhasználónak. Most még csak az útvonal fájlon keresztül tesszük ezt meg, de a későbbiekben a Controller felhasználásakor is hasonlóan fogjuk csinálni, úgyhogy érdemes már most jól elsajátítani ezeket a technikákat.

Kezdésként próbáljunk ki két dolgot, ugyanúgy a **routes / web.php** fájlnál és majd a nézet fájljaink szerkesztésénél maradunk:

1. Amikor az útvonal visszatérése még nem egy konkrét nézet, hanem csak egy szöveg. Például a főoldal útvonalánál egy **return 'Hello world';** utasítást írjunk meg és hozzuk be a főoldalt a böngészőben.
   a. Azt tapasztalhatjuk, hogy egy sima kiíratás történik meg.

2. Ezután próbáljuk ki azt, amikor nem csak egy sima szöveggel, hanem egy tömbbel térünk vissza ugyanott. Például kommenteljük ki az előző **return** utasítást előbb, majd így: **return ['foo' => 'bar'];**
    a. Ekkor a Firefox böngészőben egy JSON-t kapunk vissza és a definiált értéket, Chrome-ban nálam csak a nyers adat jelenik meg. Érdemes a böngészőben különböző fejlesztői kiegészítést használni, és akkor nem csak a nyers adat vagy a JSON fájl tartalmát tudjuk megtekinteni, hanem a válasz (*response*) fejléc részét is, amiben látjuk, hogy a Content-Type egy JSON típusú fájl (3–3. ábra). Ez nagyon hasznos akkor, ha egy API-t akarunk összeállítani.
    b. A PHP-vel egy egyszerű asszociatív (nem egész szám indexű) tömböt hoztunk létre szerver oldalon és a kliens oldalon egy JSON objektum került átadásra. Teoretikusan, ez a munkát nagyon meg tudja könnyíteni, hiszen egy *„közös kommunikációs nyelv”* alapján könnyedén tudtunk szerver oldalról adatot küldeni a kliens oldal számára úgy, hogy semmilyen átalakításokat, konvertálásokat nem kellett végrehajtanunk. Ez is a Laravel keretrendszer egyik nagy előnye, hogy ezt a kommunikációt biztosítja számunkra.



3–3. ábra: HTTP fejléc a főoldal lekérésére Firefox-ban

Arról, hogy mi is a JSON fájl, itt lehet olvasni bővebben. Számunkra talán első körben elég most annyit tudni, hogy ez egy szöveges fájl, amelyben adatokat tudunk tárolni kulcs-érték párok segítségével. Amiért nagyon jók ezek a fájlok, az főleg az egyszerűségükből adódik, valamint, hogy mindkét oldalon (kliens és szerver részen is) egyszerűen tudjuk kezelni és feldolgozni őket. Korábban erre az XML fájltípust használták *„közös nyelvként”*, azonban az eléggé el tud bonyolódni, aminek nehézkessé válhat a kezelése, úgyhogy mindenképpen inkább a JSON használatát javaslom.

3.2.1. Egyetlen változó értékének átküldése és kiíratása

Ennyi kis bevezető után, rátérhetünk a mostani fő témánkra, az adatok átadására a nézeteknek. Maradjunk az iménti példa *„útvonalán”*, vagyis először egy szöveges változóban lévő adatot adjunk át az útvonalon keresztül a nézetnek.

```
$username = 'John';
return view('welcome', [
    'name' => $username
]);
```

3–5. kódrészlet: Egyszerű szöveges változó (adat) átadása a nézetnek

Vegyük észre, hogy továbbra is a **view()** segédfüggvényt használjuk, amiben az 1. paraméter a nézet neve lesz, míg a 2. paraméter egy asszociatív tömb (erre utal a szögletes zárójel **[]** páros), amiben a **'name'** kulcshoz hozzárendeljük a **$username** változó értékét. Ez volt az első teendő, amiben az útvonalnál elküldjük az adatot, már csak a nézetben megkapott adatot kell kiírni a látogatónak.

A **welcome** nézetben a **$username** változóban lévő adat kiíratását elvégezhetjük többféle módon is. Kezdjük a kicsit *„régimódi”* PHP-s kiíratással, aztán megnézzük a modern Blade-es kiíratást is. Egy egyszerű ** **-be tegyük a kiíratásokat a **<body>** tag-eken belül (a többi ottani tartalom törölhető).

```
<body>
    <div>Hi, <?php echo $name; ?></div>
</body>
```

3–6. kódrészlet: Régimódi PHP-s változó (adat) kiíratás a nézet fájlban

Ez egy nagyon régies megoldás, sokan nem is szeretik emiatt a PHP-t, mert a kvázi HTML fájlba ilyen módon lehet teli pakolni PHP kódokkal. A PHP kódot **<?php ?>** karakterekkel kell nyitni és lezárni. Az **echo** az pedig egy sima kiíratást jelenti, méghozzá a **$name** változó kiíratását. Vegyük észre a mindig hangsúlyozott névkonvenciót. Amit küldtünk adatot az útvonalon keresztül, az egy asszociatív tömb volt (kulcs-érték pár), ebből a nézetben a kulcsra tudunk hivatkozni, és azt megjeleníteni a látogatónak. Ennél a fenti megoldásnál egy fokkal elegánsabb, de még mindig csak egy *„sima”* PHP-s kiíratás van itt (a **<body>** és a ** ** tag-ek nélkül):

```
Hi, <?= $name; ?>
```

3–7. kódrészlet: A PHP-s echo helyettesítése, rövidítése az = jellel

Gyakran előfordult az, hogy egy-egy kiíratásra használták csak a PHP-s beágyazást a HTML kódba. Emiatt ezt az egyszerűsített formát is lehet alkalmazni (közel megegyező az iménti megoldással). Viszont mi egy Blade fájlt használunk, ami még egyszerűbbé teszi az életünket, úgyhogy a kiíratásra bőven elég csak ezt használnunk:

```
Hi, {{ $name }}
```

3–8. kódrészlet: Laravel Blade – változó kiíratása

Megjegyzés: a háttérben egy PHP-s metódushívás történik meg a két kapcsoszárójel használatával így: **htmlspecialchars($name)** Ezzel egy támadástípus ellen tudunk védekezni egyszerűen.

A kiíratásokat kipróbálhatjuk egymás után a böngészőben, mindig ugyanazt az eredményt kell kapnunk.

Itt is lehet egyszerűsített módon az útvonalon keresztül a nézetnek adatot átadni, ahogy az előző 3.1. alfejezet végén láthattuk.

```
Route::view('/' , 'welcome', ['name' => 'John']);
```

3–9. kódrészlet: Egyszerűsített útvonalon keresztüli adatküldés a nézetnek

3.2.2. Tömb adatok átküldése és kiíratása

Először elküldjük a tömb adatait egy új útvonalnak, aztán rátérünk a nézetben a megjelenítésre. A **web.php**-ban regisztráljuk az új útvonalat:

```
Route::get('/pass-array', function () {
  $tasks = [
    'Go to the store',
    'Go to the market',
    'Go to the work'
  ];
  return view('tasklist', [
    'tasks' => $tasks
  ]);
});
```

3–10. kódrészlet: Tömb átadásának útvonala

Az útvonal címe **/pass-array**, ebben definiáljunk egy **$tasks** nevű tömböt, ami három szöveges elemből áll. Utána pedig, a korábban látott módon a **view()** segédmetódus második paraméterében adjuk át ezt a tömböt, vagyis küldjük el az **tasklist** nevű nézetnek. Ez eddig rendben, azonban, ha most a böngészőben le szeretnénk kérni a http://127.0.0.1:8000/pass-array útvonalat, akkor hibát kapnánk, ugyanúgy, mint korábban, amikor nem létezett még a nézet fájl. Hozzuk ezért létre előbb a **resources / views** mappában az **tasklist.blade.php** nevű fájlt. Létrehozás után pedig következhet a kiíratás, amit most is *„hagyományosan”,* beágyazott PHP kiíratással fogunk először megtenni:

```
<ul>
  <?php foreach($tasks as $task) : ?>
    <li><?= $task; ?></li>
  <?php endforeach; ?>
</ul>
```

3–11. kódrészlet: Tömb kiíratása a nézetben beágyazott PHP kóddal

Ezzel ugye egy sima *„pöttyös”* felsorolást kapunk a böngészőben egy listát a feladatokról. Ehelyett használhatjuk a Blade sablon motort is, amellyel sokkal szebb formában tudjuk kiíratni ugyanezt a listát:

```
<ul>
  @foreach($tasks as $task)
    <li>{{ $task }}</li>
  @endforeach
</ul>
```

3–12. kódrészlet: Tömb kiíratása ciklussal a nézetben Blade sablon motor használatával

A *„háttérben”* a fordító tudja, hogy ezen egyszerűsítés mögött a fenti PHP kód van. Sokkal egyszerűbb lett így a forráskód. A weboldal frissítésével folyamatosan ellenőrizhetem, hogy még mindig jól jelenik-e meg, amit én szerettem volna. Ez a végső kód nem csak egyszerűbb, de sokkal könnyebben olvasható is, mint a beágyazott PHP kódos változat.

Az útvonal fájlban végrehajthatok még egyszerűsítéseket és használhatjuk a **with()** segédmetódust. Ehhez először a **/pass-array** útvonalat feldolgozó névtelen metódusból töröljük (kommenteljük ki) az utolsó, **return** utasítást és illesszük be helyette ezt:

```
return view('tasklist')->withTasks($tasks);
```

3–13. kódrészlet: Adatátadás a with() segédmetódussal

Ha bővítem itt még a **$tasks** tömböt, akkor az az új elem is rögtön megjelenik a weboldalon egy frissítés után: például adjuk hozzá ezt az új szöveges elemet: 'Task #4' és nézzük is meg az eredményét. A bővített listát kell látnunk. A **with()** segédmetódus pedig egy kicsit *„trükkösen”* viselkedik, hiszen a nevéhez hozzáfűztük magának a **$tasks** tömbnek a nevét (nagy kezdőbetűvel) és paraméterül pedig átadtuk neki magát a **$tasks** tömböt. Mindössze ennyit csináltunk és mégis működik, ez szintén a keretrendszer működése miatt van és azáltal vagyunk erre képesek, hogy betartjuk a névhasználati konvenciót.

De magának a névkonvenciónak nem kötelező része, hogy mit fűzünk hozzá a *„with”* szóhoz, ez csupán azt határozza meg, hogy a nézetben hogyan tudunk majd rá hivatkozni. Adjunk át neki egy másik változót is (nem tömböt) és vizsgáljuk meg, hogy úgy mi történik.

```
$foobar = 'foobar';
return view('tasklist')->withTasks($tasks)->withFoo($foobar);
```

3–14. kódrészlet: Nézet adatátadásához fűzött adatok (tömb és változó)

A nézetben pedig a lista kiíratása után tudunk hivatkozni a *„withFoo”*-val átadott *„foo”* (kis kezdőbetűs) változóra:

```
<div>{{ $foo }}</div>
```

3–15. kódrészlet: A with() segédmetódussal átadott változó értékének kiírása

Egy egyszerűsítést még végrehajthatunk, amikor több adatot is a **with()** segédmetódussal adunk át az útvonaltól a nézetnek (**web.php**-ban az előző **return** utasítást kommetezzük ki, és szúrjuk be helyette ezt):

```
return view('tasklist')->with([
  'foo' => $foobar,
  'tasks' => $tasks
]);
```

3–16. kódrészlet: Több adat átadása összevontan a with() segédmetódussal

Ennek így még mindig működnie kell. Próbáljuk azért jól megfigyelni a keretrendszer névkonvencióit, amelynek köszönhetően ilyen egyszerűen működik az adatátadás az útvonaltól a nézetnek, hiszen elsőre ez biztos szokatlannak tűnhet.

3.2.3. Felhasználótól érkező útvonalbeli adatok átadása és kiíratása

Most megismerkedünk egy újabb segédmetódussal, amit a Laravel keretrendszerben használhatunk. Ez a **request()** metódus lesz, amivel felhasználói bemeneteket, adatokat (input-okat) tudunk lekezelni (*megjegyzés:* ugyanezt a **request()** segédmetódust fogjuk alkalmazni akkor, amikor majd az űrlapok feldolgozásánál felhasználói bemeneteket vizsgáljuk). Hozzunk létre ennek is egy új útvonalat, majd az ehhez kapcsolódó nézetet is ugyanúgy **/request-test** címmel.

```
Route::get('/request-test', function () {
  return view('request-inputs', [
    'title' => request('title'),
  ]);
});
```

3–17. kódrészlet: Felhasználói bemenetből származó adatok útvonala

A felhasználó által definiált útvonal adatok szintén kulcs-érték párokként jelennek meg az URL-ben. A paraméterlista **? (kérdőjellel)** kezdődik, majd utána a kulcs érték párok **= (egyenlőségjellel)** összefűzve vannak. Ha további kulcs-érték párokat is szeretnénk így megadni, akkor **& (és) jellel** kell hozzáfűzni a paramétereket. Példa: utvonalnev**?**kulcs1**=**ertek1**&**kulcs2**=**ertek2**&**kulcs3**=**ertek3

Következhet a **request-inputs.blade.php** nézetfájl létrehozása az alábbi egyszerű tartalommal:

```
<h1>{{ $title }}</h1>
```

3–18. kódrészlet: Felhasználótól érkező adat megjelenítése

Ezzel a webcímmel tudjuk tesztelni a működését: http://127.0.0.1:8000/request-test?title=MyFirstTitle

Eredményül meg kell kapnunk a kiíratását a címnek. Tesztelhetjük tovább és az URL-t szerkesztve különböző értékeket adhatunk a **title** attribútumnak. De mi van akkor, ha mi vagyunk a támadók, és meg akarjuk hack-elni az útvonalon keresztül a webalkalmazásunkat? Próbáljuk ki ezzel a webcímmel:

http://127.0.0.1:8000/request-test?title=<script>alert("Boom!");</script>

Talán azt várnánk, hogy az oldal betöltésekor felugrana a JavaScript-es figyelmeztető ablak, ami kiírná nekünk, hogy *„Boom!”*. De nem ez történik… mert ahogy említettem, a Laravel keretrendszer megvéd minket, mint gyanútlan programozókat. Ez a kiíratás a háttérben a **htmlspecialchars()** funkciót használja, hogy ellenálljon az alkalmazás az itteni *„támadásnak”*. Újra felhívom a figyelmet arra, hogy mindenre, ami a felhasználótól, látogatótól érkezik, úgy kell tekintenünk, mint ha az ördögtől kaptuk volna. Készüljünk arra, hogy támadók fogják össztűz alá venni a webalkalmazásunkat. **Kódbeszúrásos támadásnak** nevezzük azt, amikor valamilyen programkódot (akár JavaScript, PHP vagy SQL nyelven érkezik) ártó szándékkal küld valaki az alkalmazásunknak feldolgozásra. Ha pedig mi, programozók nem ellenőrizzük ezeket a felhasználótól érkező adatokat, akkor az az alkalmazásunk, egyéb rendszereink, szervereink összeomlásához vezethet. A bővebb magyarázatért érdemes átolvasni a 15.4.1. és 15.4.2. alfejezeteket.

Változtassuk meg a kiíratást erre a **<h1>** tag-eken belül a **request-inputs.blade.php** -ban:

```
<?= $title; ?>
```

3–19. kódrészlet: Felhasználói adat kiíratása PHP nyelven

Frissítsük a weboldalunkat, és vegyük észre, hogy felugrott a figyelmeztetés (*alert*) ablak, amit fentebb már elvártunk volna. Ez egy elég nagy biztonsági rés, hiszen a kódbeszúrásos támadások így sikeresen végre tudnának hajtódni, ha ezt a PHP-s kiíratás módszert használnánk. Úgyhogy inkább térjünk vissza a Blade-es **{{ $title }}** kiíratásra, és így szerezzük vissza a keretrendszer védelmét.

Ha valamilyen rejtélyes oknál fogva, mégis szeretnénk azt, hogy a nézetben a kiíratáskor hajtódjon végre valamilyen script, akkor ezt a következő Blade formátum szerint tehetjük meg:

```
{!! $title !!}
```

3–20. kódrészlet: Kód végrehajtása a nézetben (veszélyes, főleg, ha valamilyen felhasználói „bemenetet" hajtunk végre)

Most megint, ha frissítjük az oldalt, akkor megkapjuk a figyelmeztető üzenetet a böngészőben, de ez – ismétlem – **veszélyes!**

Az alfejezetben végrehajtott programkód-módosítások ebben a GitHub commit-ben érhetők el, a felülírt, megváltoztatott kódokat megjegyzésbe tettem.

Tipp: a VSCode sokat segít nekünk a megjegyzések hozzáadásában és eltávolításában, méghozzá környezet, vagyis fájl típus szerint fogja hozzáadni, elvenni a megjegyzést az adott kódsoroktól.

Jelöljünk ki több kódsort és nyomjuk meg a **Ctrl + k** majd rögtön a **Ctrl + c** billentyűkombinációkat: ekkor megjegyzésbe teszi a kijelölt kódsorainkat. Ha meghagyjuk a több sor kijelölését, és megnyomjuk egymás után a **Ctrl + k** és a **Ctrl + u** billentyűkombinációkat, akkor elveszi a soroktól a megjegyzés jelöléseket. Ez pedig működik HTML, CSS, JavaScript, PHP és további más típusú fájlokban is.

3.3. Útvonal paraméterek (wildcards)

Azt már megtapasztaltuk, hogy az útvonalak összeállításánál nagy szabadságunk van fejlesztőként, képesek vagyunk mindenfélét definiálni és regisztrálni, de az eddig megismert útvonalak mind statikusak voltak. Sokszor azonban olyan útvonalra van szükség, ami még nincsen a legelején *„kőbe vésve"*, hanem enged egy kis szabadságot nekünk, így dinamikusan változtatható paraméterekkel tudjuk ellátni őket. Ezek lesznek a *„wildcard"* paraméterrel (több ilyen is lehet) rendelkező útvonalak.

A wildcard útvonalak akkor lehetnek nagyon hasznosak, ha például van egy blog oldalunk, és a blogbejegyzéseknek szeretnék egy adott útvonal struktúrát, sablont meghatározni, amelyek bizonyos paraméterektől leszámítva ugyanúgy működnek. Például a **/posts/** fő útvonal után szeretném a bejegyzések címeit betenni az útvonalba, de nem szeretném őket mind egyesével, egyedileg regisztrálni. Ekkor használhatok wildcard-ot (vagy *joker* útvonalat). Az útvonal, amit regisztrálok a **web.php**-ben így nézhet ki: **/posts/{post}** teljes kód szerint pedig így:

```
Route::get('/posts/{post}', function () {
  return view('post');
});
```

3–21. kódrészlet: Paraméterrel (wildcard) rendelkező útvonal

Tehát kapcsos zárójelek közé teszem a dinamikusan változó paramétert, és így minden olyan útvonalra illeszkedni fog ez, ami a kezdésében megegyezik (*„base url"* plusz a *„posts"* előtag, mivel a végén van csak a paraméter), például ezekre mind illeszkedik:

- http://localhost:8000/posts/elso-bejegyzes
- http://localhost:8000/posts/masodik-bejegyzes
- http://localhost:8000/posts/12345
- stb.

A paramétert nem kötelező a végére tenni, az útvonalban bárhol el lehet helyezni és akár több dinamikusan változó paraméter is lehet az útvonalakban, például, ha egy időjárási adatsort, magát a hőmérséklet értékeket lekérő útvonalat szeretnénk definiálni, az kinézhetne így is: /temperature/{from}/{to} Amivel azt érnénk el, hogy magát a hőmérsékletet egy kezdő- és egy végdátum közötti időszakban kilistázhatnánk.

Visszatérve a blog oldalunkra: ez a korábbi útvonal még nem működik az elvárt módon, egy kicsit finomítanunk kell rajta ahhoz, hogy az adatátadás is megtörténhessen majd a megfelelő nézetnek.

```
Route::get('/posts/{post}', function ($post) {
  return $post;
});
```

3–22. kódrészlet: Paraméteres adatátadás és visszaadás útvonalon keresztül

Bár blogunk még nincsen és így blogbejegyzéseink sem, de kreatívok vagyunk és az adatátadás ilyen egyszerűen is le tudjuk tesztelni, hogy működnek-e most már a fenti felsorolásban lévő linkek.

Az adatátadás az útvonalon keresztül így már működik: ugyanazt írja ki a betöltött oldal, amit az útvonalban a paraméter helyére megadunk neki.

A sima érték átadáson és visszaadáson túl, helyezzük vissza a return utasításba a nézetet és adjuk át neki az adatot.

```
Route::get('/posts/{post}', function ($post) {
  return view('post', [
    'post' => $post
  ]);
});
```

3–23. kódrészlet: Adat átadása paraméteres útvonalon keresztül a nézetnek

Majd hozzuk létre a kapcsolódó nézet fájlt is a megfelelő helyen (resources / views / post.blade.php) az alábbi tartalommal:

```
<p>{{ $post }}</p>
```

3–24. kódrészlet: Post nézetfájl tartalma

Az adatátadás működik, a nézet megkapja az adatot és bekezdésben (paragraph) megjeleníti a számunkra, bármilyen karaktersorozatot is írunk be a címben (útvonalban) a paraméter helyére.

Közelítsünk most egy valós életbeli megoldás felé (vagy legalábbis szimuláljuk), amikor például adatbázisból nyerhetjük ki a bejegyzéseket (**post**-okat) és az adott azonosítójút szeretném átadni a nézetnek, majd megmutatni azt a felhasználónak:

```
Route::get('/posts/{post}', function ($post) {
  $posts = [
    'first-post' => 'Hello, this is my first blog post!',
    'second-post' => 'Now I am getting the hang of this blogging thing'
  ];
  return view('post', [
    'post' => $posts[$post]
  ]);
});
```

3–25. kódrészlet: Adathalmaz létrehozása az útvonalban és paraméter kérésnek megfelelő átadása a nézetnek

Az útvonal regisztrációjában egy tömb segítségével szimulálom az adathalmazt. A böngészőbe beírt link (útvonal) ezután már nem fog bármilyen paraméterrel működni, csak a tömbelemek kulcsaival (bal oldali azonosítókkal). Ha megfelelő azonosítót adott meg a felhasználó, akkor pedig vissza fogja kapni a hozzá tartozó blogbejegyzés tartalmát. Jelen állapot szerint tehát csak két paraméteres link fog működni:

1. http://127.0.0.1/posts/first-post
2. http://127.0.0.1/posts/second-post

Ha mást írok be az utolsó helyre, akkor nem fog működni, hiszen a **$posts** tömbnek nincs más indexű eleme. Ha mégis el szeretnénk kerülni, hogy a látogató valamilyen *„rémisztő"* hibaoldalt kapjon meg, akkor azt is könnyen kijavíthatjuk. Módosítsuk az útvonal **return** utasítását eszerint:

```
return view('post', [
  'post' => $posts[$post] ?? 'Nothing here yet.'
]);
```

3–26. kódrészlet: Nem létező blogbejegyzés lekérésre is értelmezhető eredményt adunk vissza

Ezzel elértük, hogy most már például a http://127.0.0.1/posts/123 linkre is értelmezhető eredményt kap vissza a felhasználó, ha nincs is még ilyen *„blogbejegyzésünk az adatbázisban"*.

Megjegyzés: a **??** operátor a PHP-ben azt jelenti, hogy ha létezik az előtte lévő elem, akkor rendben van és azt használja, ha nem létezik (vagy null), akkor a **??** utáni értéket adja vissza.

De megcsinálhatjuk azt is, hogy felhasználjuk a Laravel keretrendszer adta lehetőséget és az általa nyújtott 404-es hiba állapotkódú oldalt adjuk vissza a látogatónak. Bővítsük az útvonalunkat a **return** utasítás elé szúrjuk be a következő feltételvizsgálatot:

```
if ( ! array_key_exists($post, $posts)) {
  abort(404);
}
```

3–27. kódrészlet: Kulcskeresés (post) az adathalmazban (posts)

Így, ha nem találjuk meg az adathalmazban a látogató által beírt link szerinti paramétert (kulcsot), akkor a 404-es oldalt mutatjuk meg neki. Töltsük újra a böngészőben a http://127.0.0.1/posts/123 linket, hogy megkapjuk a 404-es oldalt.

3.3.1. Gyakran előforduló hibaoldalak (nézetek) felüldefiniálása

Vannak témakörök, amelyek a webes programozás során gyakran előkerülnek, egy ilyen a **többnyelvűsítés** (lokalizáció) is, ezzel még a későbbiekben foglalkozunk majd (15.2. alfejezet), most még csak *felüldefiniáljuk* az alapértelmezetten kapott nézeteinket. A Laravel keretrendszer ebben is segít minket, például az útvonalaknál a következőket tudjuk megtenni: a gyakran előforduló HTTP hibakódokat tartalmazó oldalakat felülírhatjuk (átszerkesztjük a kinézetét, vagy átszerkesztjük az alapértelmezett szövegeket). Ez utóbbit fogjuk tenni az imént megkapott *„404 | NOT FOUND"* nézetnél. Adjuk ki a következő utasítást a terminal-ban:

```
php artisan vendor:publish --tag=laravel-errors
```

Megjegyzés: a **vendor** mappában lévő fájlokat soha nincs értelme módosítani, mivel azok egy `composer update` parancs futtatása után (főleg az érintett csomag verzióváltása esetén) felül fognak íródni az érintett fájlok. Ezzel szemben, ha az érintett fájlokat publikáljuk a projektünk magjába (itt éppen a **resources** mappába), akkor az mindig felül fogja definiálni a **vendor** mappa mélyén elhelyezkedő szerkeszteni kívánt fájlokat.

Ennek hatására a **resources / views** mappában létrejön egy **errors** nevű mappa, amibe bekerülnek 401, 402, 403, 404, 500 stb. hibakódokat tartalmazó nézetek, amelyeket szabadon módosíthatunk, és ha a felhasználó ilyen hibakódokat kapna, akkor a mi általunk felüldefiniált nézetek jelennek meg a számukra. (További információ erről itt: https://laravel.com/docs/10.x/errors#http-exceptions)

Most, ha a **resources / views / errors / 404.blade.php** nézetben módosítjuk a **message**-t tartalmazó sorban a *„Not Found"* üzenetet bármi másra, például ilyenre:

```
@section('message', __('The blog post does not exist'))
```

3–28. kódrészlet: 404-es hibaüzenet felüldefiniálása

Akkor így már meg is kapjuk eredményül a felüldefiniált hibaoldalt a böngészőben:

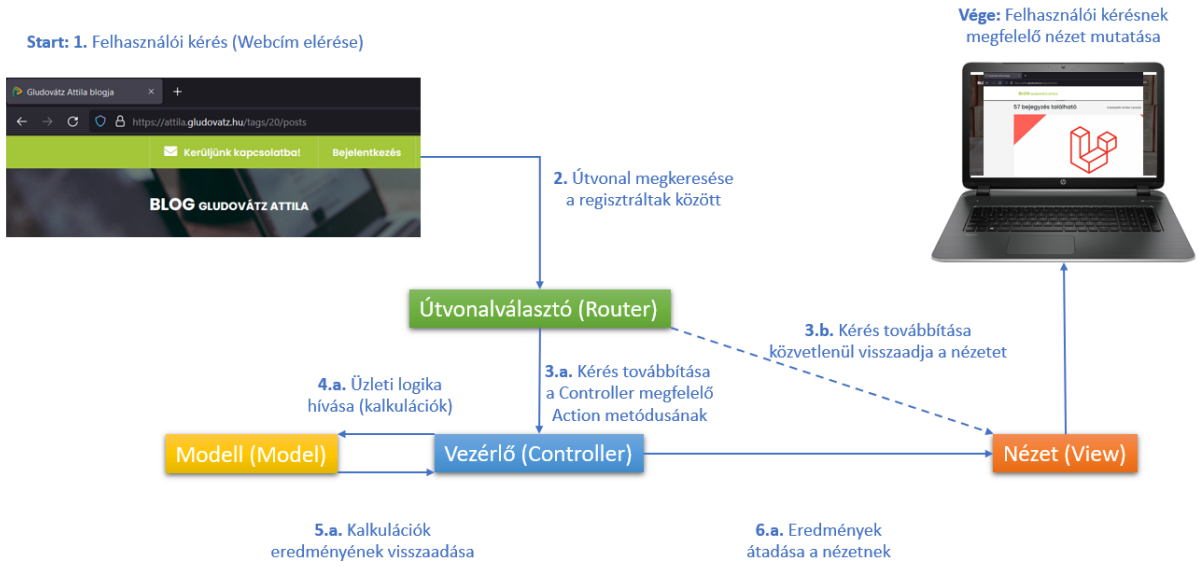
3–4. ábra: Új 404-es hibakódot és részleteit tartalmazó nézet oldal

Az oldal kinézetét a **layout.blade.php** fájlban tudjuk módosítani, de ezt majd a következő, nézetekről szóló fejezetben részletesen fogom tárgyalni.

Az alfejezetben végrehajtott programkód-módosítások ebben a GitHub commit-ben érhetők el.

3.4. Kérések kiszolgálása az MVC architektúrában a hosszabb ágon

Eddig egy egyszerűsített útvonalon haladtunk az MVC tervezési minta elemei mentén (3–5. ábra, a 3.b. ág felé haladtunk a kiszolgálásban): a felhasználói kérés után a regisztrált útvonalak felől a nézetek felé vettük az irányt (szaggatott vonalú nyíl) és vissza is adtuk az eredményt a felhasználó böngészőjének.



3–5. ábra: Felhasználói kérés kiszolgálása hosszabb ágon a vezérlővel és a modellel

Ez azonban a ritkább szituáció, és sokkal jellemzőbb, ha az útvonalaktól a vezérlők felé haladunk, mivel a vezérlők segítségével tudunk az üzleti logikáért és az adatbázisban lévő adatokért felelős modellekkel kapcsolatot fenntartani, meghívni őket. A vezérlők összeszedik tehát a szükséges adatot és információt, majd azt tovább képesek adni a nézeteknek, ami már megjelenhet a felhasználók böngészőjében (3–5. ábra: 3.a., 4.a., 5.a., 6.a. útvonalak). Most még az adatbázissal való kapcsolatról és az adatlekérésről nem fog szó esni, tehát nem teljes az a ág bejárása, de a vezérlőt és a modellt már hozzávesszük eddig áttekintett tudásanyaghoz, de még csak az útvonalak elérése (felhasználói kérések kiszolgálása) kapcsán érintjük őket.

3.4.1. Vezérlő (Controller) bevétele a kiszolgálási ágba

Először hozzuk létre magát a vezérlőt, hogy tudjunk majd rá hivatkozni az útvonal fájlból. A 2.3.2. alfejezetben megismertük a Laravel keretrendszer könyvtár struktúráját és emiatt tudhatjuk, hogy a vezérlők az **app / Http / Controllers** mappába fognak bekerülni. Megtehetnénk, hogy manuálisan létrehozzuk ide az új **PostController.php** névre hallgató fájlunkat, azonban az nem lenne túlságosan kényelmes, mivel egy üres fájl jönne létre. Mi viszont azt szeretnénk, hogy ennél azért többet kapjunk. Adjuk ki a következő parancsot a terminal-ban:

```
php artisan make:controller PostController
```

Figyeljünk a *névkonvencióra*, mert nem véletlenül lett ez a neve. Használjuk az első részt nagy kezdőbetűvel, valamint angolul (esetleg többes számban, de ez nem feltétlenül kötelező), utána pedig következhet mindig a *„Controller”* szó.

Ha a parancsról többet szeretnénk tudni, akkor adjuk ki a következő utasítást a terminal-ban, így több információhoz juthatunk a vezérlő készítésének lehetőségeiről:

```
php artisan -help make:controller
```

Maga a vezérlő létrehozási parancs meglehetősen olvasható és így érthető is. Az utasítás kiadásának hatására létre fog jönni a vezérlő fájlunk **PostController.php** fájlnévvel a megfelelő helyen és benne a **PostController** osztály a szükséges névtér definícióval és importálással együtt. Ehhez az osztályhoz adjunk hozzá egy olyan metódust, ami a blogbejegyzéseink kiírását fogja támogatni.

```
public function show($post)
{
  return $post;
}
```

3–29. kódrészlet: PostController show metódusa (action)

Ez a show metódus a **$post**-ot kapja majd meg az útvonaltól és kezdetben egyszerűen csak meg fogja jeleníteni a számunkra.

Ezután definiáljuk és regisztráljuk az útvonalat, ami elvisz majd a vezérlő felé. Attól függően, hogy hány olyan útvonalat szeretnénk majd regisztrálni, ami egy konkrét vezérlő felé viszi a kérés kiszolgálását érdemes importálni a fájlt a **web.php** fájl elején, avagy sem. Mivel most még csak egy ilyen útvonalunk van, ami a **PostController** használatát igényli, ezért hivatkozhatunk rá az útvonalon belül *„hosszú”, „teljes”* névvel.

Fontos: *adódhat a kérdés, hogy a web.php-n belül hol definiáljuk az új útvonalakat…?*

Mivel az útvonalak betöltése lineárisan történik a fájl elejéről kezdve, ezért, ha a Laravel talál egy olyan útvonalat, ami már passzol a felhasználói kérések, akkor rögtön aszerint történik meg a kiszolgálás. Ha viszont ugyanezt az illeszkedő útvonalat újra és újra, akár többször is regisztráljuk, akkor a legutolsó előfordulásnak megfelelő műveletsort fogja végrehajtani a rendszer. Például, ha egymás után háromszor regisztrálom be a kezdőoldal (‘/’) útvonalat, majd mindegyik lekezelésénél csak annyit írok ki, hogy 1., 2. és 3., akkor a rendszer a „3.”-at fogja kiírni a böngészőben a kezdőoldalon.

Ez a terminológia egy kicsit hasonlít a CSS stílusszabályok definiálására és érvényességére, mivel az ottani rangsorolásnál van az, hogy az a stílusszabály lesz érvényes (a selector illeszkedésen túl), amelyiket később (hátrébb) definiáltuk.

A blogbejegyzést megmutató útvonalat szintén **get**-es és szintén a **/posts/{post}** útvonalon lenne érdemes, ugyanúgy, mint ahogy az előző, 3.3. alfejezetben már megvalósításra került az útvonal regisztrálásánál. Két egyforma, ugyanolyan útvonalat pedig nem érdemes egymás után regisztrálni, hiszen, ahogy a *„Tipp”* részben is megfogalmazásra került, akkor a Laravel az első passzoló regisztrált útvonal szerint küldené tovább a kérést kiszolgálásra, a másodikat már nem venné figyelembe. Továbbá magunkat sem érdemes összezavarni azzal, hogy ugyanolyan útvonalakat regisztrálunk egymás után, mert ha később újra elővesszük majd a kódunkat, akkor nem tudhatjuk, hogy mi is volt a szándékunk ezzel.

Emiatt azt javaslom most, hogy az érintett útvonalat kommenteljük ki teljesen, és hozzunk most létre egy olyat, ami az új vezérlőnk irányába küldi tovább a felhasználói kérést.

```
Route::get('/posts/{post}', ['App\Http\Controllers\PostController',
'show']);
```

3–30. kódrészlet: PostController (hosszú névvel) show metódusa felé történik a kérés továbbítása

Így (3–30. kódrészlet) tudunk tehát hosszú névvel hivatkozni a **PostController**-re a **web.php** fájlban, de ha tudjuk, hogy a **PostController**-nek több metódusa felé is szeretnénk majd küldeni a felhasználói kérést kiszolgálásra, akkor érdemes a **web.php** fájl elején importálni magát a **PostController** osztályt és akkor az útvonalnál már rövidebben is hivatkozhatunk rá.

```
use App\Http\Controllers\PostController;
// a web.php fájl többi része
Route::get('/posts/{post}', [PostController::class, 'show']);
```

3–31. kódrészlet: A PostController importálása és hivatkozás rá az útvonalnál röviden

Teszteljük a böngészőnkben a következő weblinkkel: http://127.0.0.1/posts/first-post

Ez így működik és visszakapjuk az útvonalban átadott tartalmat az oldal megjelenítésekor.

A *leggyakoribb hibák* azok szoktak lenni, hogy rosszul definiáljuk az útvonalat (csak rövid névvel hivatkozunk rá, de az útvonal fájl elején nem importáljuk), esetleg még nem létezik az útvonal regisztrálásakor a Controller vagy a hivatkozott metódusa.

Végül a **PostController show()** metódusán belül ne csak visszaadjuk a kapott adatot, hanem küldjük tovább a már létező nézetnek egy újfajta módon:

```
public function show($post)
{
  return view('post', compact('post'));
}
```

3–32. kódrészlet: A compact() segédmetódus használat a nézetnek való adatküldésnél

A **compact()** segédmetódussal tudjuk leegyszerűsíteni (kevesebb kódírással) azt, amikor egy asszociatív tömböt szeretnénk átadni a nézetnek. Ez a **compact('post')** utasítás megfelel annak, mint hogy ha a **['post' => $post]** kódsorokat írtuk volna oda a **view()** metódus második paraméterébe. Főleg akkor hasznos, ha több ilyen azonos nevű kulcs-érték párt szeretnénk átadni a nézetnek, mert akkor csak a **compact()** metódus paraméterei között fel kell sorolnunk szövegesen, hogy milyen adatokat szeretnénk átadni a nézetnek (a következő alfejezetben mutatok is erre egy konkrét példát).

3.4.2. Modell (Model) bevétele a kiszolgálási ágba

Most következhet a modell fájl bevétele az ágba, ahol kiszolgáljuk a felhasználói kérést. A modell fájl segítségével számoljuk ki például a kapott „blogbejegyzés" hosszát (karakterszámát). *Fontos, hogy tudatosítsuk magunkban azt, hogy az üzleti logika, vagyis a különböző funkcionalitások megvalósításának helye a modell fájlokban van.* Bármennyire is *„csábító"* lenne most itt még a tanulmányaink kezdetén az, hogy akár az útvonal regisztrációjánál, akár a vezérlő metódusainál elvégezzük a számításokat,

műveleteket, az *nem lenne célravezető az MVC tervezési minta követése szempontjából.* Ha csak a saját, mostani példánknál maradunk, akkor egy blogbejegyzés hosszának kiszámítására számos helyen szükségünk lehet a munkánk során és ha ezt a funkcionalitást az útvonal regisztrációnál vagy a vezérlő metódusában implementálnánk, akkor sehol máshol nem tudnánk újra felhasználni ezt a kódot, csakis ide-oda másolgatással, ami aztán egy jó nagy hibaforrás lenne a webalkalmazásunkban. Minden funkcionalitást csak egyetlen helyen valósítsunk meg (valamelyik modell fájlban), de ott a legtökéletesebben, hogy aztán majd máshol tudjuk alkalmazni ezeket a funkcionalitásokat.

Hozzunk létre a terminal-ban egy új modell fájlt:

```
php artisan make:model Post
```

Az **app / Models** mappában létre is jön az ennek megfelelő **Post.php** és benne a Post osztály.

Adjunk hozzá egy új statikus metódust, mert ezt az osztályt most még csak a kalkulációra használjuk adatbázis elérés és példányosítás szükségessége nélkül.

```
public static function getLength($post)
{
  return strlen($post);
}
```

3–33. kódrészlet: A „blogbejegyzés" hosszának kiszámítása

A PHP-s **strlen()** metódussal csak megszámoljuk a paraméterül kapott szöveg hosszát és visszaadjuk a benne lévő karakterek számát.

Először a PostController osztály előtt importáljuk az App\Models\Post osztályt:

```
use App\Models\Post;
```

3–34. kódrészlet: Post Model osztály importálása a PostController.php fájlban

Ezután a **PostController** osztály **show()** metódusának magját kommentezzük ki, és legyen benne ez a két sor:

```
$length = Post::getLength($post);
return view('post', compact('post', 'length'));
```

3–35. kódrészlet: PostController osztály show() metódusának végső tartalma

Így két változót értékét is átküldjük a **post.blade.php** nézet fájlunknak, de jelenleg még csak a **$post** változónak a kiíratása történik meg benne, úgyhogy bővítsük ki azért, hogy látszódjanak a karakterszámok is:

```
<p>length: {{ $length }} character(s)</p>
```

3–36. kódrészlet: A post.blade.php bővítése a bejegyzés karakterszámának kiírásával

Teszteljük a böngészőnkben a következő weblinkkel: http://127.0.0.1/posts/first-post

first-post

length: 10 character(s)

3–6. ábra: A blogbejegyzés megjelenítése és a karakterszámának kiírása

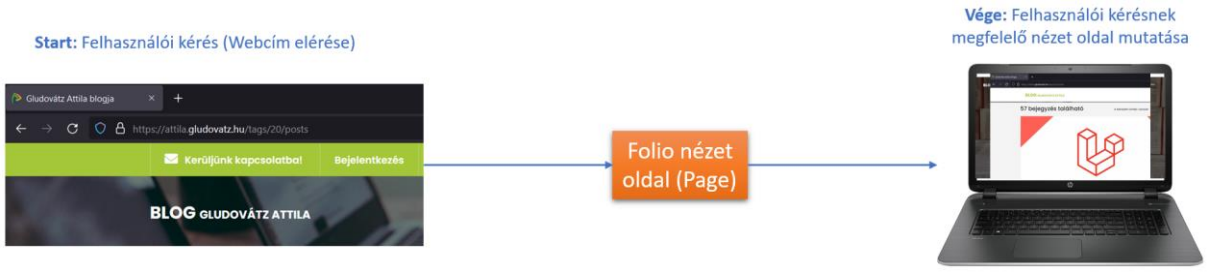
Az alfejezetben végrehajtott programkód-módosítások ebben a GitHub commit-ben érhetők el.

Az útvonalak kezelésére, újra szervezésére (*refactoring*) még többször vissza fogunk térni a könyvben, de ennyi információ a kezdéshez elegendő volt. A fejezet utolsó részében a tesztelés témakörével kezdünk el ismerkedni és tesztel fogjuk az eddigi útvonalainkat.

3.5. Új eszköz az útvonalkezelési technikákhoz: Laravel Folio

Ez egy új eszköz, amelynek használatával az útvonalválasztást szeretnék még egyszerűbbé, átláthatóbbá tenni a Laravel fejlesztői, mindezt egy hatékony oldal-alapú útvonalválasztással érhetjük el. Az első stabil verziója a Folio eszköznek 2023. augusztusában jelent meg, amikorra már a Laravel 10-es verziója jónéhány alverzió váltáson átesett, emiatt bátran nevezhetjük ezt egy új eszköznek a Laravel keretrendszerben. (*Megjegyzés:* ez nem azt jelenti, hogy kiváltja a Folio az eddig megismert, *„tradicionális"* útvonaltervezési technikákat, hanem csak azok mellett ezt is lehet majd használni, mert ez egy újfajta gondolkodásmód szerint építi fel az útvonalak kezelését.)

A motivációt talán az adhatta a Folio kitalálására és fejlesztésére, hogy bizonyos esetekben (például személyes weboldalaknál, statikus oldalaknál, blogoknál), szükség van arra, hogy a lehető legegyszerűbben szeretnék elérhetővé tenni a fejlesztők a látogatók által lekért útvonalak eredményeit. A Laravel Folio megjelenése előtt a legegyszerűbb felhasználói kérés-kiszolgálás (1) egy útvonal regisztrációjával indult, majd utána (2) az útvonalon belül a nézetet adtuk vissza a felhasználó böngészőjének. Ez eddig két lépés volt (lásd a korábbi 3–5. ábra 3.b. ágát), de a Laravel Folio-val megoldható mindez egyetlen lépéssel! Ez a technika és a mögöttes tartalom közel állhat a Next.js, Nuxt.js vagy a Svelte fejlesztőinek gondolkodásmódjához.



3–7. ábra: Felhasználói kérés kiszolgálása a Folio nézet oldalával egyszerűen

Telepítsük a Folio-t a composer-rel:

```
composer require laravel/folio
```

Most már hozzáférünk két új artisan parancshoz a folio névtérben, ha lekérjük a php artisan utasítást, meg is nézhetjük őket:

```
folio
 folio:install              Install all of the Folio resources
 folio:list                 List all registered routes
```

3–8. ábra: Laravel Folio artisan parancsai

Utána telepíthetjük hozzá a saját Service Provider fájlját (**app / Providers / FolioServiceProvider.php**), amivel be tudjuk állítani a későbbiekben, hogy hol legyenek az oldalaink (**pages** mappában).

```
php artisan folio:install
```

Illetve az imént említett **FolioServiceProvider** osztály bekerül a **config / app.php** fájlban a regisztrált szolgáltatások közé (**$providers** tömb), amely elemek mindig betöltődnek az alkalmazás működésekor.

11

Új módszer a szolgáltatások, támogató osztályok regisztrálására: újdonság a Service Provider osztályok regisztrációjánál.

A Laravel Folio telepítésekor már észre is vehetjük, hogy itt nem a **config / app.php**-be kerül be a FolioServiceProvider osztály regisztrálása, hanem a **bootstrap / providers.php** fájl visszatérési tömbje fogja tartalmazni azokat a támogatási szolgáltatásokat, amelyeket a webalkalmazásunk fejlesztése során használni szeretnénk majd.

Általánosságban elmondható, hogy a Laravel 11-ben a rendszer saját (vagy harmadik féltől származó) Service Provider osztályait már sosem kell *„manuálisan"* regisztrálni, mivel ezt mindig elvégzi helyettünk a keretrendszer vagy a csomag telepítése. Ha azonban mégis egy saját, egyedi Service Provider-t hoznánk létre, akkor azt mindig a **bootstrap / providers.php** fájl visszatérési tömbjében kell elhelyezni, ugyanúgy, ahogy a FolioServiceProvider is bekerült most oda a csomag telepítése által.

3–2. újdonság: Service Provider osztályok eltérő kezelése

A **resources / views** mappában hozzunk létre egy **pages** nevű mappát, abban pedig egy **projects.blade.php** fájlt. Mivel a Folio-t már telepítettük, így az alapértelmezetten a **resources / views / pages** könyvtárban keresi az oldalakat tartalmazó nézet fájlokat, ezért, ha elindítjuk az alkalmazásunk kiszolgálását, akkor a **/projects** útvonalon be is kell jönnie a projektjeinket tartalmazó nézet fájlnak. De előbb azért töltsük fel tartalommal a **projects.blade.php** fájlt:

```
<div>
    <h1>My Projects</h1>
</div>
```

3–37. kódrészlet: projects.blade.php Folio oldal tartalma

Az eredménye itt látható:

Ha a másik Folio-s artisan parancsot is kipróbáljuk, akkor az ki is listázza nekünk ezt az oldalt, ugyanúgy, mint korábban a route:list parancs tette:

```
php artisan folio:list
```

Eredménye a terminal-ban:

```
GET       /projects ..................................................... projects.blade.php

                                                                          Showing [1] routes
```

3–9. ábra: Folio útvonalak és oldalak listázása a terminal-ban

A php artisan route:list paranccsal lekért útvonalak közé bekerült a Folio-s útvonal is egy *„helyőrző"* wildcard-dal (a helyőrző helyére kerülhetnek az útvonal nevei, mint jelen példánkban a **projects**):

```
GET|HEAD        {fallbackPlaceholder} ..................................... laravel-folio
```

3–10. ábra: Folio-s útvonal a többi, már meglévő útvonal között

A Folio-val a paramétert (wildcard-ot) tartalmazó útvonalak is működnek. Viszont ezek leginkább majd az erőforrásokat lekérdező, létrehozó, módosító, törlő utasítások esetén lesz hasznosak (lásd 8.3. fejezet).

Ebben az alfejezetben végrehajtott programkód módosítások ebben a GitHub commit-ben találhatók meg. Emellett a Laravel 11-ben megjelenő Service Provider regisztrációjának újdonsága miatt a Laravel 11-es projektben is végrehajtottam és kipróbáltam a Folio-t, amely ugyanúgy tud működni, mint a Laravel 10 esetében. Az ehhez tartozó programkód módosítások ebben a GitHub commit-ben érhetők el.

3.6. Laravel, mint backend API

A Laravel-t, mint szerver oldali PHP programnyelv alapú keretrendszert, sokszor használják arra, hogy a háttérben működjön, akár lényegi felhasználói felület (nézetek) megvalósítása nélkül. Ekkor a backend részen működő keretrendszer csak egyfajta kiszolgálóként működik. Így a backend oldali keretrendszer csak egy olyan szereplő a felhasználói kérések kiszolgálásának folyamatában, amely fogadja a felhasználóktól a kéréseket, útvonalak megszólításán keresztül, utána ellenőrzi, hogy megfelelőek-e a kérések, és ha igen, akkor például összegyűjti a szükséges adatokat az adatbázisból, esetleg kalkulációkat hajt végre rajtuk azért, hogy helyesen ki tudja szolgálni a kéréseket, és azokra megfelelő választ adjon.

A backend oldalon történő API alapú kiszolgálási megközelítések azért is rettentően népszerűek manapság, mivel rendkívül sok kliens oldali osztálykönyvtár létezik (például React, Vue.js stb.), amelyek működését már elsajátította az adott fejlesztő. Így a kliens oldal megvalósítása után, ha szüksége van adatokra vagy egyéb háttérműveletekre, amelyeket egy backend oldali keretrendszer könnyebben, hatékonyabban tud elvégezni, akkor API útvonalakon keresztül tudja elérni annak szolgáltatásait.

A világhálón számos, ingyenesen is elérhető API létezik, mint például az IMDB[4] filmes adatbázis oldala, amely a webes felületen kívül elérhető API-on (https://developer.imdb.com/) keresztül is, vagy a Google

[4] Internet Movie Database

Maps szolgáltatása (https://developers.google.com/maps), aminek köszönhetően egy paraméter sorozat kulcs-érték párosainak megadása után akár saját weboldalunkba is beágyazhatunk interaktív térképeket. Esetleg, ha az alkalmazásunkban időjárás adatokat (https://openweathermap.org/api) vagy tőzsdei adatokat (https://exchangeratesapi.io/) szeretnénk megjeleníteni egy kis felületen, akkor a háttérben ott is egy szolgáltató API felületét kell megszólítani, és a visszakapott adatokat már úgy leszünk képesek megjeleníteni a saját alkalmazásunkban, ahogy azt mi szeretnénk. API tehát rengeteg létezik (még talán annak a weboldalnak is van ilyen elérési felülete, amelyet az Olvasó leggyakrabban böngész végig webes kéréseken keresztül). A mi alkalmazásunk, mivel a Laravel keretrendszerre épül, kiválóan alkalmas arra, hogy egyfajta API felületként szolgáljon a közvetlen felhasználói kéréseknek, vagy a backend oldal elé épített frontend (kliens) oldali keretrendszereknek.

3.6.1. API útvonal végpontok létrehozása

Még mielőtt létrehoznánk az API felület számára a regisztrált útvonalakat, tekintsük meg az **app / Providers / RouteServiceProvider.php** fájlt és annak osztályát! Alapértelmezetten két metódust tartalmaz, amelyek közül a **boot()** metódus a fontosabb számunkra, a másik metódus, **configureRateLimiting()** csak ezt a **boot()** metódust egészíti ki, segíti azáltal, hogy limitálja az egy percen belül, azonos felhasználótól vagy IP címről beérkező kérések számát. Visszatérve a **boot()** metódus magjára, itt vannak definiálva azok az útvonal *„csoportok”,* amelyeket alapból létrehozott számunkra a Laravel.

```
$this->routes(function () {
  Route::middleware('api')
    ->prefix('api')
    ->group(base_path('routes/api.php'));

  Route::middleware('web')
    ->group(base_path('routes/web.php'));
});
```

3–38. kódrészlet: RouteServiceProvider boot() metódusában definiált útvonalcsoportok jellemzőinek beállítása

Kétféle módon érkezhet eszerint kérés az alkalmazásunkhoz:

1. API felületen keresztül:
    a. Ekkor minden kérésnek egy olyan köztes rétegen (*„middleware”*) kell keresztül mennie, amely a kérés megfelelőségét vizsgálja, és csak akkor engedi át az alkalmazásunknak, ha érvényesnek találta. A middleware-ekről részletesebben a 10.1. alfejezetben lesz szó, itt most erről még nem is kell többet tudnunk.
    b. Az API-os útvonalaknak lesz egy előtagja, ami az útvonalban jelenik meg: **api**, ha tehát létezik egy útvonalunk, amelyet a **web.php**-ban már létrehoztunk, például a **/request-test** útvonalat, akkor azt a webböngészőben helyi kiszolgálás esetén a http://localhost:8000/request-test weblinken érhetnénk el. Az API alkalmazása esetében, ugyanezt az útvonal végpontot így tudnánk elérni: http://localhost:8000/api/request-test A prefix következtében tehát bekerül az útvonal elé a **/api** rész.

c. Az útvonal csoport utolsó metódushívása tartalmazza azt, hogy az API-os útvonalakat a **routes / api.php** fájlban kell regisztrálni alapértelmezetten.

2. Utána helyezkedik el a web-ről érkezők kérések csoportja. Ezt már többször használtuk korábban, de itt most láthatjuk, hogy egy webes köztes rétegen kell keresztül menniük a böngészőkből érkező kéréseknek, illetve, ahogy már azt tapasztaltuk, a **routes / web.php**-ban kell alapértelmezetten regisztrálni ezeket az útvonalakat.

Az API számára létrejövő útvonalakat, tehát az **api.php**-ban regisztráljuk be. Hozzunk is létre ide egy útvonalat a működés kipróbálásához!

```
Route::get('/hello-api', function () {
  return "Hello API !";
});
```

3–39. kódrészlet: Első saját API útvonal végpontunk

Megjegyzés: van már benne egy automatikusan létrejövő útvonal, azonban ennek elérése felhasználói hitelesítést igényelne, amelyet a mi alkalmazásunk még nem tartalmaz. A felhasználói hitelesítésről a 1. fejezetben lesz szó részletesen.

RouteServiceProvider, api.php: ezek alapértelmezetten nem léteznek a Laravel 11-ben.

A RouteServiceProvider fájl *„kiváltását"* a **bootstrap / app.php** fájlban találjuk. Itt az alkalmazásunknak útvonalait és egyéb más dolgait (köztes rétegeket – middleware, kivételeket – exceptions stb.) tudjuk regisztrálni.

A **withRouting()** metódus fog módosulni a *„web"*-es kódsor alapján. Adjuk ki a következő utasítást:

```
php artisan install:api
```

Ez módosítani fogja az imént említett **withRouting()** metódus magját, illetve létre fog jönni a **routes** mappában az **api.php** fájl is. Emellett, mivel a valós életben a legtöbb API híváshoz szükség van felhasználói hitelesítésre, ezért települni fog a Laravel Sanctum csomag (és beállítási fájl), amely ezt támogatja. Létrejön ezeken kívül egy személyes hozzáféréseket tároló tábla szerkezete is, amelyet a telepítés végén migrálhatunk (létre is hozhatunk) az adatbázisban.

A Laravel 11-hez tartozó API telepítési módosítások ebben a GitHub commit-ben találhatók meg.

Megjegyzés: a terminal-ban még kapunk egy figyelmeztetést is, hogy ne felejtsük el hozzáadni a **HasApiTokens** funkcionalitást (trait-et) a **User** Model osztályhoz, de egyelőre nekünk erre nincsen most még szükségünk.

3–3. újdonság: API eltérő telepítése, használata

3.6.2. API tesztelése Postman alkalmazással

Tipp: számos ingyenes eszköz elérhető az API kérések indításához és a válaszok megtekintéséhez. Akár a Chrome böngészőben is lehet telepíteni a *„Talend API Tester"* kiterjesztést, amely nagyon hasonlóan működik még ezen a szinten, mint ahogy a Postman-t fogjuk használni. A könyv további részében viszont a Postman alkalmazáson keresztül fogom bemutatni az API tesztelésekkel kapcsolatos példákat.

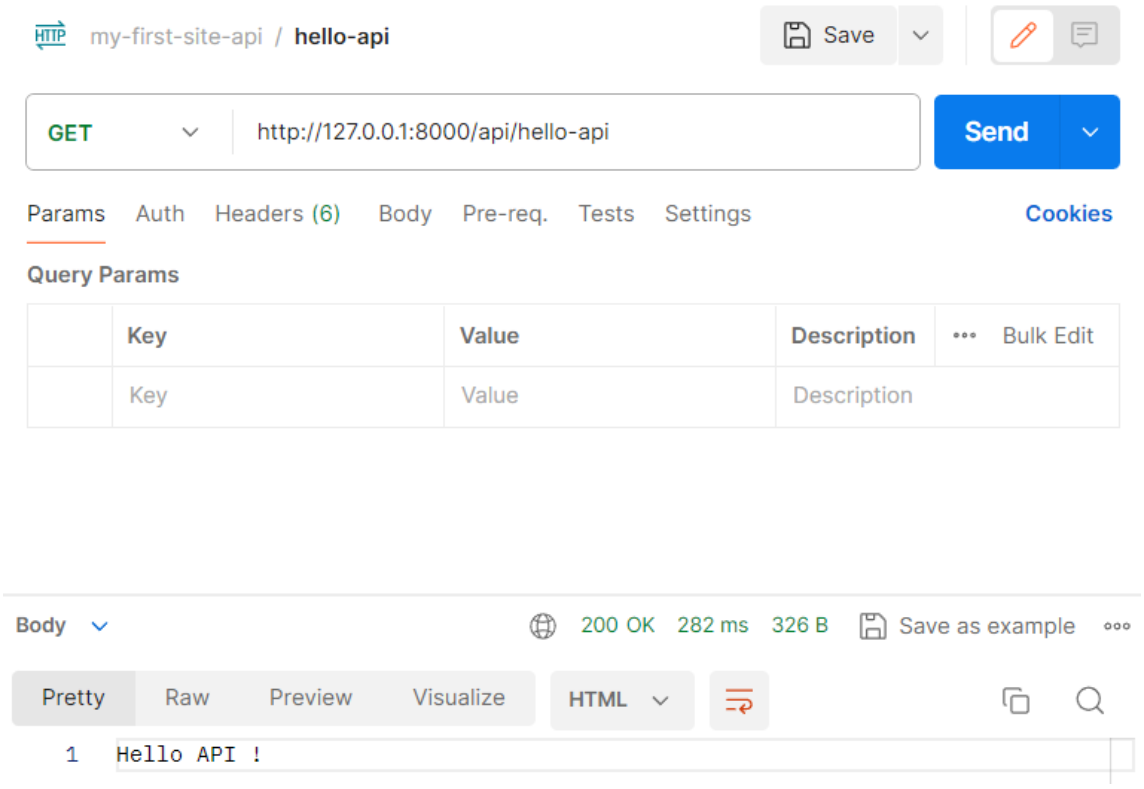
Telepítsük fel a Postman alkalmazást gépünkre! Innen letölthető: https://www.postman.com/downloads/ Telepítés után az első használatkor regisztrálnunk is kell egy felhasználót, majd bejelentkezni, de erre egy Google fiók is tökéletesen megfelelő.

Hozzunk létre egy új **workspace**-t, amit nevezzünk el **my-first-site-api** néven! Miután létrejött, a *„workspace"* neve mellett jobb szélen megjelenik három pont, ha fölé visszük az egeret. Kattintsunk rá, és válasszuk ki az *„Add request"* menüpontot! Ekkor a jobb oldali területen megjelenik az a felület, ahol lehet definiálni a Laravel webes alkalmazásunkhoz intézett API kérést.

Legelső példánkban a köszöntő útvonalunkat szeretnénk megszólítani és egy köszönést várunk el válaszként. Az útvonal a **GET** HTTP metódust használatával érhető el ezen a linken:

http://127.0.0.1:8000/api/hello-api

További beállításokra nincs is most szükség, csak a *„Send"* gombot kell megnyomni és alul meg is kapjuk a választ (lásd a következő ábrát).



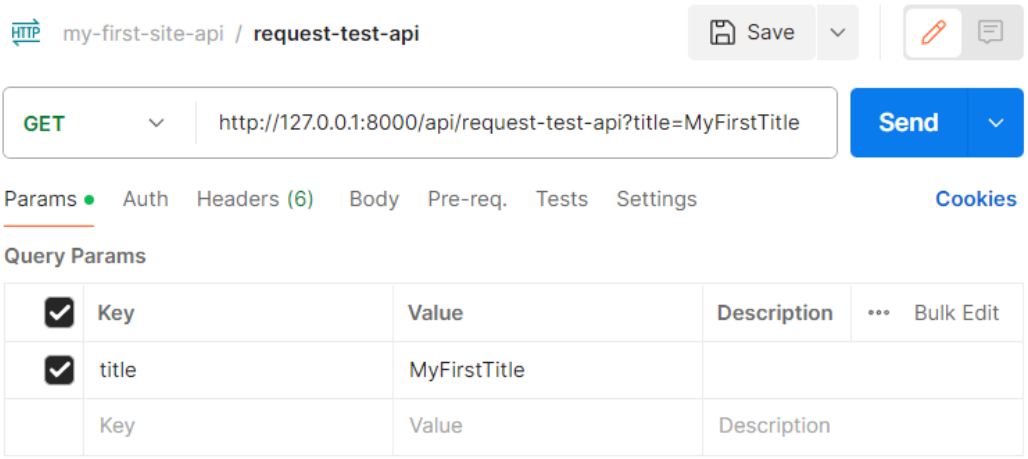
3–11. ábra: Első API hívásunk és eredménye a Postman alkalmazásban

Az iménti ábrán (és természetesen a Postman-ben is) felfedezhetjük, hogy 200-as HTTP állapotkódot kaptunk válaszként, ami az OK-nak felel meg. Válaszidőt és a válasz csomag méretét is láthatjuk a konkrét válasz felett. Ezen kívül a középső szekcióra szeretném még felhívni a figyelmet, mivel ott a *„Params"* lapfülön majd a kéréshez további paramétereket (kulcs-érték párokat) is hozzá tudunk adni. De ennek kipróbálásához először hozzunk létre, regisztráljunk egy új útvonalat az **api.php**-ban:

```
Route::get('/request-test-api', function () {
  return [
    'title' => request('title'),
// példa: http://127.0.0.1:8000/api/request-test-api?title=MyFirstTitle
  ];
});
```

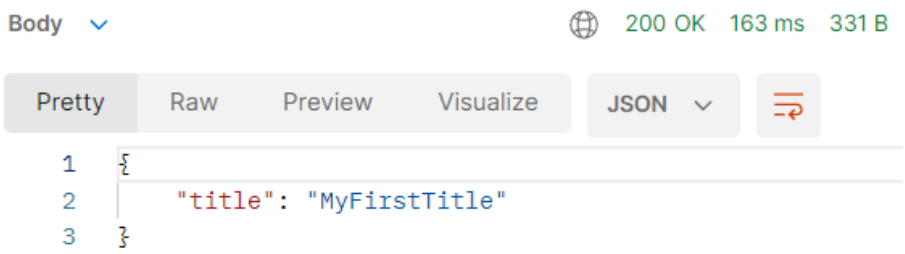
3–40. kódrészlet: Paraméteres API útvonal regisztrálása

Az útvonalat már használtuk a **web.php**-ban, itt csak egy kicsit módosítsunk rajta az iménti kódrészlet alapján (az útvonal elérésén és a **return** utasításon)! A kódrészletben található példa weblinket használhatjuk is a Postman-ben:



3–12. ábra: Paraméteres API kérés a Postman-ben

Vegyük észre, hogy a *„Params"* lapfülön az URL beillesztése után automatikusan kitöltődött az első paraméter kulcs-érték párosa, hiszen az URL már tartalmazta a *title* kulcsot és a hozzá tartozó **MyFirstTitle** értéket.



3–13. ábra: Paraméteres API kérésre kapott válasz a Postman-ben

A válasz törzsében pedig azt figyelhetjük meg, hogy amit az útvonalnál megadtunk visszatérési tömböt, az itt már JSON objektum lesz! Tehát kvázi egy JavaScript objektummal tud dolgozni a kliens oldal, ami rettentően kényelmessé teszi a további munkát az így megkapott adatokkal kapcsolatban.

A *„Params"* lapfülön kívül a *„Headers"* és a *„Body"* is rendkívül fontosak, ezek leginkább majd az erőforrásokat létrehozó, módosító, törlő API útvonal elérések esetén lesznek hasznosak számunkra, amikor kvázi egy űrlap elküldést szeretnénk szimulálni és lekezelni az alkalmazásunk használatával (lásd majd a 7.5.4. alfejezetet).

Ha eddig még nem tudtuk volna rögtön felidézni a HTTP státuszkódokat, állapotkódokat, akkor ezt a hiányosságot érdemes pótolni, mivel ezeknek az ismerete egy API felület elérése és működtetése kapcsán elengedhetetlenek. A HTTP kérésre a válasz mindig tartalmaz egy állapotkódot, azon kívül, hogy kértünk-e le vagy küldtünk-e el valamilyen konkrét erőforrást. Az állapotkódokról egy összefoglaló táblázat itt látható:

Csoport (adott számmal kezdődő háromjegyű kódok)	Csoportba tartozó státuszkódok jelentése
1xx	*Informatív* – Kérés megkapva
2xx	*Siker* – A kérés megérkezett, értelmezésre elfogadásra került, utána pozitív válasz érkezett rá.
3xx	*Átirányítás* – A kérés megválaszolásához további műveletre van szükség (vagy valami történt még a kérés megválaszolása után).
4xx	*Kliens hiba* – A kérés szintaktikailag hibás, vagy nem érhető el az erőforrás, nem teljesíthető a kérés.
5xx	*Szerver hiba* – A szerver nem tudta teljesíteni az egyébként helyes kérést.

3–1. táblázat: HTTP státuszkódok csoportjai és általános jelentésük

Ennél részletesebb listát a 17. fejezetben lévő táblázatban található hivatkozásoknál találhatunk meg. Az állapotkódokhoz két link is elhelyezésre került ott: angol és magyar értelmezésekkel, példákkal együtt is elérhet a részletes lista a hivatkozások nyomán.

Ebben az alfejezetben végrehajtott programkód módosítások ebben a GitHub commit-ben találhatók meg.

3.7. Automatikus tesztelés (útvonalak)

Az olyan komplex keretrendszerek, mint a Laravel is, teljeskörű támogatást nyújtanak a fejlesztők számára a tesztelés végrehajtásához. Mivel most először találkozunk a könyvben az automatikus tesztelés témakörével, ezért egy általános ismertetést adok hozzá, majd megnézzük, hogy a Laravel-ben hogyan is működik ez. Végül a fejezet fő témájához kapcsolódva megnézzük, hogy magukat az útvonalakat és elérésüket hogyan lehet tesztelni.

3.7.1. Tesztelési alapok és szintek

Tesztelésre azért van szükség, hogy az alkalmazásunkban meglévő hibákat még az üzembe helyezés előtt megtaláljuk, ezzel növeljük a minőségét és megbízhatóságát [4]. Abban szinte biztosak lehetünk, hogy az alkalmazásban van hiba, hiszen azt emberek fejlesztik és az emberek hibáznak. Tehát abban szinte biztosak lehetünk, hogy tesztelés előtt van hiba, abban viszont nem lehetünk biztosak, hogy tesztelés után nem marad hiba benne. A tesztelés után azt tudjuk elmondani, hogy a letesztelt részekben nincs hiba, így nő a program minősége és megbízhatósága is. Ez azt is mutatja, hogy a program azon funkcióit kell tesztelni, amiket a felhasználók legtöbbször fognak használni.

1. *Komponens teszt:* a komponensteszt csak a rendszer egy komponensét teszteli önmagában.
2. *Integrációs teszt:* az integrációs teszt kettő vagy több komponens együttműködési tesztje.
3. *Rendszerteszt:* a rendszerteszt az egész rendszert, tehát minden komponensét együttesen teszteli.
4. *Átvételi teszt:* az átvételi teszt során a felhasználók a kész rendszert tesztelik.

Az első három teszt szintet együttesen *fejlesztői tesztnek* hívjuk, mert ezeket a fejlesztő cég alkalmazottai vagy megbízottjai végzik. Ezek általában időrendben is így követik egymást.

A komponensteszt a rendszer önálló részeit teszteli általában a forráskód ismeretében (fehérdobozos vagy strukturális tesztelés). Az ilyen fehérdobozos teszteléseknél mindig a készülő struktúrát (például programkódot) tesztelünk. Az ilyen kész struktúrák a következők: kódsorok, elágazások, metódusok, osztályok, funkciók, modulok.

A kódminőség javítása elméleti és gyakorlati szempontból a 13. fejezetben kerül még inkább fókuszba, így ha valami esetleg nem érthető a tesztelés kapcsán, érdemes áttekinteni majd azt a főfejezetet.

3.7.2. Tesztelés a Laravel-ben

A Laravel-ben, mint alkalmazáson belül komponenstesztet tudunk készíteni. Az itt definiált komponens tesztek a metódusokra és a komplexebb funkcionalitásokra koncentrálódnak. Metódusokat **unit test**-ekkel, bonyolultabb funkcionalitásokat **feature test**-ekkel tesztelhetünk.

Az egységteszt (unit test) a metódusokat teszteli. Adott paraméterekre ismerjük a metódus visszatérési értékét (vagy mellékhatását). A unit test megvizsgálja, hogy a tényleges visszatérési érték megegyezik-e az elvárttal: ha igen, sikeres a teszt, egyébként sikertelen. Elvárás, hogy magának a unit test-nek ne legyen mellékhatása. A unit test-elést minden fejlett programozási környezet (Integrated Development Environment, IDE) támogatja, azaz egyszerű ilyen teszteket írni. A jelentőségüket az adja, hogy a futtatásukat is támogatják, így egy változtatás után csak lefuttatjuk az összes unit test-et, ezzel biztosítjuk magunkat, hogy a változás nem okozott hibát.

A feature test-eket összetettebb, komplexebb funkcionalitások tesztelésére lehet használni, általában valamilyen szerepkörbe (például bejelentkezett felhasználóként) végrehajtva a funkció lépéseit.

A Laravel könyvtárstruktúrája alapból tartalmaz egy **tests** nevű könyvtárat. Ebben két alkönyvtár helyezkedik el, a **Feature** és a **Unit**. A Feature-re gondoljunk úgy, mint ha egy funkcionalitást akarnánk implementálni és utána (vagy előtte) ezt szeretnénk tesztelni a projektünkben. Például az a funkcionalitásunk, hogy el akarunk adni zenéket az oldalunkon, és ezt szeretnénk végig tesztelni: (1) csak

regisztrált felhasználók vásárolhatnak, (2) csak bizonyos termékeket vehetnek, (3) a fizetési folyamatot végig követve menedzselnénk és tesztelnénk a lépéseket stb. A Unit Test egy sokkal jobban izolált, leválasztott tesztelési lehetőséget ad nekünk az alsóbb szinteken. Például adott **class** adott funkcióját teszteljük ezen a szinten. Tehát a tesztelés folyamatát a Laravel-ben úgy képzeljük el, mint ha kívülről befelé (outside → in) haladnánk: **Feature**-től indulunk majd utána a **Unit** teszt következik és az adott osztály adott függvényét akarjuk tesztelni úgy, hogy milyen lefutási eredménnyel tér vissza… ide-oda lépkedünk a két szint között, amíg a hibákat kapjuk, és ki nem javítjuk őket.

Mivel a projektünk már tartalmaz a **tests** könyvtáron belül (és almappáiban) **ExampleTest** osztályokat, ezért azok már futtathatók is. A Laravel a **PHPUnit**-ot használja tesztelésre, ami egy **xUnit** architektúrára épülő unit tesztelő keretrendszer. Ez a projektünkben a **vendor / bin / phpunit** mappában található meg. Futtassuk is le a parancsot a terminal-ban:

```
vendor/bin/phpunit
```

```
PHPUnit 10.0.11 by Sebastian Bergmann and contributors.

Runtime:       PHP 8.1.2
Configuration: C:\xampp\htdocs\jegyzet-2023\my-first-site\phpunit.xml

..                                                                  2 / 2 (100%)

Time: 00:06.176, Memory: 24.00 MB

OK (2 tests, 2 assertions)
```

3–14. ábra: PHPUnit futtatásának eredménye

Visszaadja nekünk, hogy hányas PHPUnit-ot használ a projekt, milyen futtatókörnyezetben vagyunk (PHP verziószáma) és hol található a beállítási fájl, ami alapján lefut a tesztelés. Ezután két pont kerül kiírásra és megadja nekünk, hogy 2/2 (100%-os) volt a tesztek lefutása, tehát mindkét tesztünk sikeres volt. Ha hibát kaptunk volna, akkor az adott teszt helyén a **.** helyett piros hátterű **F** betűt kapnánk. A lefutási idő és a memóriahasználat is kiírásra került végül, illetve egy lezáró üzenet, hogy 2 tesztünk volt és 2 állítás került megerősítésre (assertation). Az **assertion** egy olyan logikai állítás, amelyet a programnak valamely pontján teljesítenie kell (igaznak kell lennie). Az **assert()** utasítások a szemantikát és a teljesítményt illetően is az üres utasítással egyenértékűek. Ha az állítás valóban igaz, az **assert** utasítás hatástalan, ellenkező esetben egy Assertion hiba következik be. A Laravel 10-ben elérhető **assert** utasítások itt érhetők el.

Mivel most sokszor fogjuk használni a tesztelést, érdemes létrehozni egy *alias* nevet, amivel a fenti *„hosszú”* parancsot tudjuk meghívni csak rövidebben (PowerShell-es, vagy röviden, PS-es terminal-ban tudjuk kiadni ezt):

```
Set-Alias -Name phpunit -Value vendor/bin/phpunit
```

Utána már csak egyszerűen a `phpunit` parancs kiadásával tudjuk indítani a tesztelést. Példának okáért rögtön futtassuk ezt úgy, hogy csak egy konkrét fájlt tesztelünk le:

```
phpunit tests/Unit/ExampleTest.php
```

Látható, hogy ez működik (viszont ez csak egyetlen munkamenetig él, tehát ha bezárjuk a terminal-t, akkor utána már nem veszi figyelembe ezt az alias beállítást, *„elfelejti”*). Az utasítás kiadásának hatására már csak 1 teszt és 1 assert futott le, helyesen. Ha megnézzük a két **ExampleTest** osztályt és benne a metódusokat, akkor láthatjuk, hogy a Unit mappában lévő egy egyszerű igaz-igaz állítás egyezőségét vizsgálja meg, a **Feature** mappában lévő pedig lekéri a kezdőoldalunkat, és megvizsgálja, hogy a visszakapott válasz az 200-as HTTP állapotkódot ad-e vissza, vagyis, hogy hibamentesen betölti-e a kezdőoldalt.

A Laravel keretrendszer a PHPUnit használatával egy másik lehetőséget is biztosít a tesztjeink lefuttatására:

```
php artisan test
```

Ez visszaadja, hogy melyik osztály, melyik tesztelő metódusai futottak le helyesen (kipipálva) és helytelenül és melyek hibásan. Ha hibásan futottak le, azt is megkapjuk, hogy mi volt a hiba és hol bukik el a teszt lefutása. Például, ha a **Feature** mappában lévő **ExamleTest** osztály tesztelő metódusában nem a kezdőoldal (**/**), hanem egy másik, nem létező útvonalat szeretnénk elérni vele és arra várjuk a 200-as státusz kódot, akkor a tesztelés el fog bukni, de próbáljuk is ezt ki önállóan és nézzük meg az eredményét!

Mikor teszteljünk? Két lehetőség adódik: implementálás előtt vagy után. Előfordulhat olyan funkcionalitás, amelynek előbb a tesztjét írjuk meg, aztán pedig elvégezzük a tényleges megvalósítást, és kódolunk addig, ameddig ez az új tesztünk OK eredményt nem ad vissza. Ezt hívják *tesztvezérelt szoftverfejlesztésnek* (Test-driven Development). De teszteket akkor is érdemes írni, ha már a megvalósításunk készen van, hiszen ezzel automatikusan elérhetjük azt, hogy megbizonyosodunk róla, hogy valamely funkcionalitások jól működnek és az esetleges továbbfejlesztések után is még mindig jól működnek.

3.7.3. Tesztesetek létrehozása az útvonalakhoz (automatikus tesztelés)

Hozzunk létre egy új tesztet és közben ismerkedjünk a tesztelés szabályaival.

```
php artisan make:test RoutesTest
```

Alapértelmezetten ilyenkor a **Feature** mappába fog bekerülni az új teszt osztályunk. Ha Unit tesztet hoznánk létre, akkor írjuk utána a `-u` kapcsolót. A következő lépés, hogy hozzunk létre tesztelő metódusokat, amelyek visszajelzést adnak a webalkalmazásunk helyes működéséről. Tesztelő metódust kétféle módon hozhatunk létre:

1. Helyezzük el a következő annotációs megjegyzést a metódus előtti sorban: **/** @test */**
2. Kezdjük a metódusunk nevét így: **test**

Mindkettő jó megoldás, azt használjuk, ami számunkra kényelmesebb, logikusabb, átláthatóbb. Adjunk viszont a metódusunknak nagyon beszédes, olvasható nevet, mert a teszt futtatása során, főleg, amikor már nagyon sok lesz belőlük, akkor sokat tud segíteni, hogy rögtön átlássuk, mégis hol és mivel van probléma.

Ha valaki ismeri a *Felhasználói történet* (User Story) tervezői eszközt, annak ismerős lehet az a stratégia, hogy hogyan építsünk fel egy ilyen tesztelési metódust (kövessük tehát a **Given-When-Then**, vagyis a **GWT** sablont):

- **G:** Van az adott felhasználó, aki be van jelentkezve a rendszerbe;
- **W:** Amikor a felhasználó eléri az adott útvonalat (például: **/posts**);
- **T:** Akkor listázza ki neki a blogbejegyzéseket.

Bár a felhasználói hitelesítés témakörével még nem foglalkoztunk, szintén idekívánkozna még az útvonalak teszteléséhez, ha látogatóként (nem bejelentkezett felhasználóként) el szeretnénk érni valamilyen olyan útvonalat (például adott blogbejegyzés szerkesztését), amihez felhasználói hitelesítés szükséges, és ekkor 403-as státuszkóddal térne vissza az oldal (assertForbidden).

Közben persze mindent ellenőrizzünk le assert utasításokkal. De még mielőtt túlságosan bonyolult *Feature* tesztekbe, programfunkciókba mennénk bele. Próbáljunk tesztelni néhány egyszerűbb metódust és közben gyakoroljunk is.

Az útvonalakhoz tartozó assert utasítások közül ismerjük meg és teszteljük az alábbiakat az újonnan létrehozott **RoutesTest.php** fájlunkban (a tesztelő metódus nevét változtassuk meg erre: **test_basic_routes()**, ami beszédesebb, mint az **test_example()** volt):

1. Kérjük le a kezdőoldalt és ellenőrizzük, hogy a nézete a **welcome** lesz-e.
2. Kérjük le a **contact** útvonalat és ellenőrizzük, hogy 200-as HTTP státuszkódú, tehát létezik és elérhető-e.
3. Kérjük le a **contac** útvonalat (végső t betű nélkül, az elírást szimulálva) és ellenőrizzük, hogy nem lesz így megtalálható (tehát 404-es HTTP státuszkódot ad-e vissza az alkalmazás).
4. Kérjük le a **pass-array** útvonalat, egy másik módon is ellenőrizzük, hogy minden oké-e, illetve vizsgáljuk, hogy látható-e az oldalon a market szó, működik-e az adatátadás.
5. Kérjük le a **request-test** útvonalat egy URL-beli paraméterrel és ellenőrizzük, hogy látható-e az oldalon.

A felsorolás pontjainak megvalósítás az alábbi kódokkal valósítható meg:

```
public function test_basic_routes(): void
{
  // 1. példa
  $response = $this->get('/');
  $response->assertViewIs('welcome');

  // 2. példa
  $response = $this->get('/contact');
  $response->assertStatus(200);

  // 3. példa
  $response = $this->get('/contac');
  $response->assertNotFound();
```

```
    // 4. példa
    $response = $this->get('/pass-array');
    $response->assertOk();
    $response->assertSee('market');

    // 5. példa
    $response = $this->get('/request-test?title=MyFirstTitle');
    $response->assertSee('MyFirstTitle');
}
```

3–41. kódrészlet: Útvonalakhoz tartozó tesztek

Adjuk ki a terminal-ban a tesztelési utasítást:

php artisan test tests/Feature/RoutesTest.php

A tesztelés eredményeként láthatjuk, hogy így 1 metódust teszteltünk, amely 6 különböző assert utasítást ellenőriz. Ez rögtön rávilágít arra a problémára, hogy ha valamelyik assert elbukik a 6 közül, akkor a **basic_routes** tesztünk is elbukik. Érdemes lehet emiatt az assert utasításokat külön tagolni, esetleg csoportosítani az egy útvonalhoz tartozókat és ezáltal sokkal pontosabb képet tudunk kapni majd arról, hogy milyen tesztesetek buktak el az ellenőrzés során. Egy lehetséges megoldás például, hogy a basic_routes metódust háromfelé szedjük szét, az elsőben maradna az 1. példa, a kapcsolati oldalt tesztelő metódusba kerülne a 2. és 3. példa, míg az adatátadást tesztelő metódusba kerülhetne a 4. és 5. példa.

Az iménti csoportosítás megvalósítását mindenkire rábízom, akinek esetleg nem menne, az a GitHub commit-ben tudja ellenőrizni a megoldását az enyémmel.

Mit teszteljünk mindenképpen? Nagyon fontos, hogy az alkalmazásunk legfőbb útvonalai (főoldal, rólunk, kapcsolat stb.) mind elérhetőek legyenek, 200-as HTTP állapotkóddal térjen vissza a lekérésük. Ha pedig tesztelünk, akkor készítsünk **automatikus teszteket**, hogy a későbbiekben mindig könnyen ellenőrizhető legyen az alkalmazásunk útvonalainak helyes elérése.

Tipp: Teszteld a tudásod!

Bár még nem tudunk mindent az útvonalakról, de érdekes lehet kipróbálni az eddigi tudásunkkal, milyen teszteknek tudnánk megfelelni. Léteznek nyilvános repo-k, amelyekkel ezt a tesztelést meg tudjuk tenni, itt van például egy, ami az útvonalakhoz tartozik:

https://github.com/LaravelDaily/Test-Laravel-Routes

Ha klónozzuk a projektet és elindítjuk a tesztelést, akkor kezdetben minden tesztünk elbukik. De a **routes / web.php** és **routes / api.php** fájlokban olyan instrukciók vannak a kommentekben, amelyek alapján, ha végrehajtjuk a megvalósítást (implementálást), akkor át fognak menni a tesztjeink. További javaslatok a GitHub projekt leírásában érhetők el a könyvtár- és fájlstruktúra alatt.

Talán kialakult már annyi rutinunk, hogy a Laravel dokumentáció megfelelő részét is felhasználva megoldható az, hogy átmenjenek sikeresen a fejlesztéseink az alkalmazás útvonalainak tesztesetein.

Laravel 11: megújult struktúra (*„slimmer application skeleton”* ötlet).

A Laravel útvonalakat tartalmazó fájljait is érinti a méretbeli csökkentés, emiatt a **routes** mappának tartalma is kevesebb lett. Eltűnt az **api.php** és a **channels.php** fájl is.

Az **api.php** alapértelmezetten tartalmaz egy Laravel Sanctum által védett API útvonalat, amely hitelesített felhasználók, esetleg más alkalmazások, vagy még inkább a kliens oldali keretrendszerek számára biztosítunk amiatt, hogy elérhessék a Laravel alkalmazás logikáját vagy éppen az adatbázisban lévő adatokat egy hitelesítés után. Maga a Laravel Sanctum hitelesítést (authentication) és engedélyeztetést (authorization) is elvárhat a beérkező kérésektől, mielőtt engedné őket kiszolgálni. Az **api.php**-t publikálhatjuk a következő utasítással:

```
php artisan install:api
```

De az utasítás kiadásának hatására nem csak ez az egy fájl jön létre és kap tartalmat, hanem a **laravel/sanctum** csomag is bekerül a projektünk **vendor** (szerver oldali csomagokat tartalmazó) mappájába.

A WebSocket broadcasting (kliens-szerver közti valós-idejű kommunikációs) funkcionalitáshoz szükséges **channels.php** fájl is hasonló módon tud bekerülni a projektünkbe:

```
php artisan install:broadcasting
```

A végrehajtott programkód módosításokat ebben a GitHub commit-ben lehet áttekinteni.

3–4. újdonság: Megújult struktúra az útvonalválasztásnál

3.8. Összegzés

Bár ismereteink az útvonalakról és kezelésükről még közel sem teljes, de az alapokat ezzel elsajátítottuk. A későbbiekben sem tudjuk *„kikerülni”* ezt a területet, mivel mindig innen fog indulni a felhasználói kérések fogadása és feldolgozása.

Ebben a fejezetben elsajátítottuk az útvonalkezelés alapjait, megismertük a legfontosabb (területhez kapcsolódó) `artisan` parancsokat. Az adatátadás témakörét elméleti és gyakorlati szempontból is körül járuk. Paraméteres útvonalakat használtunk azokra az esetekre, amikor adatoktól függően regisztrálunk generikus útvonalakat. Az MVC tervezési minta szerint bejártuk a felhasználói kérések feldolgozását a rövidebb (útvonal -> View) és a hosszabb (útvonal -> Controller metódusa -> Model -> Controller -> View) ágakon is.

Áttekintettük egy új eszköz, a Laravel Folio egyszerűsített *„útvonalkezelési”* technikáit, de ezzel még a későbbiekben foglalkozunk, amikor az erőforrások foglalkozó leggyakoribb műveleteket (lekérdezés, létrehozás, módosítás, törlés) tekintjük át.

Ezután az útvonalak másik nagy csoportjára az API felület útvonalaira helyeztük át a hangsúlyt. Definiáltunk két API útvonalat (paraméter nélkülit és paraméterest), majd ezeket manuálisan teszteltük a Postman alkalmazás segítségével.

Végül az automatikus tesztelés témakörét kezdtük el vizsgálni általánosságban és a Laravel-ben egyaránt. Néhány saját teszteset készítésével zártuk az útvonalakat feldolgozó fejezetet.

Az útvonalakra még többször visszatérünk, például a következők kapcsán: RESTful Controller metódusokhoz vezető útvonalak, elnevezett útvonalak (7. fejezet), middleware-ek használata az útvonalaknál, és a felhasználói hitelesítés útvonalai (10. fejezet) stb.

4. Nézetek (Views)

A webalkalmazás nézetei azok, amelyekkel a felhasználóink először találkoznak a használat során, így egyáltalán nem mindegy, hogy milyen kezdeti benyomással rendelkeznek majd róla. Ha már elérte a webalkalmazásunkat, akkor a nézeteken keresztül tud a felhasználó interakcióba lépni vele, kéréseket küldeni neki és válaszokat kapni tőle.

A fejezetnek nem célja, hogy megismertessen egy teljeskörű kliens oldali keretrendszert, vagy hogy web designer-t faragjon belőlünk. Inkább az lehet a célkitűzésünk, hogy akár előbbiek mélyebb ismerete nélkül is képesek legyünk – egy kicsit backend oldalról szemlélve – teljesértékű frontend-et készíteni az alkalmazásunknak. Ehhez ismerjük meg az építőköveket, amelyeket a Laravel keretrendszer nyújt számunkra.

Kliens oldalon (frontend) fogunk tevékenykedni, így a HTML, CSS alapjainak ismerete fontos a munkánk során, de emellett a Laravel együtt tud működni egy úgynevezett sablon motorral (template engine), amelyet **Blade**-nek hívunk. Ezt már használtuk is a kiíratások során, sőt már egy direktíváját is alkalmaztuk, amikor egy tömb adatait ciklikusan (3.2.2. fejezet) írtuk ki a nézetben. A fejezet során a Blade sablon motorral is bővebben megismerkedünk. és teszteljük is a működését.

A Laravel-ben a nézetek tervezését kétféle irány szerint közelíthetjük meg:

1. *top -> down,* vagyis fentről lefelé haladva, vagy
2. *bottom -> up,* vagyis lentről felfelé építkezve.

Mindkét tervezési és megvalósítási irányt áttekintjük a továbbiakban. A fentről lefelé történő megvalósítást úgy kell elképzelni, hogy a weboldalunk felületén elhelyezünk olyan területeket, régiókat, amelyeket utána megtöltünk tartalommal (4.1. alfejezet). A tervezési módhoz egy sablonon keresztüli megvalósítást is rendelünk (4.2. alfejezet). Közben megismerkedünk még egy hasznos eszközzel, amely a frontend fejlesztést teszi gyorsabbá (Vite, 4.3. alfejezet).

Ezzel szemben a lentről felfelé történő építkezésnél kisebb komponensek sablonjait hozzuk létre paraméterezési lehetőségekkel (4.4. alfejezet). Aztán ezekből a különböző módokon beállított komponensekből építjük fel a weboldalunk bizonyos kinézeti elemeit. Nem ritkán úgy, hogy a manapság rendkívül népszerű Tailwind CSS keretrendszer is használjuk (4.4.1.1. alfejezet).

Majd komponensek megvalósításával egy komplex felület sablont is integrálunk egy új alkalmazásunkba (4.5. alfejezet), amelyet a későbbi fejezetekben még tovább fogunk bővíteni funkcionalitások szempontjából.

4.1. Keretes szerkezet kialakítása

Egy webalkalmazásnak számos olyan része van, amelyek ugyanazok minden felületen. Ha több nézetünk van, akkor ezek az állandó (ritkábban változó, közel azonos) részek használata minden nézetben (ide-oda másolással és beillesztéssel) rettentően megnövelné a hiba előfordulásának lehetőségét. Ilyen ritkán változó webalkalmazás felületi elemek lehetnek például a fejléc, lábléc, menüstruktúra stb. De a CSS/JS importálások ismételt (lemásolt) használata sem túl hatékony, hiszen akkor minden egyes fájlban el

kellene végezni ugyanazokat az importálásokat. Érdemes emiatt létrehozni egy *keretes szerkezetet* (layout), amelyben kialakítjuk a weboldal struktúráját, benne a statikus és a dinamikusan változó részekkel.

A frontend (még nem lefordított és nem optimalizált) részei a Laravel mappastruktúrájában a **resources** mappában helyezkednek el. Ezen belül már alapból van nekünk **css**, **js** és **views** mappánk, utóbbiban találhatók a nézet fájlok, amelyek kapcsán elvárás, hogy a fájlok kiterjesztése **.blade.php** legyen, így tudjuk alkalmazni bennük a Blade sablon motor elemeit, direktíváit.

Hozzunk létre a **views** mappában egy **layout.blade.php** nevű fájlt és mindent, ami a **welcome** nézetben van, másoljunk át ebbe az új fájlba. A **welcome** és a további nézetekbe pedig csak azokat fogjuk majd beleírni, ami azokat egyedivé teszi. Végignézve az új **layout** fájl struktúráját, a **<head>** részei jól használhatók egy közös sablon elemeként, a body lesz az, amit testre fogunk szabni: statikus és dinamikus részekkel bővítjük. A statikus rész egy menü lesz, illetve helyőrzőket fogunk elhelyezni benne a dinamikusan változó tartalmaknak.

4.1.1. Stílusok és a menü struktúra

Töröljük ki a köszönést, mert most arra nem lesz szükségünk itt.

Ennek megvalósítását rögtön egy javaslattal kezdem: a W3Schools weboldal az ilyen gyakran használt weboldal komponensekre mutat nekünk példákat: számos különböző navigációs elemmel, képnézegetővel, gombokkal, táblázatokkal és egyéb elemek felhasználásával kapcsolatban. Például, ha mi egy olyan menüstruktúrát szeretnénk, ami a weboldalunkon fent helyezkedik el, horizontálisan és még reszponzív is legyen, akkor erre mutat nekünk egy példát itt. Használjuk ezt a mi **layout** nézet fájlunkban:

1. Adjuk hozzá a HTML kódot:
   a. a *„font-awesome”* stíluslapot a **<head>** tag-be,
   b. a ** **-et pedig a **<body>** tag-be helyezzük el.
2. Adjuk hozzá a további stílus információkat:
   a. a már a fájlban lévő **<style>** tag tartalmát törölhetjük (a Tailwind CSS tartalmát),
   b. annak helyére szúrjuk be az új stílusszabályokat,
   c. a *„media query”* szabályokat se felejtsük el itt hozzáadni, mivel ezektől lesz majd reszponzív a menü struktúra.
3. A body záró tag-je elé (amikor már felépült az oldal szerkezete) helyezzünk el egy script tag párost, amibe a **myFunction()** függvényt helyezzük el.
4. Végül a menüpontjainkat írjuk felül a már meglévő saját útvonalaink alapján (4–1. kódrészlet) és mentsük el a fájlt.

```
<div class="topnav" id="myTopnav">
  <a href="/">Kezdőoldal</a>
  <a href="/contact">Kapcsolat</a>
  <a href="/pass-array">Tömb adatainak küldése</a>
  <a href="/request-test?title=MyFirstTitle">Adatküldés URL segítségével</a>
  <a href="/posts/elso-bejegyzes">Első blogbejegyzés</a>
  <a href="javascript:void(0);" class="icon" onclick="myFunction()">
```

```
    <i class="fa fa-bars"></i>
  </a>
</div>
```

4–1. kódrészlet: Meglévő útvonalaink a navigációban

Ezzel készen is vagyunk a weboldal struktúránk keretes szerkezetének stílus információval, a menü tartalmával és dinamikus működésével (reszponzívitás tulajdonság eléréséhez). A kinézet működését csak a következő alfejezet módosításai után tudjuk ellenőrizni.

4.1.2. Helyőrzők és kiterjesztések

A szerkezet fájlunkba tegyünk bele egy helyőrzőt (*placeholder*), amely azt jelzi nekünk, hogy hova tudunk majd dinamikusan változó tartalmakat betölteni. A **<body>**-ban a menü szerkezet után szúrjuk be a helyőrzőnket, amit a **@yield** direktívával jelölünk és nevet adunk neki.

```
<div>
  @yield('content')
</div>
```

4–2. kódrészlet: Helyőrző beszúrása a layout.blade.php fájlba

Ezzel megadtuk a dinamikusan változtatható tartalom helyét a HTML szerkezetben. Most hajtsuk végre a menüpontjainkhoz tartozó nézetekben ennek a szerkezet fájlnak a kiterjesztését! Kezdjük a **welcome** nézettel:

```
@extends('layout')

@section('content')
  Hi, {{ $name }}
@endsection
```

4–3. kódrészlet: Szerkezet fájl kiterjesztése a welcome nézetben

Az **@extends** direktívával jelezzük a rendszernek, hogy ki fogunk terjeszteni egy szerkezet fájlt, aminek a nevét a zárójeleken belül adjuk meg. Ezután következik a helyőrzőbe illesztendő kód, amelyet a **@section - @endsection** direktíva párossal jelzünk neki és a **@section** után megadjuk, hogy melyik helyőrző helyére akarjuk betölteni az aktuális kódot.

Ugyanezt megtehetjük a többi nézet fájllal is, amelyeket a menüpontok segítségével el tudunk érni: **contact**, **pass-array**, **request-test?title=MyFirstTitle**, **posts/elso-bejegyzes**. Legutóbbi tartalmát (4–4. kódrészlet) még ide beillesztem példaként, de a többit nem, mert a *„recept”,* amit alkalmazni kell, az ugyanaz:

```
@extends('layout')

@section('content')
  <p>{{ $post }}</p>
  <p>length: {{ $length }} character(s)</p>
@endsection
```

4–4. kódrészlet: Post nézet kiterjesztett változata

Lehetőségünk van arra is, hogy több helyőrzőt illesszünk be a **layout** fájlba a **@yield** direktívával és azokat a **@section** direktíva újbóli alkalmazásával szintén ki tudjuk terjeszteni. Például a **layout** fájlba a **<title>** tag-et bővítsük ki:

```
<title>Laravel - @yield('title')</title>
```

4–5. kódrészlet: Title (cím) helyőrző beillesztése a szerkezet fájlba

Ez azért is hasznos, mert így a webalkalmazás neve után mindig tudunk az adott menüponthoz illeszkedő címet is adni neki. Például a **contact** nézet fájlt így bővíthetjük:

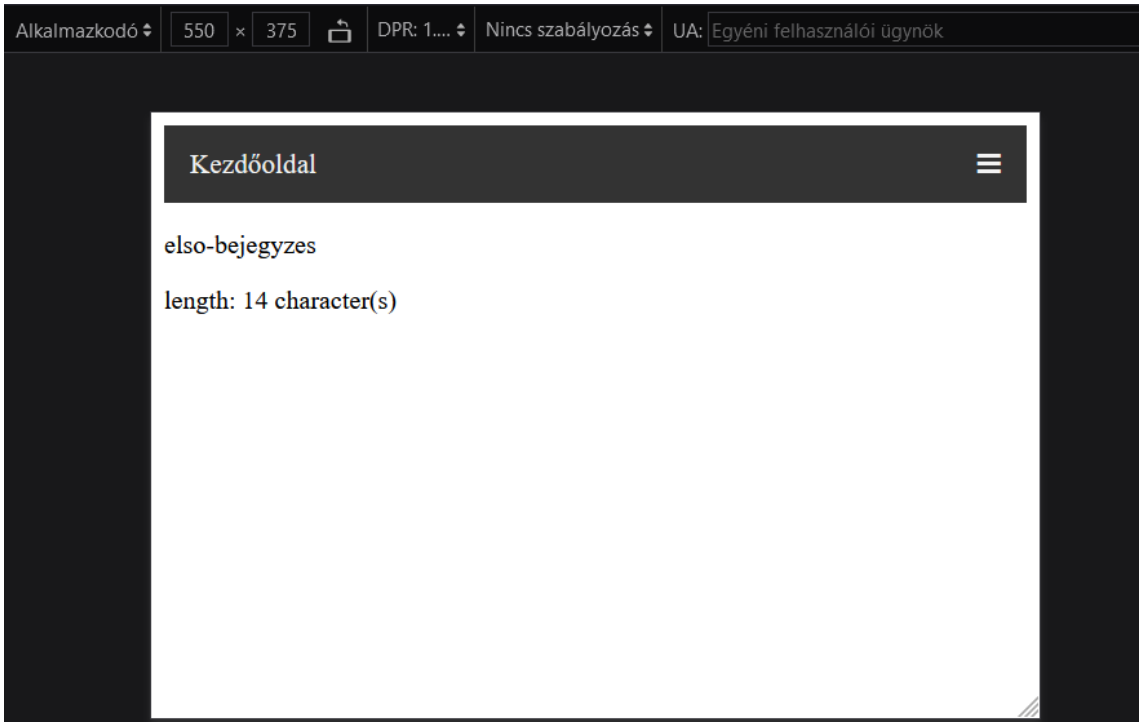
```
@section('title')
Kapcsolat
@endsection
```

4–6. kódrészlet: A kapcsolat oldal dinamikus „alcímének" megadása

Ugyanezt meg tudjuk tenni a többi nézetnél is, értelemszerűen.

Teszteljük is az alkalmazást a böngészőben, és figyeljünk a következőkre:

- A menü struktúra **600px**-nél kisebb szélességű oldalnál reszponzívvá válik és megjelenik jobb felül az oldalon a *„hamburger menü"*, ami dinamikusan működik és csak az első menüpont marad látható (4–1. ábra).
- A menüpontok működnek, az oldalak dinamikus tartalmai betöltődnek, miközben a menü struktúra állandó marad a szerkezet fájlnak köszönhetően.
- Az egyes menüpontok címének második része (*„Laravel – "* után) dinamikusan változik a tartalmak között, ezt a böngésző lapfülén és a tálcán elhelyezkedő ablaknál is megjelenik (de persze az aktuális oldal forráskódjának megtekintésénél is tudjuk ezt ellenőrizni).



4–1. ábra: Az első blogbejegyzés tartalma és a menüsor az 550px széles böngészőben

4.1.3. HTTP hibakódú nézetek keretes szerkezete

A 3.3.1. alfejezetben publikált hibanézetek felüldefiniálása is hasonló struktúrára épül, ha megnyitjuk az **errors** mappát. Itt két szerkezet fájl is található: **layout** és **minimal** névvel. Mindkét fájl tartalmazza azt a helyőrzőt, amely a hibaüzenet kiírására szolgál, a **minimal** még azt is, amelyik a HTTP státuszkódot is kiírja még emellé. Az, hogy melyiket szeretnénk alkalmazni ezek közül, az rajtunk múlik.

Az **@extends** direktívánál *„abszolút"* útvonalat kell alkalmazni. Az ilyenkor alkalmazott *„gyökér"* (kiinduló) könyvtár a **resources / views** mappa, innen indul a keresése a szerkezet fájlnak. Így tehát ha például a **404.blade.php**-ban az **@extends**-en belüli paraméter csak a **minimal**-ra vagy **layout**-ra hivatkozna, akkor az a **resources / views** mappán belül keresné ezeket a fájlokat (a layout-ot meg is találná, csak nem azt, ami az **errors**-ban automatikusan létrejött, hanem az iménti 4.1.2. alfejezetben általunk definiált szerkezet fájlba illesztené be a helyőrzőbe a részeket, ha éppen lennének olyanok, amikre a **404**-es nézet fájl szekciói hivatkoznak). Az abszolút útvonal megadásához használhatjuk a **.** (pont) karaktert a mappák elválasztására, ezért az **@extends('errors.minimal')** könyvtár- és fájlmegadás is megfelelő lenne. A kettőspont alkalmazása is megfelelő lehet itt, mivel a **minimal** az **errors** névtéren (namespace) belül van, emiatt a Laravel keretrendszer ezt is fel tudja dolgozni.

A másik különbség a **@section** direktívánál fedezhető fel. Itt a **@section** nem csak egy paramétert tartalmaz (a helyőrző nevét), hanem egy második paraméter is elhelyezhető benne akkor, ha nem túl hosszú az átadni kívánt üzenet (kódsor). Ekkor nincs szükség **@endsection** záró direktívára sem.

Az alfejezetben végrehajtott programkód-módosítások ebben a GitHub commit-ben érhetők el, ha esetleg a leírások alapján nem sikerült volna végig csinálni mindent, akkor itt könnyen átlátható formában megtalálhatók a módosítások.

4.2. Sablon alkalmazása

Az alkalmazásunk jelenleg még nagyon puritán, szegényes, semmiképpen sem nevezhető ízlésesnek, de ezen fogunk változtatni az alfejezet feldolgozása során.

Az alkalmazni kívánt sablonnal szemben fogalmazzuk meg az elvárásainkat: legyen egyszerű, letisztult, könnyen adaptálható, átlátható, modern, reszponzív… ezek lehetnek talán az általános elvárásaink, és akkor mindenki hozzáveheti még a saját, szubjektív, ízlésének és szükségleteinek megfelelő paramétereket, amelyeknek meg kell felelnie az új weboldal sablonnak.

A lehetőségek tárháza kvázi végtelen, mert rengeteg ilyen sablon érhető el az interneten: be kell csak írni a keresőbe, hogy például: *„free responsive template"* és már záporoznak is ránk a találatok. Az ingyenes kategóriában is vannak nagyon profik és abszolút átlátható megoldások. Én igyekeztem egy olyan sablont választani, ami azért egyszerűbb, nincs túlságosan túlbonyolítva CSS és JavaScript bővítményekkel, hanem csak egy tisztán átlátható struktúrára épül.

A sablon, amit fel fogok használni itt érhető el[5]: https://www.free-css.com/free-css-templates/page275/oldschool-pastel Ingyenesen letölthető és megtekinthető működés közben, reszponzív, tehát mindenféle képernyőméreten szép a megjelenése. Az összecsomagolt fájl mindössze két fájlt tartalmaz, egy **index.html**-t, ami a weboldal tartalmát (szerkezetét) fogja össze és egy **pastel.css** fájlt, ami a stílusszabályokat tartalmazza.

Amit tenni fogunk:

1. A **pastel.css** fájlt betesszük a **public** mappán belül a **css** mappába (ha nincs ilyen mappa ott, akkor létre kell hozni és oda kell beilleszteni a **pastel.css**-t).
2. Az **index.html** fájl tartalmát bemásoljuk egy újonnan létrehozandó **pastel-layout.blade.php** fájlba és módosításokat hajtunk ott végre rajta:
    a. A **pastel-layout.blade.php** fájl helyőrzőit létrehozzuk hasonlóan, ahogy azt tettük a **layout.blade.php** fájlnál.
        i. A **<head>**-ben lévő **<title>**-nél csinálhatjuk ugyanúgy.
        ii. A tartalmi részt (content) jelölő helyőrzőt a **content** azonosítójú ** **-ben a *„Page Items”* megjegyzés után illesszük be eszerint (4–7. kódrészlet).
    b. A menüstruktúrát is átmásoljuk a korábbi **layout** fájlból az új **pastel-layout**-ba, de tartsuk meg az új menünél az itt látható nav-ul-li szerkezetet a felsorolás kinézet megtartásához.
3. A többi, dinamikusan változó nézet fájlnál állítsuk be, hogy az új szerkezet fájlt terjesszék ki: **@extends('pastel-layout')**

```
<div class="pageitem">
  <h2>@yield('title')</h2>
  <p>@yield('content')</p>
</div>
```

4–7. kódrészlet: Tartalmi elemek beszúrása az oldalba

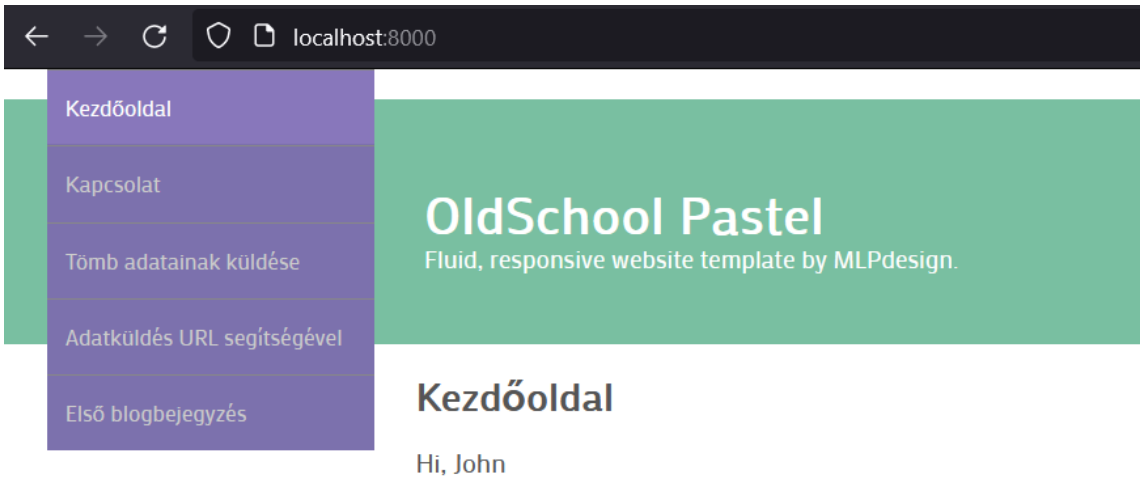
Ha most elindítjuk a webalkalmazásunk kiszolgálását és megnézzük a böngészőben a kezdőoldalt, akkor egy elég csúnyácska, szétcsúszott weboldalt találunk. Közel sem azt, amit esetleg elvártunk volna. Ennek az oka, hogy a CSS fájlunk nincs jól *„bekötve”* a stíluslapok közé a **pastel-layout** fájlban. Ezt így oldhatjuk meg a legegyszerűbben: a **pastel-layout** fájl **<head>** részében, ahol a pastel.css fájlra hivatkozunk, írjuk át erre:

<link rel="stylesheet" href="**/css/**pastel.css">

Talán feltűnhetett, hogy korábban azt írtam, a **resources** mappa fog szolgálni a webalkalmazás frontend (css, js, view stb.) elemeinek a tárolására, most mégis a **public** mappát használtuk arra, hogy eltároljuk benne a pastel.css fájlunkat. Erre még vissza fogok térni, hogy miért van, egyelőre fogadjuk el, hogy a **public** mappában lévő fájlok, mappák azok, amelyekhez nyilvánosan bárki hozzáférhet, így a **public**-ban

[5] Biztonsági mentés, ha az adott oldalon már nem lenne elérhető: OldSchool Pastel Free Website Template - Free-CSS.com.zip

lévő **css** mappához is eszerint tudunk hozzáférni még a Laravel keretrendszer *„további segítsége nélkül”* (Vite, lásd majd a következő alfejezetet).



4–2. ábra: Kezdőoldal az új sablon alapján

Teszteljük újra a böngészőben a megjelenést, a 4–2. ábra mutatja a lényegi részeket: a menüstruktúrát és a kezdőoldal tartalmi részét.

Viszont az még zavaró lehet, hogy amikor átlépünk a Kapcsolat menüpontra, akkor bár a tartalom megfelelően változik, de az aktív menülink nem kerül át a Kezdőoldalról a Kapcsolatra. Ezt is oldjuk meg, mert jelenleg az **active** osztály statikusan be van égetve kódként a Kezdőoldal menüpont **class** attribútumába. Ezt programozzuk át a következőképpen:

```
<li>
  <a href="/" class="{{ Request::path() === '/' ? 'active' : '' }}">
    Kezdőoldal
  </a>
</li>

<li>
  <a href="/contact" class="{{ Request::path() === 'contact' ? 'active' : ''
}}">
    Kapcsolat
  </a>
</li>

<li>
  <a href="/pass-array" class="{{ Request::path() === 'pass-array' ?
'active' : '' }}">
    Tömb adatainak küldése
  </a>
</li>

<li>
```

```
  <a href="/request-test?title=MyFirstTitle" class="{{ Request::path() ===
'request-test' ? 'active' : '' }}">
    Adatküldés URL segítségével
  </a>
</li>

<li>
  <a href="/posts/elso-bejegyzes" class="{{ Request::path() === 'posts/elso-
bejegyzes' ? 'active' : '' }}">
    Első blogbejegyzés
  </a>
</li>
```

4–8. kódrészlet: Aktivitás ellenőrzése az útvonalban

A Laravel **Request** osztályának **path()** statikus metódusa segítségével le tudjuk kérni, hogy mi is az éppen aktuális útvonal, ami betöltődött. Nyilván, ha a Kezdőoldalon vagyunk, akkor ez az útvonal a „**/**", míg, ha a contact útvonalon vagyunk, akkor ez a *„contact"* értéket veszi fel és így tovább. Ha pedig a megadott útvonal értéket veszi fel, akkor adja hozzá a **class** attribútumhoz az **active** osztályt. Ha nincs azon az útvonalon, akkor ne adjon hozzá semmi érdemlegeset a **class** attribútumhoz (ne legyen aktív a menüelem). *Megjegyzés:* a PHP nyelvben létezik ez a feltételvizsgálat, tehát ez gyakorlatilag egy **if**, a **?** (kérdőjel) után van lekezelve az igaz ág, a **:** (kettőspont) után pedig a hamis ág.

Ez a weboldal még tényleg nem a legszebb a világon, viszont, letisztult, könnyen használható és még reszponzív is, úgyhogy a továbbiakban ezt fogjuk használni ahhoz, hogy ismerkedjünk a frontend és a backend rejtelmeivel.

Az alfejezetben végrehajtott programkód-módosítások ebben a GitHub commit-ben érhetők el, ha esetleg a leírások alapján nem sikerült volna végig csinálni mindent, akkor itt könnyen átlátható formában megtalálhatók a módosítások.

4.3. Vite használata a gyakorlatban

A Vite egy modern frontend építő eszköz, ami extrém gyors fejlesztést tesz lehetővé és segíti a JavaScript és CSS fájlok *„bekötését"* az alkalmazásba. Már az előző alfejezetben megtapasztalhattuk, hogy magának a CSS fájlnak a bekötése sem triviális, ez azonban még fokozódik, amikor egy szerverre publikáljuk az alkalmazásunkat, nem csak saját gépen működtetjük. A Vite alkalmas még arra is, hogy az alkalmazásba bekötött JS és CSS kódok függőségeit (egyéb más csomagoktól) felügyelje, kezelje, frissen tartsa. De ezen felül még le is kell esetleg fordítani ezeket a fájlokat, plusz optimalizálni kell őket, hogy amikor majd a felhasználó gépére ténylegesen eljutnak ezek a kódok, akkor a lehető legkevesebb sávszélességet és tárhelyet használják, és így minél gyorsabban megjelenjenek ott.

Tipp: a Vite egy meglehetősen új eszköz a Laravel keretrendszerben, azért, hogy nagyon hatékonyan tudjunk **SPA (Single Page Application)** alkalmazásokat létrehozni valamilyen JavaScript-alapú osztálykönyvtárral vagy keretrendszerrel (Vue.js, React, Inertia JS stb.).

> Korábbi Laravel verzióknál (még a 9-es előtt) a **Laravel Mix**-et lehetett használni, amely a kliensoldali webfejlesztők körében népszerű **Webpack** csomagra épült. A Laravel Mix-ről és használatáról itt írtam a blogomon.

4.3.1. Telepítés

A telepítéshez a könyvben most először igazán szükségünk van a futtatókörnyezet kialakításánál jelzett Node.js-re és az npm csomagkezelőjére. A Laravel projekt mappánk már tartalmaz a gyökér könyvtárában egy **package.json** fájlt, ami minden olyan csomagot tartalmaz, amelyre majd szükségünk lesz a Vite használatához. Telepítsük a csomagokat, amelyek az alkalmazásunk kliens oldali függőségei is egyben:

```
npm install
```

A telepítés hatására létrejött a **node_modules** mappánk a projektben, amely ezeket a kliens oldali függőség csomagjait tartalmazza (többek között a Vite-ot is). A mappa mérete meglehetősen nagy a többi mappához képest, viszont ez mindig *„legenerálható”* a **package.json** fájl segítségével, emiatt a **.gitignore** fájlunk tartalmazza is ezt a mappát, hogy ne terheljük vele a verziókezelő oldalt, jelen esetben a GitHub-ot.

4.3.2. Beállítások

Van továbbá egy nagyon fontos fájlunk, amelynek elemei belépési pontként szolgálnak ahhoz, hogy lefordítsuk és optimalizáljuk a kliens oldali kódjainkat, ez a **vite.config.js** fájl.

A fájlban található **plugin()** metódus tartalmazza azt a bemeneti tömb gyűjteményt, amelyek az alkalmazás belépési pontjai lehetnek. Ez a **resource** mappában már létező **css / app.css** és a **js / app.js** fájlok alapértelmezetten.

4.3.3. Betöltés

A belépési pontok meghatározása után következhet ezeknek a fájloknak a betöltése, amivel csak annyit kell tennünk, hogy hivatkozunk rájuk a **pastel-layout** szerkezet fájlunk **<head>** címkéin belül.

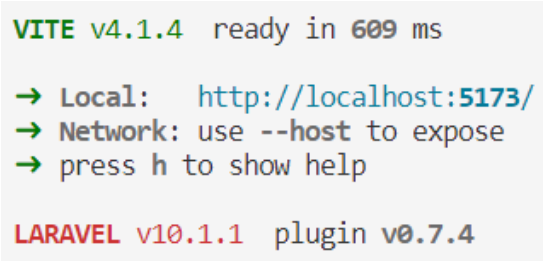
```
{{-- <link rel="stylesheet" href="/css/pastel.css"> --}}
@vite(['resources/css/app.css', 'resources/js/app.js'])
```

4–9. kódrészlet: @vite direktíva beszúrása a betöltéshez (pastel.css stíluslap felhasználásának törlése)

A **pastel.css** stíluslapot azért kommenteltem ki, hogy látszódjon majd a tényleges eredmény és azt is, hogy milyen gyorsan történik meg a módosított fájlok feldolgozása és betöltése.

Most futtassuk a terminal-ban a következő parancsot, hogy elinduljon a fejlesztési szerver:

```
npm run dev
```



4–3. ábra: Vite indulása és futása a terminal-ban

Most, ha ráfrissítünk a webalkalmazásunkra a böngészőben, akkor azt tapasztaljuk, hogy eltűnt, *„elromlott"* az oldal stílusának a szabályzása (mivel kikommenteltük a pastel.css beemelését a **pastel-layout** fájlban). Viszont, ha ezekután a **public / css / pastel.css** fájl tartalmát másoljuk és beillesztjük a **resources / css / app.css** fájlba és mentjük, akkor a mentés után – *a böngésző oldal manuális újratöltődése nélkül!* – betöltődik a friss tartalom és újra megkapjuk a webalkalmazásunk stílusos formázását. Ezután, ha bármit változtatunk az app.css fájlban és elmentjük, akkor azon nyomban láthatjuk is a böngészőben a módosítás eredményét.

Mivel Vue.js-t, React-et, Inertia JS-t nem szeretnék még használni a tudásszintünk ezen részén, emiatt egyelőre ennyit elég tudnunk a Vite-ról ahhoz, hogy hatékonyan tudjunk vele fejleszteni.

Az alfejezetben végrehajtott programkód-módosítások ebben a GitHub commit-ben érhetők el.

4.4. Blade komponensek

A webes alkalmazások kinézetét nem csak szerkezet fájlokkal, Blade direktívák alkalmazásával és sablonokkal lehet hatékonyan elkészíteni: ez a megközelítés tekinthető egy *(„top-down")* nézet tervezési és elkészítési módnak, de megközelíthetjük mindezt a másik irányból is (*„bottom-up")* és ebben fognak segíteni bennünket a Blade komponensek. Ezek a komponensek a Blade sablonok egy részhalmazának tekinthetők és lehetővé teszik számunkra, hogy újfajta, *egyénileg beállítható, paraméterezhető és újrafelhasználható* kódokat készítsünk. Gyakorlatilag egy építőkockának tekinthető egy komponens, amelyet több oldalon is felhasználhatunk, akár egy kicsit mindenhol másképpen (paraméterezéssel) és még adatot is adhatunk át neki. Ilyen építőkocka vagy komponens lehet például egy menü struktúra vagy akár egy figyelmeztető üzenet is. A Blade komponensekbe PHP és HTML kódokat is írhatunk.

A Laravel-ben kétféle komponenst tudunk használni:

1. névtelen (anonymous) vagy
2. egy teljes osztály alapú (class-based) komponenst.

Ezek megismeréséhez hozzunk létre egy új Laravel projektet:

```
laravel new l10-components
```

A projekt létrejötte után pedig hozzuk létre neki a repository-t, majd töltsük fel a GitHub-ra (ehhez segítséget a 2.4.2. alfejezetben találunk). Az általam létrehozott projekt itt érhető el, ezt fogjuk bővíteni ebben és a további fejezetekben.

4.4.1. Névtelen Blade komponensek

A **resources / views** mappában hozzunk létre egy **components** nevű mappát, abban pedig egy **layout.blade.php** fájlt (csak, hogy eltérjünk a már meglévő layout fájlunktól). VSCode-ban egy **!** jel lenyomása után **TAB** gombbal egyből le is tudjuk generálni az érvényes HTML5 szabvány szerinti fájlstruktúrát. A **<title>** tag-eken belül adjuk meg az oldal nevét: *„Blade névtelen komponensek bemutatása Tailwind CSS keretrendszerrel kiegészítve"*. A **<body>** tag-eken belül pedig egy **$slot** változó segítségével fogjuk majd kiíratni, megjeleníteni a komponensünk tartalmát (nagyon hasonlóan, mint amikor a **@yield** - **@section** direktívákkal helyeztünk el és írattunk ki tartalmakat). Ez a **$slot** változó egy speciális változó, amit arra használhatunk, hogy a komponensünk tartalmát kiírhassuk.

```
<body>
  {{ $slot }}
</body>
```

4–10. kódrészlet: $slot változó kiíratása a komponens tartalmának elhelyezése miatt

A **welcome.blade.php** tartalmát cseréljük le a következőre:

```
<x-layout>
  Hello world!
</x-layout>
```

4–11. kódrészlet: Köszöntés megjelenítése komponens alapon

Az **<x-layout>** tag-re szeretném felhívni a figyelmet, mert ez nem egy létező HTML tag. Az *„x"* utal arra, hogy ez egy komponens része lesz, méghozzá a **layout** komponensé. A Laravel keretrendszer *névkonvenciója* azt diktálja, hogy az *„x-"* után következzen a komponens neve, amit a **resources / views / components** mappában fog keresni *„komponensnév.blade.php"* formátumban. Ha ott még egy alkönyvtárba el szeretnénk szeparálni a komponenseinket, akkor az *„x-alkönyvtárnév.komponensnév"* formátumban kellene hivatkozni a komponensre (ponttal válasszuk tehát el az alkönyvtárak nevét egymástól és a komponens nevétől is).

Mentés után elindíthatjuk az alkalmazásunk kiszolgálását, és megnézhetjük a böngészőben, hogy megjelent a köszöntés. Picit csúnya még, ezért közben ismerkedjünk meg a Tailwind CSS alapjaival is, mivel ezt fogjuk használni a komponensek kezelése során.

4.4.1.1. Tailwind CSS keretrendszer telepítése

A Tailwind CSS keretrendszer lehetővé teszi a fejlesztők számára, hogy CSS kódok írása helyett a keretrendszer eszközkészletét felhasználva hozzanak létre szép weboldal szerkezeteket és elemeket paraméterezhető osztályok (**class**) segítségével. A Bootstrap keretrendszerrel ellentétben, ez a Tailwind nem egy nagy előre definiált osztálykészletet (gombok, figyelmeztető üzenetek, navigációs elemek stb.) ad nekünk, hanem például a figyelmeztető üzenet elkészítéséhez ad építőkockákat: sárga háttérszínt (**bg-yellow-200**) és vastagított szöveget (**font-bold**) a zárójelben írt osztályok használatával építhetünk fel, így nagyobb szabadsága van a fejlesztőknek és például nem csak a keretrendszer által nyújtott néhány darab színt használhatja alapértelmezetten.

Mindenekelőtt telepítsük az alapvető kliens oldali csomagjainkat a projektünkbe az **npm** csomagkezelő segítségével:

```
npm install
```

Utána következhet a Tailwind telepítése a Laravel projektünkhöz:

```
npm install -D tailwindcss postcss autoprefixer
```

Ezzel létrehozzuk a két beállítási fájlját a Laravel projektünkben **tailwind.config.js** és **postcss.config.js** névvel:

```
npx tailwindcss init -p
```

Az új **tailwind.config.js** fájlt egy sorban kell módosítani (a **content** tömb tartalmát töltsük fel így):

```
/** @type {import('tailwindcss').Config} */
module.exports = {
  content: [
    "./resources/**/*.blade.php",
    "./resources/**/*.js",
  ],
  theme: {
    extend: {},
  },
  plugins: [],
}
```

4–12. kódrészlet: Tailwind CSS beállításának Laravel specifikus bővítése

A **resources / css / app.css** fájlt nyissuk meg, és adjuk hozzá ezt a tartalmat:

```
@tailwind base;
@tailwind components;
@tailwind utilities;
```

4–13. kódrészlet: A resources / css / app.css fájl új tartalma

A terminal-ban futtassuk a következő parancsot:

```
npm run dev
```

4.4.1.2. Tailwind stílus osztályok alkalmazása a névtelen komponensekben

Most már csak össze kell kötnünk a szerkezet fájlunkat a Tailwind CSS-sel. Helyezzük el a **layout.blade.php** fájl **<head>** címkéi között a következőt:

@vite('resources/css/app.css')

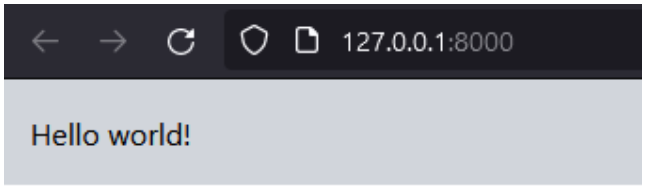
Ismételt mentés és megtekintés után, már azt tapasztalhatjuk, hogy egy kicsit más stílusú a *„világ köszöntő”* szövegünk, de ez még nem teljesen biztos, úgyhogy módosítsuk a **welcome** nézetet az alábbiak szerint:

```
<x-layout>
```

```
  <section class="bg-gray-300 p-4 mb-6">
    <div class="container">
      Hello world!
    </div>
  </section>
</x-layout>
```

4–14. kódrészlet: Tailwind osztályokkal kiegészített köszöntő nézet a szerkezet komponensben

Látható, hogy egy **<section>** és egy ** ** tag-be foglaltuk a világ köszöntését, továbbá mindkét említett tag-nek adtunk meg Tailwind specifikus osztályokat. Az eredmény itt látható:



4–4. ábra: A welcome nézet a böngészőben Tailwind osztályokkal kiegészítve (szürke háttér, térközök)

A böngészőnkben tehát így néz ki a kezdőoldalunk, amely egy komponenst használ a megjelenítésre a Tailwind CSS keretrendszer lehetőségeivel felvértezve (persze még csak nagyon egyszerűen).

Ha most azt szeretnénk, hogy a világ köszöntése háromszor jelenjen meg, akkor csak ismételjük meg az eddig használt szekció (**section**) használatát az **x-layout**-on belül. Abban a szerencsés helyzetben vagyunk, hogy a korábbi `npm run dev` parancs futtatásával a Laravel Vite mindig frissíti a kódunk megjelenítését, ha elmentjük a fájlunk tartalmát.

Ez így egész jól átlátható kezdésnek. De még szervezzük át a komponenseinket egy kicsit: hozzunk létre a **components** mappában egy **section.blade.php** nevű fájlt! A tartalma pedig majdnem megegyezik a **welcome** nézetben látható egyik **<section>**-nel, de a világ köszöntése helyett használjuk benne ismét a **$slot** változó kiíratását:

```
<section class="bg-gray-300 p-4 mb-6">
  <div class="container">
    {{ $slot }}
  </div>
</section>
```

4–15. kódrészlet: A section komponens tartalma

A **welcome** nézet tartalmát pedig módosítsuk így:

```
<x-layout>
  <x-section>
    Hello world again!
  </x-section>
</x-layout>
```

4–16. kódrészlet: A welcome nézet új tartalma két komponens használatával

Egyszer köszöntjük újra a világot, de már a **section** komponenst töltjük fel tartalommal a **layout** komponensen belül. Mindennek helyesen meg is kell jelennie a böngészőnkben egyszer.

De mi van akkor, ha többször szeretnénk köszönteni a világot, viszont mindig egy kicsit másképpen (kinézetben)? Alakítsuk át a **welcome.blade.php**-t:

```
<x-layout>
  <x-section>
    Hello world 1x!
  </x-section>
  <x-section>
    Hello world 2x!
  </x-section>
  <x-section>
    Hello world 3x!
  </x-section>
</x-layout>
```

4–17. kódrészlet: Az x-layout komponensen belül három különböző tartalmú section komponens használata

Ez működik, de túl szürke, mindegyik **section** komponensünkre ugyanaz a stílusosztály (halmaz) vonatkozik. Köszöntsük a szürkén kívül más színben a világot, ehhez pedig használjunk *egy változót és a paraméterátadást*. A változót a **section** komponensbe definiáljuk, és vizsgáljuk is meg az értékét, hogy majd mit kaphat meg. Módosítsuk erre a **section** komponens első sorát (ez csak egy sor, de az átláthatóság kedvéért több sorba van tördelve – a tördelésnek a működés szempontjából nincsen jelentősége):

```
<section class="
  {{ $type === 'success' ? 'bg-green-400' : 'bg-red-400' }}
  p-4
  mb-6
">
```

4–18. kódrészlet: A section komponens háttérszínének paraméterérték-függő beállítása

Egy feltételvizsgálattal ellenőrizzük a **$type** változó értékét, amit majd a **welcome** nézetből küldünk neki. Itt pedig az látható, hogy hogyan küldjük a **$type** értékeit:

```
<x-layout>
  <x-section type="success">
    Hello world 1x!
  </x-section>
  <x-section type="success">
    Hello world 2x!
  </x-section>
  <x-section type="unsuccessful">
    Hello world 3x!
  </x-section>
</x-layout>
```

4–19. kódrészlet: A $type változó értékadásai a komponenshívások fejében

Így az első kettő szekció zöld hátterű lesz, az utolsó pedig piros hátterű, persze egyetlen ilyen komponenssel is tesztelhetjük és annak változtatjuk meg a **type** attribútum értékét. Ha viszont most lehagyjuk a **type** attribútumot a komponenshívás fejében, akkor hibát fogunk kapni, mert a **$type** változónak kötelező értéket kapnia.

Lehetőségünk van adatokat attribútumként hozzáadni a komponensünkhöz. Szúrjuk be a **section.blade.php** fájlunk elejére ezt:

```
@props([
  'type' => 'success'
])
```

4–20. kódrészlet: A $type változónak alapértelmezett érték megadása

Ebben az esetben a **type** változó alapértelmezett értéke a **success** lesz. Le is tudjuk tesztelni, ha például a **welcome** nézetben a harmadik **<x-section>** tag-ből kivesszük a **type** attribútumot az értékadásával együtt. Ekkor eredményül azt kapjuk, hogy már mindhárom szekciónk zöld hátterű lesz a böngészőben. Vagy ha úgy gondoljuk, akkor a harmadiknál meghagyhatjuk a **type="unsuccessful"**-t, és az első kettőnél pedig kivehetjük a **type**-ot, akkor is zöld-zöld-piros hátterű eredménysort fogunk kapni. Próbáljuk ki többféle módon, és nézzük meg, hogyan viselkedik!

Ez mindaddig jó, amíg csak két színnel akarunk játszani, de ha már több van, akkor egy kicsit másképp kell alkalmazni az adathalmazt. A **section** komponens **@props** részét módosítsuk:

```
@props([
  'type' => 'success',
  'colors' => [
    'success' => 'bg-green-400',
    'unsuccessful' => 'bg-red-400',
    'doubtful' => 'bg-gray-400'
  ]
])
```

4–21. kódrészlet: Háttérszín készlet megadása kulcs-érték párokkal

Maradjon így meg a **$type** változó alapértelmezett értéke *„success”*-ként, de tegyünk bele egy **colors** asszociatív tömböt, amivel tudjuk biztosítani a típus (**type**) szerinti háttérszínt. Majd a szekció siker (**success**) esetén zöld, sikertelenségnél (**unsuccessful**) piros, kétséges (**doubtful**) kimenetelnél pedig szürke lesz (ezek természetesen csak példák, bármilyen más színek és értékek is lehetnének).

Ezután a **<section> class** attribútumában módosítsuk az eddigi feltételes elágazást eszerint:

```
<section class="
  {{ $colors[$type] }}
  p-4
  mb-6
">
```

4–22. kódrészlet: A section komponens háttérszínének beállítása típus szerint

Ekkor a komponens a típus szerinti háttérszínt fogja felvenni. Nincs is más hátra, mint a **welcome** nézetben, amikor a három **section** komponenst használjuk, az elsőnél a **type**-ot nem kell megadni, ez lesz az alapértelmezetten sikeres (zöld), a másodiknál a **type** legyen *„doubtful"*, tehát szürke háttérszínt kap majd, míg az utolsónál legyen a **type** attribútum értéke *„unsuccessful"*, így annak piros háttérszíne lesz.

```
<x-layout>
  <x-section>
    Hello world 1x!
  </x-section>
  <x-section type="doubtful">
    Hello world 2x!
  </x-section>
  <x-section type="unsuccessful">
    Hello world 3x!
  </x-section>
</x-layout>
```

4–23. kódrészlet: A section komponensek paraméterezése

Az eredménye pedig itt látható:

4–5. ábra: Komponens paraméterezésének eredménye a böngészőben

Természetesen a **section** komponensben lévő **colors** tömb értékeit bővíthetnénk például úgy, hogy ne csak háttérszínt adjunk nekik, hanem keretet is, például a **success**-nél **'border-green-500'**-at adhatnánk hozzá. Minden csak rajtunk múlik, hogy mit és hogyan szeretnénk, a Tailwind CSS keretrendszer összes lehetséges osztálya a rendelkezésünkre áll.

Végül, ha szeretnénk használni a **welcome** nézet **section** komponenseiben egy másik Tailwind osztályt a kinézet módosítására, az sajnos nem fog működni alapból. Például a harmadik **section** komponensnél állítsunk be egy **mt-10**-et (**margin-top**), ami **2.5rem** nagyságú felső margót jelentene a **section**-nek.

```
<x-section type="unsuccessful" class="mt-10">
```

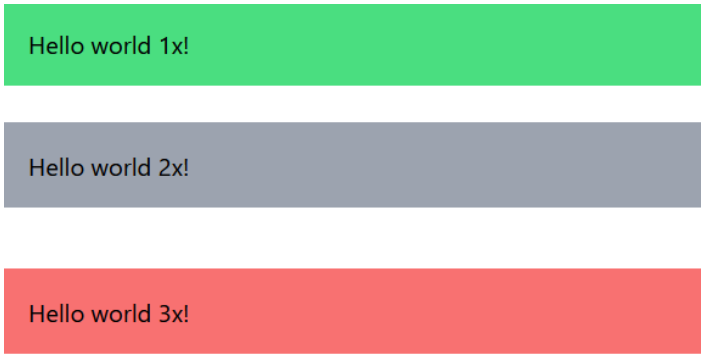
4–24. kódrészlet: További osztály hozzáadása a section komponenshez

Ennek következtében még nem változik a kinézet, nem fogja megkapni a nagyobb felső margót a harmadik **section**. Az érvényre juttatásához módosítanunk kell a **section.blade.php**-ben a **<section>** tag belső tartalmát, hogy *össze tudjuk vonni az esetlegesen kapott és az itt beállított* attribútumok értékeit.

```
<section {{
  $attributes->merge([
    'class' => "{ $colors[$type] } p-4 mb-6"
  ])
}}>
```

4–25. kódrészlet: Osztályok összevonásának lehetősége biztosítva

Így lehet az attribútum *„zsákban”* lévő paramétereket és értékeiket összefésülni. Most már működni fognak azok az osztályok is, amiket a komponens példánynál adtunk hozzá (**mt-10**).



4–6. ábra: Nagyobb felső margó az utolsó szekció előtt

Az alfejezet során láthattuk, hogy hogyan lehet végrehajtani akár többszöri kiíratásokat előre legyártott komponens sablonok segítségével. A kiíratásokat (és stílusukat) paraméter átadásokkal tudtuk beállítani, szabályozni és végül egyedileg hozzáillesztett osztályokkal is összefűzni, ha ennek láttuk szükségét.

Ez a technika akkor is nagyon hasznos lesz, amikor hasonló űrlapokat akarunk majd létrehozni, minimális paraméterezéssel újra felhasználva őket (lásd a későbbiekben a 8.2.2.2. alfejezetet).

Az alfejezetben végrehajtott programkód-módosítások ebben a GitHub commit-ben érhetők el.

4.4.2. Osztály alapú Blade komponensek

Az osztály alapú komponens – a névtelen komponensekkel szemben – egy üzleti logikát magába foglaló osztályt és metódusait is tartalmazza a nézet sablont tartalmazó osztályon kívül. Ennek szemléltetésére létre fogunk hozni egy figyelmeztetés (**alert**) komponenst a következő utasítással:

```
php artisan make:component Alert
```

Az eredményül létrejövő két osztály pedig itt található:

1. app / View / Components / Alert.php
2. resources / views / components / alert.blade.php

Előbbi fájlban lévő osztály a **Component** ősosztályból öröklődik és egy üres konstruktoron kívül tartalmaz még egy **render()** megjelenítő metódust, amely az utóbbi fájl – egyelőre üres – sablonként való felhasználásával tér vissza.

Mivel át szeretnénk látni a működését, ezért végig csinálunk egy példa felhasználást a figyelmeztető üzenet kiíratására. A figyelmeztető üzenet típusát itt is paraméterátadással (adatokkal) tudjuk majd testreszabni.

Az **app / View / Components / Alert.php** konstruktorában adjuk meg, hogy ezeket a változókat **($title, $message**) lehet majd módosítani a komponens példányosításakor:

```
public function __construct(public string $title = ''
, public string $message = '')
{
  //
}
```

4–26. kódrészlet: Alert lehetséges paramétereinek megadása

Ehhez mindössze annyit kellett tenni, hogy a konstruktor paraméterei közé felsoroltuk őket és mivel nem szeretnénk **Exception**-t kapni, ha esetleg valamelyik változónak nem adunk értéket, ezért mindkettőhöz hozzárendeltünk egy-egy üres szöveget, ami felülíródik, ha érkezik konkrét érték.

A Tailwind CSS keretrendszer szolgáltat számunkra figyelmeztetéseket, amelyek közül én a címmel ellátottat választottam ki az **alert** sablonjához. A szövegeket helyettesítsük változókkal a sablon fájlban (**alert.blade.php**):

```
<div role="alert">
  <div class="bg-red-500 text-white font-bold rounded-t px-4 py-2">
    {{ $title }}
  </div>
  <div class="border border-t-0 border-gray-400 rounded-b bg-gray-100 px-4
py-3 text-gray-700">
    <p>{{ $message }}</p>
  </div>
</div>
```

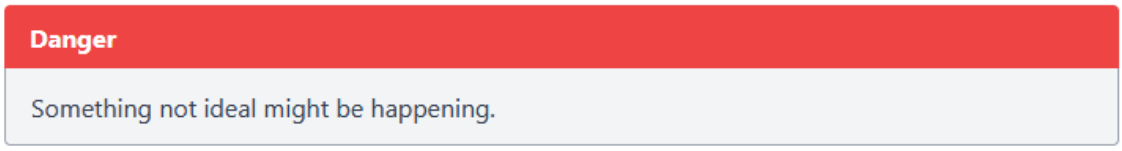
4–27. kódrészlet: Az alert sablon kialakítása

A komponens példányosítása a **welcome** nézetben történhet meg a legegyszerűbben (egy címsorral elválasztom az előzőektől és még egy margót is beállítok az új figyelmeztetésnek):

```
<h1>Class based components</h1>
<div class="m-8">
  <x-alert title="Danger" message="Something not ideal might be
happening."/>
</div>
```

4–28. kódrészlet: Figyelmeztető komponens elhelyezése a kezdőoldalon

Az eredmény ilyen lett a böngészőben:



4–7. ábra: Veszélyre figyelmeztető üzenet a kezdőoldalon

Így tehát bármiféle adatküldés nélkül elérhetővé tettük a változókat és tartalmaikat az **Alert** konstruktorán keresztül a **render()** metódusban is, ami által automatikusan az **alert** sablonú nézetben is.

Ez ugye mindenképpen valamilyen veszélyre hívja fel a figyelmet, de meg tudjuk tenni azt is, hogy más típust adunk meg neki, ami egy sikeres művelet végrehajtására hívja fel a figyelmet. Ehhez először az **Alert** osztály konstruktorát bővítsük egy újabb paraméterrel, hasonlóan a **$title**-höz és a **$message**-hez egy **$type**-ot adjunk hozzá és neki is adjunk alapértelmezetten értékül egy üres szöveget.

Ezután következhet az **alert.blade.php** sablon kibővítése:

```
@props([
  'type' => 'success',
  'colors' => [
    'success' => 'bg-green-500',
    'danger' => 'bg-red-500'
  ]
])

<div role="alert">
  <div class="{{ $colors[$type] }} text-white font-bold rounded-t px-4 py-
2">
    {{ $title }}
  </div>
  <div class="border border-t-0 border-gray-400 rounded-b bg-gray-100 px-4
py-3 text-gray-700">
    <p>{{ $message }}</p>
    <p>{{ $slot }}</p>
  </div>
</div>
```

4–29. kódrészlet: A dinamikusabb figyelmeztetés sablonja

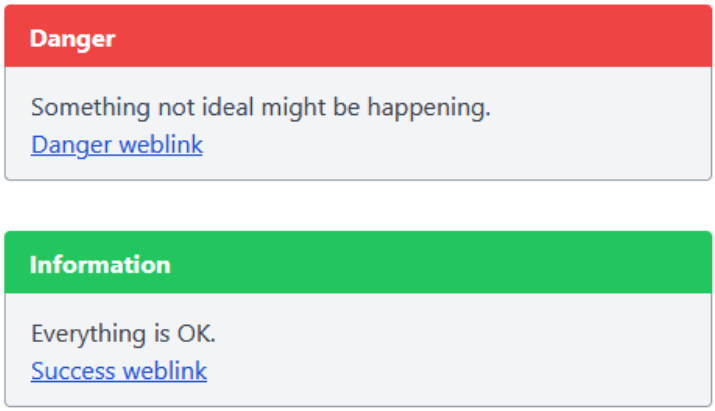
A színekkel való kiegészítés nagyon hasonló arra, amit a névtelen komponenseknél már alkalmaztunk. A **$slot** kiíratására szeretném felhívni a figyelmet, mert ezzel lehetőségünk lesz direkten beszúrni az adott helyre további HTML elemeket, például egy weblinket. A **welcome** nézetet így alakítsuk át, bővítsük:

```
<div class="m-8">
  <x-alert title="Danger" message="Something not ideal might be happening."
type="danger">
    <a href="#" class="text-blue-700 underline">Danger URL</a>
  </x-alert>
</div>
<div class="m-8">
  <x-alert title="Information" message="Everything is OK." type="success">
```

```
    <a href="#" class="text-blue-700 underline">Success URL</a>
  </x-alert>
</div>
```

4–30. kódrészlet: Weblinkek hozzáadása a figyelmeztetésekhez

Eredményül a következő két figyelmeztetést kapjuk:



4–8. ábra: Különböző komponens példányok testreszabottan

Most már többféle szerkezet, sablon, komponens alapú kinézetet megvalósítottunk a projektünkben. Innentől csak rajtunk múlik, hogy melyik megjelenítési formát választjuk a továbbiakban és mit fejlesztünk, finomítunk tovább. Természetesen arra is lehetőségünk van, hogy a két módszert ötvözzük: a keretes szerkezet és a komponensek alkalmazását vegyítsük a Blade direktívákkal!

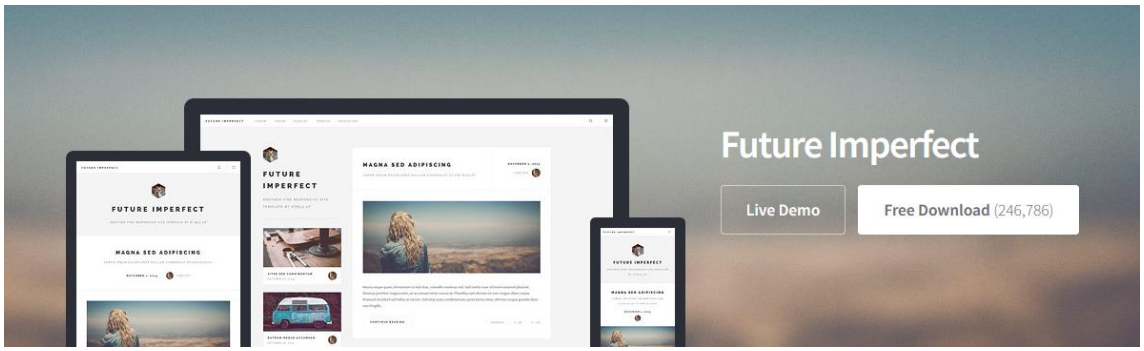
A jövőben mindig azt érdemes alkalmazni, amely jobban *„kézre áll”,* vagy az adott helyen logikusabb a használata.

Az alfejezetben végrehajtott programkód-módosítások ebben a GitHub commit-ben érhetők el.

4.5. Komplex sablon integráció Vite eszközzel

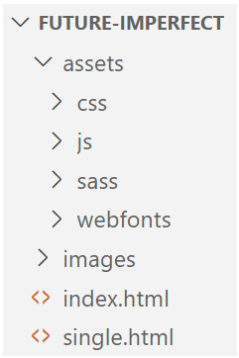
A tudásszintünk ezen pontján, azt javaslom, hogy érdemes gyakorolni és egy újabb sablont felhasználni ebben az új **l10-components** nevű projektünkben. Létezik egy másik sablon lelőhely, a https://html5up.net/ weboldal, ahol a sablonok neve és kinézete látható, valamint van két gomb is, amelyek közül az első egy élő demonstrációs oldalhoz, a második pedig közvetlenül a sablon letöltéséhez vezet minket (itt látható a népszerűsége is, hogy hányan töltötték le eddig). Az én választásom a Future Imperfect nevű sablonra[6] esett, mivel ez egy jó blog oldal kinézete. De szabadon választható bárki számára, hogy kinek melyik tetszik.

[6] Biztonsági mentés, ha az eredeti oldalon már nem lenne elérhető: html5up-future-imperfect.zip



4–9. ábra: A Future Imperfect sablon az oldalon

Az oldalon fent lévők között vannak könnyebben és nehezebben integrálhatók, ez a Future Imperfect egy nehezebben integrálható sablonnak tűnik elsőre. Azért gondolom, így, mert ha megnézzük a könyvtár és fájlszerkezetét, akkor több különféle fájl(típus) integrálását kell megvalósítani a helyes működéshez.



4–10. ábra: Future Imperfect sablon könyvtár- és fájlszerkezete

4.5.1. Tartalmi elemek áttekintése

Tekintsük át, hogy mit tartalmaz a letöltött sablon csomag (4–10. ábra)! Közben pedig gondoljuk át, hogy hol érdemes majd tárolni őket ahhoz, hogy a Vite eszközzel könnyedén fel tudjuk majd építeni a sablont a Laravel projektünkben.

1. HTML fájlok: ezek integrálása nézetek vagy komponensek segítségével megvalósítható. Kétféle HTML oldal található benne:
    a. Az **index.html** tartalmazza a főoldalt, amely a különböző komponenseket (például navigáció, blogbejegyzés lista lapozással, bemutatkozó rész, közösségi oldalak ikonjai stb.)
    b. A **single.html** egy kiválasztott blogbejegyzést mutat teljes egészében. Kiegészítésként itt mindössze a navigációs sáv van meg, alul pedig egy lábléc a közösségi oldalak ikonjaival.
2. Képek: ezek eltárolhatók a **resources** mappa **images** almappájában.
3. **Assets / Webfonts** (betűstílusok): ugyanaz igaz rájuk, ami az iménti képekre is.
4. **Assets / JavaScript**: a fájlokra szintén ugyanez igaz.
5. **Assets / CSS**: a fájlokra szintén ugyanez igaz lehet, ez a könnyebb út…
    a. …de vannak **Assets / SASS** előfeldolgozásra váró SCSS stílusfájlok is, amelyeknél egy elég komplex mappa és fájlszerkezet van kialakítva, részletesebben is meg kell tekinteni, hogy hogyan implementálható majd a mi alkalmazásunkba, ha ezekből akarjuk felépíteni.

4.5.2. HTML fájlok integrálása

A HTML fájlok felelősek az oldal tartalmáért. Nekünk két HTML fájl tartalmát kell beépítenünk a saját alkalmazásunk sablonjába:

1. index.html tartalma kerüljön a layout.blade.php -be
2. single.html tartalma kerüljön a post.blade.php -be

Dolgozzunk közösen az első említett fájllal! Azonosítsuk a szekciókat a layout.blade.php fájlban, amelyek külön nézetekbe kerülhetnek, ezáltal könnyebb lesz velük majd dolgozni is. Hozzunk létre nekik egy includes mappát a resources / views mappán belül. Az új includes mappába pedig helyezzük el a következő fájlokat:

1. _header.blade.php -be másoljuk a layout.blade.php-ből a header szekciót
2. _menu.blade.php -be másoljuk a layout.blade.php-ből a menu szekciót
3. _sidebar.blade.php -be másoljuk a layout.blade.php-ből a sidebar szekciót (ez magába foglalja a bemutatkozást – intro, mini blogbejegyzéseket – mini posts, blogbejegyzés listát – posts list, a rólunk – about és a lábléc – footer részt). Legutóbbi kell majd az egyszeri blogbejegyzés nézetnél is, úgyhogy annak hozzunk létre egy _footer.blade.php include fájlt is, amit majd a _sidebar.blade.php-n belül is beemelünk (include)
4. _scripts.blade.php -be másoljuk a layout.blade.php-ből a scripts szekciót

Azért érdemes megkülönböztetni már a nevükben ezeket a nézeteket a *„teljes értékű"* nézetektől, mert ezeket csak beemeljük (include) egy másik nézetbe részegységként, emiatt _ (aláhúzással) kezdtem a nevüket. Így, ha a jövőben rápillantunk, akkor rögtön láthatjuk, hogy ezek csak *„résznézetek",* ez csak a saját magunk számára meghatározott névkonvenció, a Laravel számára ez nem lényeges.

A header, menu, footer, js kódok importálása megegyezik a layout és a post keretes fájlokban is.

A beemeléshez majd az @include Blade direktívát tudjuk használni, de előbb a megfelelő részegységeket másoljuk át az újonnan létrehozott fájlokba. Például a <header> címkét és belső tartalmát a _header.blade.php-ba. Majd a layout-ban maradt helyen simán be tudjuk emelni így: @include('includes._header')

Átalakítások után így néz ki a layout.blade.php fájlunk tartalmi része a body címkék között (*megjegyzés:* a post elemeket nem töröltem, csak a VSCode segítségével a kód sorszáma mellett összecsuktam őket – lásd 4–11. ábra):

```
11      {{-- {{ $slot }} --}}
12
13      <!-- Wrapper -->
14      <div id="wrapper">
15
16          <!-- Header -->
17          @include('includes._header')
18
19          <!-- Menu -->
20          @include('includes._menu')
21
22          <!-- Main -->
23          <div id="main">
24
25              <!-- Post -->
26              <article class="post">…
49              </article>
50
51              <!-- Post -->
52              <article class="post">…
75              </article>
76
77              <!-- Post -->
78              <article class="post">…
101             </article>
102
103             <!-- Pagination -->
104             <ul class="actions pagination">
105                 <li><a href="" class="disabled button large previous">Previous Page</a></li>
106                 <li><a href="#" class="button large next">Next Page</a></li>
107             </ul>
108
109         </div>
110
111         <!-- Sidebar -->
112         @include('includes._sidebar')
113
114     </div>
115
116     <!-- Scripts -->
117     @include('includes._scripts')
```

4–11. ábra: A szerkezet fájl (layout) tartalmi része (body tag-eken belüli része)

Az már ebből a kódrészletből (4–5. kódrészlet) is látszódik, hogy a post-oknak érdemes lenne szintén egy szekciót létrehozni, amit utána majd egy ciklussal (például a **@foreach** Blade direktívával) többször egymás után kiíratunk a főoldalra, de most egyelőre ezzel még ne foglalkozzunk.

Megjegyzés: a fő tartalmi rész (**main**) magába foglal egy olyan blogbejegyzést (post), ami teljesen ki van kommentelve. Ez egy olyan bejegyzés, amiben példát mutat számos olyan dologra, amit a sablon nyújthat nekünk a jövőben, hogy hogyan használjunk táblázatokat, űrlap elemeket, listákat, címsorokat stb. Ez a több száz soros forráskód megjegyzés törölhető a **layout.blade.php** fájlunkból, de az **index.html**-ben maradjon meg komment nélkül, hogy emlékezzünk rá, ezeket a kiegészítő sablon elemeket is tudjuk majd használni a jövőben.

4.5.3. Külső erőforrás (kép, JavaScript, CSS és betűstílus) fájlok integrálása

A sablon csomagban lévő **images** mappát másoljuk és illesszük be a projektünk **resources** mappájába! Ugyanezt tegyük meg az **assets / webfonts** mappával is, kerüljön be a projektünk **resources** mappájába.

Majd a JavaScript fájlokkal tegyünk ugyanígy (**assets / js** tartalmának másolása és beillesztése a **resources / js** mappába). Végül a CSS fájlokkal is tegyük meg ugyanezt (**assets / css** tartalmának másolása és beillesztése a **resources / css** mappába).

4.5.3.1. Kép és betűstílus fájlok

Ha a JavaScript vagy CSS fájljainkban szeretnénk használni a képeket például háttérként, akkor a Vite-nak meg kell ezt adni, hogy tudja őket kezelni és még verzióval is el tudja látni őket (a Vite a háttérben ezt kezeli). Ehhez nyissuk meg a **resources / js / app.js** fájlunkat és adjuk hozzá a következőket:

```
import.meta.glob([
  '../images/**',
  '../webfonts/**',
]);
```

4–31. kódrészlet: Képek és betűstílusok mappájának bekötése

Ezáltal ezek a tartalmak feldolgozásra kerülnek, amikor futtatjuk a következő parancsok közül valamelyiket:

- npm run dev
- npm run build

Az első a fejlesztés során hasznos és alkalmazandó, a második akkor, amikor a végső weboldalunkat szeretnénk publikálni egy nyilvános helyen és minden kliens oldali kódot le kell fordítani, optimálissá kell tenni.

Ezután koncentrálhatunk a képekre és a **layout.blade.php** fájlban keressünk meg az összes **<img>** tag-et (6 helyen), és alakítsuk át az **src** attribútumon belüli hivatkozást ilyenre:

```
<img src="{{ Vite::asset('resources/images/pic01.jpg') }}" alt="" />
```

4–32. kódrészlet: Oldal statikus erőforrásainak beemelése

Ugyanezt tegyük meg a **resources / views / includes / _sidebar.blade.php** fájlban is az **<img>** tag-ekkel (~14 helyen).

Közben, ha figyeljük az oldalunkat működés közben, akkor már láthatjuk, hogy meg is jelentek a képek (bár a sablon maga csak ilyen elmosódott képeket adott nekünk, de ne legyünk telhetetlenek).

4.5.3.2. JavaScript fájlok

A JavaScript fájlokra való hivatkozást a **resources / views / includes / _scripts.blade.php** fájlban találjuk. Alakítsuk át itt is az **src** attribútumot, hasonlóan, ahogy a képeknél tettük az imént.

```
1   <script src="{{ Vite::asset('resources/js/jquery.min.js') }}"></script>
2   <script src="{{ Vite::asset('resources/js/browser.min.js') }}"></script>
3   <script src="{{ Vite::asset('resources/js/breakpoints.min.js') }}"></script>
4   <script src="{{ Vite::asset('resources/js/util.js') }}"></script>
5   <script src="{{ Vite::asset('resources/js/main.js') }}"></script>
```

4–12. ábra: JavaScript fájlok beemelése

4.5.3.3. CSS stíluslap fájlok

A működő stíluslap megjelenítéséhez két dolgot kell tennünk:

1. A **vite.config.js** fájl input tömbjében fel kell sorolni a **resources / css / main.css** fájlt is
2. A **layout.blade.php** fájlban a **@vite** direktíva paraméterei között helyezzük el elsőnek (!) a **main.css** fájlt. Ha második helyre tennénk az app.css után, akkor csak egy pillanatra jelenne meg jól az oldalunk betöltődéskor, utána felülíródnának a stílusszabályok és helytelenül jelenne meg az oldal.

4.5.3.4. CSS alternatíva: SASS mappa SCSS fájljai

Ha a main.css helyett az előfeldolgozott SASS fájlokat szeretnénk használni, akkor erre is van lehetőségünk. Először másoljuk be a sablonból az **assets/sass** mappát a projektünk **resources** mappájába. Utána csináljuk meg a **resources / css / main.css** fájl helyett a **resources / sass / main.scss** fájllal, hogy betesszük és hivatkozunk rá a **vite.config.js**-ben, majd a layout nézetben is a **@vite** direktívában. Ha ezeket végrehajtjuk, akkor azonban hibaüzenetet kapunk a böngészőben (a terminal-ban is hibaüzenetet kapunk, ahol a Vite fut).

```
[plugin:vite:css] Preprocessor dependency "sass" not found. Did you install it?

C:/xampp/htdocs/l10-components/resources/sass/main.scss
```

4–13. ábra: Hiányzó SASS feldolgozó

Ekkor állítsuk le a Vite futását a terminal-ban (Ctrl + c billentyűkombinációval), majd telepítsük a SASS csomagot így:

```
npm install --save-dev sass
```

Ezután újraindíthatjuk a Vite-ot:

```
npm run dev
```

Az oldal betöltődésekor már egy másik hibaüzenetet fogunk kapni:

```
[plugin:vite:css] [postcss] Failed to find 'fontawesome-all.min.css'
  in [
    C:/xampp/htdocs/l10-components/resources/sass
  ]

C:/xampp/htdocs/l10-components/resources/sass/main.scss
```

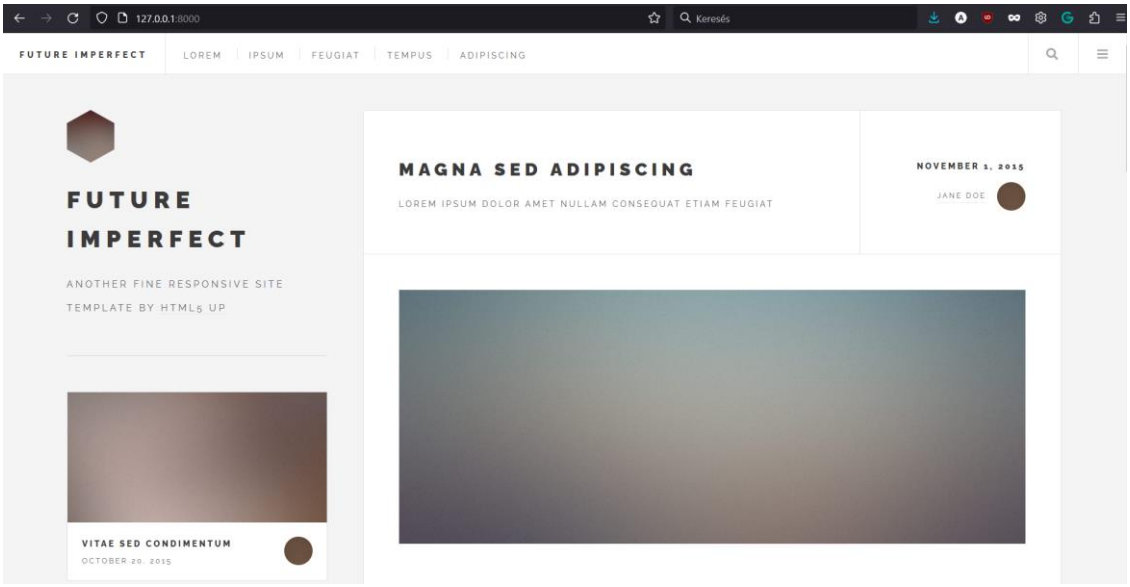
4–14. ábra: Forrásfájl rossz elérési úttal

A **resources / sass / main.scss** fájlban rossz a **fontawesome-all.min.css** fájlnak az elérési útja. Javítsuk ki erre az importáló sorban a fájl elérését: @import '**/resources/css/**fontawesome-all.min.css';

Most már megfelelő lesz az oldalunk megjelenítése nem csak a main.css segítségével, hanem a SASS fájlokkal együttműködésben is. *Megjegyzés:* a Vite-ot futtató terminal jelez nekünk figyelmeztetéseket, amik bár nem rontják el az oldal stílusát, érdemes lehet őket kijavítani.

4.5.3.5. Végső módosítás a főoldalon: welcome nézet

Végül a **welcome** nézet tartalmát törölhetjük vagy kikommentelhetjük, és csak egy **<x-layout/>** tag-et hagyjunk benne, így megkaphatjuk a végső, működő sablont követő kezdőoldalunkat:



4–15. ábra: A futó alkalmazásunk az új sablonnal

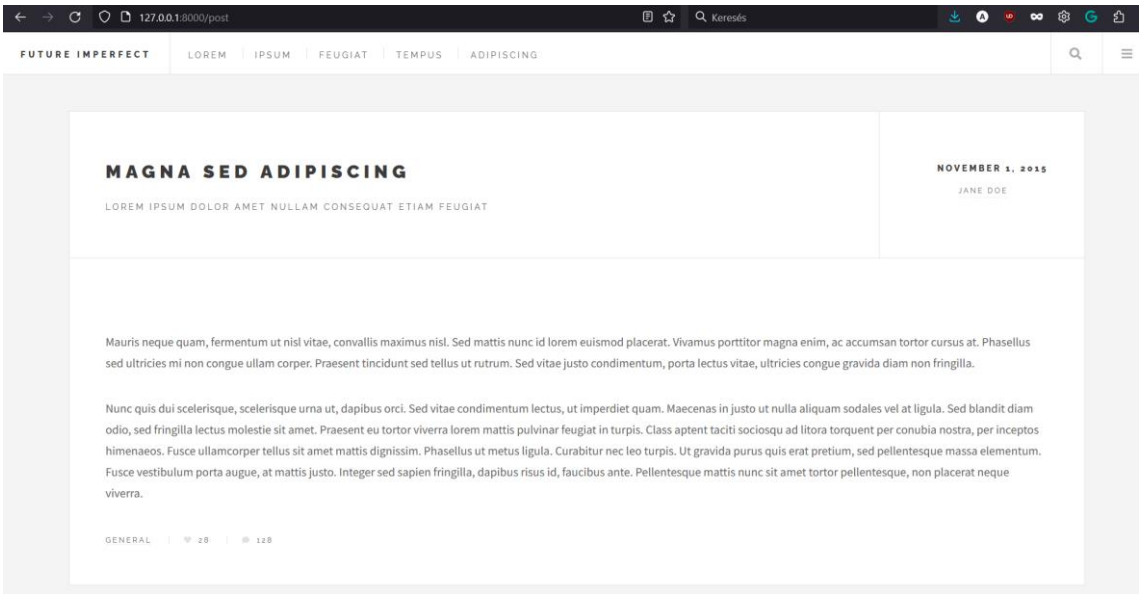
4.5.3.6. Post nézet és útvonal

Ezekután a post nézet (ami a single.html tartalmát kapta meg) átalakítása már nagyon hasonlóan működik. A **<head>** tag-ben a **@vite** direktívát kell ugyanúgy elhelyezni, a <body> részben pedig a beemeléseket az include fájlokból kell megfelelően elvégezni. Ezután már csak egy útvonalat kell létrehozni a **routes / web.php** fájlban:

```
Route::get('/post', function () {
    return view('post');
});
```

4–33. kódrészlet: Új post útvonal létrehozása teszteléshez

A fentiek végrehajtását már mindenkire rábízom, hogy elvégezze önállóan, de a fejezetet összefoglaló GitHub commit-ben ez is benne lesz, ha valaki elakadna vele.



4–16. ábra: A post útvonal és sablon végső kinézete

Természetesen a statikus tartalmakat (szövegeket, képeket stb.), amelyek jelenleg vannak a nézet fájlokban, úgy alakítjuk át, ahogy mi azt szeretnénk, hiszen ezek csak példaként vannak bennük elhelyezve. Ezzel megvalósítottunk egy komplex sablon integrációt, a webes alkalmazásunk pedig készen áll arra frontend oldalról nézve, hogy beindítsunk egy blogot (vagy ha más típusú sablont integráltunk, akkor egyéb más weboldalt).

4.5.3.7. Kiegészítés: új külső csomagok használata

Előfordulhat, hogy az új sablon számos egyéb kliens oldali csomagot is tartalmaz, mint ami a mi projektünkben alapból használatba került. Ilyenkor érdemes ellenőrizni a sablon **package.json** fájlját és ellenőrizni, hogy milyen eltérő csomagokat használ az új sablon. A csomagok nevét a saját projektünk **package.json** fájljába is át kell írni (be kell szúrni), majd következhet egy `npm install` parancs, amivel a saját projektünk **node_modules** mappájába is telepítjük ezt az új csomagot.

4.6. Automatikus tesztelés (nézetek)

A nézetek tesztelése szorosan összefügg az útvonalak tesztelésével (3.7.3). Abban az alfejezetben már kipróbáltuk az **assertSee()** metódust. Ezzel kapcsolatban fontos megjegyezni, hogy a rendszer nem *„megnézi"* a weboldalt és azt vizsgálja, hogy látja-e az adott paraméterül kapott szöveget, hanem a lekért útvonal nézetének *forráskódját* vizsgálja meg és abban keresi az adott szöveget. Az **assertSee()** ellentettje az **assertDontSee()**, ami elvárja, hogy a paraméteréül kapott szöveg ne legyen megtalálható az oldal forráskódjában. Hozzunk ehhez létre egy tesztelő osztályt:

```
php artisan make:test ViewTest
```

Az **assertDontSee()** tesztelésére valamilyen olyan szöveget adhatunk át neki, amit biztosan nem szeretnénk látni az adott nézetben. Ekkor ugye kiesnek azok az elemek, amelyek a szerkezetet adják és a legtöbb nézetben ugyanazok (menü, fejléc, lábléc stb.). Egy lehetséges megoldás, hogy a kezdőoldalon

nem szeretnénk látni a jelenlegi kapcsolat oldal elemeit. Adjuk hozzá az új **ViewTest.php** fájlhoz a következő metódust:

```
public function test_does_not_contain_contact_element(): void
{
  $response = $this->get('/');
  $response->assertDontSee('Ez egy kapcsolati oldal.');
}
```

4–34. kódrészlet: assertDontSee() használata

A teszt futtatásával pozitív eredményt kell kapnunk:

```
php artisan test tests/Feature/ViewTest.php
```

Kipróbáltuk még az útvonalas teszteknél a **assertViewIs()** metódust is, amely a névkonvenciót betartva (a .blade.php kiterjesztés nélkül) hivatkozik a nézetre magára, hogy tényleg azt adja-e vissza az útvonal lekérésre.

Végül nézzük meg két teszt és az **assertViewHas()** alkalmazásának segítségével, hogy az adott nézetek megkapják-e az adatokat, amelyeket küldenünk kell nekik.

```
public function test_view_contains_data()
{
  $response = $this->get('/');
  $response->assertViewHas('name');

  $response = $this->get('/pass-array');
  $response->assertViewHas('tasks');
}
```

4–35. kódrészlet: assertViewHas() használata

A teszt futtatása után megkapjuk eredményül, hogy a kezdőoldal (**/**) megkapja a **name** adatot (az útvonalnál az átadott asszociatív kulcs eleme), a **/pass-array** útvonalnak pedig elküldésre került a **tasks** tömb. Ha nem hinnénk a tesztek lefutásának, akkor direkt el is ronthatjuk ezeket az értékeket és a teszt futtatása rögtön jelzi, hogy hiányoznak az „elrontott" nevű adatok, tehát működnek a tesztjeink.

Az alfejezetben végrehajtott programkód-módosítások ebben a GitHub commit-ben érhetők el.

Tipp: Teszteld a tudásod!

Bár még nem tudunk mindent a nézetekről, de érdekes lehet kipróbálni az eddigi tudásunkkal, milyen teszteknek tudnánk megfelelni. Léteznek nyilvános repo-k, amelyekkel ezt a tesztelést meg tudjuk tenni, itt van például egy, ami az Blade sablon motor alapjaihoz tartozik:

https://github.com/LaravelDaily/Test-Laravel-Blade-Basics

Ha klónozzuk a projektet és elindítjuk a tesztelést, akkor kezdetben minden tesztünk elbukik. De a GitHub oldalon lévő Task lista (a könyvtár- és fájlstruktúra alatt) pontosan megmutatja, hogy milyen fájlokat kell szerkeszteni ahhoz, hogy átmenjenek a tesztjeink.

Még nem tanultunk mindenről, így azokat nem kötelező figyelembe venni, amiről még nem tudunk, a saját tudásunk felmérésénél. Viszont kialakulhatott már annyi rutinunk, hogy esetleg a Laravel dokumentáció megfelelő részét használva is megoldható az, hogy átmenjenek sikeresen a fejlesztéseink az alkalmazás nézeteinek tesztesetein.

4.7. Összegzés

Bár a webes frontend fejlesztés önmagában is egy hatalmas témakör és egy külön szakma, ha valaki ért hozzá. A Laravel kapcsán szerettem volna egy áttekintést adni, hogy mit hogyan érdemes használni a kliens oldal fejlesztése során.

A fejezet során áttekintettük a Laravel keretrendszer kliens (frontend) oldali elemeit. Kétféle technikával is építettük nézeteinket: fentről lefelé (helyőrzők és szekciók) és lentről felfelé (komponensek segítségével). A két technika együtt, egymást kiegészítve is használható. Két sablont is felépítettünk a technikák alkalmazásával. A továbbiakban a *„bonyolultabb"* (Future Imperfect) sablont integráló projektet használjuk arra, hogy bemutassak összetettebb dolgokat és felépítsünk egy teljesértékű webes alkalmazást, egy blogot.

A fejezet utolsó részében két olyan folyamatot, elemet is bemutattam, amelyek egy nagyobb webes alkalmazásnál elengedhetetlenek a kliens oldal szempontjából: többnyelvűsítés és a nézetek tesztelése.

Természetesen itt nem ér még véget a nézetekkel való ismerkedésünk, hiszen a következő fejezetekben is mindig elő fog kerülni az, hogy használjuk a nézeteket a felhasználókkal való kapcsolattartásra.

Végül egy jó tanács a folytatás előtt: bár a Laravel keretrendszer maga leginkább a szerver oldali fejlesztést támogatja, de a felhasználóink először a webes alkalmazásunk kinézetével fognak találkozni és ha az nem kellően megnyerő, használható, akkor utána őket már nem annyira fogja érdekelni, hogy mit is rejt a rendszer magja. Emiatt sem érdemes elhanyagolni az alkalmazásunk kliens oldalát, főleg mivel, ahogy a példák is mutatták, a projektjeinkbe könnyedén tudjuk integrálni az elérhető és később testre szabható sablonokat.

5. Adatbázis-hozzáférés (Database connection and access)

Az alkalmazások jelentős része adatokra épül, a webes világban sincs ez másképp. Emiatt is nagyon fontos, hogy tisztában legyünk a Laravel keretrendszer adatbázis-kezelőkhöz való kapcsolódási lehetőségeivel. A fejezetben több különböző adatbázis-kezelőhöz is csatlakozunk a webes alkalmazásainkból: valamit alapértelmezetten (MySQL, SQLite) és nagyon hatékonyan támogat a Laravel, de van olyan is (Microsoft SQL Server), ahol egy kicsit foglalkoznunk kell a háttérben megtalálható driver-ekkel is azért, hogy a kapcsolódás zökkenőmentes legyen.

A fejezet második részében a kapcsolódáson túl már tényleges adattábla-manipuláló műveleteket is végrehajtunk. Ezzel megalapozzuk a további fejezetek összetett adatkezelési eljárásait.

5.1. Kapcsolat létrehozása különböző adatbázis-kezelőkhöz

Mindegyik alfejezetben egy adatbázist hozunk létre, majd webes oldalról csatlakozunk rá, adatokat mentünk ki rá, és le is kérjük az adatokat belőlük. Az adatbázisoknál alkalmazzunk egy névkonvenciót, amelyet a könyv feldolgozása során magunkra nézve kötelezőnek tekintünk. Minden új adatbázis nevének megadásánál használjunk előtagot és utótagot is, vagy más néven prefixet (**l10_**) és postfixet (**_db**) is. Így a későbbiekben, ha már sok fejlesztést végrehajtottunk, akkor tudni fogjuk, hogy ezek az adatbázisok a könyv feldolgozásához kellettek.

5.1.1. Kapcsolódás a MySQL adatbázis-kezelő szerverhez

A MySQL relációs adatbázis-kezelő rendszer webfejlesztői körökben rendkívül nagy népszerűségnek örvendett mindig és jelenleg is, annak ellenére, hogy már nem közösségi fejlesztésű, hanem az Oracle cég tulajdona 2010 óta. A használata – bizonyos korlátozásokkal – továbbra is ingyenes és a XAMPP rendszer részeként települt is a számítógépünkre, ha követtük a futtatókörnyezetet a 2.1.1. alfejezet alapján.

Maradunk tehát az **l10-components** projektünknél. Az adatbázis-kezelő rendszer eléréséhez az **.env** fájl tartalmaz információt. Ez a fájl a projekt mappánk gyökerében található meg és a neve az angol *„environment”* szóra utal, ami a környezetet jelenti. Ennek a fájlnak a segítségével tudunk *„kulcs=érték”* párok segítségével környezeti beállításokat elhelyezni az alkalmazásunkban. Például itt tudjuk megadni az alkalmazásunk nevét, URL-jét, az adatbáziskapcsolódási információkat, a levelezési paramétereket stb. Ezek a beállítási paraméterek a **config** mappában lévő fájlokban is megtalálhatók, alapértelmezett értékeikkel együtt. Nekünk viszont lehetőségünk van az **.env** fájl segítségével felülírni az alapértelmezett értékeket. Ha ezt az **.env** fájlt és a beállításait módosítjuk, akkor gyakorlatilag egy API-n keresztül tudjuk módosítani az adatbázis kapcsolódás paramétereit, mindenféle plusz programozás nélkül.

Laravel 11: új **config** mappa tartalom.

A Laravel 11-es verziójú keretrendszerben a **config** mappa fájljait a fejlesztői változatokban még teljesen eltüntették, alapértelmezetten nem látszódtak, nem kerültek be a **config** mappába, ha pedig módosítani szerettük volna valamelyik benne lévő beállítást, akkor publikálnunk kellett a mappa fájljait. A publikálás itt annyit jelent, hogy

a **vendor** mappában lévő keretrendszer elemek közül a kiválasztottak átkerülnek (átmásolódnak) a **config** mappánkba.

`php artisan config:publish`

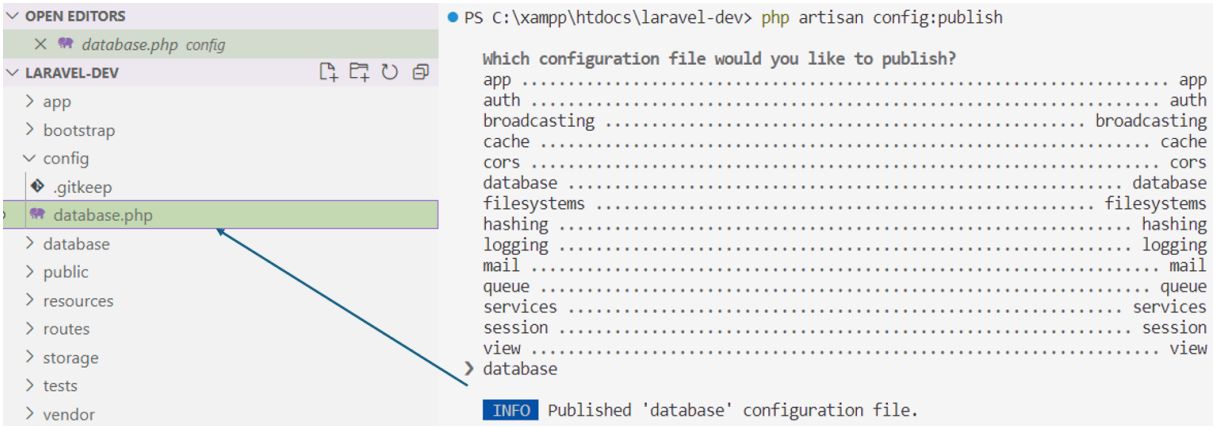
Az 5–1. ábra azt mutatja, hogy a database-hez tartozó beállítási fájlt publikáltuk, így az létre is jött a **config** mappánkban. Ha tudjuk, hogy melyik config fájlt szeretnénk publikálni, és nem a listából akarjuk kiválasztani, akkor meg is adhatjuk neki paraméterként már a parancsban:

`php artisan config:publish database`

De, ezek a változások ugyanakkor még nem voltak véglegesek a fejlesztői verziókban, a Laravel fejlesztői úgy döntöttek a visszajelzéseket meghallgatva, hogy újra automatikusan elérhetővé teszik a **config** mappában lévő fájlok többségét.

A végső Laravel 11-ben pedig ténylegesen újra elérhetővé tették a config fájlok többségét, úgy, mint például az **app.php**-t, **auth.php**-t, **database.php**-t stb. Amelyik beállítási fájl még mindig hiányozna nekünk, azt az imént leírt módon lehet publikálni a **config** mappába.

5–1. újdonság: config mappa tartalma



5–1. ábra: A config mappa kezdetben üres, ahova aztán egyesével publikálhatjuk a beállítási fájlokat

De visszatérve a Laravel 10-hez, ha az alkalmazásunk éppen kiszolgálás alatt van a `php artisan serve` utasítás hatására, és módosítjuk az **.env** fájl tartalmát majd elmentjük, akkor az alkalmazás kiszolgálását végző szerver a terminal-ban automatikusan újra fog indulni, és az új beállítások lépnek rögtön életbe.

Az **.env** fájl tartalmán belül most az adatbázis kapcsolódási beállításokra fogunk koncentrálni. Ezek **DB_**-vel kezdődnek a beállítási paramétereknél. Ezek a beállítási paraméterek a következők:

```
DB_CONNECTION=mysql
DB_HOST=127.0.0.1
DB_PORT=3306
DB_DATABASE= l10_blog_db
DB_USERNAME=root
```

```
DB_PASSWORD=
```

5–1. kódrészlet: MySQL adatbázis kapcsolódás alapértelmezett beállításai az .env fájlban

Ezzel beállítjuk a Laravel alkalmazásunkban, hogy egy MySQL adatbázis-kezelőhöz szeretnénk csatlakozni, tehát használja a **mysql driver**-t ehhez (automatikusan működik, nem kell hozzá semmi további dolgot megtennünk). A **DB_HOST** az adatbázis-kezelő szerver IP címe, ami jelen esetben a saját gép vagy **localhost**. A **DB_PORT** a **mysql** alapértelmezett port-ja, amelyen kommunikálhatunk vele. A DB_DATABASE az adatbázis neve, aminek léteznie kell az adatbázis-kezelő szerveren (a Laravel projektünkben ez telepítéskor beállításra kerül automatikusan ugyanarra a névre, ami a projekt neve is). Ez az adatbázis jelenleg még nem létezik, de nemsokára, létre fogjuk hozni. Viszont megjegyezhetjük, hogy bár az adatbázis nem létezik és jelenleg még az adatbázis-kezelő szerverhez való kapcsolódás is kétséges, ennek ellenére az alkalmazásunkat tudtuk futtatni hibamentesen. Mindez azért van, mert eddig nem használtunk benne adatbázis specifikus részeket, így nem is okozhatott ez a kapcsolódási feladat problémát. A **DB_USERNAME** és **DB_PASSWORD** az alapértelmezett adatbázis-kezelő szerver eléréséhez szükséges felhasználónév és jelszó páros. Ez utóbbiak kapcsán arra figyeljünk, hogy a **root** elérése csak a helyi gépen működik alapértelmezetten, ha már egy másik gépről / gépre és annak adatbázis-kezelőjére akarunk csatlakozni ezzel a felhasználóval, az már nem fog ilyen könnyedén működni.

Ezután nézzünk be egy kicsit a *„színfalak mögé”,* és megvizsgáljuk, hogy honnan jönnek ezek a (jelen esetben **DB_**-vel kezdődő) attribútumok és értékei! Ahogy említettem, ehhez a **config** mappát kell kinyitnunk és az abban lévő fájlokat érdemes megvizsgálni. A **database.php** fájl, akárcsak a többi **config** mappában lévő fájl, szerencsére tele van kommentekkel, amelyek elmagyarázzák az egyes sorok működését. A fájlban számos helyen láthatjuk az **env()** segédmetódust. Ez megpróbál hozzáférni az **.env** fájlhoz, annak is a metódus szerinti első paraméteréhez. Ha ez definiálva van az **.env** fájlban, akkor rendben van és onnan fogja venni az általunk beállított értéket. Ha nincs olyan attribútum az **.env** fájlban, akkor az **env()** metódus második paraméterét veszi alapértelmezettnek, amit beállít az első paraméter szerinti változóba.

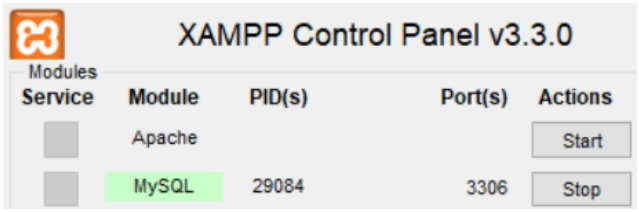
```
'default' => env('DB_CONNECTION', 'mysql'),
```

5–2. kódrészlet: Példakód egy beállításra a config / database.php fájlból

Itt például látszódik, hogy az **env()** segédmetódus megpróbálja lekérni a **DB_CONNECTION** változó értékét az **.env** fájlból és ha ott nem definiáltuk volna, akkor beállítja hozzá alapértelmezetten a **mysql** értéket.

Ha legörgetünk a fájlban, akkor láthatjuk a **'connections'** beágyazott tömb elemei között, hogy többféle adatbázis kapcsolatot vagyunk képesek definiálni az alkalmazásunkhoz (sorrendben itt megtalálhatóak: a **sqlite** – SQLite, a **mysql** – MySQL, a **pgsql** – PostgreSQL, az **sqlsrv** – Microsoft SQL Server). Ebben az alfejezetben a MySQL adatbázis-kezelő szervert használjuk a lehetőségek közül.

Indítsuk el (Start) a MySQL adatbázis-kezelő szerver kiszolgálót a XAMPP Control Panel segítségével:



5–2. ábra: MySQL adatbázis-kezelő szerver kiszolgáló elindult és fut

Ha zöld háttérszíne lesz a *„MySQL"* feliratnak, akkor problémamentesen fut és a 3306-os porton figyel alapértelmezetten. Indítsuk el a MySQL Workbench adatbázismenedzser alkalmazást. Csatlakozzunk egy kattintással a XAMPP-os adatbázis-kezelőhöz! Majd kattintsunk a fenti ikonok közül a legelsőre, amivel egy új SQL utasítást tudunk írni a szerkesztési felületre.

```
CREATE DATABASE l10_blog_db;
```

5–3. kódrészlet: l10_blog_db adatbázist létrehozó SQL utasítás

Az utasítás felett lévő villám ikon megnyomásával végrehajtható az utasítás és létrejön az adatbázis, ami a Navigator panelen a *„Schemas"* lapfülön látszódik is, ha megnyomjuk a *„Schemas"* lapon jobb felül a frissítési gombot. Az adatbázis kezdetben üres, nem tartalmaz semmilyen adattáblát, de ha jól állítottuk be az **.env** fájlban a kapcsolódási paraméter értékeket, akkor hozzá fogunk tudni férni ehhez az adatbázishoz (és létre fogunk tudni hozni adattáblákat a Laravel webes alkalmazásunk segítségével).

```
php artisan db:show
```

Az utasítás kiadásának hatására kapunk egy kérdést a terminal-ban, miszerint a projektünknek szüksége van a Doctrine DBAL csomagra ahhoz, hogy az adatbázis-kezelőhöz való kapcsolódást, magát a kapcsolat létre tudja hozni. Írjuk be, hogy *yes* és üssünk Enter-t. Ekkor elvileg hiba nélkül települni fog a csomag. De ha mégsem települne, mert a Laravel 10-ben előfordulhat, hogy ez a csomag néhány másik csomaggal való *„konfliktusa"* miatt nem sikerülhet automatikusan, akkor a következő utasítással mi magunk is telepíthetjük a csomagot, így nem állhat fenn ütközés:

```
composer require doctrine/dbal
```

Laravel 11: az új alkalmazások már nem függenek a **Doctrine DBAL** csomagtól.

Az egyéni Doctrines típusok regisztrálása már nem szükséges a különböző oszloptípusok megfelelő létrehozásához és módosításához, ezekhez korábban egyéni típusokra volt szükség.

5–2. újdonság: Csomagfüggőségi változás (Doctrine DBAL)

Miután települt a csomag, újra ki kell adnunk a fenti utasítást, hogy ellenőrizzük az adatbáziskapcsolat létezésének helyességét.

```
PS C:\xampp\htdocs\l10-components> php artisan db:show

  ..........................................................................................................
  Database ..................................................................................... l10_blog_db
  Host ......................................................................................... 127.0.0.1
  Port ............................................................................................ 3306
  Username ..................................................................................... root
```

5–3. ábra: Beállított paraméterek szerinti adatbáziskiszolgálóhoz kapcsolódás működik

Táblánk még nincsen benne, de a kapcsolat élő, és létre tudott jönni a beállítások segítségével. Ha most lekapcsoljuk a XAMPP Control Panel-en a *„Stop"* gombbal a MySQL kiszolgálót, majd újra kiadjuk a fenti db:show utasítást, akkor hibát fogunk kapni:

```
PS C:\xampp\htdocs\l10-components> php artisan db:show

  PDOException

  SQLSTATE[HY000] [2002] No connection could be made because the target machine actively refused it

  at vendor\laravel\framework\src\Illuminate\Database\Connectors\Connector.php:65
     61 |      * @return \PDO
     62 |      */
     63 |     protected function createPdoConnection($dsn, $username, $password, $options)
     64 |     {
  ➜  65 |         return new PDO($dsn, $username, $password, $options);
     66 |     }
     67 |
     68 |     /**
     69 |      * Handle an exception that occurred during connect execution.

  1   vendor\laravel\framework\src\Illuminate\Database\Connectors\Connector.php:65
      PDO::__construct("mysql:host=127.0.0.1;port=3306;dbname=l10_blog_db", "root", Object(SensitiveParameterValue)
, [])
```

5–4. ábra: Beállított paraméterek szerinti adatbáziskiszolgálóhoz kapcsolódás nem működik

Itt az SQLSTATE-ben jelzett hibaüzenet azt írja, hogy a kapcsolódási paraméterek szerinti adatbázis-kezelő szerver elutasította a kapcsolatfelvételt, mivel ugye leállítottuk korábban. Ha visszakapcsoljuk és újra lekérjük az információkat az utasítással, akkor már újra jó lesz. Ezzel tehát teszteltük a kapcsolódást a Laravel alapú webes alkalmazásunkból a MySQL adatbáziskiszolgáló felé.

5.1.1.1. Verziókövetési kiegészítés

Az alfejezetben végrehajtott módosítások csak az **.env** fájlt érintették. Ez azonban egy olyan fájl, amely környezet-specifikus, tehát minden programozónál más-más értékeket tartalmazhat, emiatt ezt a projekt gyökerében található **.gitignore** fájl tartalmazza, így ezt nem *„verziókövetjük"*, nem fog feltöltésre kerülni a GitHub-ra. Hiszen könnyedén előfordulhat, hogy nálad, Olvasónál más környezeti beállítások vezetnek például az adatbázis-kezelő rendszerhez. Ekkor egy tipikusan alkalmazott módszer szokott lenni, ha valaki klónozni szeretné a projektünket, a GitHub repo-nkat, hogy az **.env.example** fájl tartalmát lemásolja és beilleszti az újonnan nála létrehozott **.env** fájlba, majd elvégzi a beállítási paraméterek megadását a saját környezeti specialitásait figyelembe véve.

A verziókövetésről még annyit, hogy az adatbázis-kezeléshez telepítettük a **doctrine/dbal** csomagot, ami be is került a **vendor** mappába (a saját függőségeivel együtt: **doctrine/cache**, **doctrine/deprecations**, **doctrine/event-manager**, **psr/cache**). A **vendor** mappa tartalmazza a projektünk, a Laravel webes alkalmazásunk szerver oldali függőségeit, csomagjait, amelyek a működést támogatják. Ezt a **vendor**

mappát szintén tartalmazza a **.gitignore** fájl, mivel ez a **vendor** mappa meglehetősen nagyra tud nőni (jelenleg 47,9 MB) és egy *„recept"* alapján bármikor, bárki gépén újra felépíthető, ezért nincs szükség arra, hogy például a GitHub-on eltároljuk és verziókövessük. A *„receptre"* viszont szükség van a verziókövető rendszerben, ez a **composer.lock** fájl (vagy a **composer.json**, ha a **.lock** nem létezik), ami tartalmazza, hogy milyen csomagokat telepítettünk az alkalmazásunk működéséhez. Ezért, ha valaki klónozni szeretné a webes alkalmazásunkat, akkor miután létrejött nála a projektünk mappája, le kell futtatnia a következő utasítást:

```
composer install
```

Ezáltal a **composer.lock** (vagy a **composer.json**, ha a **.lock** nem létezik) receptúra alapján fel fogja építeni a klónozott repó-ban is a **vendor** mappát, és benne a projekt működéséhez szükséges csomagok könyvtárait.

Ha egy meglévő projektünkben akarjuk frissíteni a **vendor** mappában lévő csomagokat, akkor a következő utasítást használjuk:

```
composer update
```

Ez a csomagokat a **composer.json** fájl alapján fel fogja frissíteni, majd utána frissíti a **composer.lock** fájlt is.

Emiatt az alfejezetben végrehajtott programkód-módosítások csak a **composer.lock** és **composer.json** fájlokat tartalmazzák, ezek ebben a GitHub commit-ben érhetők el.

5.1.2. Kapcsolódás az SQLite fájl alapú adatbázis-kezelőhöz

Az SQLite egy kicsi, gyors, magas rendelkezésre állású, teljeskörű, fájl-alapú relációs adatbázis-kezelő rendszer. Az egyik leggyakrabban használt adatbázis-kezelő motor a világon, ami leginkább abból adódik, hogy az okostelefonok belső adatbázisait ez hajtja meg.

Maga az SQLite fájl formátum egy stabil, platformfüggetlen és visszafelé kompatibilis megoldás. Mindemellett a forráskódja nyilvános és ingyenes a használata bárki számára. Talán emiatt is, a Laravel egy beépített driver-t tartalmaz, amivel könnyedén el tudja érni az SQLite alapú adatbázis fájlokat, majd kezelni (lekérdezni és manipulálni) tudja őket, ezáltal mi programozók is hatékonyan tudjuk használni ezeket.

Az előző alfejezetben használt MySQL adatbázis-kezelő megoldás is szintén megfelelő a számunkra, azonban gondoljunk bele, hogy mennyire egyszerű egy ilyen SQLite alapú megoldás. Hiszen, ha valakinek készítünk egy webes alkalmazást, ami adatbázisra épül, akkor neki nem kell telepíteni adatbázis kiszolgálót, mivel akár egyetlen fájllal megoldható az adatbázis kezelése.

Ennyi előzetes információ megismerése után, vágjunk bele a tényleges használatába! Ahogy az a **config** mappában lévő **database.php** beállítási fájlban is látszódik, az SQLite használatához van a Laravel-nek egy **driver**-e, amivel könnyedén el tudja érni és menedzselni tudja az ilyen adatbázisokat.

```
'sqlite' => [
  'driver' => 'sqlite',
```

```
    'url' => env('DATABASE_URL'),
    'database' => env('DB_DATABASE', database_path('database.sqlite')),
    'prefix' => '',
    'foreign_key_constraints' => env('DB_FOREIGN_KEYS', true),
],
```

5–4. kódrészlet: SQLite adatbázis elérési beállítások

Látható, hogy itt nem kell majd annyi mindent beállítanunk, mint a MySQL kapcsolat esetén és alapértelmezetten a Laravel projektünk **database** mappájában fog keresni egy **database.sqlite** fájlt a rendszer, hacsak felül nem írjuk mi ezt az attribútumot az **.env** fájlunkban. Nekünk ez első körben megfelelő, úgyhogy hozzuk létre ezt az új, **database.sqlite** nevű fájlt a **database** mappában. Az **.env** fájl adatbázis specifikus részét pedig módosítsuk erre:

```
DB_CONNECTION=sqlite
DB_HOST=127.0.0.1
```

5–5. kódrészlet: SQLite adatbázis kapcsolódás beállításai az .env fájlban

Így az **.env** fájlból kitörölhetjük a nem szükséges adatbázisokat érintő attribútumokat és csak az SQLite specifikus részt hagyjuk meg benne.

A terminal-ban kiadott php artisan db:show utasítás ismét a jó eredménnyel fog visszatérni:

```
PS C:\xampp\htdocs\l10-components> php artisan db:show

  SQLite ..........................................................................................
  Database ..................................... C:\xampp\htdocs\l10-components\database\database.sqlite
  Host ............................................................................................
  Port ............................................................................................
  Username ........................................................................................
  URL .............................................................................................
  Open Connections ................................................................................
  Tables ..................................................................................... 0
```

5–5. ábra: Sikeres kapcsolódás az SQLite adatbázis fájlhoz

Ha esetleg nem bíznánk meg ebben, akkor a **database / database.sqlite** fájl nevét átírhatjuk valami eltérő névre és akkor, ha újra futtatjuk a terminal-ban a parancsot, már hibát fogunk kapni, mivel a rendszer nem fogja megtalálni a keresett helyen a **database.sqlite** fájlt.

A következő utasítással mindig tudjuk ellenőrizni, hogy milyen adatkapcsolataink vannak és azok közül mennyi van használatban az alkalmazás által:

php artisan db:monitor

```
PS C:\xampp\htdocs\l10-components> php artisan db:monitor

  Database name ..................................................................... Connections
  sqlite ..................................................................................... [] OK
```

5–6. ábra: Információ kérés: adatkapcsolatok és számuk

Az adatkapcsolat megvan az SQLite felé, de nincs hozzá nyitott kapcsolat egy darab sem, mivel ténylegesen még nincsen használatban maga az adatbázis.

A **database.sqlite** fájl nevét is alapértelmezetten tartalmazza a **.gitignore** fájl, így nem fogjuk verziókövetéssel ellátni, a későbbiekben majd rávilágítok arra, hogy miért nem (5.2.1. alfejezet).

5.1.3. Kapcsolódás a Microsoft SQL Server adatbázis-kezelőhöz

Főleg nagyvállalati környezetben népszerű a Microsoft SQL Server használata, amely egy robosztus, nagy hatékonyságú, gyors adatbázis-kezelő rendszer. Az imént bemutatott ingyenes megoldásokon túl, ez is egy kiváló választás lehet, főleg, ha tudjuk, hogy nagy adatmennyiségekkel kell majd dolgoznia a Laravel webes alkalmazásunknak.

Megjegyzés: ha tudjuk, hogy nem lesz szükségünk erre az adatbázis-kezelő rendszerre a munkánk során, akkor ez az 5.1.3. alfejezet kihagyható, mert nem épülnek rá logikailag további fejezetek.

5.1.3.1. SQL Server telepítése

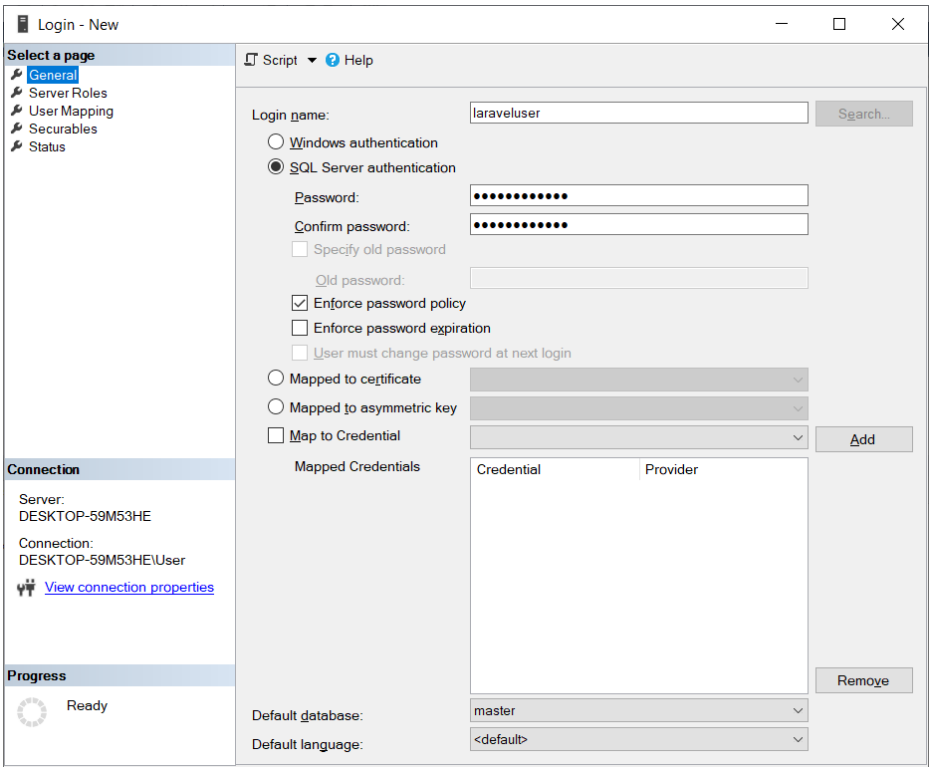
Fejlesztéshez és teszteléshez letölthető hozzá ingyenes verzió, amit mi is tudunk letölteni, telepíteni és használni. Nekünk megfelelő a Developer és az Express verzió is, én az előbbit telepítettem a saját gépemre.

Utána pedig letöltöttem hozzá egy SQL Server Management Studio-t innen, amit utána telepítettem is. Ezzel hatékonyan lehet kezelni az adatbázisokat. Ennek az alfejezetnek nem témája, hogy hogyan is kell Microsoft SQL Server-t és a Management Studio-t telepíteni, úgyhogy ettől eltekintenék és csak arra koncentrálok, hogy a Laravel alkalmazásunkkal, hogy tudjuk majd elérni az SQL Server-en létrehozott adatbázisunkat.

5.1.3.2. Adatbázis létrehozása és hozzáférés beállítása

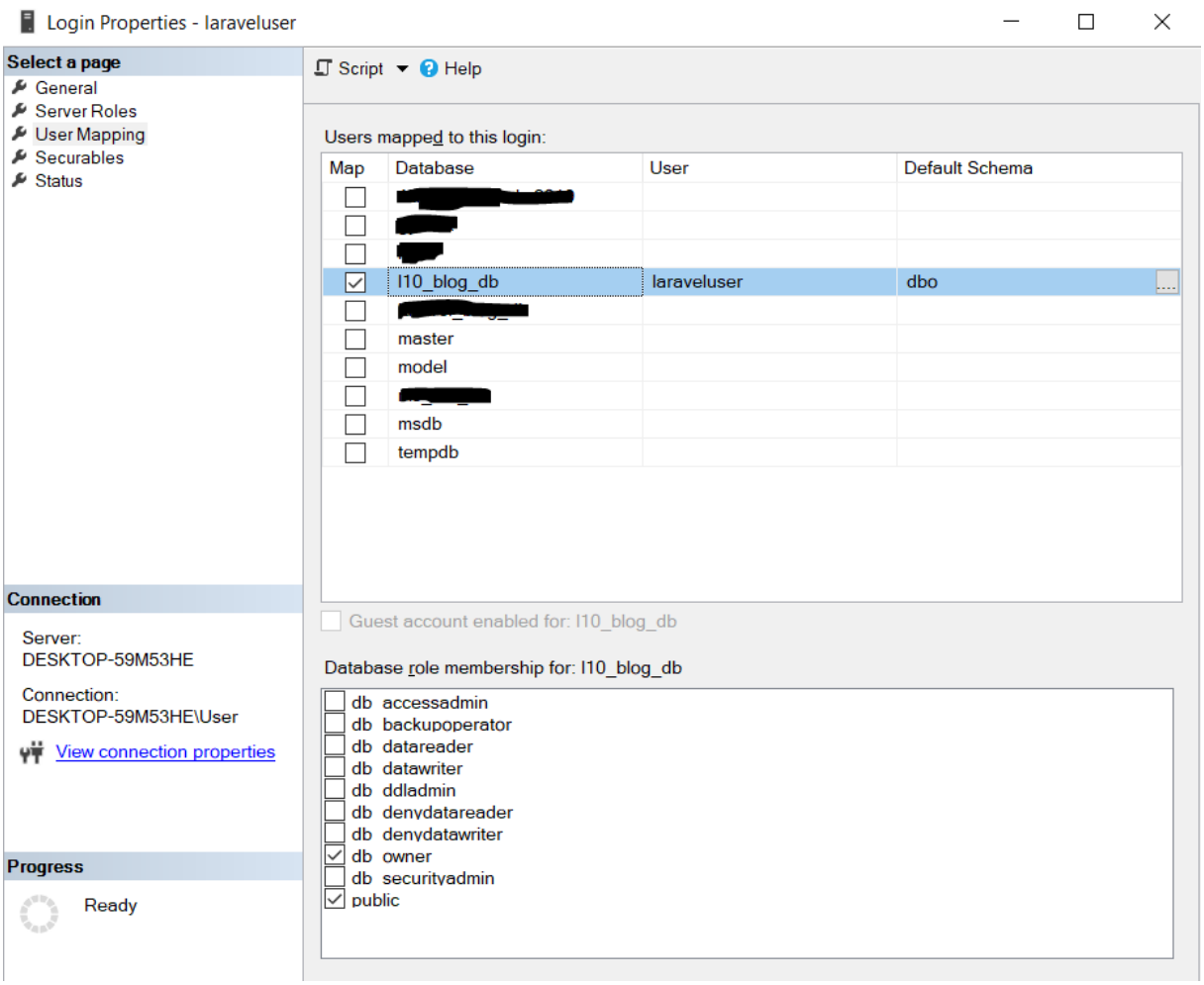
Tegyük is meg ezt a létrehozást: a Management Studio-ban grafikusan is megtehetnénk (jobb egérgomb a *„Databases"* mappán és *„New Database…"*), de én egy *„New Query"* ablakban adom ki ugyanazt (5–3. kódrészlet) utasítást, amit a MySQL-es adatbázis-kezelőben már kiadtam egyszer. Az *„Object Explorer"* panel tetején nyomjuk meg a frissítés gombot, ha nem látszódna az új adatbázisunk.

Az adatbázis létrehozása után fontos kiemelni, hogy mi magunk az SQL Server Management Studio-ban Windows-os felhasználói hitelesítéssel (esetleg **sa** szerver adminisztrátori felhasználóval) csatlakoztunk a szerverhez. Ez azonban a webes alkalmazásunknak nem lesz megfelelő, mivel egy elég nagy biztonsági rés keletkezne ezáltal az adatbázis szerverünkön, ahova be lehetne esetleg törni a webes alkalmazáson keresztül. Ezt mindenképpen el kell kerülnünk, ezért létrehozok egy olyan felhasználót, aki majd tulajdonosa lesz ennek az újonnan létrehozott **l10_blog_db** adatbázisnak. Kattintsunk a *„Security"* fő mappában a *„Logins"* mappára jobb egérgombbal kattintva tudok *„New Login..."*-t kiválasztani. *„Login name"*-nek adjuk meg például azt, hogy **laraveluser**, *„SQL Server authentication"*-t válasszunk a következő részben, majd adjunk meg egy aránylag hosszú jelszót (például: ASDFqwer1234), majd az *„Enforce Password expiration"*-nél vegyük ki a pipát.



5–7. ábra: Új bejelentkezési azonosító beállítása az SQL Server-hez (1. rész)

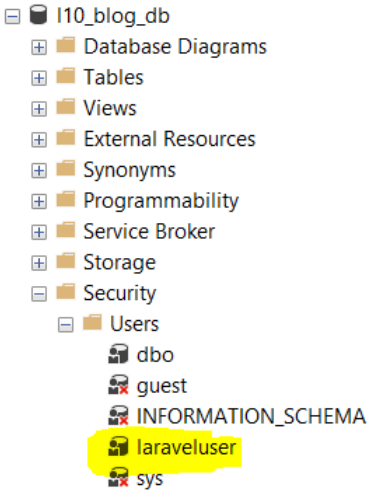
Ezután még ne OK-ézzuk le, hanem bal oldalt a menüben válasszuk ki a *„User Mapping”* menüpontot és felül a Map oszlopban pipáljuk be az új adatbázisunknál a checkbox-ot, alul pedig a **db_owner** szerepkört jelöljük be pipálással.



5–8. ábra: Új bejelentkezési azonosító beállítása az SQL Server-hez (2. rész)

Ezután már leokézhatjuk a párbeszédpanelt és létrejön az új hozzáférés az adatbázis-kezelőhöz és az adatbázishoz, amelyet a webes alkalmazásunk tud majd használni.

Ellenőrzésként megtehetjük, hogy az *„Object Explorer"*-ben a *„Databases"*-t nyissuk le, majd plusszozzuk ki a **l10_blog_db** -t és a *„Security"*-n belül a *„Users"* mappában ott kell lennie az imént hozzárendelt **laraveluser** felhasználónak.

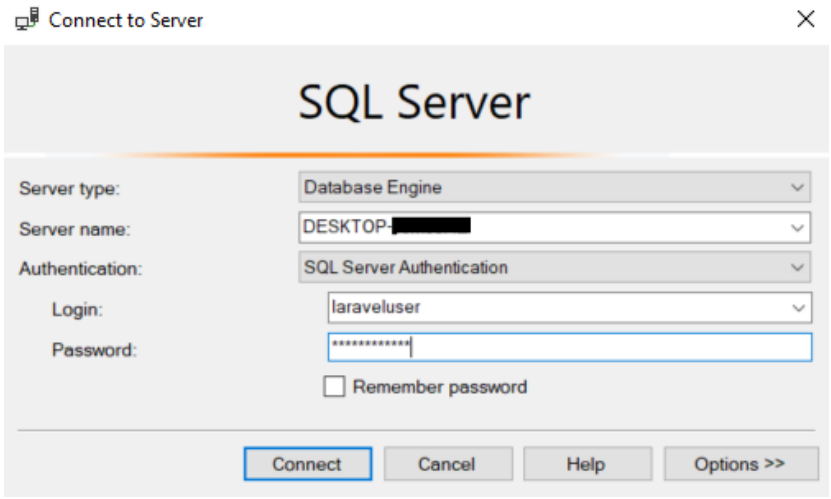


5–9. ábra: Új felhasználó (laraveluser) hozzárendelésre került az új adatbázisunkhoz

Ezzel az adatbázis készen is áll arra, hogy fogadja a kapcsolatfelvételi igényeket, vagyis az adattáblák elérését és menedzselését ezzel az új felhasználóval. Vannak viszont olyan beállítások, amelyeket el kell végeznünk akkor, ha egy PHP alapú alkalmazásból (Laravel) szeretnénk elérni a Microsoft SQL Server-ét.

5.1.3.3. Ellenőrzés és a csatlakozási hiba javításához javaslat

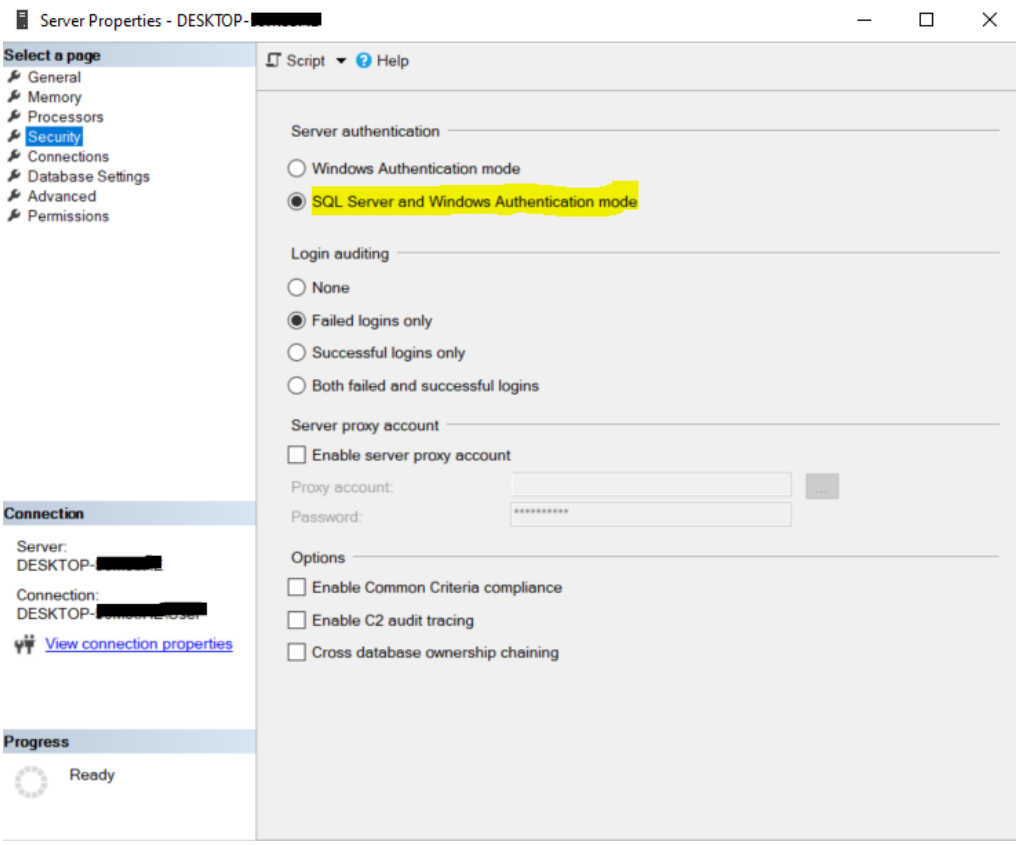
Ellenőrizzük le, hogy az újonnan létrehozott felhasználó hozzáfér-e az SQL Server-hez. A Management Studio-ban a bal oldali *„Object Explorer"* tetején kattintsunk a *„Disconnect"* gombra, majd utána a *„Connect"* gombra. Hajtsuk végre a következő beállításokat:



5–10. ábra: SQL Server-hez kapcsolódás a laraveluser felhasználóval

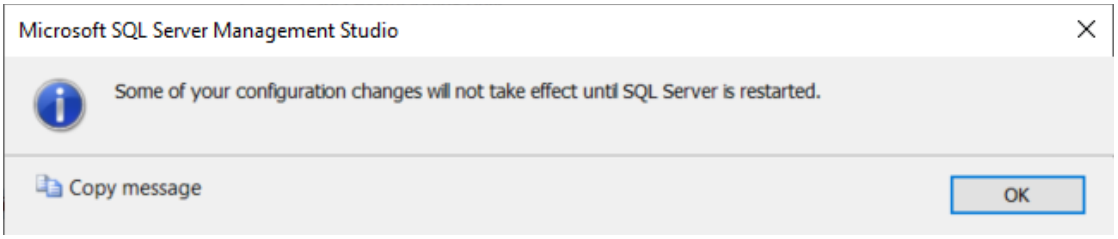
Ha itt esetleg hibát kapunk és nem tudunk csatlakozni, akkor válasszuk ki ismét az *„Authentication"* részben a *„Windows Authentication"*-t és kapcsolódjunk újra a szerverhez.

Utána kattintsunk az *„Object Explorer"* részben a szerver nevére jobb egér gombbal (ha a telepítéskor nem változtattuk meg, akkor *„DESKTOP-"* előtaggal kell kezdődnie. Válasszuk ki a *„Properties"* menüpontot. A felugró kisablakban pedig a bal oldali menüben menjünk rá a *„Security"* menüpontra, a *„Server authentication"* szekcióban pedig az *„SQL Server and Windows Authentication mode"*-ot kell kiválasztani, majd alul *„OK"*.



5–11. ábra: SQL Server csatlakozási módjának átállítása

Ezután kapunk egy figyelmeztető üzenetet, hogy az elvégzett beállítás módosítás csak az SQL Server (nem a Management Studio) újraindítása után fog érvényre jutni.



5–12. ábra: SQL Server újraindítására figyelmeztető üzenet

Újraindítás után próbáljunk meg újra csatlakozni a **laraveluser** felhasználónkkal és remélhetőleg ezután már sikeres lesz (5–10. ábra).

5.1.3.4. SQL Server specifikus PHP kiterjesztések telepítése

Az én fejlesztő és futtatókörnyezetem XAMPP alapú és ha követted az iránymutatásaimat, akkor nálad is ez a helyzet. Előfordulhat persze, hogy más a környezeted, de PHP-ra akkor is szükséged van, egyszóval nálad is meg kell lennie a PHP fordítónak és a beállításainak… de én maradok a XAMPP-nál és aszerint fogom elmagyarázni, hogy milyen beállításokra, módosításokra van szükség ahhoz, hogy Microsoft SQL Server-t tudj használni a Laravel projektedben.

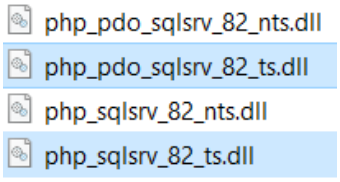
Először derítsd ki mindenképpen, hogy milyen verziószámú a PHP-d: ezt a terminal-ba bárhol beírhatod és visszaadja:

```
php -v
```

Nyissuk meg ezt a weboldalt: https://github.com/Microsoft/msphpsql/releases

A legfrissebb kiterjesztéseket tartalmazza a PHP-hoz, hogy tudjunk csatlakozni az SQL Server-hez. A bejegyzések tartalmaznak mindig egy *„Assets"* részt, amelyet bontsunk ki és tegyünk láthatóvá (*„Show all …"*). Keressük meg a *„Windows"-os* csomagokat és a PHP verziónknak megfelelő zip fájlt töltsük le. (Nálam ez most a 8.2.) *Megjegyzés:* elvileg jó, hogy ha mindig a legfrissebb verziószámú csomagokat használjuk, de a legfrissebbeknél van a legnagyobb esély arra, hogy okozhatnak valamilyen verzióproblémát, ha más csomagokkal akarunk együttműködni velük.

Letöltöm a **Windows-8.2.zip** fájlt. Kicsomagolás után találok benne két mappát, amely az x64 és x86 processzor verziókra utal. Haladjunk tovább abba a mappába, amely a mi saját rendszerünket jellemzi. Mivel nekem 64 bites operációs rendszerem van, ezért a zip fájlban azt a mappát választom ki.



5–13. ábra: PHP verziónak megfelelő csomag tartalma és releváns részei

Ez a négy fájl van a mappában. Nekünk arra van szükségünk, amelynél a fájlnév *"_ts"*-re végződik a .dll kiterjesztés előtt. Ezt a két fájlt kell átmásolni a **C:\xampp\php\ext** mappába.

Ha ez megvan, akkor elvégezhetjük a további beállításokat. Nyissuk meg szövegszerkesztővel a **C:\xampp\php** mappában a **php.ini** nevű fájlt. Az ini fájlban a **;** (pontosvesszővel) kezdődő sorok megjegyzések. Keressük meg a *"Dynamic Extensions"* szekciót, nálam a 892. sorban van, tehát valahol akörül érdemes keresni. Ennek a szekciónak a végére, de még a *"Module Settings"* szekció elé kell létrehozni ennek a két új fájlnak, kiterjesztésének a beemelését:

extension=php_pdo_sqlsrv_82_ts.dll

extension=php_sqlsrv_82_ts.dll

Ha ezt megcsináltuk, akkor készen is vagyunk a telepítési résszel.

5.1.3.5. Kapcsolódás az SQL Server-hez

Nincs más hátra, mint visszatérni a Laravel projektünkhöz, annak is az **.env** fájljához és beállítani az MS SQL Server-nek megfelelő kapcsolódási paramétereket.

```
DB_CONNECTION=sqlsrv
DB_HOST=127.0.0.1
DB_PORT=1433
DB_DATABASE=l10_blog_db
DB_USERNAME=laraveluser
DB_PASSWORD=ASDFqwer1234
```

5–6. kódrészlet: Microsoft SQL Server adatbázis kapcsolódás alapértelmezett beállításai az .env fájlban

Nincs más hátra, mint futtatni a már *„szokásos"* utasításunkat:

php artisan db:show

```
PS C:\xampp\htdocs\l10-components> php artisan db:show

  SQL Server 2012 ..........................................................................
  Database ..................................................................... l10_blog_db
  Host ........................................................................... 127.0.0.1
  Port ..................................................................................... 1433
  Username ..................................................................... laraveluser
  URL ......................................................................................
  Open Connections ..............................................................................  1
  Tables ..............................................................................  0
```

5–14. ábra: Sikeres kapcsolódás a Microsoft SQL Server adatbázisához

A kapcsolódás sikeres. Ha valamelyik beállítást *„direkt elrontjuk"* és úgy próbáljuk futtatni a fenti parancsot, akkor hibát fogunk kapni a terminal-ban.

Megjegyzések:

1. Egy hallgatónál azt tapasztaltuk, hogy az **1433**-as **port**-on keresztül nem működött a csatlakozás, mivel egy másik alkalmazás már lefoglalta magának korábban ezt a portot, így az **.env** fájlban az 1434-es port-ot használva tudtunk csatlakozni az SQL Server-hez.
2. Még a PostgreSQL adatbázis-kezelő volna hátra, de az nagyon hasonlóan használható, mint a MySQL, így annak megismerését meghagynám az Olvasóra, ha szüksége lenne rá.

Tipp: a Microsoft Azure felhőszolgáltatás is biztosít flexibilis MySQL-es adatbázis-kezelő szervert (2023-ban 1 évig ingyenesen használható). Így akár azt is használhatjuk adataink eltárolására egy-egy féléves feladat, vagy Szakdolgozat készítéséhez.

Az Azure felhőbeli adattárolásról itt írtam a blogomon.

5.2. Adatbázis-kezelés a webalkalmazással

Az előző alfejezetben három különböző adatbázis-kezelőhöz is sikeresen létesítettünk kapcsolatot, most már csak az irányt kell kiválasztanunk, hogy melyiket alkalmazzuk a jövőben. Az én választásom a MySQL adatbázis-kezelőre esett, miután különböző szempontokat számításba vettem (ingyenesség, hatékonyság stb.).

A Laravel keretrendszer úgy segít nekünk, hogy gyakorlatilag (kisebb módosításokat leszámítva) teljesen mindegy, hogy milyen adatbázis-kezelőt használunk az alkalmazásunk mögötti adatrétegben, azt elfedi nekünk egy adatkezelő réteggel. Így az adatkezelő réteg lesz az, ami *„lefordítja"* a mi Laravel-ben írt programozott utasításainkat az adatbázis-kezelőnek, illetve a neki értelmezhető SQL nyelvre.

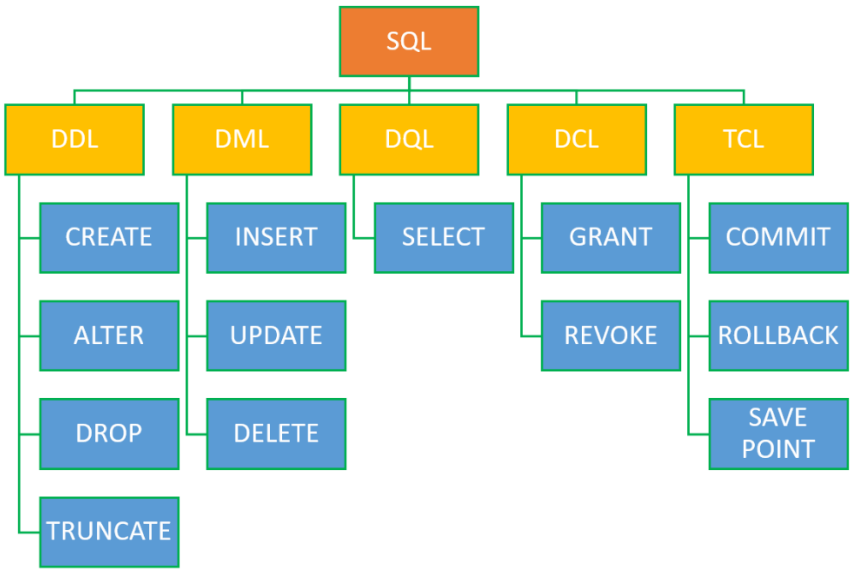
Itt most a sikeres kapcsolatfelvétel után az adattáblák létrehozására, törlésére és módosításaira koncentrálunk.

5.2.1. Bevezetés a migrációk világába

A migrációs fájlok arra szolgálnak, hogy az adatbázisunk és adattábláink struktúráját menedzseljük. Az SQL nyelvet az adatbázisok menedzselésére találták ki, ez a nyelv öt fő részből épül fel:

1. SQL **DDL** (Data Definition Language = Adat Definíciós Nyelv): adattáblák, oszlopok létrehozása, módosítása, törlése stb. Ennek megvalósításában segítenek a **migrációs fájlok** a Laravel-ben. További funkcióiról ebben az alfejezetben lehet olvasni.
2. SQL **DML** (Data Manipulation Language = Adat Manipulációs Nyelv): adatok beszúrásával, frissítésével foglalkozik egy adattáblában.
3. SQL **DQL** (Data Query Language = Adat Lekérdező Nyelv): adatok különböző módon történő lekérdezésével foglalkozik.
4. SQL **DCL** (Data Control Language = Adatelérést Vezérlő Nyelv): ezzel szabályozhatjuk, vezérelhetjük majd az adatelérések körülményeit és azok végrehajtását.
5. SQL **TCL** (Transaction Control Language = Tranzakció Vezérlő Nyelv): ezzel utasításokat vagy utasítás csomagokat tudunk érvényre juttatni vagy akár vissza is vonatni a rendszerrel.

Az utolsó négy felsorolási elem alkalmazása az **Eloquent ORM (Object-Relational Mapping)** és a **Query Builder** segítségével működik a Laravel-ben. A 5. fejezetben lesz róluk szó részletesen és majd a későbbiek során is folyamatosan alkalmazzuk őket.



5–15. ábra: SQL nyelv részei

A migrációs fájlok használatának legfőbb céljai és funkciói:

- Adattáblák létrehozása, oszlopokkal (mezőkkel)
  - Az oszlopoknak nevet, típust, különböző kényszereket (például: alapértelmezett érték) tudunk beállítani.
  - Adattábla szintű kényszereket (például egy mező külső kulcsát egy másik tábla kulcsára) tudunk beállítani.
- Az adattáblákat és mezőket tudjuk frissíteni, törölni.

- Mindezeket verziókövetéssel látja el nekünk a rendszer: ezáltal képesek leszünk adatstruktúrákat migrálni az adatbázisba és vissza is tudjuk vonni ezeket a migrációkat, ezáltal törlődnek az adattáblák.
- A migrációs fájlok alapján az adatbázis struktúra bárkinél (a saját környezetében) újra előállítható egyetlen parancs kiadásával.

A felsorolásból látható, hogy *a migrációs fájlok a DDL SQL kategóriába tartoznak.*

A felsorolásban felsorolt funkcionalitásokat úgy tudjuk megtenni, hogy egyáltalán nem kell önállóan működő SQL utasításokat írnunk. Ahogy az 5.1. alfejezetben láttuk, ezeket a migrációs fájlokat több különböző adatbázis-kezelő rendszerben tudjuk futtatni. Ez persze nem jelenti azt, hogy most az adatbázisokkal kapcsolatos tudásunkat el is felejthetjük... dehogy! Csak a Laravel keretrendszerbe beágyazottan fogjuk az adatbázis-kezeléssel kapcsolatos műveleteket végrehajtani. Tehát egy kicsit másképp kell gondolkodnunk az adatbázissal és a táblák menedzselésével kapcsolatosan.

Egy másik nagyon nagy előnye ennek azon túl, hogy az adatbázis kezelését migrációs fájlokkal fogjuk megvalósítani, hogy ezt a területet is *verziókezeléssel* támogatja meg a Laravel keretrendszer.

5.2.2. Migrációs fájlok gyakorlati alkalmazása

Egy blog alkalmazás építésébe kezdünk bele, amely elég komplex lesz ahhoz, hogy a Laravel keretrendszer különböző lehetőségeivel ismerkedjünk a projekt fejlesztése során.

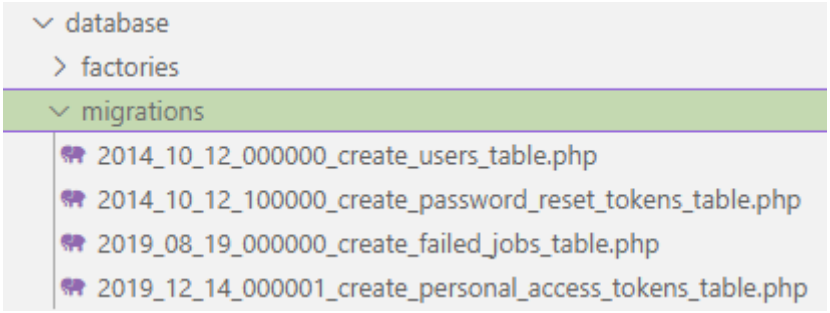
A bloghoz kiválóan alkalmas az **l10_components** projektünk eddigi felépítése és főleg a nézetek szempontjából kialakított szerkezete és sablonja.

5.2.2.1. Adatbázis kapcsolat ellenőrzése

A kapcsolat létrejöttét továbbra is a `php artisan db:show` utasítással tudjuk ellenőrizni (ehhez esetleg ebben a projektben még telepítenünk kell a **doctrine/dbal** csomagot, ha még nem tettük meg korábban).

5.2.2.2. Első migrációs fájlunk: posts table

A címben azt írtam, hogy az első, de ez egy kis csalás volt, mivel a projektünk tartalmaz már alapból migrációs fájlokat. Ezeket a **database / migrations** mappában találhatjuk meg mindig.



5–16. ábra: Migrációs fájlok helye és a kezdeti fájlok a projekt mappájában

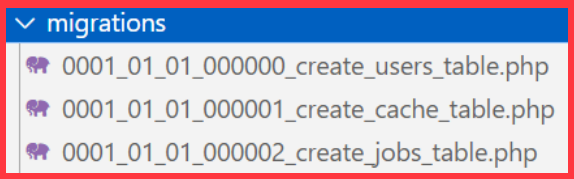
Négy darab migrációs fájlunk már van az említett mappában. Mielőtt az első saját migrációs fájlunkat elkészítenénk, vizsgáljuk meg ezeket a meglévőeket.

A **fájlok nevei** egy *időbélyeggel* (dátum plusz idő) kezdődnek, ez a létrehozásuk dátumát jelzi és egyúttal egy időrendi sorrendet is ad a fájloknak, amivel gyakorlatilag majd a végrehajtásuk (lefuttatásuk) sorrendjét is meghatározzuk. Ha például mi létrehozunk majd egy új migrációs fájlt, akkor az majd 2023-mal kezdődő időbélyeget fog kapni és az ötödik migrációs fájl lesz, amit végrehajt a rendszer az adatbázis migrálás során. Az időbélyegek után is *beszédes neveket* találunk, mivel ezek a migrációs fájlok funkcióira (háttérben rejlő SQL utasításaira) utalnak, például az elsőnél a **create_users_table**, az egy **users** táblát fog létrehozni. Fontos itt még az, hogy a táblák nevei (**users**, **failed_jobs** stb.) mind *többesszámban* vannak (angolul), ez egy Laravel *névkonvenció*, tehát a helyes működés szempontjából számunkra is fontos, hogy betartsuk ezt a szabályt.

Alapértelmezetten létrejövő migrációs fájlok nevében az időbélyeg megváltozott

Korábban egy konkrét dátummal és idővel (lásd az iménti ábrát) kezdődött a migrációs fájlok neve, a Laravel 11-ben viszont *„újradátumozták"* őket, így a konkrét dátumtól függetlenül az alapértelmezetten meglévők végrehajtásával indul mindig a migrálás.

Az új fájlok neve ez lett:



5–17. ábra: Alapértelmezetten létrejövő migrációs fájlok neve megváltozott a Laravel 11-ben

5–3. újdonság: Alapértelmezetten létrejövő migrációs fájlok nevei

A fájlok nevei után vizsgáljuk meg a belső tartalmukat és szerkezetüket! Minden ilyen fájlban van egy **up()** és egy **down()** metódus. Az **up()**-ban lévő utasításokat hajtja végre a rendszer, amikor a verziókövetésben *előre lép,* a **down()**-ban lévő kódot pedig akkor, amikor a verziókövetésben *visszalép*. A már meglévő migrációs fájlokban emiatt az **up()** metódusok egy-egy adattábla létrehozást (és a mezőik létrehozását) tartalmazza, a **down()** metódusok pedig egy-egy adattábla törlését tartalmazzák.

Hozzuk most létre az első saját migrációs fájlunkat, amiben a blogbejegyzéseket fogjuk tudni eltárolni.

```
php artisan make:migration create_posts_table
```

Ebben a fájlban tudjuk programozottan kezelni a **posts** adattábla szerkezetét. Koncentráljunk először az **up()** metódusra. Itt tudjuk megadni azokat a mezőket, amiket szeretnénk eltárolni az adattáblánkban:

```
public function up(): void
{
  Schema::create('posts', function (Blueprint $table) {
    $table->id();
    $table->string('slug');
    $table->text('body');
    $table->timestamp('published_at')->nullable();
    $table->timestamps();
  });
```

```
}
```

5–7. kódrészlet: A posts tábla létrehozása és kezdeti szerkezete

Alapértelmezetten az **id()** és a **timestamps()** segédmetódusokat tartalmazta a két sor, ezeket minden adattábla tartalmazza majd, ne módosítsuk, mivel ezek a mi segítségünkre lesznek a jövőben. Hozzáadtunk három sort (mezőt), de nem ez lesz a végleges táblaszerkezet, viszont kiindulásnak megfelelő lesz.

Ha nem is teljesen értünk minden sort, az azért kivehető, hogy ez egy CREATE TABLE utasítást tartalmaz *„rejtetten”* SQL nyelven: létrehozza a **posts** nevű táblát, **id**, **slug**, **body** és *„időbélyeges”* mezőkkel (oszlopokkal). Az adattípusokról (string, text, timestamp stb.) itt olvashatunk bővebben. Az id() segédmetódusnál bár úgy tűnik, hogy nem tartalmaz típust, de rejtetten mégis: ez egy előjel nélküli big integer (nagy egész szám, Unsigned Bigint), ami az elsődleges kulcs (Primary Key) lesz a táblában, valamint egy automatikus léptetést (Auto Increment) is hozzáad beszúrások (INSERT INTO) esetén (MySQL-ben ez AUTO INCREMENT névre hallgat, Microsoft SQL-ben pedig Identity).

A migrációs fájl(ok)ban lévő **up()** metódusokban lévő kódok fognak végrehajtódni egymás után, ha kiadjuk a következő utasítást:

```
php artisan migrate
```

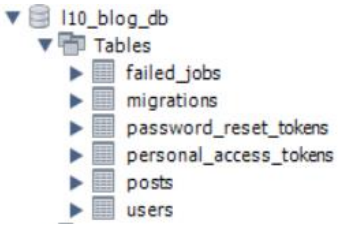
```
PS C:\xampp\htdocs\l10-components> php artisan migrate

  INFO  Preparing database.

  Creating migration table ..................................................... 133ms DONE

  INFO  Running migrations.

  2014_10_12_000000_create_users_table ......................................... 99ms DONE
  2014_10_12_100000_create_password_reset_tokens_table ......................... 72ms DONE
  2019_08_19_000000_create_failed_jobs_table ................................... 75ms DONE
  2019_12_14_000001_create_personal_access_tokens_table ........................ 43ms DONE
  2023_03_31_165920_create_posts_table ......................................... 19ms DONE
```

5–18. ábra: Migrációs fájlok lefutása és sorrendjük

A fájlok végrehajtási sorrendje a fájlok nevében lévő időbélyeg szerinti növekvő sorrendben fog végrehajtódni. Alapértelmezetten voltak már migrációs fájljaink, most azok mellé került be ez a sajátunk (**create_posts_table**) újként, az új időbélyeggel.

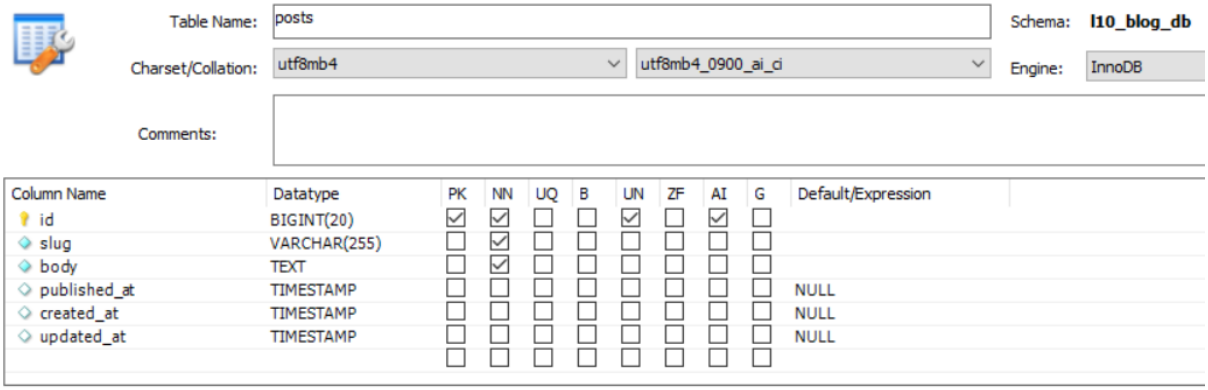
Egy dolog még feltűnhet nekünk, legelőször, az adatbázis előkészítése során létrejött egy migrációs (**migrations**) adattábla, ennek a verziókövetésben lesz szerepe, amit a következő alfejezetben részletezek majd.

Ellenőrizzük le a Workbench-ben, hogy milyen táblák kerültek migrálásra a rendszerben:



5–19. ábra: Migrálás eredményeként létrejövő adattáblák az adatbázisban

Bekerült tehát az adatbázisba a **posts** nevű táblánk is, a megadott oszlopnevekkel és típusokkal.



5–20. ábra: posts adattábla részletei és a mezők típusai, kényszerei

Az **id()** segédmetódusról és lefutásának eredményét már részleteztem, de nézzük meg a **timestamps()** segédmetódus lefutásának eredményét is! Fontos, hogy ez angolul többesszámban van, tehát több időbélyeges oszlop jött létre ennek következtében. Ezek az oszlopok a **created_at** és az **updated_at** mezők, amelyek olyan hasznos kiegészítő információt nyújtanak majd mindig nekünk, hogy automatikusan eltárolják azokat az időbélyegeket, amikor a táblában egy újabb sor beszúrásra került (**created_at** mezőben látható), vagy módosításra került a sor (**updated_at** mezőben látható). A **published_at** időbélyeget mi hoztuk létre manuálisan a migrációs fájl egy külön sorában, ez nem tartozik az alapértelmezetten létrejövő mezők közé.

A **posts** tábla létrehozása után rájöttünk, hogy a **title** mezőt kihagytuk a tábla szerkezetéből, vizsgáljuk meg, hogy mit tehetünk ilyenkor, milyen lehetőségeink vannak.

5.2.2.3. Migrálás és verziókövetés

Először is, el kell felejtenünk, hogy manuálisan (az adatbázis menedzselő alkalmazásunkkal vagy CREATE / ALTER / DROP TABLE utasításokkal) kezeljük az adattábláinkat és szerkezetüket. Ha így tennénk, akkor az elrontaná a verziókövetést, amelyet a Laravel biztosít a számunkra a **migrations** adattáblán keresztül.

id	migration	batch
1	2014_10_12_000000_create_users_table	1
2	2014_10_12_100000_create_password_reset_...	1
3	2019_08_19_000000_create_failed_jobs_table	1
4	2019_12_14_000001_create_personal_access_...	1
5	2023 03 31 165920 create posts table	1

5–21. ábra: Migrációs (migrations) adattábla tartalma az első migrálás után

A **migrations** tábla tartalmazza a végrehajtott migrációs fájlokat sorrend szerint és *„batch-enként”* vagyis csomagokban. Jelenleg mind az öt migrációs fájl az első csomagba tartozik, mivel egyszer hajtottuk végre a migrációt.

A mi célunk most az, hogy hozzáadjuk a **title** mezőt a **posts** táblához. Módosítsuk a **create_posts_table** migrációs fájlt és adjuk hozzá ezt a mezőt (mondjuk az **id** és a **slug** mezők, sorok közé):

```
$table->string('title');
```

Majd újra migráljunk:

```
php artisan migrate
```

Azt fogjuk visszakapni, hogy nincs mit migrálni (*„Nothing to migrate.”*). Mindez azért van, mert a rendszer nem érzékeli azt, hogy lenne újabb migrációs fájl, amit migrálni kellene, hiszen nincs is, mert mi csak a már meglévő és migrált **create_posts_table** migrációs fájlt módosítottuk.

Azt tehetjük ilyenkor, hogy visszalépünk egy (esetleg több) batch-et a **migrations** adattábla szerinti verziókban:

```
php artisan migrate:rollback
```

Majd újra migrálunk:

```
php artisan migrate
```

Ekkor már be fog kerülni a **title** mező is a **posts** táblánkba. *De gondoljuk át, hogy mi ezzel a gond…? Törlődtek az adatbázisban lévő tábláink, minden adatukkal együtt!* Ez most még csak azért nem okozott problémát nekünk, mivel minden adattáblánk üres volt, de ez azért elég ritka szituáció, amikor egy működő rendszerrel dolgozunk.

A migrációs `rollback` hatására a migrációs fájlok **down()** metódusa hajtódott végre, amelyek mind az öt esetben adattábla törlését (kvázi `DROP TABLE`) végzik el.

A visszalépések helyett, ha úgy szeretnénk, akkor egyetlen utasítással újraépíthetjük az adattábláinkat a migrációs fájlok segítségével az elejétől a végéig. Hajtsuk végre az utasítást, de előbb töröljük ki a **title** mező hozzáadását a **posts** táblát létrehozó migrációs fájlból, mivel egy másik módon fogjuk majd hozzáadni ezután.

```
php artisan migrate:fresh
```

A fentieknél egy sokkal jobb megoldás az (főleg akkor, ha már vannak adataink az adattáblákba), ha egy új migrációs fájlt hozunk létre és abba helyezzük el a **title** mező hozzáadását a **posts** táblához:

```
php artisan make:migration add_title_to_posts_table
```

Már a neve is beszédes, ha rápillantunk, egyből tudhatjuk is, hogy mit tartalmaz a migrációs fájl: hozzáadja a **title** mezőt a **posts** táblához. Ha pedig így adjuk meg a fájl nevét, akkor a névkonvenciók miatt a Laravel rögtön eszerint készíti el nekünk az **up()** és **down()** metódusokat: a **posts** tábla módosítását elvégző utasításokat helyez el bennük.

```php
public function up(): void
{
  Schema::table('posts', function (Blueprint $table) {
    $table->string('title');
  });
}

public function down(): void
{
```

```
  Schema::table('posts', function (Blueprint $table) {
    $table->dropColumn('title');
  });
}
```

5–8. kódrészlet: A title mezőt hozzáadó és elvevő migrációs fájl

A migrálással lefut az **up()** metódusban lévő **title** mező hozzáadás, a migrálás visszavonásával (rollback) törli a **posts** táblából a **title** mezőt. Próbáljuk is ki ezeket, oda vissza és közben figyeljük meg, hogy a **migrations** táblában is megtörténik a verziókövetés az új sornál a batch megadásával (csomagonként építi fel és le az adatbázis szerkezetének módosításait).

1. php artisan migrate
2. php artisan migrate:rollback
3. php artisan migrate

migration	batch
2014_10_12_000000_create_users_table	1
2014_10_12_100000_create_password_reset_...	1
2019_08_19_000000_create_failed_jobs_table	1
2019_12_14_000001_create_personal_access_...	1
2023_03_31_165920_create_posts_table	1
2023_04_02_180057_add_title_to_posts_table	2

5–22. ábra: Új migrálási csomag a migrations adattáblában

Így viszont a **title** mezőt a **posts** táblában az utolsó helyre illesztette be a rendszer. Lépjünk egyet vissza a migrációval, és módosítsuk az **up()** metódus tartalmát az alábbi szerint (a **title** mezőt az **id** után szúrja be a táblaszerkezetbe):

$table->string('title')**->after('id');**

Most már újra migrálhatunk és jó helyre fog kerülni a **title** mező a **posts** táblában.

Megjegyzés: az **after()** sajnos az SQLite-ban nem működik, csak a többi adatbáziskezelő rendszerben, így az SQLite-ban továbbra is az oszloplista végén marad a **title** mező.

Ha tovább akarunk ismerkedni ezzel a migrate paranccsal és a rollback-en kívüli egyéb paraméterezéseivel, akkor futtassuk ezt az utasítást: php artisan és keressük meg a migrate szekciót.

```
migrate
 migrate:fresh          Drop all tables and re-run all migrations
 migrate:install        Create the migration repository
 migrate:refresh        Reset and re-run all migrations
 migrate:reset          Rollback all database migrations
 migrate:rollback       Rollback the last database migration
 migrate:status         Show the status of each migration
```

5–23. ábra: Migrációs lehetőségek a php artisan parancs használatakor

Például a php artisan migrate:status utasítás kiadásával a **migrations** adattábla tartalmát kapjuk vissza a terminal-ban.

Továbbá szeretném felhívni mindenképpen a figyelmet a fresh és a refresh parancsokra, mert ezek *eléggé veszélyesek*, főleg akkor, ha éles környezetben dolgozunk. Ezek hatására ugyanis teljesen újraépíti a migrációs fájlok alapján az adatbázis szerkezetet a Laravel, különösebb megerősítő kérdés nélkül, azonnal végrehajtja őket. Ezt most kipróbálhatjuk, mert most még igazából nem okozunk vele kárt, de a későbbiekben nagyon veszélyes lenne ezek használata. Figyeljük meg, hogy ha végrehajtjuk, akkor az add_title_to_posts_table is az 1-es csomagba (batch-be) fog kerülni, mivel már létezik és első migrálási körben a rendszer a meglévő migrációs fájlokat mind végrehajtja az első csomagban.

5.2.2.4. További különbségek az adatbázis kiszolgálók között

A migrációs fájlok, mint egy adatstruktúra menedzselő réteg helyezkednek el az adatbázisunk felett és az ő segítségével tudjuk kezelni az adatbázisban lévő táblaszerkezeteinket. Az adatkapcsolat létrehozása több különböző adatbázis-kezelő rendszerhez lehetséges és a migrációs fájlok a legtöbb műveletüket mindegyik adatbázis-kezelőben végre tudják hajtani (adattábla létrehozása, törlése, oszlopok módosítása stb.). Előfordulhat azonban, hogy találkozunk olyan különbséggel, ami például a MySQL-ben működik, de az SQLite-ban egy kicsit másképp van, egy ilyen lehetőségre példa az idegen kulcsok eldobása, ami MySQL-ben simán működik, de az SQLite-ban egy kicsit *„nyakatekert”* megoldást kell erre alkalmaznunk, mivel az SQLite csak nagyon limitált lehetőségeket biztosít az ALTER TABLE SQL utasítás keretein belül. SQLite-ban ez úgy valósítható meg, hogy egy új táblát kell létrehoznunk az idegen kulcs kényszer nélkül és át kell másolni az eredeti tábla tartalmát az új táblába. Ekkor egy megoldási lehetőség az lehet a migrációs fájlban, hogy megvizsgáljuk a művelet (dropForeign()) végrehajtása előtt, hogy milyen adatkapcsolatunk van, és csak akkor hajtjuk végre, ha nem SQLite. Ekkor a migrációs fájlon belül az up() vagy down() metódusok valamelyikében a séma létrehozó vagy módosító utasítás szekcióban elvégezhetjük a feltételes utasítás végrehajtását.

```
if (env('DB_CONNECTION') !== 'sqlite')
  $table->dropForeign(['thumbnail_id']);
```

5–9. kódrészlet: Példa kód: adatbáziskapcsolat-függő utasítás végrehajtása a migrációs fájlban

Ebben a példakódban egy thumbnail_id nevű idegen kulcsot és a hozzá tartozó oszlopot töröljük akkor, ha nem sqlite-os az adatkapcsolat.

Ez csak egy ilyen példa volt, az összeset nem is célom bemutatni, csak rá szerettem volna világítani, hogy ha mi fejlesztünk egy alkalmazást a saját környezetünkben, az általunk kiválasztott adatbázis-kezelőben és a migrációs fájljainkkal. Akkor ettől még egy másik környezetben, egy másik adatbázis-kezelőnél ugyanezek a migrációs fájlok okozhatnak problémákat, amelyek kijavításáról gondoskodni kell.

Tipp: Teszteld a tudásod!

Bár még nem tudunk mindent a migrációról, de érdekes lehet kipróbálni az eddigi tudásunkkal, milyen teszteknek tudnánk megfelelni. Léteznek nyilvános repo-k, amelyekkel ezt a tesztelést meg tudjuk tenni, itt van például egy, ami a migrációs témakör alapjaihoz tartozik:

https://github.com/LaravelDaily/Test-Laravel-Migrations

Ha klónozzuk a projektet és elindítjuk a tesztelést, akkor kezdetben minden tesztünk elbukik. De a GitHub oldalon lévő **Task** lista (a könyvtár- és fájlstruktúra alatt) pontosan megmutatja, hogy milyen fájlokat kell szerkeszteni ahhoz, hogy átmenjenek a tesztjeink.

Még nem tanultunk mindenről, így azokat nem kötelező figyelembe venni, amiről még nem tudunk, a saját tudásunk felmérésénél. Viszont kialakulhatott már annyi rutinunk, hogy esetleg a Laravel dokumentáció megfelelő részét használva is megoldható az, hogy átmenjenek sikeresen a fejlesztéseink az alkalmazás migrációinak tesztesetein.

Az alfejezetben végrehajtott programkód-módosítások ebben a GitHub commit-ben érhetők el.

5.3. Összegzés

A fejezet feldolgozása során megvizsgáltuk, hogy a Laravel keretrendszer milyen adatbázis-kezelő rendszerekhez biztosít alapértelmezetten kapcsolódási felületet (MySQL, SQLite, Microsoft SQL Server). Az **.env** környezeti beállításokat tartalmazó fájl segítségével ténylegesen csatlakoztunk is ezekhez az adatbázis-kezelő rendszerekhez. Ahol szükséges volt (SQL Server) ott további beállításokat is végrehajtottunk annak érdekében, hogy kapcsolódni tudjunk a szerveren lévő adatbázishoz.

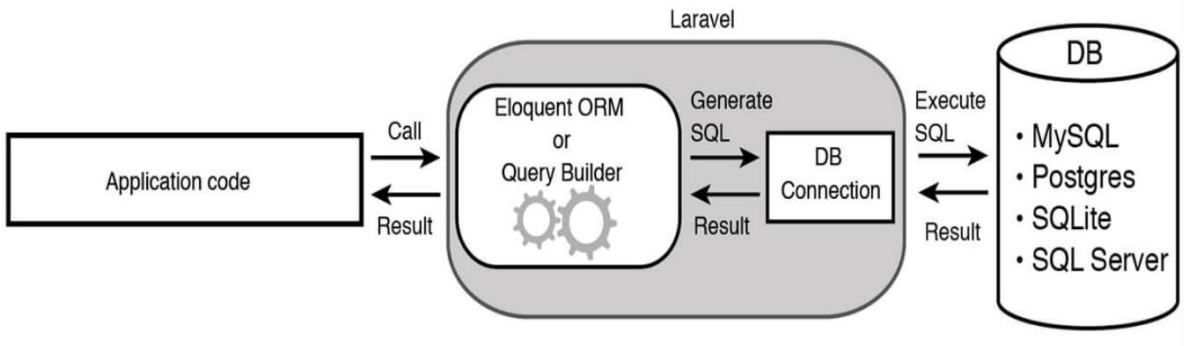
Ezután megismerkedtünk a migrációk világával, amely fájlok segítségével verziókövetéssel ellátott adatbázis manipulációs (tábla / mező / kapcsolat létrehozó és módosító) műveleteket hajtottunk végre a gyakorlat során.

6. Adatbázis-kezelés (Database management)

Most már többféle adatbázishoz tudunk csatlakozni, a migrációs fájlok segítségével pedig tudunk táblaszerkezeteket létrehozni és módosítani is (SQL DDL). Következhet az adatbázis-kezelés (SQL) manipulációs (DML) és lekérdezési (DQL) része.

6.1. Bevezetés az adatbázis-kezelés használatába Laravel-ben

A 6–1. ábra megmutatja, hogy használhatunk mindenféle adatbázis-kezelő rendszert (mi is alkalmaztunk már a MySQL-en kívül SQLite-ot, SQL Server-t, és akár felhőbeli adatszolgáltatást is). Az .env fájl DB_ prefixű attribútumai felelősek voltak azért, hogy ezekhez a típusú rendszerekhez fel tudjuk építeni az alkalmazásunkból a kapcsolatot. Most pedig rátérünk arra, hogy hogyan lehet majd manipulálni az adattábláinkat (INSERT, UPDATE, DELETE stb.) illetve lekérdezni belőlük (SELECT).



6–1. ábra: A webalkalmazás adatbázis kapcsolódási, lekérdezési és manipulációs folyamatai, eszközei

A Laravel keretrendszer kétféle lehetőséget nyújt a számunkra:

1. **Eloquent**: ez egy **ORM (Object-Relational Mapping)** eszköz, ami lehetővé teszi, hogy objektumokat használjunk az adatbázis műveletek elvégzésére, a művelet eredményét pedig könnyedén feldolgozzuk a segítéségével. Ezzel egy olyan eszközhöz jutunk, amellyel könnyedén tudunk ún. **CRUD** *(Create-Read-Update-Delete),* vagyis a leggyakrabban használt műveleteket végrehajtani az adatbázisunkban. A használatuk célja az, hogy adatokhoz szeretnénk az Eloquent-tel hozzájutni vagy manipulálni őket könnyedén. Gyakorlati alkalmazására a 6.2. alfejezetben láthatunk példákat.
2. **Query Builder:** ez egy olyan eszköz, amelynek a segítségével az SQL utasításokra kísértetiesen hasonlító lekérdezéseket rakhatunk össze, építhetünk fel utasításrészletek egymáshoz fűzésével. Ezzel a témakörrel akkor fogunk foglalkozni, ha az Eloquent-et már kellőképpen megismertük. Gyakorlati alkalmazására a 6.3. alfejezetben mutatok be példákat.

Az Eloquent és a Query Builder a használatuk céljából egyenértékűek, vagyis mindkettő használható az adatok manipulációjára és lekérdezésére, az Eloquent alkalmazásakor egy objektumon vagy magán a Model osztályon keresztül érjük el az adatokat az adatbázisban és az osztályban írhatunk hozzájuk támogató metódusokat. A Query Builder esetén többnyire nagyobb adathalmazokat kérünk le az adatbázisból, amelyek aztán egy-egy gyűjteményként (lista, tömb stb., angolul collection) jelennek meg a webalkalmazásban és azokat tudjuk programozottan kezelni, vagy majd módosítani az adatbázisban.

Miért is használjuk ezeket? Miért nem írunk csak saját, felépített SQL lekérdezéseket? A választ több oldalról is megvizsgálom, mivel ezen fenti két eszköz alkalmazása olyan előnyöket nyújt a számunkra, amelyeket vétek lenne nem kihasználni, például:

- Helyes és optimalizált SQL lekérdezéseket generálnak a háttérben, kisebb a hibázási lehetőség.
- A lekérdezések könnyebben frissíthetők, karbantarthatók, újra felhasználhatók.
- Megvédik a használóját (minket) attól, hogy SQL beszúrásos (injection) támadást tudjanak végrehajtani az alkalmazásunk kódjai ellen.
- Segítségével lehetőségünk van arra (amit korábban már csináltunk is), hogy könnyedén leválthatjuk az alkalmazásunk alatt futó adatbázis-kezelő rendszert, és nem kell emiatt megváltoztatnunk (vagy csak nagyon minimálisan) az Eloquent ORM-mel vagy a Query Builder-rel összerakott lekérdezéseinket.
- A programozók úgy manipulálhatják a kódjukat, mintha objektumokkal, gyűjteményekkel dolgoznának.

Az Eloquent ORM használatához szükség van arra, hogy az eddig nem igazán érintett MVC elemhez nyúljunk, ez pedig a Model. A Model-ek két fontos dologért felelnek:

1. Ők biztosítják az üzleti logikát, tehát olyan adattagokat és metódusokat, amelyek az alkalmazásunk funkcióinak működését teszik lehetővé.
2. Ők azok, akik az Eloquent ORM segítségével elérik a *hozzájuk tartozó (!) adatbázistáblát*. A hozzá tartozó adatbázistáblát alapértelmezetten a Laravel a névkonvenció miatt tudja. A Model nevét angolul egyesszámban használjuk, így a Model többesszámú nevét viselő adattáblát tudjuk elérni. Például a **Post** Model fájl a **posts** adattáblát fogja elérni, használni, esetleg manipulálni. (Ezt a névkonvenciót persze felüldefiniálhatjuk majd, de alapértelmezetten ez a Laravel működési elve.)

Megjegyzés: az első pontra ismét felhívnám a figyelmet, mert nagyon sokan összekeverik az *„üzleti logika megvalósítása"* szempontjából a Model-t a Controller-rel. Fontos, hogy a Model biztosítja az üzleti logikát, vagyis az egyes metódusokat, amelyekkel például számításokat tudunk végezni az adatokon, így lesz csak később újra felhasználható az, amit implementálunk, a Controller-ben ez csak nagyon nehezen és a Laravel szabályaival szembe menve tudna csak működni, ezért inkább ezt ne tegyük!

6.2. Eloquent alkalmazása

Elsőként az Eloquent-tel fogunk megismerkedni gyakorlati szempontból, ahogy eddig is tettük. A másik (**my-first-site** nevű) projektünkben már a 3.4. alfejezetben létrehoztuk a Controller és a Model fájlokat. Itt is hasonlóképpen fogunk eljárni, de most az adatbázistábla elérése, lekérése és feltöltése szempontjából egyelőre csak a Model fájlra lesz szükség, amit kezdetben a Tinker alkalmazással, majd utána az útvonalakon keresztül fogunk menedzselni.

6.2.1. A Model fájl

Hozzunk létre a projektünkben a blogbejegyzéseknek egy Eloquent ORM fájlt, ami egy Model lesz:

```
php artisan make:model Post
```

Ennek hatására bekerül az **app / Models / Post.php** fájl a projektünkbe. Ez a Model ősosztályt terjeszti ki, VSCode-ban akár meg tudjuk nézni az osztály tartalmát: számos változóval és metódussal dolgozik, amelyeket előre definiál nekünk, de mi felül is írhatjuk majd őket.

Készítsünk *„manuálisan"* az adatbázistáblába való beszúrással egy blogbejegyzést.

```
INSERT INTO `l10_blog_db`.`posts`(
  `title`,
  `slug`,
  `body`,
  `published_at`
)
VALUES (
  'My first blog post',
  'my-first-blog-post',
  'The content of post',
  '2023-04-10 12:11:00'
);
```

6–1. kódrészlet: posts tábla feltöltése INSERT INTO SQL utasítással

Ha végrehajtottuk a lekérdezést, beszúródott a sor, utána el is köszönhetünk ettől a módszertől, ugyanis mostantól kezdve sosem fogunk kiadni így SQL utasítást az adatbáziskezelő rendszer menedzselő alkalmazásban, mindig csak programkódból fogunk adatlekérdező és -manipuláló műveleteket végrehajtani.

6.2.2. Lekérdező és manipuláló lekérdezések futtatása Tinker alkalmazással

Az Eloquent Model fájlunkat használjuk a Tinker (REPL[7]) eszközben. Ez lehetőséget biztosít a számunkra, hogy kapcsolatot teremtsünk a Laravel alkalmazásunkkal működés közben, méghozzá programozottan, PHP utasítások kiadásával.

Indítsuk is el ezt a terminal-unkban:

```
php artisan tinker
```

Írjuk be az induló Tinker-be, hogy 2+2 és üssünk entert, megkapjuk, hogy 4. Tehát ahogy a lábjegyzetben olvasható, megkapja a felhasználói bemeneteket, kiértékeli őket, majd kiírja az eredményt.

PHP kódsorokat is képes kiértékelni:

```
$name = 'Attila';

echo "Hi, " . $name;
```

6–1. utasítások: PHP kód futtatása a Tinker-ben

[7] Read-Eval-Print Loop = egy olyan környezet, amelyben felhasználói bemeneteket tud beolvasni, kiértékelni és kiírni az alkalmazás. Ezen felül a Laravel alkalmazást (Tinker indításkori verzióját!) tudjuk megszólítani és műveletek végrehajtásával megbízni.

A két sor végrehajtása után köszönt engem a konzolon keresztül, tehát a változó értékét hozzáfűzte a köszönéshez.

Megjegyzés: nem okoz hibát, hogy ha lehagyjuk a sor végén lévő pontosvesszőt (;), de én a pontosság kedvéért mindig ki fogom írni a példáimban, mert így akár majd a webes alkalmazásunkban is használhatók lesznek ezek az utasítások kódsorként.

6.2.2.1. Egyszerű lekérdezés (SELECT)

De nem csak ilyen egyszerű PHP kódokat tudunk megadni a Tinker-nek, amit ő majd kiértékel és reagál rá, hanem a Tinker indításakor meglévő állapotú Laravel alkalmazásunkat is meg tudjuk szólítani és működésre tudjuk bírni.

```
App\Models\Post::all();
```

6–2. utasítások: A posts táblában lévő összes sor lekérése az Eloquent Model segítségével

Eredményül az **all()** metódus visszaad nekünk egy gyűjteményt, amelyben minden elem az **App\Models\Post** osztály egy példánya, objektuma, ami SQL nyelven kvázi egy SELECT * FROM **posts;** utasításnak felel meg, amivel a **posts** adattábla összes sora és összes mezője lekérésre kerül.

```
> App\Models\Post::all();
= Illuminate\Database\Eloquent\Collection {#4477
    all: [
      App\Models\Post {#4436
        id: 1,
        title: "My first blog post",
        slug: "my-first-blog-post",
        body: "The content of post",
        published_at: "2023-04-10 12:11:00",
        created_at: null,
        updated_at: null,
      },
    ],
  }
```

6–2. ábra: A Post Eloquent Model fájlon keresztül lekérjük az posts adattábla tartalmát, gyűjteményt kapunk vissza

Megjegyzés: a kiíratásnál megjelenő azonosító számok (például az iménti ábrán a #4477 vagy a #4436) véletlenszerűen generált értékek, amelyek az adott munkamenetben érvényesek, de ha ti futtatjátok ezeket a parancsokat a Tinker-ben, akkor biztosan más értékeket fogtok ott látni.

Kiegészítés: a Laravel rengeteg gyűjteményt kezelő metódussal segíti a munkánkat, ezek közül itt most csak néhányat emelek majd ki, de ennél sokkal több hasznos funkció érhető el hozzájuk, amelyeket érdemes áttekinteni a hivatalos dokumentációban. Mivel az adatbázisból érkező adatok gyűjteményével foglalkozunk itt, ezért az adatok manipulálásával kapcsolatban léteznek gyűjtemény kezelést támogató metódusok, amelyekről itt olvashatunk részletesebben és találunk hozzájuk példákat. Ez egy kibővítése annak az amúgy is óriási metódus-halmaznak, amelyet a Laravel biztosít számunkra az adatbázis-kezeléstől független gyűjtemények kezeléséhez, amelyeket a 12.1. alfejezetben én is bemutatok számos gyakorlati példával együtt.

Teszteljük tovább a lekérdezést, az **all()** helyett lekérhetjük a legelső példányt (adattábla sort) és annak mezőit is.

```
$post = App\Models\Post::first();

$post->title;
```

6–3. utasítások: Kiírjuk a posts adattábla legelső sorának title mezőjének értékét

Ha csak a **title** attribútumának kiíratást akarjuk elvégezni (kvázi, mintha az SQL SELECT utasításban ennek az egy oszlopnak a nevét írnánk ki), akkor nem is feltétlenül lenne itt szükség a **$post** objektum létrehozására, hanem csak a **first()** metódus után tudnánk folytatni a **->title** lekérésével.

6.2.2.2. Beszúrás (INSERT)

Hozzunk létre egy új blogbejegyzést a Tinker segítségével: először egy **Post** objektumra lesz szükség, amit utána felparaméterezünk, végül elmentjük, így bekerül majd a **posts** adattáblába az új sor.

```
$post = new App\Models\Post;

$post->title = 'My second blog post';

$post->slug = 'my-second-blog-post';

$post->body = 'The content of the second blog post';

$post->published_at = '2023-04-10 17:39:00';

$post->save();
```

6–4. utasítások: Új blogbejegyzés létrehozása a Tinker-ben

Ellenőrzéshez futtassuk az 6–2. utasításokat és láthatjuk, hogy bekerült az adattáblánkba a beszúrt sor. Ellenőrizhetjük mindezt az adatbázis-kezelő alkalmazásunk segítségével is és lekérhetjük a **posts** tábla tartalmát, a következőt fogjuk tapasztalni:

- benne van az új sor az adattáblánkban,
- az **id** mezőben automatikusan léptetésre került az új sor egyedi azonosítója,
- annak ellenére, hogy a **created_at** és **updated_at** mezőknek nem adtunk értéket, a Laravel keretrendszer ezeket automatikusan kitölti az új sor beszúrásakor, viszont két (esetleg egy) órával kevesebb értéket tartalmaznak,
    - ez azért van, mert az alkalmazás projektünkben a beállítások között a **config / app.php** az időzóna (**timezone**) értéke **UTC**-re van állítva, ha azt szeretnénk, hogy az időzóna ne okozhasson eltérést az időbélyegekben, akkor írjuk át a UTC-t erre: **Europe/Budapest**

Új környezeti változó: az **.env** fájlban lévő **APP_TIMEZONE**

A Laravel 11-ben már az **.env** fájlban tudjuk definiálni az alkalmazásunkra vonatkozó időzónát az **APP_TIMEZONE** beállítás segítségével.

6–1. újdonság: Új környezeti változó az alkalmazott időzónáról

Ezzel viszont megváltoztattuk a Laravel alkalmazásunk kódját, amit viszont még a Tinker nem érzékel. Ezért a programkód módosítások érvényre juttatásához, illetve, hogy az alkalmazásunk legfrissebb verziójával *„kommunikáljunk"* a Tinker-en keresztül, indítsuk újra a Tinker-t, Ctrl + c billentyűkombinációval zárjuk be, majd `php artisan tinker` paranccsal indíthatjuk ismét.

Szerencsére a Tinker *„emlékszik",* hogy milyen parancsokat futtattunk benne a bezárás előtt, így a felfelé mutató nyíllal tudunk lépkedni a korábban kiadott parancsok között és felülírhatjuk őket a következők szerint:

```
$post = new App\Models\Post;
$post->title = 'My third blog post';
$post->slug = 'my-third-blog-post';
$post->body = 'The content of the third blog post';
$post->published_at = '2023-04-10 18:00:00';
$post->save();
```

6–5. utasítások: Új blogbejegyzés létrehozása a Tinker-ben, frissített időzóna szerint

A parancsok futtatása után, az 6–2. utasításokat ismét futtatva, vagy közvetlenül az adattáblánk tartalmát lekérve az adatbázis-kezelő menedzserben ellenőrizhetjük, hogy most már jó értéke kerülnek be a **created_at** és **updated_at** mezők értékeibe az új sorunknál.

6.2.2.3. Frissítés (UPDATE)

Egy meglévő (lekért) objektum/adatsor adattagjainak lekérése, majd módosítása és mentése megfelel egy sor frissítésének az adatbázisban. Mindez a Tinker-ben így néz ki: kérjük le az első blogbejegyzést (első sort a **posts** táblából), a visszakapott példányban módosítsuk például a bejegyzés címét (**title**) és mentsük el.

```
$post = App\Models\Post::first();
$post->title = 'My first modified blog post';
$post->save();
```

6–6. utasítások: Frissítés bemutatása a Tinker-ben

Itt tehát nem hoztunk létre új példányt a **Post** osztályból, hanem az Eloquent Model segítségével az adattábla első sorát és mezőit kértük le és tároltuk el a **$post** objektumban. Utána az objektumnak módosítottuk a **title** mezőjét majd elmentettük, amivel gyakorlatilag egy frissítést (UPDATE-et) hajtottunk végre a **posts** adattáblában. Ennek ellenőrzését ugyanúgy hajthatjuk végre, mint korábban. Figyeljük meg, hogy az első sor (vagy objektum, attól függően, hogy hol ellenőrizzük), most már tartalmazza az **updated_at** mezőjében az aktuális időbélyeget, ami a frissítés hatására automatikusan bekerült a mezőbe.

6.2.2.4. Törlés (DELETE)

Ha pedig egy adatsort lekérünk egy objektumba, majd töröljük, akkor az adatbázisban egy sor törlését hajtjuk végre. A **find()** metódussal tudjuk megkeresni (és megtalálni) az adott azonosítóval (**id**-val) rendelkező blogbejegyzést. Ez tehát **id** szerint fog keresni és adja vissza az eredményt nekünk.

```
$post = App\Models\Post::find(2);

$post->delete();
```

6–7. utasítások: Törlés bemutatása a Tinker-ben

Ha olyan azonosítójú sort szeretnénk lekérni az adattáblából az objektumba, ami nem létezik, akkor **null**-t ad vissza a Tinker és természetesen null-t nem is tudnánk törölni a **delete()** metódus hívásával.

6.2.2.5. Soft Delete

A rákérdezés (*„Biztosan szeretnéd törölni az adott elemet?"*) nélküli törlés egy kicsit problémás lehet néha, ha nem szeretnénk rögtön elveszteni egy – még rosszabb esetben több – adatsort (akár egy rosszul összeállított szűrés miatt). Ennek elkerülése céljából az Eloquent ORM-ben lehetőségünk van úgynevezett *„soft delete"* használatára. Ennek alkalmazásával nem fog törlődni ténylegesen az adatsor a táblából, hanem csak egy új **deleted_at** időbélyeg mezőben el tudjuk tárolni, hogy az adott sort mikor törölték. Ehhez bővíteni kell a **Post** Model osztályt:

```
// importálás az osztály előtt:
use Illuminate\Database\Eloquent\SoftDeletes;


// tényleges használat az osztályon belül:
use HasFactory, SoftDeletes;
```

6–2. kódrészlet: SoftDeletes importálása és használata a Post Model fájlban

Lépjünk ki a Tinker-ből és hozzuk létre a **deleted_at** mezőt migráló fájlt:

```
php artisan make:migration add_deleted_at_to_posts_table
```

```
// up() metódus séma bővítése:
$table->softDeletes();


// down() metódus séma szűkítése:
$table->dropSoftDeletes();
```

6–3. kódrészlet: Soft Delete a migrációs fájlban

A migrációs fájlokkal való ismerkedés során egy kicsit másképp adtunk hozzá és vettünk el oszlopokat a táblából, de itt ezeket a segédmetódusokat specifikusan a *„soft delete"* működéséhez biztosítja nekünk a Laravel. Az utasítások viszont egyenértékűek lennének azzal, mintha hozzáadnánk egy ilyen utasítást az **up()** metódusban:

```
$table->timestamp('deleted_at')->nullable();
```

A **down()** metódusban pedig törölnénk ezt az oszlopot.

Hajtsuk végre a migrációt és ellenőrizzük le utána, hogy valóban létrejött a **deleted_at** timestamp típusú mező a **posts** adattáblánkban.

```
php artisan migrate
```

Ha most a Tinker újraindítása után lefuttatjuk az alábbi utasítás sorozat első két sorát, tehát egy meglévő (például az 1.) sorhoz tartozó objektummal végrehajtom a törlést, akkor most már nem fog törlődni, hanem a **deleted_at** mezőbe kerül bele az adott sorban az aktuális időbélyeg.

```
$post = App\Models\Post::first();
$post->delete();
App\Models\Post::all();
App\Models\Post::withTrashed()->get();
$post->restore();
App\Models\Post::all();
```

6–8. utasítások: Post objektum (sor) törlése és helyreállítása kiegészítve lista lekérésekkel

Viszont a rendszer a 3. sor végrehajtására úgy reagál, ahogy az elvárható, tehát a *„törölt"* sort nem fogja visszaadni eredményül a gyűjteményben. Ha szeretnénk, hogy a *„soft delete"* módon törölt sorokat is visszaadja nekünk a rendszer, akkor a **withTrashed()->get()** metódusok láncolatát kell hívnunk. Végül az utasítások 5. sorában helyreállítottam (**restore()**) a blogbejegyzést, vagyis ezzel töröltem az adott sorban a **deleted_at** mező értékét és így már nem törölt, hanem helyreállított lesz, utána már újra meg tudom kapni az **all()** metódus hívásával a nem törölteket.

A tényleges sortörlésről akkor sem kell lemondanunk, hogy ha a *„soft delete"* technikát alkalmazzuk az Eloquent Model fájlunkban, mindössze arra van szükség, hogy a **delete()** metódus helyett a **forceDelete()** metódust hívjuk meg az objektumnál.

A törlések akkor is működnek, ha nem csak egy sort (objektumot) kérünk le, hanem többet (kvázi egy gyűjteményt), akkor ezeket is tudjuk együttesen törölni.

Tesztelés és gyakorlás szempontjából érdemes létrehozni néhány blogbejegyzést, frissíteni őket, esetleg törölhetjük is őket a *„soft delete"* eljárás gyakorlásához azért, hogy a következőkben több adattal tudjunk dolgozni.

A Model osztály neve után két kettősponttal (**::**) *statikus* metódus hívása történik meg, amiről annyit érdemes tudni, hogy (ez az Objektumorientált Programozás témaköréhez kapcsolódik) ilyen statikus metódusok akkor is meghívhatók, ha az osztályból nem hoztunk létre példányt. Viszont a további metódushívások már nyíllal (**->**) vannak hozzáfűzve az utasításhoz.

Háttérinformáció: *Mik is azok a trait-ek a Laravel-ben?*

Előfordulhat, hogy bizonyos üzleti logikai funkcionalitásokat egy olyan különálló, elszeparált fájlba szeretnénk kihelyezni, amelyet utána újra, többször is fel tudunk használni akár Model-ekben, akár Controller-ekben, így hozzáadva azokhoz bizonyos viselkedési formákat.

Ilyenre példa az imént használt **SoftDeletes**, amelyet a **Post** Model osztályon belül importáltunk, de ide került a **HasFactory** is. Ezek mindegyike egy-egy olyan funkcionalitás halmazt rendel a Model osztály példányaihoz, amely a viselkedésüket, működésüket befolyásolja. Mivel a Laravel nyílt forráskódú, ezeknek a keretrendszerbeli trait-eknek a tartalmát szabadon tudjuk böngészni és tanulmányozni.

Ha mi is szeretnénk saját trait-eket létrehozni, akkor azokat az **app / Traits** mappába tegyük meg. PHP fájlok legyenek és a **<?php** címke után a trait kulcsszóval kezdődjön, mintha egy osztályt (class) definiálnánk. Elnevezési konvenció miatt érdemes követni, de nem kötelező, hogy ezen fájlok neve *„Trait.php”*-ra végződjön: **app / Traits / ExampleTrait.php**

```php
<?php

trait ExampleTrait {
  public function exampleMethod() {
    return "Ez egy példa metódus az ExampleTrait-ből";
  }
}
```

6–4. kódrészlet: Példa a saját trait létrehozására Laravel-ben

Az őt felhasználó osztály pedig így nézhet ki:

```php
<?php

class ExampleClass {
  use ExampleTrait;
}

$example = new ExampleClass;
echo $example->exampleMethod(); // Kimenet: ez egy példa metódus az ExampleTrait-ből
```

6–5. kódrészlet: ExampleTrait felhasználása egy példa osztályban

11

Új parancsok: újdonságként érkezett négy `artisan make` parancs. Ezekkel **enum**, **class**, **interface** és **trait** elemeket lehet létrehozni a Laravel projektünkben.

A fenti példa alapján az **ExampleTrait**-et így lehetne létrehozni az új paranccsal:

`php artisan make:trait Traits/ExampleTrait`

Ennek köszönhetően az új trait-et a megfelelő helyre, könyvtárba (**app / Traits**), a megadott névvel és a megfelelő névtérrel generálja le nekünk a keretrendszer és nem kell manuálisan üres fájlt hozzáadnunk majd azt feltölteni a kezdeti helyes tartalommal.

6–2. újdonság: Új artisan parancsok az enum, class, interface, trait elemek létrehozásához

6.2.2.6. Összetett lekérdezés (SELECT, WHERE, ORDER BY)

Aki már használt az SQL-ben összetettebb lekérdezéseket, az tudja, hogy nem mindig a teljes adathalmazt akarjuk lekérni, hanem néha szűrten szeretnénk őket megkapni (mint például az előző alfejezet végén egy speciális lekérdezéssel a *„soft delete"* eljárással törölt blogbejegyzéseket) és valamilyen műveletet is elvégezhetünk velük. A WHERE kulcsszó segít minket az SQL-ben és nincs ez másként a Laravel-ben sem.

Például tudunk egy dátum mező (**published_at**) szerint szűrni. Ha az itt korábban bemutatott három blogbejegyzést hozzuk létre az adattáblában, akkor ez az alábbi egy olyan szűrés lehet, amely két (2-3.) bejegyzéseket fogja visszaadni egy gyűjteményben:

```
App\Models\Post::where('published_at', '>', '2023-04-10 17:00:00')->get();
```

6–9. utasítások: Szűrt eredménylista lekérése

A lekért (teljes vagy részleges) gyűjtemény a **posts** adattábla (összes vagy szűrt) sorait adja vissza. A gyűjtemény tartalmazza a példányokat, objektumokat, amelyek egy vagy több sort jelentenek az adattáblában (egy példány = egy sor).

De nem csak lekérni tudjuk őket és eltárolni egy gyűjteményben, hanem meg is tudjuk őket számolni (**count()**) vagy akár *„törölni"* frissítéssel (válaszként azt mondja meg a Tinker, hogy hány elemnél hajtotta végre az utasítást) és helyreállítani is, majd lekérni a gyűjtemény tartalmát, ezek láthatók az alábbi utasításokban.

```
$collection = App\Models\Post::where('published_at', '>', '2023-04-10 17:00:00');

$collection->count();

$collection->update(['deleted_at' => date('Y-m-d H:i:s') ]);

$collection->restore();

$collection->get();
```

6–10. utasítások: Szűrt eredménylista elemeivel műveletek végrehajtása

Így látható, hogy akkor is működik ez, ha nem csak egy adatsort (objektumot) kérünk le, hanem többet (kvázi egy gyűjteményt), és ezeket is tudjuk együttesen frissíteni, törölni, helyreállítani stb. Ha a **date()** metódusnak csak az 'Y-m-d' paramétert adnánk át, akkor ez csak a dátumot tartalmazná és pontosan éjfélre (00:00:00) állítaná be az időt.

Az ORDER BY SQL kulcsszavak is használhatóak az Eloquent lekérdezésekben. Az alábbi példák szerint kilistázhatjuk a blogbejegyzéseket létrehozásuk szerint a régebbit bejegyzésektől az újabbak felé haladva, majd a legutóbb publikáltat törölhetjük is.

```
App\Models\Post::orderBy('id')->get();

App\Models\Post::orderBy('published_at', 'desc')->first()->delete();
```

6–11. utasítások: Rendezett listázás és rendezés szerinti utolsó elem törlése

Az SQL ismeretek tehát itt is nagyon hasznosak lesznek, nem lehet őket figyelmen kívül hagyni, ha adatbázis-kezelésről van szó, mert bár a Laravel-ben nem kell konkrét SQL lekérdezéseket írnunk, de a mögöttes logikáját ismernünk kell ahhoz, hogy jól összefűzött metódusokkal a megfelelő adathalmazt kérdezzük le az objektumokba, gyűjteményekbe.

Megjegyzés: vegyük észre, hogy eddig a Post Model osztályunkat jóformán egyáltalán nem szerkesztettük, csak a *„soft delete"-es* technika megvalósítása miatt minimálisan, de minden egyéb dolgot ez az Eloquent osztály automatikusan nyújt nekünk.

6.2.2.7. Beszúrás kompakt módon (INSERT)

Egy blogbejegyzést a 6.2.2.2. alfejezetben bemutatott megoldáshoz képest létre tudunk hozni (be tudunk szúrni az adattáblába) egyszerűbben, mindössze egy utasítás végrehajtásával is.

```
App\Models\Post::create([
        'title' => '4th post',
        'slug' => '4th-post',
        'body' => 'content of 4th post',
        'published_at' => '2023-04-13 13:00:00'
]);
```

6–12. utasítások: Blogbejegyzés létrehozása Eloquent Model-en keresztül kompakt módon (egy utasítás több sorba tördelve)

Ennek működnie kellene, de még hibaüzenetet kapunk, amit a Laravel keretrendszer biztonsági beállítása miatt jelez nekünk. A Model mezőit (a tábla oszlopait) hozzá kell adnunk a kitölthető tulajdonsághoz a **Post** Model-ben:

```
protected $fillable = ['title', 'slug', 'body', 'published_at'];
```

6–6. kódrészlet: $fillable attribútum az Eloquent Model fájlban

Ez egy védelmi mechanizmus a Laravel-ben, amely miatt csak azok a mezők tölthetők fel adattal, amelyek itt a **$fillable** attribútumban szerepelnek. Azokat a mezőket kell hozzáadni, amelyek nem automatikusan generálódnak (**id, created_at, updated_at** stb.), ha kihagyok belőle olyan mezőt, amelyet az adatbázistábla kényszerei miatt elvárna (NOT NULL), akkor ugyanúgy hibát fogunk kapni az adatsor létrehozásakor (beszúrásakor).

A Tinker-t indítsuk újra és most már menni fognak a 6–12. utasítások, így létrejön az adattáblában a legújabb blogbejegyzés.

A **$fillable** tulajdonságot visszájára is fordíthatom, és az ellentettjét, a **$guarded** attribútumot hívhatom segítségül. Amikor ezt beállítom és egy üres tömböt adok neki értékül, akkor azt mondom a rendszernek, hogy nem védek egyetlen mezőt sem az adatfeltöltéstől, bármit elfogadok és feltöltöm az adattáblába, de azért látjuk, hogy ez nagyon veszélyes is egyben, hiszen ekkor minden felhasználótól érkező bemenetet elfogadok majd.

```
protected $guarded = [];
```

6–7. kódrészlet: Működő, kényelmes, de veszélyes megoldás a $guarded attribútummal az Eloquent Model fájlban

Egy tipikus megtámadása lehet ennek, ha valaki *„belecsempészi"* az **id** változó értékét is a küldött adatcsomagba, a rendszerünk pedig ellenőrzés nélkül fogja ezt fogadni, majd feldolgozni.

Próbáljuk ki, hogy ezzel a **$guarded** attribútummal is menni fog a mentés és megadhatjuk neki, hogy melyik mezőket nem kell védeni.

```
protected $guarded = ['id', 'created_at', 'updated_at', 'deleted_at'];
```

6–8. kódrészlet: Működő, biztonságos megoldás a $guarded attribútummal az Eloquent Model fájlban

A Tinker-t indítsuk újra és most már menni fognak a 6–12. utasításokat (kisebb módosítással: írjuk át a blogbejegyzések számozását 5-re), így létrejön az adattáblában a legújabb blogbejegyzés.

Összefoglalásként, a **$fillable** tulajdonságot akkor használjuk, ha meg akarjuk mondani a **create()** metódusnak, hogy mit *engedünk* feltölteni az adattáblába, a **$guarded** tulajdonságot pedig akkor használjuk, amikor meg akarjuk mondani a **create()** metódusnak, hogy mit *nem engedünk* semmiképpen feltölteni.

6.2.3. Lekérdezések naplózása

A fejezet feldolgozása során gyakorlatilag végig teszteléseket hajtottunk végre a Tinker alkalmazással, a létrehozott **posts** táblával és adataival.

Talán elsőre az Eloquent utasítások mögött meghúzódó SQL utasításokat nehezebb felismerni. Ehhez segítséget nyújt nekünk a Tinker-en keresztül a Laravel:

Először engedélyezzük az adatbázis lekérdezések naplózását, majd futtassunk példaként két Eloquent parancsot, és kérjük le a naplót:

```
DB::enableQueryLog();

App\Models\Post::orderBy('published_at', 'desc')->first();

App\Models\Post::where('published_at', '>', '2023-04-10 17:00:00')->update(['deleted_at' => date('Y-m-d') ]);

DB::getQueryLog();
```

6–13. utasítások: Naplózás engedélyezése, példák futtatása, napló lekérdezése

A köztes két Eloquent utasítást már alkalmaztuk korábban, így azok lefutásának eredményeit ismerjük, úgyhogy most koncentráljunk az utolsó utasításra, ami lekéri nekünk a napló tartalmát és egy gyűjteményt ad vissza:

```
= [
    [
      "query" => "select * from `posts` where `posts`.`deleted_at` is null order by `published_at` desc limit 1",
      "bindings" => [],
      "time" => 0.8,
    ],
    [
      "query" => "update `posts` set `deleted_at` = ?, `posts`.`updated_at` = ? where `published_at` > ? and `posts`.`deleted_at` is null",
      "bindings" => [
        "2023-04-13",
        "2023-04-13 17:31:21",
        "2023-04-10 17:00:00",
      ],
      "time" => 3.52,
    ],
  ]
```

6–3. ábra: Naplózott Eloquent (SQL) utasítások

Így akár tudjuk ellenőrizni azt is, hogy az adatbázis-kezelő rendszer menedzserében ott lefuttatva a lekérdezéseket megvizsgálhatjuk, hogy az elvárt eredményt kapjuk-e. A napló, azaz a gyűjtemény elemei az SQL lekérdezések és jellemzőik, kapcsolódó elemeik. Az első lekérdezés mellett nem található paraméter, csak a lefutási idő. A második naplózott utasításnál már vannak paraméterek, amelyeket sorban be lehet illeszteni a *„query”* részben lévő SQL utasítás kérdőjelei (**?**) helyére ahhoz, hogy az adatbázis-kezelő menedzser alkalmazásban is tudjuk futtatni a lekérdezést. Emiatt láthatjuk azt is, hogy bár mi csak a **deleted_at** mező értékét akartuk frissíteni az adattáblában, ez azzal járt, hogy az **updated_at** mező értéke is frissült, hiszen ez automatikusan történik meg az adatsor bármilyen változásakor.

Tipp: Teszteld a tudásod!

Bár még nem tudunk mindent a migrációról, de érdekes lehet kipróbálni az eddigi tudásunkkal, milyen teszteknek tudnánk megfelelni. Léteznek nyilvános repo-k, amelyekkel ezt a tesztelést meg tudjuk tenni, itt van például egy, ami a migrációs témakör alapjaihoz tartozik:

https://github.com/LaravelDaily/Test-Laravel-Eloquent-Basics

Ha klónozzuk a projektet és elindítjuk a tesztelést, akkor kezdetben minden tesztünk elbukik. De a GitHub oldalon lévő **Task** lista (a könyvtár- és fájlstruktúra alatt) pontosan megmutatja, hogy milyen fájlokat kell szerkeszteni ahhoz, hogy átmenjenek a tesztjeink.

Még nem tanultunk mindenről, így azokat nem kötelező figyelembe venni, amiről még nem tudunk, a saját tudásunk felmérésénél. Viszont kialakulhatott már annyi rutinunk, hogy esetleg a Laravel dokumentáció megfelelő részét használva is megoldható az, hogy átmenjenek sikeresen a fejlesztéseink az alkalmazás Eloquent osztályainak és metódusainak tesztesetein.

Az alfejezetben végrehajtott programkód-módosítások ebben a GitHub commit-ben érhetők el.

6.3. Query Builder alkalmazása

Az adatbázis-kezelés másik technikája a Laravel-ben a Query Builder, ami – ahogy a neve is jelzi – egy lekérdezés építésével valósulhat meg az alkalmazásban. Az SQL nyelv ismerete itt talán még hasznosabb,

mint az Eloquent esetében volt, mert amiket felépítünk lekérdezéseket, azok rettentően hasonlítanak majd az SQL nyelven megfogalmazott és felépített lekérdezésekre.

Ebben az alfejezetben számos példát mutatok arra, hogy hogyan lehet használni ezeket a lekérdezéseket, de a legteljesebb listát mindig a Laravel kapcsolódó dokumentációjában találhatjuk meg. Az Eloquent-es példákat a Tinker alkalmazás segítségével mutattam be, itt azonba már az alkalmazásunkba fogom beépíteni a Query Builder által visszakapott eredményeket (gyűjteményeket, objektumokat stb.). De fontos azt is kiemelni, hogy az ott használt Eloquent lekérdezések ugyanilyen módon alkalmazhatók lennének a Laravel alkalmazásban is, nem csak a Tinker-ben.

Az Eloquent-tel ellentétben, a Query Builder használatához nincs szükség külön Model osztályokra, mert ezeknél az úgynevezett Facade[8] lesz segítségünkre, ami a Laravel keretrendszer majdnem az összes funkcionalitásához biztosít ilyen hozzáférést. A **DB** Facade az adatbázis-kezeléshez biztosít eszközkészletet, kvázi mint egy *„minden részletre kiterjedő segédosztály”,* aminek a funkcionalitásait alkalmazva tudjuk hatékonyan kezelni az adatbázisokat. A DB Facade alkalmazása egy lekérdezés építéséről szól a továbbiakban, amelynél a **table()** metódussal jelöljük ki mindig, hogy melyik táblával szeretnénk elsődlegesen foglalkozni. Lekérdezések építésénél pedig a **get()** metódussal egy gyűjteményt kapunk vissza, a **first()** metódussal egy adott objektumot (adattábla sorának értékeivel).

6.3.1. Lekérdezések (SELECT)

Query Builder-t leggyakrabban lekérdezések építésére szoktuk használni. Ebben az alfejezetben számos gyakorlati példát megnézünk ezzel kapcsolatban.

6.3.1.1. A legalapvetőbb lekérdező metódusok (get, first, find, table, select)

A legegyszerűbben ezt a **routes / web.php** fájlban tudjuk kipróbálni. A **dd()** a Laravel keretrendszer egyik leghasznosabb metódusa, amelyet hibakeresésre tudunk használni, a **dd** rövidítés a *„dump & die”*-ra utal, amivel mindenféle elemet (szöveget, objektumot, gyűjteményt stb.) ki tudunk íratni és láthatjuk az értékét, értékeit.

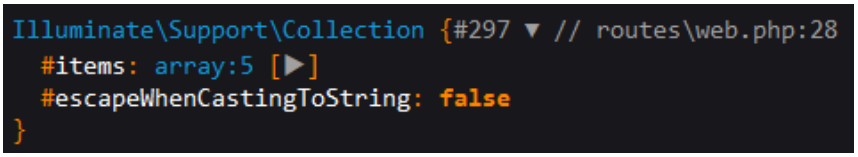
Megjegyzés: ha nem szeretnénk megállítani (die) az oldal működését, akkor a **dd()** helyett használhatjuk a **dump()** metódust is, és akkor a további (utána lévő) kódok is le fognak futni.

```
// importáljuk a DB Facade-ot
use Illuminate\Support\Facades\DB;

Route::get('/posts', function () {
  $posts = DB::table('posts')->get();
  dd($posts);
});
```

6–9. kódrészlet: Első egyszerű lekérdezés a Query Builder-rel

[8] Ezeknek a háttér működéséről bővebben majd a 12.2.4. alfejezetben lesz szó, itt egyelőre annyit elég tudni a Facade-okról, hogy ezek egy statikus felületet biztosítanak azokhoz a Laravel osztályokhoz, amelyek különböző szolgáltatásokat tesznek elérhetővé a Laravel funkciói közül A DB Facade például abban segít minket, hogy az adatbázissal kapcsolatos műveleteket egyszerűen és hatékonyan tudjuk elvégezni.

Az http://localhost:8000/posts/ útvonal betöltésével a böngészőben megkapjuk a következő eredményt:



6–4. ábra: Az egyszerű lekérdezés eredménye

Látható, hogy a **get()** metódushívás eredményéül egy nem-Eloquent specifikus gyűjteményt (**collection**) kaptunk vissza, amelynek az elemeit (**items** tömb) is meg tudjuk tekinteni, ha a sor jobb szélén lévő kis háromszögre kattintunk. Ezután pedig az egyes elemek részleteit is meg tudjuk nézni.

A fenti utasításban nincs szó példányosításról, csak a Model osztály segítségével lekérünk az adatbázisból kettő darab adatsort, amit visszaadunk egy gyűjteményként és ezt kapja meg a **$posts** változónk (PHP-ban ugye van ilyen szabadságunk, hogy egy változónak sokféle dolgot eredményül adhatunk), itt most egy gyűjteményt adtunk meg neki, ami listaként vagy tömbként is felfogható és kezelhető úgy.

6–5. ábra: Az 5. blogbejegyzés részleteinek tallózása

Ha ebből a gyűjteményből szeretnénk megkapni csak a blogbejegyzések címét, akkor használhatjuk a **pluck()** metódust, amely egy hasznos gyűjtemény-kezelő:

```
dd($posts->pluck('title'));
```

6–10. kódrészlet: Blogbejegyzések címeinek kiíratása a gyűjtemény mezőszűrésével

Az így kapott eredmény szintén egy gyűjtemény:

6–6. ábra: Gyűjtemény szűrése egy konkrét mező (title) értékeire

Ha a 6–9. kódrészletben lévő **get()** metódushívást **first()**-re cseréljük, és a **pluck()** metódushívást elhagyjuk a kiíratásnál, akkor az *items* tömb legelső elemének részleteit kapjuk vissza eredményül a böngészőben, tehát a gyűjtemény legelső objektumát. Ekkor, ha nem az egész objektummal (adatsorral) szeretnénk dolgozni, akkor lekérhető objektum mezőként bármelyik attribútuma az alábbi szerint:

```
dd($posts->title);
```

6–11. kódrészlet: Objektum mezőjének kiírása

Ha nem a legelső elemet szeretnénk visszakapni a gyűjteményből, akkor **id** alapján is tudunk szűrni például a 4. elemre így:

```
$posts = DB::table('posts')->find(4);
```

6–12. kódrészlet: A find() segédmetódus használata

Ha olyan azonosítóval (**id**) rendelkező sort szeretnénk lekérni, amelyik nem létezik (például a 99.-et), akkor a böngészőben null értéket kapunk vissza, ami alapesetben nem okozna gondot, de ha a kódunk nincs felkészítve arra, hogy null is lehet a lekérdezés eredménye és például annak szeretnénk kiíratni a **title** mezőjét, akkor már **Exception**-t fogunk kapni, mivel a **null**-nak nincsen **title** mezője.

Ha nem szeretnénk a teljes adatsorokat lekérni, akkor a **select()** metódusnak megadhatjuk, hogy mely mezőkre van konkrétan szükségünk.

```
$posts = DB::table('posts')->select('title', 'body')->find(4);
```

6–13. kódrészlet: Mezők szerint szűrt adatok (select()-tel)

De használható ugyanerre a **find()** metódus második, tömb paramétere is:

```
$posts = DB::table('posts')->find(4, ['title', 'body']);
```

6–14. kódrészlet: Mezők szerint szűrt adatok (find()-dal)

Előfordulhat, hogy csak valamilyen összesítő információra van szükségünk, ekkor használhatjuk az SQL nyelvből már jól ismert MIN, MAX, COUNT, SUM stb. *összesítő (aggregáló) függvények* valamelyikét az adat összegyűjtésére, majd utána kiírhatjuk. Ez a példa kód a blogbejegyzések számát írja ki nekünk:

```
Route::get('/posts-count', function () {
  $postsCount = DB::table('posts')->count();
  dd($postsCount);
});
```

6–15. kódrészlet: Összesítő függvény alkalmazása az adatoknál

6.3.1.2. Eloquent és Query Builder lekérdezések összehasonlítása

Vizsgáljuk meg, hogy milyen egy Eloquent specifikus gyűjtemény a webes alkalmazásban megjelenítve és hasonlítsuk össze a korábbi Query Builder által visszaadott gyűjteménnyel (például a 6–5. ábra mutat ilyet). Bővítsük az útvonalainkat az alábbival:

```
Route::get('/eloquent-post', function () {
  $post = Post::where('id', 3)->get();
  dd($post);
});
```

6–16. kódrészlet: Eloquent specifikus gyűjteményre példa a webalkalmazásban

A Post Model osztály importálásáról ne feledkezzünk meg az útvonal használata előtt. Az útvonal lekérése után a böngészőben egy sokkal részletesebb gyűjteményt kapunk vissza, amelynek a releváns részeit *„kinyitothatjuk a háromszögekre kattintással”.*

Ezen látható (6–7. ábra), hogy az Eloquent specifikus gyűjtemény egy eleme nem csak a puszta mező értékeket tartalmazza, hanem magát az Eloquent osztályt és annak (a mezők értékein túl) a releváns információit, például, hogy milyen az adatbáziskapcsolat, melyik táblához kapcsolódik, mi annak az elsődleges kulcsa, mely mezői tölthetők fel az adattáblának (**$fillable**) vagy hogy engedélyezve van-e **forceDeleting**, ami kvázi a *„soft deleting”* ellentéte.

6–7. ábra: Eloquent specifikus gyűjtemény eleme

6.3.1.3. Szűrés (WHERE)

A WHERE feltételek alkalmazásával lehet az adathalmazt szűrni. Alapesetben, ha a lekérdezés felépítésénél több **where()** segédmetódust használunk, akkor AND, tehát ÉS kapcsolattal köti össze a logikai feltételeket, így a szűrt eredmények között csak azok jelennek meg, amelyekre mindegyik igaz.

```php
Route::get('/filtered-posts', function () {
  $posts = DB::table('posts')
    ->where('published_at', '>', '2023-04-09')
    ->where('published_at', '<', '2023-04-11')
    ->whereNotNull('updated_at')
    ->get();
  dd($posts);
});
```

6–17. kódrészlet: Szűrt blogbejegyzések lekérdezése

A 6–17. kódrészletben több feltételt is ÉS kapcsolattal kötöttem össze, így mindegyiknek érvényesnek kell lennie az adathalmaz szűrt soraira. A harmadik **whereNotNull()**-os feltétel azt jelzi, hogy az **updated_at** mező nem lehet NULL az adatsorban, így mindenképpen csak a szerkesztett sorokat fogjuk visszakapni, amelyek a publikálás dátuma a megadott két érték közé kellett, hogy essen. A Laravel dokumentáció tanulmányozása során feltűnhet, hogy ilyen esetekben használható lenne egy egyszerűsítés is a **whereBetween()** metódussal, ami a szűrendő mező megnevezése után egy kételemű tömbben várja a minimum és maximum értékeket. Ez az egyszerűsítés a következő szerint történhet meg:

```php
Route::get('/filtered-posts', function () {
  $posts = DB::table('posts')
    // ->where('published_at', '>', '2023-04-09')
    // ->where('published_at', '<', '2023-04-11')
    ->whereBetween('published_at', ['2023-04-09', '2023-04-11'])
    ->whereNotNull('updated_at')
    ->get();
  dd($posts);
});
```

6–18. kódrészlet: Újraszervezés: whereBetween alkalmazása dátumokra és számszerű értékekre is működik

A 6–18. kódrészletben látható, hogy a kommentelés így is működik, annak ellenére, hogy csak egyetlen utasítás van több sorba tördelve.

Ha azt szeretnénk, hogy az **ÉS** kapcsolat helyett **VAGY** kapcsolat legyen a feltételek között, akkor a **where()** helyett az **orWhere()** metódust kell használni.

```php
Route::get('/filtered-posts', function () {
  $posts = DB::table('posts')
    ->whereBetween('published_at', ['2023-04-09', '2023-04-11'])
    ->orWhere('title', 'LIKE', '%th%')
    ->get();
  dd($posts);
});
```

6–19. kódrészlet: OR (VAGY) kapcsolat a feltételek között és példa a LIKE használatára

Az OR feltétel összekapcsoláson túl, ez az utóbbi példa a LIKE használatára is példát mutat nekünk a lekérdezéseknél, ami egy mintaillesztést szemléltet: csak azok a blogbejegyzések jelenjenek meg, amelyek nevében szerepel valahol a *„th”* (angol sorszámozásra) utaló két betű.

Az eddig bemutatott módokon kívül, még számos módon lehet használni a **where** feltételeket, érdemes a konkrétumok áttanulmányozása a Laravel keretrendszer kapcsolódó dokumentációjában.

Újdonság a Laravel 10-ben: még a Laravel 11 kiadása előtti egyik utolsó nagyobb frissítés kapcsán a Laravel 10.47-es verziójában megjelent két új kulcsszó. Ezek akkor a leghasznosabbak, ha egy keresési funkcionalitást szeretnénk megvalósítani az alkalmazásunkban mindössze egyetlen keresési bemeneti mezővel, amelybe keresési szöveget adhatunk meg, és így akár több adattábla oszlop tartalmában is együttesen tudunk keresni.

- **whereAll()**: az összes benne definiált feltételnek teljesülnie kell (**AND**)
- **whereAny()**: legalább egy benne definiált feltételnek teljesülnie kell (**OR**)

További információk és példakódok az új szűrési kulcsszavak használatáról itt találhatók: https://github.com/laravel/framework/pull/50344

6.3.1.4. Csoportosítás (GROUP BY és a HAVING) és az összesítő (aggregate) függvények

Adatainkat tudjuk csoportosítani és valamilyen összesítő, kalkulált információt tudunk kinyerni belőlük. Jelenleg az adataink még nem túlságosan szerte ágazóak és főleg dátum típusú mezőket tartalmaznak, emiatt egy olyan példán keresztül mutatom be ezt a lekérdezést, ami szintén a dátumokkal kalkulál.

Építsünk egy lekérdezést azért, hogy kiszámoljuk, adott napon mennyi blogbejegyzés került publikálásra.

```
Route::get('/posts-by-day', function () {
  $posts = DB::table('posts')
    ->select(DB::raw('DATE(published_at) as day, COUNT(*) AS posts_by_day'))
    ->groupBy('day')
    ->get();
  dd($posts);
});
```

6–20. kódrészlet: Blogbejegyzések száma naponta

Ekkor ugye a **select()** metódusba csak olyan mezők kerülhetnek be, amelyek a **groupBy()**-ban is szerepelnek, vagy az azokhoz tartozó kalkulált értékek. A példában a publikálás dátumából a **DB::raw()** metódus segítségével számíthatunk ki értékeket, mint például a sorok számát (COUNT(*)), és hívhatunk SQL specifikus függvényeket, mint például a DATE(), ami a dátum típusú mezőből elhagyja az idő részt, így napokat generál a **published_at** mezőből, amit ezekután már használhatunk a csoportosítás (GROUP BY) során.

Ha nem csak egy-egy mező nevét szeretnénk használni a **select(), where(), groupBy()** és még további építési segédmetódusokon (**having(), orderBy()**) belül, akkor alkalmazhatjuk őket a **Raw** utótaggal, ami után nem kell külön a **DB** Facade-on keresztül elérni a leírandó SQL utasításrészt. Így az épített lekérdezés **select()** eleme átalakítható eszerint:

```
Route::get('/posts-by-day', function () {
  $posts = DB::table('posts')
  //->select(DB::raw('DATE(published_at) as day, COUNT(*) AS posts_by_day'))
```

```
    ->selectRaw('DATE(published_at) as day, COUNT(*) AS posts_by_day')
    ->groupBy('day')
    ->get();
  dd($posts);
});
```

6–21. kódrészlet: selectRaw() metódus használata

A HAVING utasítás ugyanúgy hozzáfűzhető, ahogy az az SQL nyelv iránymutatása szerint használható, tehát már a csoportosítás során elvégzi a szűrést, nem csak utána. Ennek a Laravel-en belüli működése leginkább a **where()**-re hasonlít, hiszen itt is valamilyen feltételt tudunk megfogalmazni, csak a fókusza nem akármelyik mezőn van, hanem a csoportosítás során alkalmazott mező(kö)n. A **where()**-rel való hasonlósága miatt itt is használható például a **havingBetween()**, amely két érték közötti feltétel szerint szűri majd le az összekapcsolás alapjául szolgáló mező értékeit.

```
Route::get('/posts-by-day', function () {
  $posts = DB::table('posts')
  //->select(DB::raw('DATE(published_at) as day, COUNT(*) AS posts_by_day'))
    ->selectRaw('DATE(published_at) as day, COUNT(*) AS posts_by_day')
    ->groupBy('day')
    ->having('day', '<>', '2023-04-10')
    ->get();
  dd($posts);
});
```

6–22. kódrészlet: Csoportosítás alapját képező mező értékeinek szűrésére példa

6.3.1.5. Rendezés (ORDER BY)

A rendezés használható az **orderBy()** segédmetódussal, amelynek egy mező nevét kell megadni első paraméterében, illetve a második paraméterben a rendezés irányát (növekvő – **asc**, csökkenő – **desc**, alapértelmezetten, ha nem adjuk meg, akkor növekvő sorrendben fogja rendezni a gyűjteményeket).

```
Route::get('/latest-posts', function () {
  $posts = DB::table('posts')
    ->orderBy('published_at', 'desc')
    ->get();
  dd($posts);
});
```

6–23. kódrészlet: Blogbejegyzések rendezése

A legújabb blogbejegyzést itt is lekérhetjük és megjeleníthetjük, nem csak a már ismert Eloquent technikával. A **latest()** lekérdezést építő metódus alapértelmezetten a **created_at** mező alapján fogja rendezni csökkenő sorrendben a gyűjteményünket (tehát, mintha azt írnánk helyette: **orderBy('created_at' 'desc')**, csak a **latest()** sokkal rövidebb önmagában), utána a **first()** metódussal pedig kiválasztjuk közülük a legelsőt, így a legfrissebb blogbejegyzést kapjuk meg. Arra is lehetőségünk van, hogy a **created_at** mező helyett másik timestamp alapú mezőt használjuk a rendezésre, ekkor csak meg kell a **latest()**-nek adni paraméterül a mező nevét.

```
Route::get('/latest-post', function () {
```

```
  $post = DB::table('posts')
    ->latest()
    ->first();
  dd($post);
});
```

6–24. kódrészlet: Legfrissebb blogbejegyzés lekérése

A **latest()** ellentéte az **oldest()**, amely alapértelmezetten a **created_at** mező alapján növekvő sorrendbe rendezi a visszaadott adathalmazt.

Arra is lehetőségünk van, hogy a gyűjtemény egy véletlenszerűen kiválasztott darabját kérjük le, például egy random blogbejegyzést akarunk ajánlani majd a felhasználóinknak, akkor az **inRandomOrder()** építési segédmetódus áll a rendelkezésünkre, amely az alábbi példa szerint használható:

```
Route::get('/random-post', function () {
  $post = DB::table('posts')
    ->inRandomOrder()
    ->first();
  dd($post);
});
```

6–25. kódrészlet: Véletlenszerű blogbejegyzés lekérése

Ha most frissítjük a böngészőben ezt a címet: http://localhost:8000/random-post akkor mindig másik blogbejegyzés részleteit fogja visszaadni.

6.3.1.6. Összekapcsolás (JOIN)

A táblák összekapcsolása is nagyon hasonlóan működik a Laravel-ben, mint az SQL utasítások esetén. Kulcsok egyezősége szerint tud a leghatékonyabban működni, bár más mezők mentén is összeilleszthetők lennének a táblák, de ha a mezők nem lennének kulcsok, akkor az lassítaná a folyamatot.

Az adattáblák összekapcsolásával és a kulcsokkal a 6.4. alfejezetben foglalkozok, így a **join()** (INNER JOIN) művelet bemutatására is után térek ki.

6.3.1.7. Ellenőrzés, lekérdezés tesztelés

Ha azt gondoljuk, hogy jól építettük fel a lekérdezéseket, de valami mégsem működik megfelelően, akkor meg tudjuk nézni a **get()** eredménylista vagy a **first()** eredmény helyett magát az SQL utasítást a **toSql()** metódus alkalmazásával.

```
Route::get('/latest-posts-sql', function () {
  $posts = DB::table('posts')
    ->orderBy('published_at', 'desc')
    ->toSql();
  dd($posts);
});
```

6–26. kódrészlet: SQL lekérdezés kinyerése a Query Builder-ből

A **dd()** segédmetódussal kiírható a lekérdezés eredménye, ahogy az látható is az alábbi ábrán.

6–8. ábra: SQL lekérdezés kiíratása

Az SQL lekérdezés kinyerése után ez lefuttatható bármelyik adatbázis-kezelő menedzser alkalmazásban és ellenőrizhető, hogy tényleg ugyanazt az eredményt kaptuk-e meg, mint amit elvártunk előtte.

Ha az SQL utasításban lenne egy paraméter érték, például egy szám vagy egy szöveg, akkor azt külön jelezné nekünk a **toSql()** egy kérdőjellel a paraméter helyén, ami után külön le kellene kérnünk a parméter értékét. Ezzel szemben hasznosabb, ha a **toRawSql()** segédmetódust használjuk, mert akkor rögtön meg tudjuk kapni az adott paraméternek az értékét is. Szemléltetésül itt van a két paraméteres lekérdezés futtatása a Tinker-ben:

```
> DB::table('posts')->where('id', 25)->toSql();
= "select * from `posts` where `id` = ?"

> DB::table('posts')->where('id', 25)->toRawSql();
= "select * from `posts` where `id` = 25"
```

6–9. ábra: Paraméterrel rendelkező lekérdezések SQL utasításainak kinyerése toSql() és toRawSql() segédmetódusokkal

Ha szeretnénk megkapni a paraméterek értékeit, akkor a fenti két segédmetódus helyett (vagy mellett) használhatjuk a **getBindings()** segédmetódust is, amely egy tömbben visszaadja a paraméterek konkrét értékeit. De ebben a már megismert és sokat használt **dd()** metódus is segíthet nekünk:

```
> DB::table('posts')->where('id', 25)->dd();
"select * from `posts` where `id` = ?" // vendor\laravel\framework\src\Illuminate\Database\Query\Builder.php:3935
array:1 [
  0 => 25
] // vendor\laravel\framework\src\Illuminate\Database\Query\Builder.php:3935
```

6–10. ábra: Paraméterrel rendelkező lekérdezés SQL utasításának kinyerése dd() segédmetódussal

Kiíratás menet közben: ahogy láttuk a Laravel 10 későbbi verziójában és a Laravel 11-ben már azt is megtehetjük, hogy nem külön utasításban írjuk ki a számunkra érdekes információkat.

Ezt úgy tehetjük meg, hogy az összefűzött metódusláncba beillesztjük a **dump()** vagy **dd()** metódusokat és a kiíratás rögtön megtörténik, például így:

DB::table('posts')->inRandomOrder()->**dump()**->first();

Ez az utasítás először kiírja az adatbázis SQL lekérdezést, majd végre is hajtja és a legelső elemet kiválasztja a gyűjteményből.

De a Laravel 11 egy **Dumpable** nevű **trait**-et is ad nekünk, amely két metódussal: **dd()** és **dump()** rendelkezik. Ezt a két metódust azokban az osztályokban, amelyek importálják a

trait-et, felül is tudjuk definiálni, és egy kiíratást tudunk végezni az osztály használata közben.

6–3. újdonság: Részlekérdezés eredményének kiírása további végrehajtás előtt

Ezzel szintén meg tudjuk nézni ugyanúgy, hogy mi a háttérben *„meghúzódó"* SQL lekérdezés és annak mi a paramétere. Egyetlen *„probléma"* vele itt, hogy ezzel a Tinker-ből ki is léptet minket a rendszer.

A segédmetódusok mindegyike működik Eloquent-es (Model-en keresztüli) lekérdezés és Query Builder-es lekérdezés alkalmazása esetén is.

6.3.2. Adatmanipuláció Query Builder-rel

Adatok beszúrását, frissítését és törlését is el tudjuk végezni a Query Builder segítségével. Ezt tekintem át példákon keresztül ebben az alfejezetben.

6.3.2.1. Beszúrás (INSERT)

A Query Builder-rel egy, illetve több adatsor is beszúrását is el tudjuk végezni, használjuk hozzá az **insert()** építési segédmetódust. Arra kell mindenképpen figyelni, hogy az adattábla kötelező és alapértelmezett érték nélküli mezőinek mindenképpen adjunk értéket a beszúró műveletben. Beszúrásnál a **created_at** és **updated_at** is csak NULL értéket kapnak alapértelmezetten.

A beszúrás teszteléséhez használjuk a *FakerPHP* lehetőségeit, mivel ezt alapértelmezetten tudja kezelni a Laravel (tehát nem kell telepíteni hozzá semmit) és egy **fake()** metóduson keresztül számos tesztértéket tudunk generálni a működéshez. Bővebben a FakerPHP tesztértékeiről itt tudunk tájékozódni: https://fakerphp.github.io/

```
Route::get('/insert-post', function () {
  DB::table('posts')->insert([
    'title' => fake()->sentence(1),
    'slug' => fake()->slug(2),
    'body' => fake()->paragraph(3),
    'published_at' => fake()->dateTime(),
    'created_at' => now(),
    'updated_at' => now(),
  ]);
});
```

6–27. kódrészlet: Egy blogbejegyzés beszúrása

A **fake()** segédmetódussal tudunk létrehozni – a teljesség igénye nélkül példa mondatokat, neveket, egyedi e-mail címeket, URL-eket, dátumokat és még rengeteg mindent, akár lokalizáltan (például magyar neveket használva), amire csak szükségünk lehet. A megismeréséhez mindenképpen javaslom a fenti címen áttekinteni a lehetőségeit és a keresőt használva példákat keresni.

```
Route::get('/insert-posts', function () {
  $postsCount = fake()->randomDigitNotNull();
  $posts = [];
  for ($i=0; $i < $postsCount; $i++) {
```

```
    $posts[] = [
      'title' => fake()->sentence(1),
      'slug' => fake()->slug(2),
      'body' => fake()->paragraph(3),
      'published_at' => fake()->dateTime(),
      'created_at' => now(),
      'updated_at' => now(),
    ];
  }
  DB::table('posts')->insert($posts);
});
```

6–28. kódrészlet: Több (1-9 közötti darab) blogbejegyzés létrehozása

A böngészőben betölthetjük ezeket az útvonalakat és az adatbázistábla tartalmát lekérve láthatjuk, hogy hogyan bővül tesztadatokkal.

6.3.2.2. Frissítés (UPDATE)

A frissítésnél arra érdemes ügyelni, hogy jól fogalmazzuk meg azt a **where()**-ben lévő feltételt, amellyel bizonyos sorra, sorokra szűrni szeretnénk a frissítés végrehajtása előtt. Utána maga a frissítés már könnyedén megy.

Példaként frissíthetjük azokat a blogbejegyzéseket, amelyeket véletlenszerűen generált adatokkal töltöttünk fel az előző alfejezetben. Koncentráljunk a publikálás dátumára és nézzük meg, hogy milyen értékek kerültek bele a **posts** táblánkba! Alább látható az én táblámról egy részlet:

id	title	slug	body	published_at	created_at	updated_at	deleted_at
11	Sit ullam.	adipisci-numquam	Expedita placeat ut deleniti quod natus. Quis hi…	1995-10-19 00:08:17	2023-04-23 12:04:43	2023-04-23 12:04:43	NULL
12	Aspernatur.	corrupti-aperiam-consequatur	Deleniti velit aut omnis quisquam minima enim. M…	2006-06-06 21:40:28	2023-04-23 12:04:52	2023-04-23 12:04:52	NULL
13	Iste nemo.	pariatur-et-id	Sunt aut rerum deleniti. Aliquam quia laboriosam…	2014-11-28 20:41:25	2023-04-23 12:04:52	2023-04-23 12:04:52	NULL
14	Est.	aspernatur-ipsa-sit	Voluptatum eaque voluptatem assumenda omni…	1996-08-14 17:18:04	2023-04-23 12:04:52	2023-04-23 12:04:52	NULL
15	Quia quasi.	pariatur-et	Voluptatem aut inventore numquam repellendus…	1979-01-08 08:17:29	2023-04-23 12:04:52	2023-04-23 12:04:52	NULL
16	Molestiae.	quia-reiciendis-perspiciatis	Nesciunt suscipit et aut nihil magnam soluta nes…	2022-10-25 20:48:06	2023-04-23 12:04:52	2023-04-23 12:04:52	NULL
17	Est.	facere-praesentium	Exercitationem nisi expedita labore accusantium…	2021-06-11 19:18:44	2023-04-23 12:04:52	2023-04-23 12:04:52	NULL
18	Consequatur.	est-quia	In odio quibusdam accusantium adipisci quos. C…	1993-06-18 05:17:39	2023-04-23 12:04:52	2023-04-23 12:04:52	NULL
19	Quia qui.	illum-quo-in	Voluptatum impedit sed accusantium libero solut…	2003-07-04 19:08:39	2023-04-23 12:04:52	2023-04-23 12:04:52	NULL
20	Quia.	omnis-exercitationem-sapiente	Earum quisquam dolores sapiente architecto eu…	1990-08-14 06:26:52	2023-04-23 12:04:52	2023-04-23 12:04:52	NULL

6–11. ábra: A posts adattábla részlete

Bekerült nálam bőven 2000. előtti blogbejegyzés is, miközben lehet, hogy akkor még el sem indult az alkalmazásban kezelt blog. Frissítsük fel az **update()** utasítással az iménti beszúrásoknál létrehozott adatsorok (a **created_at** mező értéke nagyobb mint a tegnapi nap 24 órás értéke) publikálási dátumait mostanra az új útvonal meglátogatásával:

```
Route::get('/update-posts', function () {
  DB::table('posts')
    ->where('created_at', '>', now()->subDays(1)->endOfDay())
    ->update(['published_at' => now()]);
});
```

6–29. kódrészlet: Random blogbejegyzések publikálási dátumainak frissítése mostanra

6.3.2.3. Törlés (DELETE)

A törlésnél szintén érdemes arra figyelni, hogy jól fogalmazzuk meg a feltételt, mert ha nem adjuk meg helyesen, akkor értékes adatsorokat törölhet nekünk az alkalmazás. Törléshez használjuk a **delete()** építési segédmetódust.

Ha például azt szeretnénk elérni, hogy a legutóbbi blogbejegyzést töröljük, akkor rendezhetjük a bejegyzéseket létrehozás szerinti azonosítóval csökkenő sorrendbe és lekérhetjük a **limit()** segédmetódussal az utolsó egy darabot közülük.

```
Route::get('/delete-latest-post', function () {
  DB::table('posts')
    ->orderBy('id', 'desc')
    ->limit(1)
    ->delete();
});
```

6–30. kódrészlet: Legutóbbi blogbejegyzés törlése

Az iménti kódrészletben bár nem adtunk meg konkrétan feltételt a **where()**-ben, mivel nem is használtuk, de az **orderBy()** és a **limit()** kombinációja funkcionalitásban megegyezett ezzel.

Ha az adattábla teljes tartalmát szeretnénk törölni, akkor használhatjuk a következő két utasítás valamelyikét az útvonalon belül:

```
Route::get('/delete-posts', function () {
  DB::table('posts')->delete();
  DB::table('posts')->truncate();
});
```

6–31. kódrészlet: posts tábla kiürítése

A **delete()** és a **truncate()** is kitörli a tábla teljes tartalmát. A különbség köztük annyi, hogy ha **delete()**-tel törlünk, akkor a legutoljára beszúrt sor id mezőjének az értékéhez képest fog eggyel növekedni az újonnan beszúrt sor **id** mezője. A **truncate()** esetén újraindul az **id** számozása 1-től újabb sorok beszúrásánál.

Az alfejezetben végrehajtott programkód-módosítások ebben a GitHub commit-ben érhetők el.

6.4. Adatkapcsolatok

Eddig az adattáblákat önmagukban vizsgáltuk, kértük le az adataikat és manipuláltuk őket. Innentől kezdve viszont megismerkedünk az adatkapcsolatokkal, így kettő vagy akár több tábla együttes kezelését is elsajátíthatjuk. Az adatkapcsolatok minden részletre kiterjedő áttekintését a Laravel dokumentációjában itt találjuk.

6.4.1. Legegyszerűbb adatkapcsolat (1-1)

Bár a valós életben talán ritkábban használunk 1-1-es kapcsolatot, hiszen ez helyettesíthető azzal, hogy az egyik tábla mezőit elhelyezzük a másik táblában. Viszont előfordulhat, hogy a két tábla elemeit külön-külön szeretnénk menedzselni, mivel sok mezőt tartalmazhatnak, ekkor lehet létjogosultsága ennek a kapcsolati formának.

A legegyszerűbb adatkapcsolat két Model osztály és ezáltal két adattábla között, az 1-1 kapcsolat. Ami annyit tesz, hogy mindkét oldalról 1-1 szereplő fog részt venni a kapcsolatban. Ha a megértést segítendő valós életbeli példát szeretnék mondani, akkor biztos azzal kezdeném, hogy minden egyes (14+) embernek van egy darab személyazonosító igazolványa és fordítva is igaz ez, mert egy ilyen igazolvány egy személyhez tartozik. Ha ezt a terminológiát átültetjük a mi blog oldalunkra, akkor azt mondhatnánk, hogy minden egyes blogbejegyzésnek van egy konkrét értékelése (angolul rating) és egy értékelés (az összes adatával együtt) egyetlen bejegyzéshez tartozik.

Hozzuk létre az értékelések menedzselését végző Eloquent Model osztályt:

```
php artisan make:model Rating -m
```

6.4.1.1. Eloquent kapcsolat

Az -m kapcsolóval rögtön migrációs fájlt is készít nekünk a rendszer a **Rating** osztályhoz. De még mielőtt módosítanánk az értékelés migrációs fájlját, előbb bővítsük ki a **Post** Model osztályt a következő funkcióval:

```
public function rating()
{
  return $this->hasOne(Rating::class);
}
```

6–32. kódrészlet: Kapcsolatot képviselő metódus

Ezzel a metódussal fogjuk majd lekérni az adott blogbejegyzéshez (**Post** osztály példányához) tartozó értékelést (**Rating** osztály példányát).

6.4.1.2. Idegen kulcsot képviselő mező

Ahhoz, hogy ezt meg tudjuk tenni, az adatbázisban meg kell lennie a **ratings** táblának, ami a migrációs fájljával tudunk létrehozni. Adjuk hozzá az értékelés pontszámát (**score**) és *egy idegen kulcs (illetve egy annak megfelelő típusú és nevű!) mezőt* az **[időbélyeg]_create_ratings_table.php** fájlban lévő osztályhoz (az **id** után a **timestamps()** elé szúrjuk be ezt a két sort a **create()** metóduson belül:

```
$table->double('score');
$table->unsignedBigInteger('post_id')->unique();
```

6–33. kódrészlet: ratings tábla bővítése a migrációs fájllal

*Ezzel még nem hoztuk létre a kapcsolatot a migrációs fájl segítségével, csak egy olyan mezőt hozunk létre, ami típusában és nevében alkalmas lesz arra, hogy idegen kulcsként tudjon működni majd a jövőben, de maga az idegen kulcs **hivatkozás** itt még nem jön létre.*

Egy kicsit ismertetném, hogy mi is az az **idegen kulcs**: ez gyakorlatilag egy olyan mező, amin keresztül kapcsolatot tudunk teremteni egy másik adattábla (jelen esetben: 1 sor lesz kapcsolatban a másik tábla 1 sorával, ezért 1-1 kapcsolatnak hívjuk) adott sorával annak **id** mezőjén keresztül.

Ahogy azt már tudjuk, a Laravel-ben vannak névkonvenciók, szabályok, amelyeket itt is érdemes betartani ahhoz, hogy a rendszer úgy működjön, ahogy azt mi elvárjuk és a legnagyobb segítséget nyújtsa nekünk.

Kezdjük az idegen kulcsnak szánt mező típusával: **unsignedBigInteger**, ez azt jelenti, hogy előjel nélküli (tehát nem negatív) egész számokat lesz képes eltárolni ez a mező az adattáblájában. *Miért fontos ez?* Azért, mert az adatkapcsolathoz szükséges mező típusának meg kell egyeznie annak a mezőnek a típusával, amire hivatkozik majd. Amikor pedig a migrációs fájlokban kiadjuk ezt az utasítást a **create()** metóduson belül: **$table->id();** akkor ez gyakorlatilag azt jelenti majd, hogy létre fog jönni egy id nevű mező, aminek a típusa BIGINT (azon túl persze, hogy ez egy elsődleges kulcs lesz, ami egyértelműen azonosít minden egyes sort az adattáblájában). Tehát a hivatkozó (**ratings** tábla **post_id**) mező és a hivatkozott (**posts** tábla **id**) mező típusainak meg kell egyeznie, ez az elvárás! Ha nem így lenne, hibát adna a Laravel tényleges idegen kulcs létrehozása esetén. A másik névkonvenció a hivatkozó **post_id** mező neve: itt az aláhúzás előtti részben kisbetűvel kell beírni a hivatkozott Model nevét (vagy ha úgy tetszik, az adattábla nevét angolul egyes számban): **post**, az aláhúzás után pedig azt, hogy annak melyik mezőjére hivatkozunk (**id**). *Ez utóbbi olyan konvenció, amelyet bár felülírhatnánk önkényesen, de miért is tennénk meg azt, hogy mi a Laravel-től eltérő névkonvenciót alkalmazunk az idegen kulcs mezőnév megadására...? Ne tegyünk logikátlan dolgokat, mert az csak bonyodalmakat szülne!*

További megszorítás még a **post_id mezővel** kapcsolatban, hogy legyen egyedi, ezért a **unique()** kényszert is alkalmazzuk a mezőre, mivel egy értékelés egy blogbejegyzéshez fog tartozni és nem lehet majd egy blogbejegyzésnek több értékelése.

Mentés után mehet a migrálás!

6.4.1.3. Eloquent Model testreszabása

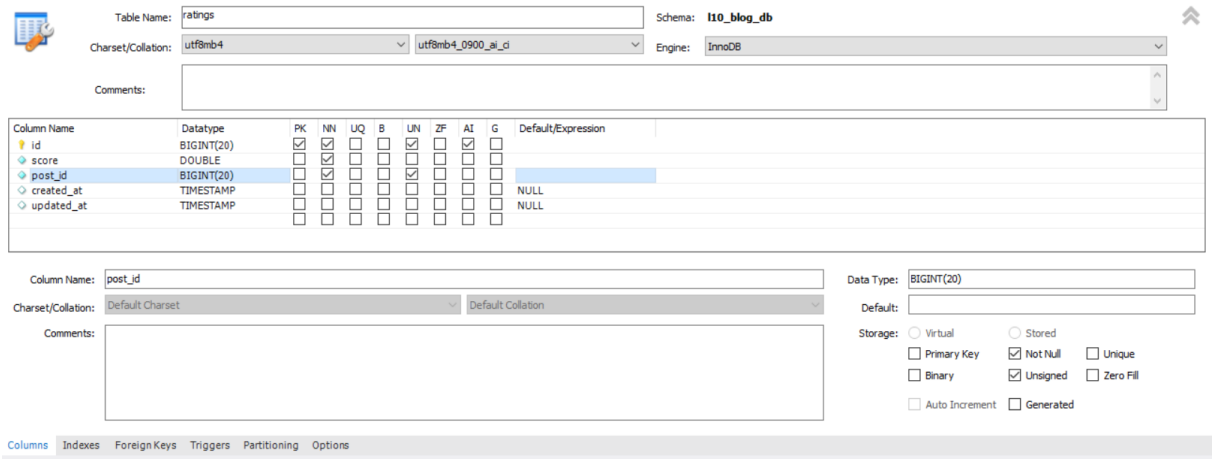
Ha mégis szeretnénk eltérni a tanácsom ellenére a Laravel névkonvenció alkalmazásától ebben az esetben, mert van valamilyen olyan külső körülmény, amelynek meg szeretnénk felelni, akkor az Eloquent Model osztályokban van lehetőségünk felülírni bizonyos értékeket. Példák:

1. Ha a **Post** Model osztállyal nem a **posts** adattáblához szeretnénk hozzáférni, akkor azt egy változó felüldefiniálásával tehetjük meg:
    a. **protected $table = 'masik_tabla_neve';**
    b. A migrációs fájl segítségével előtte érdemes létrehozni az adattáblát.
2. Ha a **posts** adattáblánkban nem szeretnénk (létrehozási **created_at** és módosítási **updated_at**) időbélyegeket eltárolni, akkor hasonlóan megtehetjük ezt is:
    a. **public $timestamps = false;**
    b. A migrációs fájlban töröljük ki a **timestamps()** (figyelem, többes számban van, mivel a **created_at** és az **updated_at** mezőket is létrehozná) tartalmazó sort a migrálás előtt.
3. Ha a **posts** adattáblánkban nem szeretnénk az **id** mezőt használni elsődleges kulcsként, akkor erre is lehetőségünk van:
    a. **protected $primaryKey = 'elsodleges_kulcs_oszlop_neve';**
    b. A migrációs fájlban állítsuk be egy másik mezőhöz az elsődleges kulcsot, például, ha számláink, megrendeléseink vannak, akkor ezt így tudjuk megtenni egy szöveges mezőnél:
        i. **$table->string('order_number')->primary();**

Számos egyéb lehetőségünk van még az alapvető logikától való eltérésre (ha ezt szeretnénk megtenni, akkor nézzük meg a Model ősosztály lehetőségeit és programkódját az ottani megjegyzésekkel együtt), én azonban nem tanácsolnám ezeket a túlzott eltéréseket, mert tapasztalataim szerint jobb betartani a *„játékszabályokat”,* mintsem önkényesen felülírni őket.

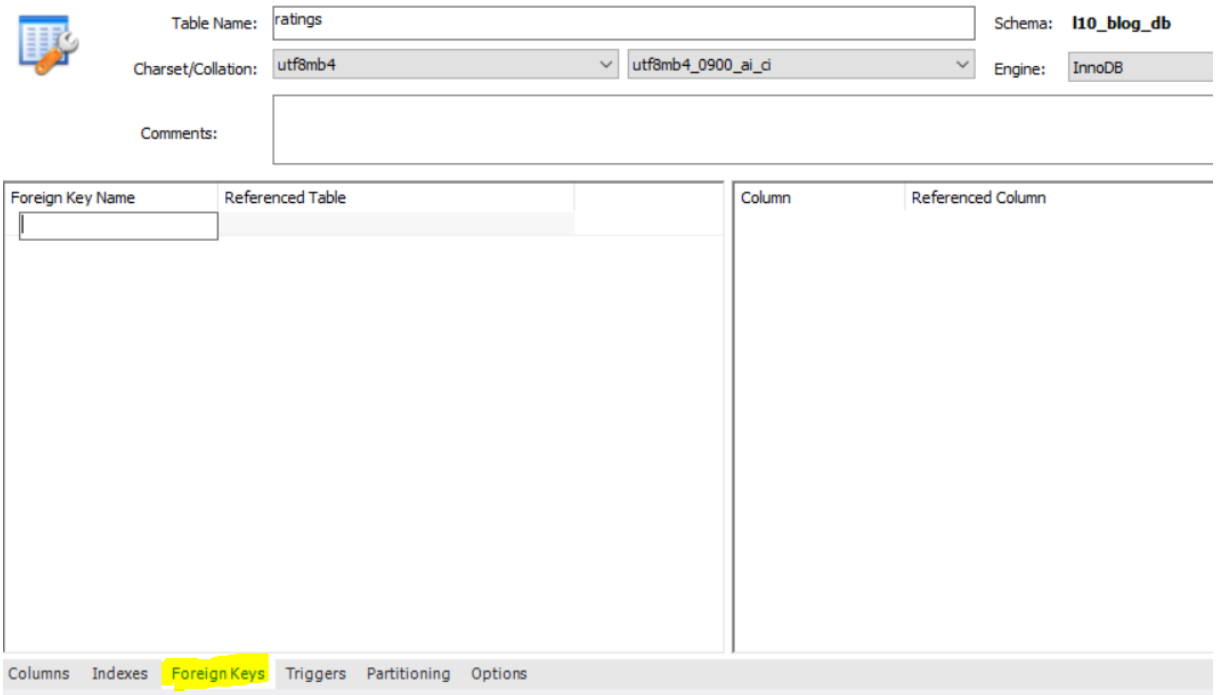
6.4.1.4. Idegen kulcs kapcsolat

Ha nem módosítottuk a fentieket, akkor a migrálás után ellenőrizhetjük a Workbench, hogy mi jött létre a **ratings** adattáblában.



6–12. ábra: post_id a ratings táblában és kényszerei

MySQL Workbench segítségével az adatbázisunkban a tábla neve fölé tudjuk vinni az egeret és válasszuk a sor jobb szélén a *„csavarkulcs”* ikont, ekkor mutatni fogja a mezőket és láthatjuk, hogy létrejött a **post_id** BIGINT típusú mező (aminél alul ki vannak pipálva: NOT NULL, nem lehet ilyen érték nélkül beszúrni új sort a táblába, Unsigned, vagyis előjel nélküli módosítókkal).



6–13. ábra: Még nem idegen kulcs a post_id

Ha alul kiválasztjuk a sárgával kiemelt *„Foreign Keys”,* idegen kulcsokat tartalmazó listát, akkor láthatjuk, hogy ez még üres, tehát még nem idegen kulcs a **post_id,** csak a mezőjét készítettük így elő neki.

Az idegen kulcs kapcsolat létrehozásához először lépjünk vissza a migrációban egy lépést, mivel még úgyis üres a tábla, így lehet a legegyszerűbben a táblaszerkezet definícióját módosítani:

```
php artisan migrate:rollback
```

Majd bővítsük a **ratings** tábla létrehozásának definícióját a mezők definiálása után (tehát a **timestamps()** mezőket tartalmazó sor után):

```
$table->foreign('post_id')->references('id')->on('posts');
```

6–34. kódrészlet: Idegen kulcs kapcsolat létrehozása

Így itt definiáltuk, hogy a **post_id** mező lesz az idegen kulcs, ami hivatkozni fog a **posts** tábla **id** mezőjére. Mindezt újra ellenőrizhetjük a migrálás után a 6–13. ábra alapján, és látni fogjuk az idegen kulcs nevét (**ratings_post_id_foreign**) és hogy melyik táblára hivatkozik.

6–14. ábra: Idegen kulcs meglétének ellenőrzése

6.4.1.5. Rating Model és a kapcsolat tesztelése

Teszteljük az alkalmazásunkat: először adatfeltöltésre lesz szükségünk, de szerencsére a blogbejegyzések (**posts**) táblában vannak már adataink, így most csak az értékeléseket kell feltöltenünk. Viszont a **Rating** Model osztályt bővítenünk kell a **$fillable** attribútummal, hogy egyszerűsített módon tudjuk majd létrehozni az új adatsorokat:

```
protected $fillable = ['score', 'post_id'];
```

6–35. kódrészlet: Feltölthető mezők a ratings táblában

Teszteléshez a már korábban alkalmazott Tinker lesz a segítségünkre, indítás után már írhatjuk is a kódsorokat:

```
php artisan tinker
```

Nálam 1-től 9-ig léteznek a **posts** táblában blogbejegyzések, így azoknak tudok majd értékelést adni az idegen kulcs kényszer miatt.

```
App\Models\Rating::create(['score' => 5.7, 'post_id' => 2]);

$post = App\Models\Post::find(2);

$post->rating;

$post->rating->score;
```

6–14. utasítások: Új értékelés létrehozása a blogbejegyzéshez, majd a blogbejegyzésen keresztül lekérjük az új értékelést

A 6–14. utasítások szekcióban a létrehozott értékeléshez tartozó **Post** Model osztály objektumát kérjük le. Utána pedig az Eloquent kapcsolat (**Post** Model osztály **rating()** metódusa) segítségével lekérhetem a hozzá kapcsolódó értékelést. Még csak nem is metódusként hívni, hanem mezőként, és működni fog, az eredménye alább látszódik.

```
> App\Models\Rating::create(['score' => 5.7, 'post_id' => 2])
= App\Models\Rating {#3773
    score: 5.7,
    post_id: 2,
    updated_at: "2023-05-01 19:58:11",
    created_at: "2023-05-01 19:58:11",
    id: 1,
  }

> $post = App\Models\Post::find(2)
= App\Models\Post {#4735
    id: 2,
    title: "Consequatur.",
    slug: "minus-eius-eaque",
    body: "Itaque praesentium ut pariatur ratione. Illo vero odit eaque perferendis distinctio. Inventore quaerat fuga quaerat sit ipsum laudantium.
",
    published_at: "1984-01-30 23:22:17",
    created_at: "2023-04-23 21:55:46",
    updated_at: "2023-04-23 21:55:46",
    deleted_at: null,
  }

> $post->rating
= App\Models\Rating {#4719
    id: 1,
    score: 5.7,
    post_id: 2,
    created_at: "2023-05-01 19:58:11",
    updated_at: "2023-05-01 19:58:11",
  }

> $post->rating->score
= 5.7
```

6–15. ábra: Az Eloquent utasítások és eredményeik

Ilyen egyszerű, és visszakapjuk, hogy 5.7. A háttérben lefutó SQL utasítás ez volt: SELECT * FROM **ratings** WHERE **post_id = 2;** Ezzel megkapjuk az adatsort, majd utána a **score** mező értékét.

6.4.1.6. Kapcsolat megfordítása, tükrözése

Az Eloquent kapcsolatok, mint ahogy a relációs adatbázis kapcsolatok is többoldalúak, többszereplősek (leggyakrabban kétszereplősek). Emiatt a **Rating** Model osztályunkban is hasonlóan hozzuk létre a kapcsolatért felelős metódust, mint ahogy a **Post** Model osztályban tettük már.

```php
public function post()
{
  return $this->hasOne(Post::class);
}
```

6–36. kódrészlet: Rating osztály Eloquent kapcsolata

Így most már nem csak a blogbejegyzés értékelését, hanem az értékelés blogbejegyzését is le tudjuk kérni. Mivel ez egy programkódbeli módosítás, indítsuk újra a Tinker-t, hiszen az a programkódunk korábbi aktuális állapotával kommunikál, ez pedig egy új változtatás. Egyetlen értékelésem van, annak szeretném lekérni a blogbejegyzését, akkor még hibát kapnék, mivel a másik táblában (**posts**) nincs még meg az értékeléshez vezető idegen kulcs mező (**rating_id**) és kapcsolat.

```
> App\Models\Rating::first()->post

  Illuminate\Database\QueryException  SQLSTATE[42S22]: Column not found: 1054 Unknown column 'posts.rating_id' in 'where clause' (Connection: mysql
, SQL: select * from `posts` where `posts`.`rating_id` = 1 and `posts`.`rating_id` is not null and `posts`.`deleted_at` is null limit 1).
```

6–16. ábra: Hiányzó idegen kulcs mező (posts.rating_id)

Észrevehetjük a hibaüzenetben, hogy a Laravel be szeretné tartani a névkonvenciót és egyből azt a mezőnevet keresi az SQL utasításban, ami utal a másik táblára és annak **id** mezőjére.

Hogyan oldhatnánk ezt meg hatékonyan? Semmiképpen se a **posts** adattáblát létrehozó migrációs fájlba módosítsunk bele, hiszen, ha újra migrálnánk, akkor elvesznének az adataink a táblából. Hozzunk inkább létre neki egy új migrációs fájlt, ami az idegen kulcsos bővítést tartalmazza:

```
php artisan make:migration add_rating_id_to_posts_table
```

Az új fájl **up()** metódusába először adjuk hozzá a külső kulcs mezőt (a **published_at** mező után), majd a kapcsolatot is definiáljuk.

```
$table->unsignedBigInteger('rating_id')->nullable()
->after('published_at');
$table->foreign('rating_id')->references('id')->on('ratings');
```

6–37. kódrészlet: Külső kulcs migrálása

A **nullable()** módosító azt jelzi, hogy a **rating_id** mező értéke lehet NULL. Erre azért van szükség, mivel a posts adattáblában már van néhány adatsor és ha enélkül szeretnénk migrálni, akkor hibát kapnánk az idegen kulcs kapcsolat definiálásakor. Hiszen, mindössze az egyik sorban lévő blogbejegyzésnek van konkrét értékelése, de a kezdetben ennél is és a többinél is NULL érték kerülne be a **rating_id** mezőbe, amit az adattábla a NOT NULL kényszer miatt nem engedélyezne. Így viszont a **nullable()** miatt engedélyezni fogja már a kapcsolat létrehozását. Megoldás lehetne még *(bár az már nem 1-1-es kapcsolat lenne),* hogy ha egy alapértelmezett értéket (**default(1)**) állítunk be neki, így a **ratings** táblában lévő, mindenképpen létező adatsor **id** értékét kapná meg, de ez nem feltétlen magyarázható vagy érthető ebben az esetben. Sokkal logikusabb, ha a **nullable()** kényszerrel ruházzuk fel a mezőt, amely a későbbiekben még módosítható ez, ha az összes blogbejegyzésnek lesz már értékelése.

A **down()** metódusba a fenti kettő utasításnak a *fordított sorrendben való megszüntetését* kell kiadni utasításként, hiszen visszavonáskor (migrate:rollback) először meg kell szüntetni a külső kulcs kapcsolatot, majd utána törölni kell a mezőt.

```
$table->dropForeign('posts_rating_id_foreign');
$table->dropColumn('rating_id');
```

6–38. kódrészlet: Külső kulcs migrálásának visszavonásakor végrehajtódó utasítások

A külső kulcs kapcsolat visszavonásának neve is névkonvenció szerint épül fel, amire már láthattunk példát a 6–14. ábra bal oldali oszlopában, csak ott még a kapcsolat *„tükörképe"* volt megnevezve. A névkonvenció így épül fel: **táblanév_mezőnév_foreign**

Ha elmentjük a fájlt, akkor végrehajthatjuk a migrálást és annak visszavonását, hogy közben tudjuk ellenőrizni az idegen kulcs mező és a kapcsolat hozzáadását, elvételét.

6.4.1.7. Post Model és a kapcsolat tesztelése

A **Post** Model osztályban a már meglévő **$fillable** attribútum tömbjét mindenképpen érdemes kibővíteni a **rating_id** mezővel. De a mostani tesztelést egy már meglévő blogbejegyzéssel hajthatjuk végre:

```
$post = App\Models\Post::find(2);

$post->update(['rating_id' => 1]);

App\Models\Rating::first()->post;
```

6–15. utasítások: A 2. blogbejegyzéshez beállítjuk az 1. értékelést, majd az 1. értékelésnek lekérjük a blogbejegyzését

Működik tehát a kapcsolat, így most már oda-vissza lekérhető a blogbejegyzés értékelése és az értékeléshez tartozó blogbejegyzés is (nyilván azokhoz, amelyeknek be van állítva a párjuk).

Gyakorlásként végig gondolhatjuk, hogy Query Builder segítségével hogyan lehetne megoldani a releváns adatsor **rating_id** mező értékének frissítését. A megoldást a következő összefoglalás bekezdés alatt megírom.

Összefoglalás: az 1-1-es kapcsolatok bár elég ritkák lehetnek, mivel ilyenkor a *„kapcsolat társ"* elemei egy-az-egyben beépíthetők a kapcsolódó táblába. De adatbázistáblák kezelése szempontjából az az előnye volt, hogy megismerkedtünk a kapcsolat létrehozásával magával (idegen kulcson keresztül), és így a kapcsolatban lévő táblák közül bármelyikbe elhelyezhetjük a másik táblára vonatkozó idegen kulcsot. Nincs megkötés arra vonatkozólag, hogy melyikben kell szerepelnie, de nyilván, ahogy láttuk is, akkor annak megfelelően kell kialakítanunk a kódjainkat, hogy melyik *„oldalon"* szerepel az idegen kulcs.

A fenti gyakorló kérdés megoldása: **DB::table('posts')->where('id',2)->update(['rating_id' => 1]);**

Megjegyzés: ez működik úgy is, hogy egy rövidítést használunk a **where()** segédmetódusnál, mégpedig úgy, hogy kiemeljük a *„where"* után nagy kezdőbetűkkel (aláhúzás nélkül, például **id** esetén **whereId()**, **created_at** esetén: **whereCreatedAt()**) az oszlop nevét és paraméterbe csak a konkrét értéket írjuk, amely a feltételvizsgálatra vonatkozik:

DB::table('posts')->whereId(2)->update(['rating_id' => 1]);

6.4.1.8. Összekapcsolás Query Builder-rel

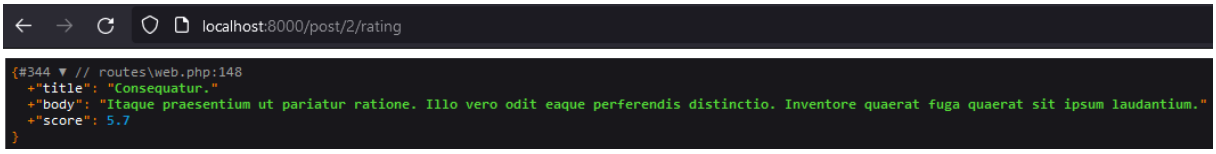
Az alfejezetben láthattuk, hogy az Eloquent Model milyen metódust és adattábla mezőt követel meg a lekérdezés eredményének megjelenítéséhez (vagy a kapcsolat kialakításához). Ezzel szemben a Query Builder-rel akkor is működne az összekapcsolás, ha nem hoztuk volna létre a Model osztályokban a kapcsolat kialakításáért felelős metódusokat. A második blogbejegyzés értékelését például így kérhetnénk le vele, akkor a **web.php** útvonal fájlban adjuk hozzá ezt az új útvonalat:

```
Route::get('/posts/{post}/rating', function ($post) {
  $result = DB::table('ratings')
    ->join('posts', 'ratings.id', '=', 'posts.rating_id')
    ->select('title', 'body', 'score')
    ->where('post_id', $post)
    ->first();
  dd($result);
});
```

6–39. kódrészlet: Blogbejegyzés és az értékelésének lekérése

Az útvonal, amivel tesztelhető az útvonal: http://localhost:8000/post/2/rating

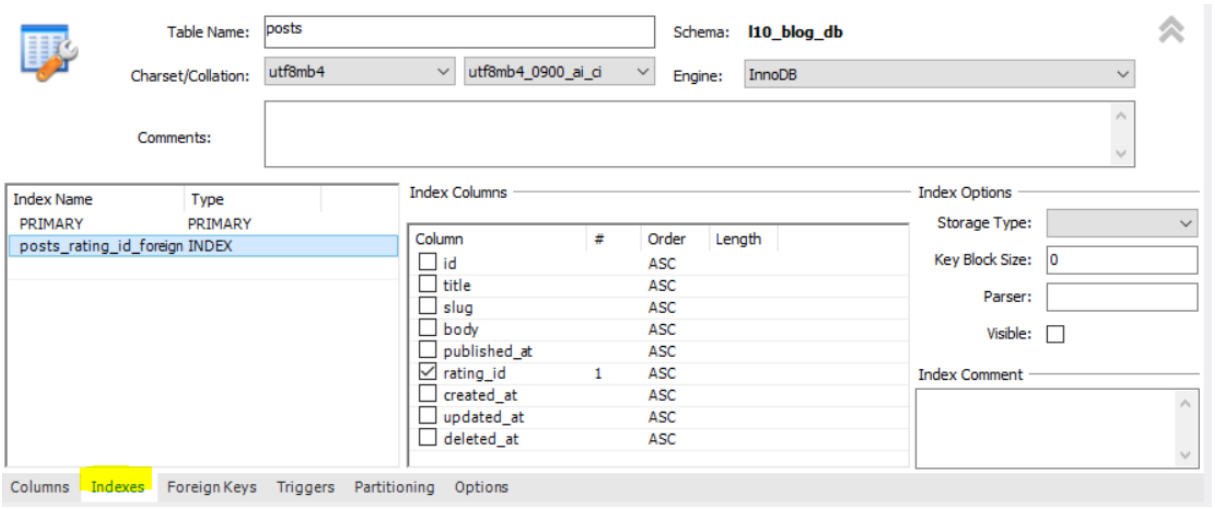
Az útvonal lekérésének eredménye a böngészőben:



6–17. ábra: Query Builder összekapcsoló lekérdezés eredménye

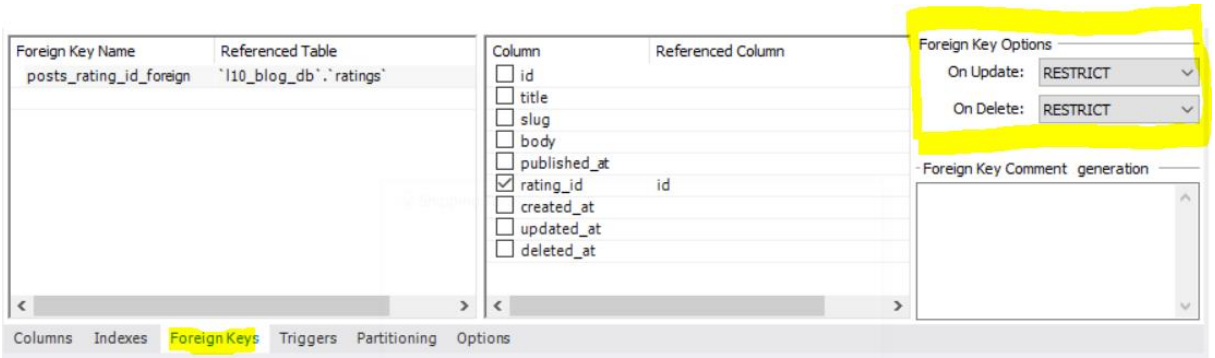
Az összekapcsolási művelet (**join**) akkor is működne, ha a **rating_id** mező értéke meg lenne adva, de a külső kulcs kapcsolat nem lenne definiálva. Ezt azonban nem javasolnám elhagyni, mivel nagy tábláknál az adatbázis-kezelők a külső kulcs kapcsolatot létesítő mezőkre automatikusan index fát építenek, így gyorsabban szűrhetőek, rendezhetőek, kereshetőek ezeknek a mezőknek az értékei és gyorsabban végezhető el az összekapcsolás annál, mint amikor csak a mező létezik a konkrét kapcsolat nélkül. Tehát elsősorban teljesítmény javítás szempontjából fontos ez az idegen kulcs kapcsolat.

Mindez látszódik is a Workbench-ben, amikor a **posts** tábla részletes beállításait lekérjük és az alsó lapfülön kiválasztjuk az *„Indexes”* lapot, majd rákattintunk az index nevére.



6–18. ábra: Idegen kulcs index fája

Ugyanitt a Workbench-ben maradva a *„Foreign Keys”* lapfület kiválasztva láthatjuk a blogbejegyzések és az értékelések közötti idegen kulcs kapcsolatot.



6–19. ábra: Idegen kulcs beállításai

A másik főbb ok az idegen kulcs kapcsolat használatára pedig az iménti képen jobb oldalon láthatjuk, ami a *„Foreign Key Options”* részben van: itt tudjuk meghatározni azokat a kényszereket, hogy mi történjen akkor a hivatkozó (**posts**) tábla soraival, ha a hivatkozott táblában (**ratings**) mondjuk frissítésre (**On Update**) vagy törlésre (**On Delete**) kerül az a sor, amire a **posts** hivatkozott. Egy beszédesebb példa: mi történjen azokkal az értékelésekkel, amik adott blogbejegyzésekhez tartoznak akkor, amikor töröljük ezt a blogbejegyzést. Megmaradjanak (tárolódjanak a táblában) ezek az értékelések, amik a törölt blogbejegyzésekhez tartoznak vagy töröljük őket az értékelések táblából is? (Ugyanígy igaz ez a frissítésre.) Ezek vizsgálatára és beállítására vissza fogok térni a következő alfejezetben.

Az alfejezetben végrehajtott programkód-módosítások ebben a GitHub commit-ben érhetők el.

Megjegyzés: a továbbiakban az 1-1-es kapcsolat eredeti változatát (a **ratings** tábla sorai hivatkoznak a **post_id**-n keresztül a blogbejegyzésekre) és nem a megfordítását fogjuk alkalmazni.

6.4.2. Egy-több (1-n) adatkapcsolat és az adatgyárak (Factory osztályok)

Egy sokkal gyakrabban használt relációs adatbázis kapcsolatot fogunk áttekinteni, ez az *„egy-több”*-es (1-n) vagy *„több-egy”*-es (n-1) adatkapcsolat lesz, attól függ, hogy honnan nézzük. A lényeg, hogy a kapcsolat egyik oldalán (táblában) egy, míg a másik oldalán (táblában) több szereplő (adatsor) vesz (vehet) részt a kapcsolatban. De példákon keresztül talán jobban megérthető ez: a blog egy kategóriájához több blogbejegyzés tartozik, de egy blogbejegyzés csak egy kategóriához tartozhat. Vagy például egy Facebook bejegyzésnek lehet több kommentje, egy felhasználónak lehet több bejegyzése, egy embernek lehet több autója, egy projektnek lehet több mérföldköve és így tovább lehetne sorolni. Mi ezek közül a legelső példát fogjuk megnézni. A kapcsolat megfordítása pedig így néz ki: egy blogbejegyzés csak egy kategóriához tartozhat (adott pillanatban). A külső (idegen) kulcsokat fogjuk használni a kapcsolat létrehozásához. A kapcsolat teszteléséhez pedig példa adatokat fogunk létrehozni adatbázis gyárak segítségével.

6.4.2.1. Eloquent kapcsolatok

A kulcsszavak, amit fogunk használni az Eloquent kapcsolat kialakításához: **belongsTo()** (vagyis: valami tartozik valamihez) és a **hasMany()** (vagyis: valaminek van több valamije). Kezdjük ez utóbbival, mivel a kategóriák még nem léteznek:

```
php artisan make:model Category -m
```

Ez létrehozza a Model fájlt és a migrációs fájlt is. A **Category** Model osztályba illesszük be a **$fillable** változót azért, hogy a **name** mezőt lehessen majd kitölteni, továbbá ezt a metódust is:

```php
public function posts() {
  return $this->hasMany(Post::class);
}
```

6–40. kódrészlet: hasMany() kapcsolat létrehozása

Így tehát kifejeztük azt, hogy egy kategóriába több blogbejegyzés tartozhat, és rögtön definiáljuk az ellenoldalát is a Post Model osztályba: egy blogbejegyzés csak egy kategóriához tartozhat.

```php
public function category() {
  return $this->belongsTo(Category::class);
```

```
}
```

6–41. kódrészlet: belongsTo() kapcsolat létrehozása

Érdemes megfigyelni a névkonvenciókat is. Amikor a kategóriának több blogbejegyzése lehet, akkor a **posts()** metódus neve többesszámban van. Amikor a blogbejegyzés egy kategóriához tartozik, akkor a **category()** metódus neve egyesszámban van. Az ellentétes osztályok importálására nincsen szükség.

6.4.2.2. Idegen kulcs kapcsolat

A kategóriákat tartalmazó migrációs fájlba csak a kategória nevét helyezzük el a séma létrehozásának definiálásában az **id()** és a **timestamps()** sorok közé.

```
$table->string('name')->unique();
```

6–42. kódrészlet: categories tábla migrációs fájljának bővítése

Mivel az adatkapcsolat *„több”* (n) oldalára, táblájába kell elhelyezni az idegen kulcsot a másik tábla **id** mezőjére, ezért a **posts** táblát kell bővítenünk egy **category_id** idegen kulccsal.

```
php artisan make:migration add_category_id_to_posts_table
```

Ezt megtehetjük több sor megadásával is úgy, hogy először magát a mezőt és a típusát, majd az idegen kulcs kapcsolatot definiáljuk úgy, ahogy a 6.4.1.6. alfejezetben is megtettük már. Viszont van neki egy még annál is egyszerűbb módja, ami mindössze egy sornyi utasítást tartalmaz, ezt szúrjuk be az **up()** metódusban lévő séma módosító utasításba.

```
$table->foreignId('category_id')->constrained()
->onUpdate('cascade')->onDelete('cascade');
```

6–43. kódrészlet: Idegen kulcs mező, kapcsolat és beállításainak definiálása egyetlen kódsorral

A **down()** metódus magjának kitöltését hasonlóan végezzük el, mint a 6–38. kódrészletben már megtettük, betartva a névkonvenciókat (ha nem menne, akkor érdemes megnézni majd az alfejezet végén megtalálható GitHub commit tartalmát).

De magyarázat szempontjából maradjunk még az **up()** metódus magjánál. A **foreignId()** paramétere a külső kulcs mező lesz, ami után a **constrained()** metódusba nem is muszáj beírni a hivatkozott tábla nevét, ha betartjátok a névkonvenciót, akkor a rendszer tudja, hogy a **categories** táblára hivatkozunk, annak is az elsődleges kulcs mezőjére, az **id**-ra. Hozzáfűztem még további mezőkényszereket, de az utasítás amúgy végrehajtható lenne **onUpdate()** és **onDelete()** nélkül is. Viszont, ha már ott vannak, akkor meg is magyarázom őket: onUpdate és onDelete esetén is a **cascade** (vagy *„tovább gyűrűzés”*) opciót választottam. Ami a példámban azt fogja jelenteni, hogy ha mondjuk törlünk egy kategóriát, aminek vannak blogbejegyzései, akkor az blogbejegyzések is törlődni fognak az **posts** táblából, tehát a törlés tovább gyűrűzik... és ugyanez igaz lenne, ha a kategóriának módosítanám az **id** (azonosítóját), akkor utána az tovább gyűrűzne a **posts** táblában is és a hozzákapcsolt blogbejegyzéseknél is frissülne a **category_id** mező értéke.

Az *onUpdate* és *onDelete* megszorításoknak a **cascade** mellett további értékek is beállíthatók:

1. **restrict**: ekkor addig nem engedi törölni/frissíteni a kategóriát, amíg van neki legalább egy darab blogbejegyzése.
2. **set null**: ha töröljük a kategóriát vagy frissítjük az azonosítóját (**id**), akkor a hozzá tartozó blogbejegyzések **category_id** mezőjének helyére null érték fog bekerülni.
3. **no action**: ha töröljük a kategóriát vagy frissítjük az azonosítóját, akkor a hivatkozó blogbejegyzések (**posts**) táblában nem fog történni semmi a kapcsolódó blogbejegyzés sorok **category_id** mező értékével, tehát kvázi *„rosszak"* lesznek, hiszen olyan mező értékre fognak hivatkozni (**categories.id**), ami már nem létezik vagy megváltozott.

Nekünk most ez a *cascade*, tovább gyűrűzés megfelelő, úgyhogy majd migráljunk, viszont a **posts** tábla tartalmaz sorokat, ami miatt esetleg most problémába ütközhetnénk (ha a **category_id** mezőnek nem lesznek nullázható értékei a migrálás után), ezért inkább építsük fel újra a teljes migrációs folyamatot, úgysem kerültek még be lényegi adatsorok a táblákba, illetve majd a következő alfejezetben úgyis megismerjük, hogy hogyan lehet tesztadatokkal feltölteni a táblákat egyszerűen.

```
php artisan migrate:fresh
```

Ezzel törlődnek a tábláink, majd utána rögtön újra felépíti a rendszer a migrációs fájlok **up()** metódusain végig haladva a teljes szerkezetet, de ekkor már az összes eddigi migrációs fájl az első csomagba (batch-be) fog bekerülni a **migrations** adattáblában.

Azt láthattuk a 6–19. ábra alapján, hogy alapból RESTRICT-re lett állítva a **posts_rating_id_foreign**-s külső kulcs kapcsolat beállítása frissítés és törlés szempontjából, viszont, ha most ellenőrizzük a **posts_category_id_foreign**-s idegen kulcs kapcsolat beállításait, akkor CASCADE-re, tovább gyűrűzésre lesz mindkettő beállítva. Az index fa pedig mindkét oszlopra felépült és majd bővül, ahogy új értékek kerülnek be a sorok ezen mezőinek értékeibe.

Mindezt a terminal-ban is leellenőrizhetjük:

```
php artisan db:table posts
```

```
PS C:\xampp\htdocs\l10-components> php artisan db:table posts

  posts ..........
  Columns ..... 9
  Size ..... 0.05MiB

  Column ..... Type
  id autoincrement, bigint, unsigned ..... bigint
  category_id bigint, unsigned ..... bigint
  title string ..... string
  slug string ..... string
  content text ..... text
  published_at datetime, nullable ..... datetime
  created_at datetime, nullable ..... datetime
  updated_at datetime, nullable ..... datetime
  deleted_at datetime, nullable ..... datetime

  Index ..........
  PRIMARY id ..... unique, primary
  posts_category_id_foreign category_id ..........

  Foreign Key ..... On Update / On Delete
  posts_category_id_foreign category_id references id on categories ..... cascade / cascade
```

6–20. ábra: Adattábla szerkezetének megtekintése a terminal-ban

Itt látszódnak az oszlopok (mezők) és kényszereik, valamint a tábla kényszerek, kulcsok (és az indexek).

6.4.2.3. Tesztelés: adatgyár (factory) osztályok

Most viszont üresek az adattábláink, pedig nagyon jó lenne, ha lennének a **posts** táblában soraink, amikkel tudnánk tesztelni az alkalmazás működését. Ebben ugye korábban segített nekünk a Tinker és manuálisan új objektumokat hoztunk létre, amelyeket elmentettünk, és ez végrehajtotta az adatbázistáblába a mentést (beszúrást). Aztán ezt egy kicsit egyszerűsítettük és az osztályok **create()** metódusával is képesek voltunk új adatsorokat létrehozni. De milyen jó lenne, ha tudnánk ezt egy kicsit automatizálni, vagy legalábbis a tömeges feltöltést megkönnyíteni. Ezt fogjuk majd csinálni itt, de előbb ismerkedjünk meg az adatbázis gyárak működésével a felhasználók létrehozásán keresztül (azért ezen keresztül, mert ehhez alapértelmezetten létezik az alkalmazás projektünkben már ilyen gyár). A Tinker-re most is szükségünk lesz, úgyhogy indítsuk el:

```
php artisan tinker
```

Mellette azért nézzük a projekt mappáinkat és fájljainkat is, mivel azt írtam, hogy egy létező adatbázis gyárat fogunk használni. Ez a legelső adatgyárunk a **database** (adatbázis) könyvtárban lévő **factory** (gyár) könyvtárban lesz **UserFactory.php** néven. Ezt alkalmazhatjuk felhasználók tömeges létrehozására a példa kedvéért. Itt látható a **UserFactory** osztály **definition()** metódusa, amelyben igazából csak egy tömb van visszatérési értékként:

```
return [
  'name' => fake()->name(),
  'email' => fake()->unique()->safeEmail(),
  'email_verified_at' => now(),
  'password' =>
'$2y$10$92IXUNpkjO0rOQ5byMi.Ye4oKoEa3Ro9llC/.og/at2.uheWG/igi', // password
  'remember_token' => Str::random(10),
];
```

6–44. kódrészlet: UserFactory osztály definition() metódusának visszatérési tömbje

Az itt látható megoldással már találkoztunk a 6.3.2.1. alfejezetben, amikor az adatbázistáblába való beszúráshoz használtuk fel a **fake()** segédmetódust példaadatok generálására. Itt is pontosan ez történik, amikor a felhasználóknak generál majd a rendszer:

- nevet,
- egyedi, biztonságos e-mail címet,
- e-mail cím megerősítési időbélyeget,
- hashelt (kódolt) jelszót, amely alapja a password szó és az **.env** fájlban definiált **APP_KEY** attribútum értéke van hozzáadva a hasheléshez, emiatt különböző környezetekben más-más hash képe lesz ugyanannak a password szónak,
- az emlékező token pedig egy véletlenszerűen generált 10 karakter hosszú szöveg.
- *Megjegyzés:* az alatta lévő **unverified()** metódus segítségével még olyan felhasználót is tudnánk szimulálni, aki nem erősítette még meg az e-mail címét.

Laravel 11: az **APP_KEY** értékének megváltoztatása már nem probléma!

A Laravel keretrendszer korábbi verzióiban, ha megváltoztattuk az **APP_KEY** attribútum értékét, az az adatbázisban eltárolt bizonyos adatok *„elromlásához"* vezetett. Főleg azoknál, ahol az **APP_KEY** értékét használta a rendszer a hash-eléshez, kódoláshoz, titkosításhoz például a felhasználói jelszavaknál, ha újrageneráltuk az **APP_KEY** értékét, akkor onnantól kezdve a teljes **users** táblában lévő **password** mező értékkészlete elromlott, a felhasználók nem tudtak utána már bejelentkezni többet.

A 11-es verzióban, ha megváltoztatjuk az **APP_KEY** értékét, akkor egy **APP_PREVIOUS_KEYS** változóban eltárolódik a korábbi érték, az új értékkel pedig automatikusan újra titkosítja az adatokat a rendszer az adatbázisban.

6–4. újdonság: APP_KEY kezelésének változása

Ez a felhasználó gyár segít nekünk tömeges teszt adatokat előállítani az adatbázisban, méghozzá úgy, hogy *„betartja"* a mezők értékeire itt megfogalmazott szabályokat.

```
User::factory()->make();
```

6–16. utasítások: Egy felhasználó tesztadatait mutató utasítás

Eredményül megkapjuk a **User** Model osztály egy példányát, a fenti szabályrendszer szerint generált mező értékekkel, de ez csak egy darab, én pedig azt ígértem, hogy bármennyit egyszerűen le tudunk gyártani... szóval nézzük meg, ehhez mit kell tennünk:

```
User::factory()->count(3)->make();
```

6–17. utasítások: Három felhasználó tesztadatait mutató utasítás

Mindössze annyit csináltam, hogy belefűztem a **count(3)** metódushívást és most már nem 1 hanem 3 felhasználó készült el. Könnyen beláthatjuk, hogy ha a 3 helyére 100-at írnánk, akkor 100 felhasználó készülne el, mindössze egyetlen utasítás kiadásával! Ellenőrizzük le az adatbázisban, hogy létrejött-e ez a 1+3 = 4 darab adatsor a **users** táblában!

De azt fogjuk látni, hogy a **users** táblánk üres, úgyhogy itt valami nem teljesen jó még. Adjuk ki a Tinker-ben ezt az utasítást:

```
User::factory()->create();
```

6–18. utasítások: Egy felhasználót tesztadatokkal létrehozó utasítás

Észrevehetjük, hogy most is visszakaptunk egy példa felhasználót a példa adataival, de most kaptunk még hozzá három mezőt: **updated_at**, **created_at**, **id**. Ezek pedig onnan lehetnek ismerősek, hogy az adattábláinkba ezek azok a mezők, amiket alapból mindig beszúr a Laravel, így én élnék a feltételezéssel, hogy most már beszúrásra is került 1 sor az adattáblánkba... de ellenőrizzük le!

Valóban ott is van az adatsor. Ha pedig kiadjuk a **count()**-os belefűzés helyett a **factory()** metódusnak a paramétert, például 99 felhasználóval, akkor létre fog jönni 99 újabb sor a users adattáblánkba a már meglévő 1 mellé.

```
User::factory(99)->create();
```

6–19. utasítások: 99 felhasználót tesztadatokkal létrehozó utasítás

A **users** táblánk ezután már 100 sort fog tartalmazni.

6.4.2.4. Saját adatgyárak létrehozása és működtetése

Ennyi előzetes ismeret után már képesek vagyunk arra, hogy létrehozzuk a saját adatgyárunkat. Mivel majd a kategóriákhoz szeretnék blogbejegyzéseket rendelni, ezért először érdemesebb a kategóriákhoz létrehozni az adatgyárat:

```
php artisan make:factory CategoryFactory
```

Az utasításból a Laravel, mivel feltételezi, hogy betartottuk a névkonvenciókat (egyes számban használtuk a fájl nevében a **Category**-t), ezért tudja, hogy a **Category** Model osztályhoz hoztuk itt létre az adatgyárat. A létrejövő új fájlunkban (**database / factory / CategoryFactory.php**) az osztályunk már tartalmazza is a szükséges dolgokat, legfőképpen a **definition()** metódust. A metódus visszatérési tömbjéhez adjuk hozzá ezt:

```
'name' => fake()->unique()->word,
```

6–45. kódrészlet: CategoryFactory definition() metódusának visszatérési tömb eleme

Konkrétan kategórianeveket nem tartalmaz a FakerPHP, emiatt egy sima szót generálunk így majd kategóriaként, de egyedien, hogy ne jöhessenek létre ugyanolyan nevű kategóriák. Egy másik nagyon egyszerű gyár legyen a következő:

```
php artisan make:factory RatingFactory
```

Ennek összeállításához viszont végezzük el egy ésszerűsítést: az 1-1-es adatkapcsolatnál a blogbejegyzés és az értékelés között ne legyen mindkét oldalon idegen kulcs mező és kapcsolat a másik táblára, hanem elegendő a **ratings**-ben a **post_id** és a **posts**-ban a **rating_id** felesleges. Új migrációs fájl létrehozásával oldhatjuk ezt meg.

```
php artisan make:migration drop_rating_id_from_posts_table
```

Kicsit furcsa ez a helyzet, mivel az **up()** metódusba kerül most az idegen kulcs és a mező *eldobása*, míg a **down()** metódusba kerül majd a mező és az idegen kulcs kényszer *hozzáadása*:

```
/**
 * Run the migrations.
 */
public function up(): void
{
  Schema::table('posts', function (Blueprint $table) {
    $table->dropForeign('posts_rating_id_foreign');
```

```
    $table->dropColumn('rating_id');
  });
}

/**
 * Reverse the migrations.
 */
public function down(): void
{
  Schema::table('posts', function (Blueprint $table) {
    $table->unsignedBigInteger('rating_id')->nullable();
    $table->foreign('rating_id')->references('id')->on('ratings');
  });
}
```

6–46. kódrészlet: Migrációs fájl mező és idegen kulcs kényszer eldobására

A fájl elmentése után futtathatjuk a migrálást (php artisan migrate) és el is tűnik a **rating_id** a **posts** táblából. Ezzel együtt az Eloquent Model kapcsolatot is töröljük ki a **Rating.php**-ből (**post()** metódust), mivel a kapcsolat ilyen módon való felhasználása megszűnt.

Visszatérhetünk a **RatingFactory** osztályra: a mező, amibe kötelező itt értéket generálnunk: **score** (a **post_id** másképp fog adódni). A **score** egy lebegőpontos szám, aminek az értékét 0.0 és 9.9 közötti értékként határozom meg a példában (a **numerify()**-ról részletesen itt lehet olvasni).

```
'score' => fake()->numerify('#.#'),
```

6–47. kódrészlet: Lebegőpontos szám generálása 0.0 és 9.9 között

A **post_id** mező értékei úgy adódnak az 1-1-es kapcsolat miatt, hogy a **PostFactory** létrehozó művelete után megtörténik a **RatingFactory** meghívása (lásd rögtön). Így majd annyi új blogbejegyzés és új értékelés (score) kerül létrehozásra, ahányszor a **PostFactory**-t beüzemeljük (magát a **RatingFactory**-t önmagában ne indítsuk el, mert egy-egy új blogbejegyzés után van értelme az értékelést is létrehozni ahhoz.

A **Post** Model osztály adatgyárát is hozzuk létre:

php artisan make:factory PostFactory

A létrejövő fájlban megint a **definition()** metódus visszatérési tömbjére koncentrálhatunk. Segítségül hívhatjuk a már létrehozott **/insert-posts** útvonalunkat a **web.php**-ból, mivel ott már a példa blogbejegyzésekhez definiáltunk néhány mezőt, most ezt a listát alakítsuk át az alábbiak szerint:

```
'title' => fake()->sentence(1),
'slug' => fake()->slug(2),
'body' => fake()->paragraph(3),
'published_at' => fake()->dateTimeThisMonth(), //
https://fakerphp.github.io/formatters/date-and-time/
'category_id' => Category::inRandomOrder()->first()->id,
```

6–48. kódrészlet: PostFactory - a posts tábla mezőinek példa értékei

A **title, slug, body** mező lehetséges értékeit már az **/insert-posts** útvonal létrehozásakor megismerhettük. A **published_at** mező értékei az elmúlt 1 hónapból fognak származni.

Az **App\Models\Category** osztály importálására szükség van a fájl tetején. A **category_id** mező értékeit úgy generáljuk, hogy egy már létező, véletlenszerűen kiválasztott adatsor **id** mező értékét választjuk a **categories** táblából.

A **configure()** metódus segítségével meg tudjuk határozni, hogy mi történjen az adatgyár egy-egy lefutása után. A mi célunk itt az, hogy ha létrejön egy blogbejegyzés az alap adataival, akkor jöjjön létre hozzá egy értékelés is, amit az alábbi módon tudunk definiálni a **PostFactory** osztályon belül:

```
public function configure()
{
  return $this->afterMaking(function (Post $post) {
    Rating::factory()->make(['post_id' => $post->id]);
  })->afterCreating(function (Post $post) {
    Rating::factory()->create(['post_id' => $post->id]);
  });
}
```

6–49. kódrészlet: A blogbejegyzés után egy értékelést is definiálunk rögtön hozzá

Indítsuk újra a Tinker-t és kezdjük el működésre bírni a gyárainkat a következő utasítások egymás utáni végrehajtásával! *Megjegyzés:* ha bizonytalanok vagyunk, hogy megfelelően működik-e az adatgyárunk, először a **create()** metódus helyett használjuk a **make()** metódust.

```
App\Models\Category::factory(10)->create();

App\Models\Post::factory(20)->create();
```

6–20. utasítások: Category és Post adatgyárak beüzemelése

Az első utasítással létrejött 10 különböző kategória. A második utasítással pedig létrejött 20 adatsornyi blogbejegyzés, amelyek véletlenszerűen lettek hozzárendelve a 10 kategóriához és mind a 20 blogbejegyzés létrehozásakor létrejött nekik 1-1 (összesen szintén 20) értékelés.

Arra is lehetőségünk van, hogy felüldefiniáljuk a Factory fájlban definiált mezők értékeit. Alább például arra mutatok példát, hogy egy konkrét kategóriához rendelhetünk hozzá több blogbejegyzést is (ugyanezt a technikát alkalmaztam a 6–49. kódrészletnél is, amikor konkrétan meg kellett határozni, hogy egy új blogbejegyzés létrehozása utáni értékelés létrehozása a legújabb blogbejegyzéshez tartozzon ténylegesen):

```
App\Models\Post::factory(5)->create(['category_id' => 3]);
```

6–21. utasítások: 5 blogbejegyzés létrehozása a 3. kategórián belül

Utóbbi utasítást meg is tudjuk *„fordítani”:* tehát létrehozunk egy konkrét kategóriát és ahhoz hozunk létre 5 darab blogbejegyzést:

```
App\Models\Category::factory()->has(App\Models\Post::factory()->count(5))->create();
```

```
App\Models\Category::factory()->hasPosts(5)->create();
```

6–22. utasítások: Új kategória és ahhoz új blogbejegyzések létrehozása

Az iménti két utasítás eredménye megegyezik, előbbi részletesebb, míg a második talán egy kicsit egyszerűbb (mágikus metódus), az átalakítási névkonvenció látható a kettő között.

Az adatokat meg tudjuk tekinteni az adatbázis tábláinkban. A kapcsolatok működését pedig ellenőrizhetjük a Tinker-ben:

```
App\Models\Category::find(5)->posts;
App\Models\Post::find(8)->rating;
App\Models\Post::find(8)-> category;
App\Models\Post::find(8)-> category->name;
App\Models\Post::find(8)-> rating->score;
```

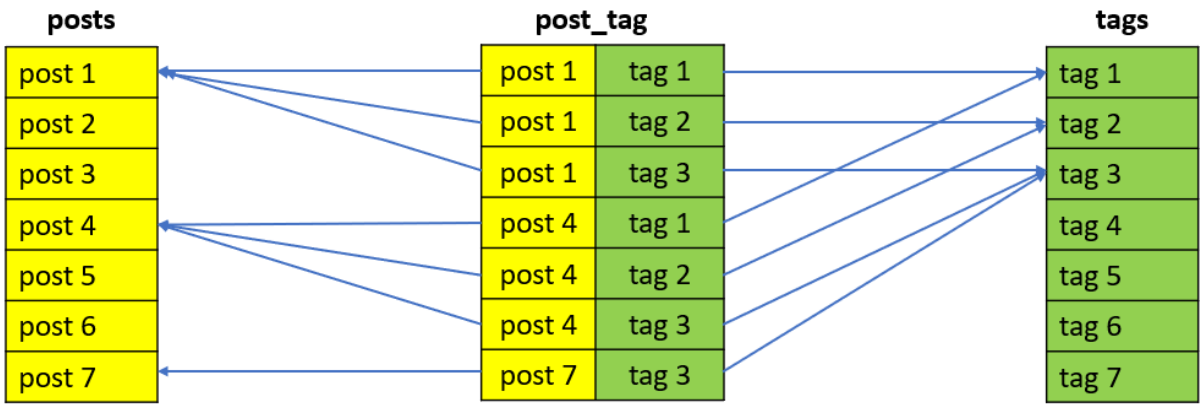
6–23. utasítások: Kapcsolatok működését ellenőrző parancsok

Az első utasításban lekértem az 5-ös azonosítójú kategóriához tartozó blogbejegyzéseket (érdemes olyat választani, amelyiknek több van). A második utasításban a 8-as blogbejegyzéshez tartozó értékelést kértem le, aztán a harmadik utasításban további információként, annak a kategóriáját is lekértem. Még tovább lehetne fűzni és a kategória nevét is meg lehetne jeleníteni a **->name** mező lekérésével. Ebből adódóan az utasításnál az értéléshez a konkrét pontszám is lekérhető a **->score** mezővel.

Az alfejezetben végrehajtott programkód-módosítások ebben a GitHub commit-ben érhetők el.

6.4.3. Több-többes (n-n) adatkapcsolat és a kapcsolótábla

Ebben az alfejezetben áttekintjük a n-n-es, vagy több-többes adatbáziskapcsolatot. Példaként gondoljunk a diákokra és a tantárgyakra. Egy diák több tantárgyat is felvehet egy félévben, míg egy tantárgyhoz több diák is tartozhat egy félévben. Ez tehát egy több-többes kapcsolat tipikus esete. Mi meg fogjuk vizsgálni Laravel-es környezetben ezt a kapcsolattípust úgy, hogy folytatjuk a blogos példánk bővítését: a blogbejegyzéseinkhez most már tag-ek (címkék) is fognak tartozni, míg egy **tag**-hez több **post** is tartozhat.



6–21. ábra: Több-többes kapcsolat ábrázolása blogbejegyzésekkel és címkékkel

Hozzuk létre a **tag**-ekhez tartozó Model-t, migrációs fájlt és az adatgyárat is, mindössze egyetlen utasítással:

```
php artisan make:model Tag -mf
```

6.4.3.1. Migrációs fájl (tags, post_tag)

A migrációs fájlban adjunk hozzá az ott meglévőkhöz egy új oszlopot (**name**), és ugye szeretnénk összekapcsolni a **Post** és **Tag** Model osztályokat, *de hogyan is kellene ezt?* Ha a **tags** táblában hoznánk létre egy **post_id** idegen kulcsot, akkor minden tag csak egy blogbejegyzéshez tartozhatna, ami ugye nem túl valószerű. Ugyanez fordított esetben: ha az **posts** táblában hoznánk létre egy **tag_id** oszlopot, akkor az azt jelentené, hogy csak egy tag tartozna hozzá, ami ugye megint csak nem reális... akkor tehát adódik a kérdés, hogy hol és hogyan is valósítsuk meg ezt a kapcsolatot? Egy *kapcsolótáblát* fogunk ehhez használni. Adatbázis szempontjából ez az az új tábla, amely az összekapcsolni kívánt táblákra hivatkozó külső kulcsokat fogja tartalmazni és így valósítható meg a több-többes adatkapcsolat.

Ezt az új kapcsolótáblát definiáljuk ugyanebben a **Tag**-hez tartozó migrációs fájlban és annak **up()** metódusában, természetesen úgy, hogy betartjuk az ilyenkor szokásos névkonvenciókat:

- A tábla neve a kapcsolatban lévő másik két táblából adódjon össze egy aláhúzással elválasztva: **post_tag**
- Mindkét tábla neve legyen egyes számban az összetételben: **post** és **tag**.
- A táblák sorrendje legyen ABC sorrendben (**post** előrébb van, mint a **tag**).

Amikre figyelni kell a létrehozás magjában:

- Az **id** és **timestamps** mezők megmaradhatnak, hiszen a kapcsolatok létrehozásának idejéről és módosulásáról ezek szolgálhatnak majd információval a későbbiekben.
- Biztosan szükség lesz két külső kulcs mezőre: **post_id** és **tag_id**. Ezek ugye a nevükből is látható táblákra fognak hivatkozni. Ezekhez pedig a külső kulcs hivatkozást is hozzuk létre úgy, ahogy az előző alfejezetben megtettük.
- Viszont ez még így nem lesz elég, mert meg kell adni egy olyan feltételt is, hogy az adott konkrét értékekkel rendelkező **post_id** és **tag_id** csak egyszer fordulhasson elő (például az 2-es számú blogbejegyzést csak egyszer lehessen hozzákötni a 4-es számú **tag**-hez, többször ne, ezt fogja biztosítani nekünk az egyediséget kikényszerítő feltétel).

Ezekután így nézhet ki a **Tag** migrációs fájljának **up()** metódusa:

```
Schema::create('tags', function (Blueprint $table) {
  $table->id();
  $table->string('name')->unique();
  $table->timestamps();
});

Schema::create('post_tag', function (Blueprint $table) {
  $table->id();
  $table->unsignedBigInteger('post_id');
  $table->unsignedBigInteger('tag_id');
```

```
  $table->timestamps();

  $table->foreign('post_id')->references('id')->on('posts')
    ->onDelete('cascade')->onUpdate('cascade');
  $table->foreign('tag_id')->references('id')->on('tags')
    ->onDelete('cascade')->onUpdate('cascade');

  $table->unique(['post_id', 'tag_id']);
});
```

6–50. kódrészlet: tags és post_tag adattáblák migrációs definíciói

A migrációs fájl **down()** metódusában *fordított sorrendben „dobjuk el”* (töröljük) a táblákat, mint ahogy az **up()**-ban létrehoztuk őket:

```
Schema::dropIfExists('post_tag');
Schema::dropIfExists('tags');
```

6–51. kódrészlet: post_tag és tags adattáblák migrációt visszagörgető kódsorai

Mentés után hajtsuk végre a migrálást! Vegyük észre, hogy csak egy PHP fájl került migrálásra, de abban két táblát is definiáltunk, amelyek létre is jöttek az adatbázisunkban.

6.4.3.2. Model fájlok (Post, Tag)

Bővítsük a Model osztályainkat az Eloquent kapcsolatot biztosító metódusok beillesztésével.

A **Post** Model osztályhoz adjuk hozzá ezt a metódust:

```
public function tags()
{
  return $this->belongsToMany(Tag::class);
}
```

6–52. kódrészlet: Post Model osztály bővítése

A **Tag** Model osztályhoz adjuk hozzá a **$fillable** attribútumot és a kapcsolatot lehetővé tevő metódust:

```
protected $fillable = ['name'];

public function posts()
{
  return $this->belongsToMany(Post::class);
}
```

6–53. kódrészlet: Tag Model osztály bővítése

Megjegyzés: ha nem szeretnénk betartani a névkonvenciókat (amit én nem tanácsolok), akkor lehetőségünk van arra is, hogy a kapcsolótábla nevét a **belongsToMany()** metódusok második paraméterében megadjuk, például lehetne így használni többes számban, de ekkor mindkét kapcsolati résztvevő kapcsolatért felelős metódusában ugyanezt a táblanevet adjuk meg:

return $this->belongsToMany(Post::class, 'post_tags');

6.4.3.3. Adatgyár (TagFactory)

A tag-ek mintaadatainak generálásához mindössze a **name** mező értékét kell megadni. Mivel konkrét tag-eink még nincsenek, használjuk a színeket tag-ként, de egyedien, hogy ne jöhessen létre ugyanolyan nevű **tag**-ek.

```
'name' => fake()->unique()->colorName,
```

6–54. kódrészlet: Színek generálása címkeként

Tinker megnyitása után hozzunk is létre néhány tag-et példaként:

```
App\Models\Tag::factory(5)->create();
```

6–24. utasítások: Címkék létrehozása az adatgyár segítségével

A kapcsolótábla feltöltését először a Tinker segítségével fogjuk tudni végrehajtani, mert annak az adatgyára az adatsorok egyediségének kényszere (lásd: 6–50. kódrészlet **unique()** része) miatt egy kicsit bonyolultabban működik.

6.4.3.4. Kapcsolati példaadatok hozzáadása

A kapcsolótáblánkat több módon is tudjuk adatokkal bővíteni, illetve azokat eltávolítani belőle. Legelőször a legegyszerűbb **attach()** és **detach()** metóduspárossal ismerkedünk meg. Előbbivel hozzáadni, utóbbival törölni tudunk adatokat a kapcsolótáblánkból. Adjuk ki a Tinker-ben az alábbi utasításokat, utána pedig sorról sorra elmagyarázom, hogy mi is történik.

```
App\Models\Tag::find(2)->posts()->attach(1);

App\Models\Tag::find(3)->posts()->attach([1,2,3]);

App\Models\Tag::find(3)->posts()->attach(1);

$posts = App\Models\Post::findMany([7,8,9,10]);

App\Models\Tag::first()->posts()->attach($posts);
```

6–25. utasítások: Kapcsolótábla feltöltése az Eloquent-es adatkapcsolat segítségével

Megjegyzés: mivel az utasítások kiadásánál többször használjuk a Tag osztályt úgy, hogy végig kiírjuk a névterével együtt, ez egyszerűsíthető a Tinker-ben is úgy, hogy elsőként a **use App\Models\Tag** importáló utasítást kiadjuk, majd utána már csak a Tag osztály nevét használhatjuk.

A legelső utasítással a 2-es azonosítójú tag-hez a **posts()** kapcsolati metóduson keresztül az **attach()** metódussal hozzáfűztük az 1-es azonosítójú **post**-ot (blogbejegyzést). A második utasításban azt láthattuk, hogy nem csak egyesével tudunk hozzáfűzni a **tag**-hez **post**-okat, hanem egy tömb megadásával több elem is hozzá illeszthető adott **tag**-hez. *A harmadik utasítással végrehajtásánál hibát (QueryException) kellett kapnunk, amely arra vonatkozik, hogy ezzel az ismételt hozzáfűzéssel megsértenénk a kapcsolótábla adatsor egyediségét biztosító kényszert, ezért ezt nem fogja engedni nekünk.* Pont emiatt a hiba miatt lesz szükség arra, hogy más kapcsolat hozzáadási és elvételi metódusokkal is megismerkedjünk a továbbiakban. De egyelőre még maradjunk itt a negyedik utasításnál, amelyben egy új segédmetódust a

findMany()-t ismerhetjük meg. Ez egy gyűjteményt vár paraméteréül, majd visszaadja azoknak az objektumoknak a halmazát, amelyek az érintett adattáblában ilyen azonosítóval rendelkeznek. Ezt először eltárolja egy változóban, majd ezt a gyűjtemény változót szintén lehet az attach() metódussal hozzáfűzni a kapcsolatok halmazához (új adatsorok keletkeznek a kapcsolótáblában).

Ugyanezen utasítások természetesen nagyon hasonlóan működnének fordított esetekben is, ha nem a tag-ekhez fűznénk hozzá post-okat, hanem a post-okhoz tag-eket (figyelni kell persze arra, hogy léteznek-e az adott azonosítójú tag-ek és post-ok, illetve az egyediségre is).

Igény szerint nyugodtan hozzáadhatunk még bármennyi kapcsolatot a post_tag táblánkhoz, ha megsértenénk az egyediségi kényszert, azt a Tinker úgyis jelezni fogja nekünk.

6.4.3.5. Kapcsolati példaadatok lekérdezése

A példaadatok aktuális halmazát mindig ellenőrizhetjük az adatbázis-kezelő menedzserünk segítségével, a kapcsolótábla adatainak lekérésével.

De a Tinker-ben alkalmazhatjuk a Model osztályok kapcsolatért felelős metódusainak meghívását is és a 6.2.2. alfejezetben megismert lekérdező utasítások segítségével tudjuk őket megjeleníteni akár szűrésekkel, rendezésekkel együtt is. Az alábbi két utasítás segítségével lekérjük példaként a 7. blogbejegyzéshez tartozó tag-eket, majd utána a 3. tag-hez tartozó blogbejegyzéseket.

```
App\Models\Post::find(7)->tags;
App\Models\Tag::find(3)->posts;
```

6–26. utasítások: Kapcsolódó post-ok és tag-ek lekérése az Eloquent-es adatkapcsolatok segítségével

6.4.3.6. Kapcsolati példaadatok törlése

Kapcsolat törléséhez a detach() metódust tudjuk segítségül hívni, ami nagyon hasonlóan működik, mint az imént bemutatott attach(), csak fordítva (ez nem létrehozza a kapcsolatokat a kapcsolótáblában, hanem törli őket):

```
App\Models\Post::first()->tags()->detach(3);
App\Models\Tag::first()->posts()->detach([8,9]);
```

6–27. utasítások: Kapcsolódó post-ok és tag-ek törlése az Eloquent-es adatkapcsolatok segítségével

Az iménti utasítások közül először az első post-nál töröljük a kapcsolatot, ami a hármas tag-gel köti össze. A második utasításban az első tag kapcsolatai közül töröljük azokat, amelyek a 8, 9 azonosítójú post-okkal kötötte össze őt.

Megjegyzés: talán hiányérzetünk van amiatt vagy zavaró, hogy a kapcsolótáblában a hozzáadásnál és a szinkronizálásnál nincsenek meg a *„szokásos"* időbélyeg mezők (created_at, updated_at) értékei. Hiszen ez azt jelenti, hogy igazából nincs is értelme létrehozni ezeket a mezőket a kapcsolótáblában. A kapcsolatok létrehozásának időbélyegeit viszont könnyedén pótolhatjuk, ha a két Model osztály (Post és Tag) kapcsoló függvényeit tovább fűzzük a belongsToMany() után, így az egyik (Post) osztályban és a másikban (Tag) is hasonlóképpen:

```
return $this->belongsToMany(Tag::class)->withTimestamps();
```

6–55. kódrészlet: Kapcsolat bővítése időbélyegek értékekkel (created_at, updated_at)

Mentések és a Tinker újraindítása után ellenőrizhetjük is a következő utasítással:

```
App\Models\Tag::first()->posts()->attach([11,12,13]);
```

6–28. utasítások: Beszúrás a kapcsolótáblába időbélyegekkel

6.4.3.7. Kapcsolati példaadatok szinkronizálása

A kapcsolatokat szinkronizálni is tudjuk. Az **attach()** metódussal szemben a **sync()** metódusnál nem kaphatunk **QueryException** hibát a már meglévő kapcsolat *„újradefiniálása"* miatt. A **sync()** paraméterül szintén id-kat kaphat a kapcsolótábla építéséhez, viszont ez *„elfelejti",* hogy milyen kapcsolat létezett korábban és csak azokat hagyja meg, amiket éppen aktuálisan megkap a **sync()** paraméterül. Ha például az első **post**-nak volt kapcsolata az 1, 2, 3 **tag**-ekkel, majd a **sync([2, 4, 6])** lefutása után csak a 2, 4, 6 fog megmaradni.

```
App\Models\Post::find(2)->tags()->sync([2,3,4]);

App\Models\Post::find(2)->tags()->syncWithoutDetaching(5);

App\Models\Post::find(2)->tags()->sync(5);
```

6–29. utasítások: Szinkronizáció hozzáfűzésekkel és elválasztásokkal (illetve azok nélkül is)

A fenti példában ugyanazzal az egy 2-es azonosítójú **post**-tal tevékenykedünk. Az első utasítás a 2-es **post**-nak volt már kapcsolat a 3. **tag**-gel, ezt a **sync()** érintetlenül hagyja, nem törli, de pluszban hozzáadja még a 2. és 4. **tag**-ekkel való kapcsolatot is. A második utasításban kipróbáltuk a **syncWithoutDetaching()** metódust, amely csak az 5. **tag**-et kapja értékül, de a már meglévő 2, 3, 4 **tag**-ekkel nem törli a kapcsolatot. Nem úgy az utolsó utasítás, amely ugyanúgy az 5. **tag**-et kapja csak paraméterül, de ez már törölni fogja az eddig meglévő 2, 3, 4 **tag**-ekkel való kapcsolatot. Az utasítások eredményei itt láthatók:

```
> App\Models\Post::find(2)->tags()->sync([2,3,4])
= [
    "attached" => [
      2,
      4,
    ],
    "detached" => [],
    "updated" => [],
  ]

> App\Models\Post::find(2)->tags()->syncWithoutDetaching(5)
= [
    "attached" => [
      5,
    ],
    "detached" => [],
    "updated" => [],
  ]

> App\Models\Post::find(2)->tags()->sync(5)
= [
    "attached" => [],
    "detached" => [
      3,
      2,
      4,
    ],
    "updated" => [],
  ]
```

6–22. ábra: Szinkronizáló utasítások futtatásának eredményei

Ha újra futtatjuk a **syncWithoutDetaching()**-os utasítást, akkor sem fogunk hibát kapni, mivel a rendszer egyszerűen csak tudomásul fogja venni, hogy erre a kapcsolatra szükségünk van és nem adódik hiba belőle.

Érdemes minél több példa utasítást futtatni a Tinker-ben, hiszen ezeket fogjuk tudni majd alkalmazni a későbbi programkódjainkban is, ezáltal a kapcsolatokat létre tudjuk hozni és törölni egyre ügyesebben. A kapcsolatok innentől kezdve mindig használni fogjuk és részletesen ismertetem majd a felépítésüket a webalkalmazás megjelenítő, hozzáadó, szerkesztő és törlő funkcionalitásai között is.

Az alfejezetben végrehajtott programkód-módosítások ebben a GitHub commit-ben érhetők el.

6.4.4. Mohó betöltés (Eager Loading)

A munkánk során az Eloquent adatkapcsolatokhoz úgy is hozzáfértünk, hogy attribútumként hívtuk meg őket, gondoljunk csak például egy adott blogbejegyzés kommentjeire (**$post->comments**), ekkor a blogbejegyzéshez kapcsolódó kommentek *„lazán töltődtek be" (Lazy loading)*. Ez azt jelenti, hogy a kapcsolat adatai addig nem töltődnek be, ameddig az attribútumon keresztül egyszer le nem kértük őket. Az Eloquent azonban képes a kapcsolatok *„mohó betöltésére"* a szülő Model lekérdezésekor, és így a *„mohó betöltés"* enyhíti az *„N+1 lekérdezés problémáját"*. A problémát úgy kell értenünk, hogy ha van egy kategória, és annak van például 25 blogbejegyzése, akkor az összesen 25+1 adatbázis lekérdezést fog jelenteni (maga a kategória lekérése egy lekérdezés, a további 25 pedig a blogbejegyzések lekérése az adatbázisból).

De mi ezt a Laravel *„eager loading”* technikájával megoldhatjuk mindössze két adatbázis lekérdezéssel, de mindennek vizsgálatához használjuk a *„Debugbar for Laravel”* csomagot.

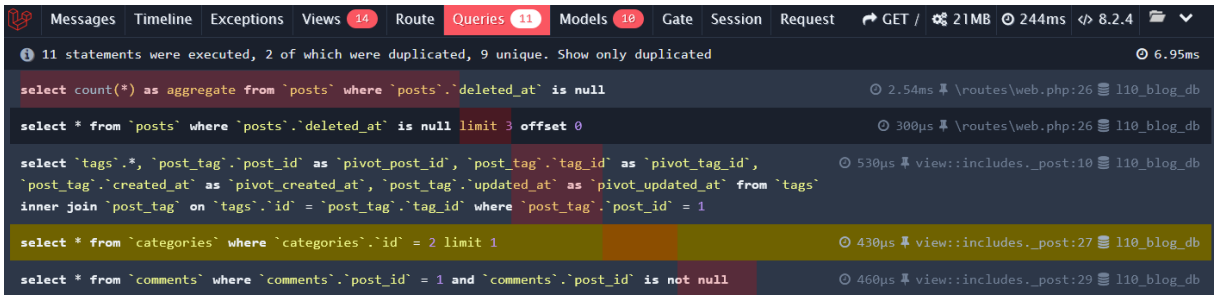
> **Tipp:** az egyik leghasznosabb Laravel csomag, amit a fejlesztés során (lehetőleg már a kezdetén) érdemes telepítenünk a Laravel projektünkbe, az a *„Debugbar for Laravel”*. Itt érhető el: https://github.com/barryvdh/laravel-debugbar
>
> Telepítés után csak akkor működik, ha az **.env** fájlban az **APP_DEBUG** attribútum **true**-ra van állítva. Aktiválás után megjelenik egy alsó sáv a böngészőben, amely tartalmazza az oldal betöltődés idejét, a forgalom méretezését, üzeneteket, kivételeket, nézeteket, útvonalakt és ami most számunkra a legfontosabb: az adatbázis lekérdezéseket és azok idejét, helyét (melyik fájlban és ott hol történt) is nyomon tudjuk követni az oldal betöltések során.

6.4.4.1. Debugbar telepítése és használatba vétele

Telepítsük a csomagot a terminal-ban a leírás alapján a composer segítségével:

```
composer require barryvdh/laravel-debugbar --dev
```

Az alkalmazás elindítása után bal alul látnunk is kell már a Laravel logóját, amire a rákattintás után megjelenik a Debugbar.



6–23. ábra: Debugbar Laravel csomag működés közben (1)

A *„Queries”* lapfülön látszódnak az adatbázis lekérdezések, ami a példában látható képen 11-et mutat, ebből 9 egyedi, 2 duplikált, úgyhogy azt a kettőt érdemes külön megvizsgálni. Láthatók a konkrét lekérdezések, paraméterekkel együtt, illetve, hogy melyik fájl melyik sorában hajtódtak végre és mennyi idő alatt. Így a későbbiekben az időre is tudunk optimalizálni, a hosszú lefutású lekérdezéseknél érdemes hatékonyságnövelést végrehajtani.

6.4.4.2. Adatbázis lekérdezések tesztelés a Debugbar segítségével

De most térjünk vissza az *„N+1 lekérdezés problémájára”*, amit a saját projektünkben a blogbejegyzések és a kommentjeik példáján keresztül fogunk szemléltetni, ha megnyitjuk ezt az útvonalat: http://127.0.0.1:8000/posts. Így betöltődik az összes, *„nem törölt”* blogbejegyzés (**deleted_at** mező értéke null). Ez lesz az 1 az *„N+1”-ből* és utána lekérdezésre kerül a (**posts.index** nézetben lévő) **$post->tags** valamint a **$post->comments** miatt a blogbejegyzések címkéi valamint kommentjei. Ez az én példámban 29-29 további lekérdezést takar a 29 blogbejegyzéshez egyenként. De most csak a kommentekre koncentráljunk, hiszen utána a probléma feloldása a címkékkel is ugyanúgy megoldható. Egy példa lekérdezés a 29 kommentre vonatkozóból:

select * from `comments` where `comments`.`post_id` = 35 and `comments`.`post_id` is not null

Ez a lekérdezés látható összesen 29-szer a listában úgy, hogy csak a **post_id** értékei változnak aszerint, hogy melyik blogbejegyzés *„nem törölt”*. Ezt a 29 lekérdezést kellene lecsökkenteni 1-re az *„eager loading”* segítségével.

Ehhez mindössze annyit kell tennünk, hogy a **PostController index()** metódusában az adatok átadását kell egy kicsit módosítanunk a **with()** segédmetódus meghívásával, amely paraméterként a **Post** Model-ben található kapcsolat metódus nevét kell, hogy megkapja, vagyis a **'comments'**-et:

```
'posts' => Post::with('comments')->orderBy('published_at', 'desc')->get()
```

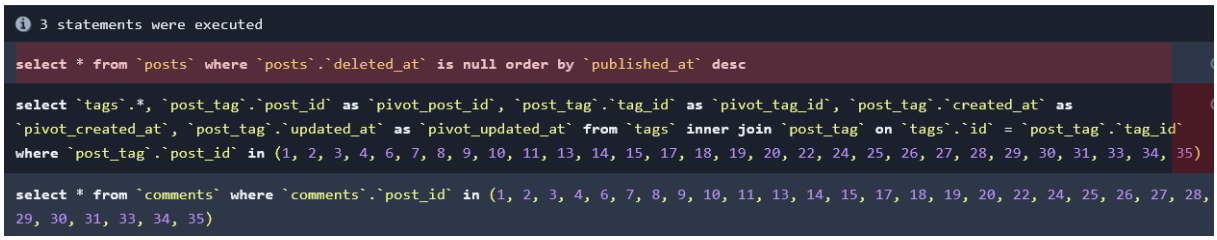
6–56. kódrészlet: Eager Loading implementálása a posts.index nézet adatainál

Ezek után, ha most újra betöltjük az oldalt a böngészőben, akkor látható lesz, hogy a kommentekre vonatkozó 29 lekérdezésből 1 lett, mégpedig ez a lekérdezés:

select * from `comments` where `comments`.`post_id` in (1, 2, 3, 4, 6, 7, 8, 9, 10, 11, 13, 14, 15, 17, 18, 19, 20, 22, 24, 25, 26, 27, 28, 29, 30, 31, 33, 34, 35)

Nyilván a számok mindenkinél mások, a lényeg, hogy a sok (N = 29 nálam) lekérdezést lecsökkentettük 1 darab lekérdezésre. Így sokkal gyorsabbá válik az oldal betöltése. Innentől a **tag**-ekre vonatkozó **with()** segédmetódus hívást beilleszteni a 6–56. kódrészletbe már nagyon egyszerű lesz, és ekkor már csak összesen 3 darab lekérdezés fog végrehajtódni az oldal betöltésekor. A **with()** segédmetódus kaphat tömböt (akár asszociatívan) is paraméterül, így nem kell kétszer meghívni kapcsolatonként külön-külön: **with(['tags', 'comments'])**

Az eredmény itt látható:



6–24. ábra: Debugbar Laravel csomag működés közben (2)

Előfordulhat, hogy amikor egy Model (adatbázis tábla) adatait le szeretnénk kérni, akkor automatikusan hozzá szeretnénk illeszteni a kapcsolataiból érkező adatokat is. Ezt a Model osztály **$with** attribútumában tudjuk megtenni. Ha például a **Post** Model lekéréseinél mindig szükségünk lenne az adott blogbejegyzés kommentjeire és címkéire is, akkor adjuk hozzá a **Post** Model osztályhoz ezt:

```
protected $with = ['tags', 'comments'];
```

6–57. kódrészlet: Kapcsolatok adatainak automatikus hozzárendelése lekérdezéskor (Post Model osztály)

Ezután, ha lekérjük például a **Post::all()** metódus segítségével az összes blogbejegyzést, akkor a címkéket és kommenteket is ki fogja listázni nekünk az adott blogbejegyzés mellett.

Ha ezekután mégis valamelyik kapcsolat adatai nélkül szeretnénk használni az adott Model-en keresztül érkező adatokat, akkor a **without()** segédmetódust használhatjuk az adatlekéréskor (például **Post::without()**). Ha pedig nem az összes **$with** attribútumban meghatározott kapcsolat adataira volna szükségünk, akkor a **withOnly()** segédmetódust használhatjuk, a szükséges kapcsolat megadásával paraméterként (például: **Post::withOnly('comments')**).

Példával illusztrálva tehát itt látható egy *„lazy loading"* lekérdezés:

```
Post::find(1)->comments();
```

Itt pedig egy *„eager loading"* lekérdezés:

```
Post::with('comments')->get();
```

Az alfejezetben módosított programkódokat ebben a GitHub commit-ben lehet megtalálni.

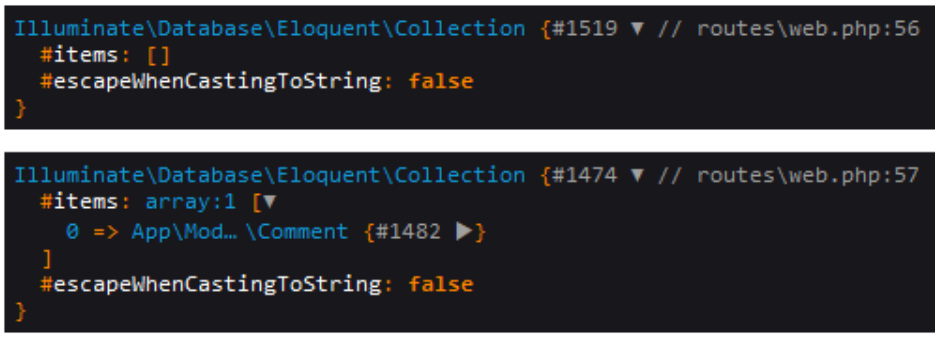
6.4.4.3. Eager loading probléma Laravel 10-ben és javítása a Laravel 11-ben

Tegyük fel, hogy van két blogbejegyzésünk és mindkét blogbejegyzéshez 2-2 komment érkezett. Előfordulhat, hogy az alkalmazásunkban van egy olyan felület, például a főoldalon, ahol csak a blogbejegyzések listázásánál csak a legutóbbi kommentet szeretnénk láthatóvá tenni. Erre az Eloquent Eager loading sajnos nem adott még jó megoldást a Laravel 10-ben. Így nézhet ki egy olyan útvonal magja, amelyben kilistáznánk az első két blogbejegyzést és legutóbbi kommentjüket.

```
Route::get('/eloquent-posts-with-comments', function () {
  $posts = Post::withOnly([
    'comments' => fn ($query) => $query->latest()->first()
  ])->paginate(2);
  dump($posts[0]->comments);
  dump($posts[1]->comments);
});
```

6–58. kódrészlet: Első két blogbejegyzés és legutóbbi kommentjeiknek lekérése

Eredményül azt kapjuk, hogy csak a legutolsó blogbejegyzésnél történik meg a komment megjelenítése:



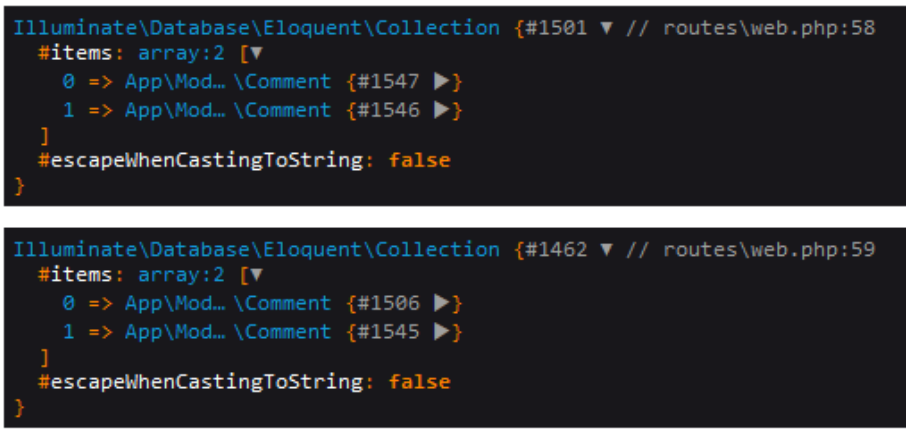
6–25. ábra: Szűrt kapcsolatból érkező hibás csatolt kommentek

Miközben, ha a **withOnly()** metódus tömbjében a 'comments' utáni részt megjegyzésbe teszem:

```
'comments'// => fn ($query) => $query->latest()->first()
```

6–59. kódrészlet: A nem szűrt kapcsolatból adódó két blogbejegyzés 2-2 kommentjének lehívása

Akkor az eredményben jól látható, hogy mindkét kiíratott blogbejegyzésnél ott van a 2-2 komment is:



6–26. ábra: A nem szűrt kapcsolatból adódó két blogbejegyzés 2-2 kommentje

Ezt az SQL-t érintő problémát a Laravel 11 már megoldja nekünk, és a szűrt visszatérési gyűjtemény a **with()** (és **withOnly()**)segédmetódusban már jó eredménnyel adja vissza a blogbejegyzések legutóbbi kommentjeinek eredményeit.

Ugyanakkor ez kiszervezhető az üzleti logikába, vagyis a **Post** Model osztályba is a **comments()** kapcsolatból kiindulva.

```
public function latestComment()
{
  return $this->hasMany(Comment::class)->latest()->first();
}
```

6–60. kódrészlet: Legutóbbi komment kapcsolat a Post Model osztályban

Ekkor pedig frissíthetjük az útvonalunkban lévő kapcsolatra hivatkozást és egyszerűbbé tehető a 6–58. kódrészlet lényegi tartalmához képest:

```
$posts = Post::with('latestComment')->paginate(2);
```

6–61. kódrészlet: Új kapcsolat alkalmazása a blogbejegyzések legutóbbi kommentjéhez

Ez a Laravel 10-ben nem működik így (**addEagerConstraints()** problémát jelez), csak harmadik féltől származó csomag telepítésével. Míg a Laravel 11-ben ez gond nélkül működni fog önmagában. Az alfejezetben módosított programkódok ebben a GitHub commit-ben találhatók meg.

6.5. Több adatgyár együttes működtetése (Seeder)

Ebben a fejezetben már rengeteg manuális tesztelést hajtottunk végre, most azonban először a tesztadatok létrehozására koncentrálunk, aztán pedig megvizsgáljuk, hogy egy-egy Eloquent vagy Query Builder lekérdezés mögött összesen ténylegesen hány adatbázisművelet húzódhat meg. Ebben egy Laravel kiterjesztési csomag, a Debugbar is segítségünkre lesz.

Tesztadatokra nagyon sokszor szükségünk van, már csak azért is, hogy a webalkalmazásunk funkcionalitásait is tudjuk vizsgálni működés szempontjából. Sikeresen tudom-e lekérni a kategóriákhoz tartozó blogbejegyzéseket, vagy a blogbejegyzések tag-jeit meg tudom-e jeleníteni az oldalon... ezekhez

mindig tesztadatokra van szükségünk. Az adatgyárak működését már megismertük, Tinker használatával futtattuk őket, most pedig rátérek a **Seeder** osztályokra, amelyek segítségével *az adatgyárakat csoportosan tudjuk meghajtani* és majd *egyetlen utasítással* egy teljes adatbázis struktúrát tudunk feltölteni tesztadatokkal.

A Seeder osztályok a **database / seeders** mappában helyezkednek el. A mappa nem üres, tartalmaz egy **DatabaseSeeder.php** nevű fájlt, amelyben a **DatabaseSeeder** osztály tartalmaz egy **run()** metódust. Említettem, hogy adatgyárakat közvetlenül a parancssorból nem tudtunk még meghívni, csak a Tinker segítségével. Azonban most, ha kivesszük a kommenteket a **run()** metódus magjában lévő felhasználói gyárakból, akkor a következő utasítással tudjuk elindítani őket:

```
php artisan db:seed
```

Ezt utána **users** adattáblában ellenőrizhetjük is, az első adatgyár 10 példa felhasználót, az utána lévő egy beégetett névvel és e-mail címmel rendelkező felhasználót hoz létre a táblánkban.

id	name	email
1	Emmanuel Murazik III	juvenal.rogahn@example.net
2	Mr. Vicente Rice	bhegmann@example.net
3	Vicenta Tromp Sr.	koepp.amara@example.com
4	Dr. Kariane Simonis	anika04@example.net
5	Norma McKenzie	frederique09@example.org
6	Gabe Raynor	micaela48@example.org
7	Lamar Kuhn	jones.kathryn@example.com
8	Malika Sawayn	istiedemann@example.com
9	Mr. Cortez Turcotte	ezequiel.weissnat@example.org
10	Prof. Noemi Torphy	riley.steuber@example.org
11	Test User	test@example.com

6–27. ábra: users tábla tartalma a db:seed után

Azt adatgyárak így tehát simán működésre bírhatók egyetlen utasítással. A további Seeder osztályok hasznát abban láthatjuk, hogy csoportba lehet szervezni az egyes Factory osztályok futtatását, és nem kell az összes gyárat külön-külön futtatni, hanem a **DatabaseSeeder** osztályba integrálhatjuk így.

A példa kedvéért hozzunk létre két olyan Seeder osztályt, amely a következők szerint fogja működtetni a gyárakat:

1. Kategória (**Category**), blogbejegyzés (**Post**), értékelés (**Rating**) szerint generál példa adatokat.
2. Címke (**Tag**), blogbejegyzés (**Post**) – így közvetve az értékelés (**Rating**) – és a **post_tag** kapcsolótáblájukba generál példa adatokat.

Utána ezt a két csoportot fogjuk meghívni az alapértelmezett **DatabaseSeeder** osztályunkba és a `db:seed` utasítással futtatjuk őket. Kezdjük a Seeder osztályok létrehozásával (az iménti felsorolási pontok első osztályához rendelem őket):

```
php artisan make:seeder CategorySeeder
```

```
php artisan make:seeder TagSeeder
```

A **CategorySeeder** osztályban hozzunk létre adatgyárakkal (amelyek már léteznek is), 5 kategóriát és 10 blogbejegyzést (a **PostFactory** osztályban automatikusan létrejön a 10 értékelés is, így a **RatingFactory** osztályt itt nem kell külön megfuttatnunk):

```
public function run(): void
{
  Category::factory(5)->create();
  Post::factory(10)->create();
}
```

6–62. kódrészlet: CategorySeeder osztály lényegi tartalma (a Category és Post osztályokat importáljuk is felül!)

Ha egy konkrét Seeder osztályt szeretnénk futtatni, akkor így tehetjük meg:

```
php artisan db:seed --class=CategorySeeder
```

A **TagSeeder** osztály tartalmazzon 20 címkét, amelyet a **TagFactory** osztály segítségével létre tudunk hozni, 10 újabb blogbejegyzést és 30 adatkapcsolatot a **Post-Tag** osztályok között. Talán ez utóbbit egy kicsit nehezebb még megvalósítani, de itt látható rá a példa:

```
public function run(): void
{
  $tags = Tag::factory(20)->create();

  Post::factory(10)->create()
    ->each(function ($post) use ($tags) {
      $post->tags()->sync($tags->random(2));
    });
}
```

6–63. kódrészlet: Tag és Post adatok gyártása kapcsolati adatokkal együtt

A 20 **tag** létrehozása most már triviális, az utasítás eredményét eltároltuk a **$tags** gyűjteményben, amelyet utána felhasználtunk a 10 blogbejegyzés létrehozása során és amikor egy blogbejegyzés létrejött a **posts** táblában, akkor véletlenszerűen kiválasztott nekünk a rendszer a kapcsolat segítségével melléjük 2 darab **tag**-et. Mivel a **sync()** metódust használtam, ezért hiba nem adódhat ebből, ha esetleg a randomizáció ugyanazt a két értéket generálná neki.

A **database / DatabaseSeeder.php** fájl **run()** metódus magja ezt az utasítást tartalmazza:

```
$this->call([
  CategorySeeder::class,
  TagSeeder::class
]);
```

6–64. kódrészlet: Erőforrásokat (kategória, blogbejegyzés, címke) létrehozó Seeder-ek futtatásához

A seed-elő parancs futtatása előtt ellenőrizzük, hogy a **categories, ratings, tags, posts, post_tag** tábláink mennyi adatsorral rendelkeznek, majd futtathatjuk a seed-elést:

```
php artisan db:seed
```

A futtatás eredménye itt látható:

```
PS C:\xampp\htdocs\l10-components> php artisan db:seed

  INFO  Seeding database.

  Database\Seeders\CategorySeeder ..... RUNNING
  Database\Seeders\CategorySeeder ..... 489.06 ms DONE

  Database\Seeders\TagSeeder ..... RUNNING
  Database\Seeders\TagSeeder ..... 109.08 ms DONE
```

6–28. ábra: Seeder osztályok futtatása a terminal-ban

Utána ellenőrizzük újra az adattábláinkban lévő adatsorok mennyiségét és hogy megfelelő adatokkal lettek-e feltöltve, de ha korábban a Factory osztályokat jól definiáltuk, akkor itt sem lehet ebből probléma.

Az alfejezet végére egy *tippet* tartogattam: ha egy Model-hez a Laravel-ben szeretnénk egyből Factory és Seeder osztályt is létrehozni, akkor a következő parancsot alkalmazhatjuk:

php artisan make:model **ModelName** -fs

Az alfejezetben végrehajtott programkód-módosítások ebben a GitHub commit-ben érhetők el.

Megjegyzés: a kimondottan adatbázisok tesztelésére használt **assert** utasítások itt érhetők el. Az ilyen assert utasításokat végrehajtó automatikus tesztelésekre a következő fejezet végén (7.6.2. alfejezetben) láthatunk majd példákat.

Tipp: Teszteld a tudásod!

Bár még nem tudunk mindent a kapcsolatokról, de érdekes lehet kipróbálni az eddigi tudásunkkal, milyen teszteknek tudnánk megfelelni. Léteznek nyilvános repo-k, amelyekkel ezt a tesztelést meg tudjuk tenni, itt van például egy, ami az adatkapcsolatokhoz tartozik:

https://github.com/LaravelDaily/Test-Eloquent-Relationships

Ha klónozzuk a projektet és elindítjuk a tesztelést, akkor kezdetben minden tesztünk elbukik. De a GitHub oldalon a fájlok alatt olyan instrukciók vannak, amelyek alapján, ha végrehajtjuk a megvalósítást (implementálást), akkor át fognak menni a tesztjeink.

A többalakú (polymorphic) kapcsolatok kezelésről még nem tanultunk, így azokat nem kötelező figyelembe venni a saját tudásunk felmérésénél. Viszont kialakulhatott már annyi rutinunk, hogy esetleg a Laravel dokumentáció megfelelő részét használva is megoldható az, hogy átmenjenek sikeresen a fejlesztéseink az alkalmazásban definiált teszteseteken.

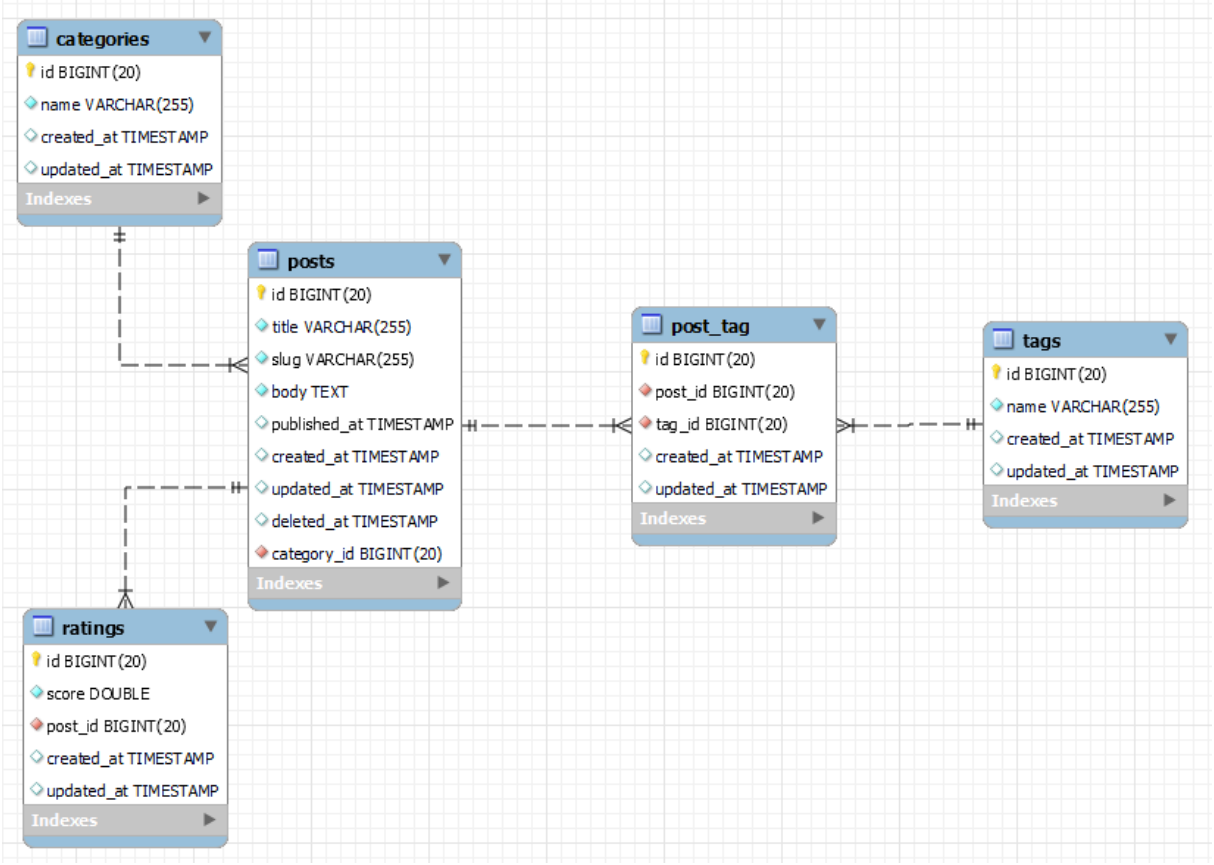
Gyakorlati szempontból az adatbázis értesítéseknél (12.3.4. alfejezetben) mi is fogunk még találkozni ilyen típusú (**morph()**) kapcsolattal.

6.6. Összegzés

A fejezet során áttekintettük a Laravel adatbázis-kezelést támogató két legfontosabb eszközét: az Eloquent ORM-et és a Query Builder-t. Mindkét eszközzel részletes lekérdezéseket (SELECT) és adatmanipulációs műveleteket (INSERT, UPDATE, DELETE) futtattunk egy-egy adattáblát érintve.

A fejezet további részében az adatkapcsolatok működésével ismerkedtünk meg (1-1, 1-n, n-n) a Laravel-ben. A keretrendszer ezeken kívül tartalmaz még egyéb Eloquent ORM kapcsolati típusokat, azonban azokat ez a könyv már nem tárgyalja, viszont a hivatalos dokumentáció bemutatja őket.

Végül kialakult ezáltal egy olyan adatbázis struktúra (releváns részét lásd: 6–29. ábra), amelyet a következő fejezetben már a webes alkalmazással tudunk lekérdezni (kiolvasni) és az adatait manipulálni (létrehozni, szerkeszteni, törölni).



6–29. ábra: Kibővített Egyed-Kapcsolat modell (MySQL Workbench alkalmazással készült, a diagram pontos értelmezéséhez az [5] jegyzet 4. fejezete ad háttértudást, nekünk ebből elég látni a táblákat és kapcsolataikat)

Az alkalmazást tesztelés alá vettük adatbázis szempontból, adatokat töltöttünk bele Seeder osztályokkal is. Legvégül pedig az N+1-es adat lekérdezési problémát oldottuk meg a Laravel mohó betöltése (*„eager loading”*) segítségével.

7. CRUD útvonalak, vezérlő metódusok és a RESTful API (CRUD routes, controller actions and the RESTful API)

A fejezetben áttekintjük a leggyakrabban alkalmazott *CRUD (Create-Read-Update-Delete)*, vagyis a létrehozó, lekérdező, frissítő és törlő műveletek végrehajtásához szükséges hét útvonalat és a hozzájuk kapcsolódó vezérlő metódusokat.

7.1. Bevezetés, áttekintés

A Laravel a REST (REpresentational State Transfer) architekturális tervezési mintát követő működést az útvonalain, vezérlő metódusain keresztül valósítja meg. Ez nem egy protokoll, viszont szorosan kapcsolódik a HTTP protokollhoz, amely ugye a webes szereplőket és kommunikáció részleteit határozza meg. Ha egy ilyen alkalmazás és architektúrája (mint például a Laravel is) megfelel a következő megszorításoknak, akkor szokták *„RESTful"*-nak nevezni: (1) kliens-szerver architektúra, (2) állapotmentesség – statelessness, (3) gyorsítótárazhatóság – cacheable, (4) réteges felépítés – layered, (5) igényelt kód – opcionális feltétel, (6) egységes felület – interface.

Modern webes alkalmazásokban a frontend és a backend API-n keresztül webszolgáltatások segítségével kommunikálnak egymással. Ez annyit jelent, hogy a felhasználó elér meghatározott útvonalakat a böngészőjében a frontend-en keresztül, amely utána megszólítja a backend-et és lekér (GET) tőle, feltölt (POST), módosít (PUT/PATCH) vagy töröl (DELETE) erőforrásokat, majd utána az eredményt visszaküldi a frontend oldalra és megjeleníti a felhasználónak.

Művelet	HTTP metódus	SQL utasítás
Create (létrehozás)	POST	INSERT
Read (kiolvasás)	GET	SELECT
Update (frissítés)	PUT vagy PATCH	UPDATE
Delete (törlés)	DELETE	DELETE

7–1. táblázat: CRUD műveletek és a hozzájuk tartozó HTTP metódus és SQL utasítás

A leggyakoribb műveletek megvalósításához ezekre a HTTP metódusokra (GET, POST, PUT/PATCH, DELETE) és erőforrás azonosítókra (URI) van szükség (7–2. táblázat). A HTTP metódusokat és az erőforrás azonosítókat együttesen kell kezelni (a táblázat első két oszlopa), ezeknek együtt kell egyedinek lenniük, ezért fordulhat elő, hogy például a **/tags** többször is szerepel a táblázatban, de egyszer GET máskor POST metódussal párosítva. A HTTP metódusokat és az erőforrás azonosítókat együttesen útvonal végpontoknak (endpoint) nevezzük.

Mi itt most még nem építünk API-t, de az alapját biztosító CRUD útvonalakat és vezérlő metódusokat megvalósítjuk a webes alkalmazásunkban. De talán egy táblázat többet mond ezer szónál, az alábbiakban a **tag**-ek kezelését bemutató útvonalakat és vezérlő metódusokat sorolom fel és magyarázom.

HTTP metódus	Erőforrás azonosító (URI)	Vezérlő metódus (Controller action)	Működés leírása *Példa*	*HTML űrlap metódus (HTML form method)*
GET	**/tags**	*Index*	Lekérjük az összes tag-et és megjelenítjük őket	-
GET	**/tags/{id}**	*Show*	Lekérünk egy konkrét tag-et (id alapján) és megjelenítjük	-
GET	**/tags/create**	*Create*	Megjelenítjük a tag létrehozáshoz szükséges nézetet (üres űrlapot)	-
POST	**/tags**	*Store*	Eltárolunk egy tag-et a tags adattáblában (INSERT)	*POST*
GET	**/tags/{id}/edit**	*Edit*	Lekérünk egy konkrét tag-et (id alapján) és a szerkesztéséhez szükséges űrlapot megjelenítjük	-
PUT	**/tags/{id}**	*Update*	Frissítünk egy konkrét tag-et (id alapján) (UPDATE)	POST, de PUT vagy PATCH metódussal
DELETE	**/tags/{id}**	*Destroy*	Törlünk egy konkrét tag-et (id alapján) (DELETE)	POST, de DELETE metódussal

7–2. táblázat: CRUD útvonalak (1. és 2. oszlop) és vezérlő metódusok (3. oszlop) a RESTful működéshez

A 7–2. táblázat sorainak színezésével próbáltam jelezni az összetartozó párokat. Sárgával (citrom és narancs) jelöltem az adatbázis erőforrások lekérését (SELECT). A GET HTTP metódussal képesek vagyunk lekérni a '/tags' és a '/tags/{id}' erőforrásokat. Mindkét útvonal elérésekor a böngészőben valamilyen erőforrást kérünk le, nevezetesen a tags adatbázistábla tartalmát az index metódusban teljes egészében, míg a show() metódusban csak egy darab sort belőle (amit az id azonosít).

Zölddel (világos és sötét) a létrehozáshoz szükséges két útvonalat és vezérlő metódust jelöltem. A világoszöld sor egy GET-es elérést mutat, ami a create() vezérlő metódus végrehajtásához vezet. Az útvonal lekérés eredményeképpen a felhasználó böngészőjének átadunk egy olyan üres (nem kitöltött) űrlapot, amelynek kitöltésével megadhatja a Tag osztály példányának, és implicit módon a tags tábla új adatsorának mező értékeit. A sötétzöld sorban jelzett POST-os útvonal végpont elérése a create űrlap kitöltése után a küldés gomb megnyomásával élesedik. Az útvonal a vezérlőben a store() metódushoz vezet, amely ténylegesen létrehozott az új Tag objektumot és a tags táblába egy új sort szúr be (INSERT) a felhasználótól kapott értékekkel. A táblázat sorainál, mivel ezek (create és store) összefüggőek, bekereteztem őket.

A táblázat következő két sorában az adattáblában már meglévő sort tudunk frissíteni (UPDATE), vagy ha az MVC tervezési minta szerint szeretném megfogalmazni, akkor a példaként használt Tag Model osztály már meglévő példányának adattagjait tudom szerkeszteni és frissíteni. Kezdjük az Edit-tel, hiszen ez vonatkozik a szerkesztésre. A GET-es útvonal meglátogatásával egy űrlapot kap vissza a felhasználó, de már nem üreset, mivel ez az adatsor a tags adattáblában már létezik és most az űrlap mezői ennek a sornak az értékeit kapják meg. Utána a mezők értékét a felhasználó átírhatja, majd, amikor végzett a szerkesztéssel akkor a küldés (vagy frissítés) gomb megnyomásával kerül meghívásra az update() vezérlő metódus a PUT-os útvonalon keresztül (lehetőségünk van a PUT helyett PATCH-et is használni, a kettő között látszólag nincsen különbség, ugyanúgy működnek, de míg a PUT esetében a teljes erőforrás –

minden elemével – frissítésre kerül, addig a PATCH esetében az erőforrásnak csak a megváltozott része fog frissülni). Ebben a metódusban már nem egy új sort szúrunk be a **tags** adattáblába, hanem a meglévő sor mezőit frissítem az érkező esetlegesen új értékekkel. A táblázat sorainál, mivel ezek (Edit és Update) is összefüggőek, bekereteztem őket, továbbá világos és sötét kék háttérszínnel láttam el őket.

A táblázat legutolsó sorában látszódik a **destroy()** metódus, amelyet a DELETE-es útvonalon keresztül tudunk elérni. Ez a paraméterként megkapott adatbázis sort törölni (DELETE) fogja a **tags** adattáblában.

Minden olyan útvonal, ami GET-es, azt a felhasználó kéri le a böngészője címsorába beírva a címet, vagy pedig egy linkre rákattintva. A POST, PUT/PATCH, DELETE HTTP metódusokat az űrlapok elküldésével tudjuk végrehajtani.

Fontos, hogy a HTML űrlap elküldését csakis GET vagy POST **method** attribútum értékkel tudjuk elküldeni, a szabvány ezt engedi meg nekünk, de mivel nekünk szükségünk van PUT/PATCH és esetleg DELETE metódusok szerinti küldésre is (a 6. és 7. útvonalaknál), ezért utóbbiak esetén egy kis trükköt alkalmazunk: POST-ként küldjük el az űrlapot, majd az űrlapon belül állítjuk be a PUT / PATCH, DELETE értékeket.

Ahogy az talán leszűrhető volt a leírásból, most minden felhasználói kérést úgy fogunk kiszolgálni, hogy az MVC tervezési minta hosszabb ágát járjuk be: útvonalak az útvonalválasztóban, vezérlők, modellek, nézetek (lásd: a 3.4. alfejezet).

A továbbiakban a CRUD műveletek ismertetését adatkapcsolatok szerint csoportosítva mutatom be:

1. először olyan adattáblával, amelyben nincsen idegen kulcs,
2. majd olyan táblával, amiben már van idegen kulcs egy másik tábla mezőjére,
3. végül pedig egy több-többes adatkapcsolatban résztvevő táblával és annak kapcsolótáblájával szemléltetem a CRUD műveletek végrehajtását.

7.2. Adatkapcsolat nélkül vagy az „1-n” adatkapcsolat „1” oldalán

A legegyszerűbb megoldást akkor tudjuk alkalmazni, ha az aktuális adattábla nem rendelkezik idegen kulcs kapcsolattal. Ez alapján könnyedén meg tudunk ismerkedni a CRUD műveletek implementálásának folyamatával. Mindig az útvonal végpont regisztrációs fájlból (**routes / web.php**) indulunk, ott létrehozzuk az útvonalat. Ezután következik az erőforráshoz tartozó Controller osztály (**app / Http / Controllers**), amelyben az aktuális metódussal kell foglalkozni, amelyhez az útvonaltól megérkeztünk. A Controller metódusában a céltól függően vagy egy nézetet adunk vissza a felhasználónak, vagy pedig végrehajtjuk az erőforrás létrehozását, frissítését, törlését. Nézet megjelenítés szempontjából először csak a lényegre szeretnék koncentrálni és miután minden működik, beépítjük a nézeteket a webalkalmazás sablonunkba.

A gyakorlati projektünkben a kategóriákat tartalmazó tábla nem tartalmaz külső kulcsot, bár a blogbejegyzések hivatkoznak rájuk, így az adatkapcsolat (1-n) fennáll, de egyelőre az egyszerűbb kezelését a **categories** táblának itt meg fogjuk tudni tenni az alfejezetek ismertetése során, mivel az „1” oldalon van a **categories** tábla.

7.2.1. Több adatsor kiolvasása (index, 1-n)

Először létrehozzuk a legelső útvonalat, ami kilistázza a tag-eket. Mindenekelőtt a web.php fájl elején importáljuk az app / Http / Controllers / CategoryController osztályt, majd jöhet az útvonal, amelyhez a 7–2. táblázatot mindig használhatjuk *„puskaként”:*

```
Route::get('/categories', [CategoryController::class, 'index']);
```

7–1. kódrészlet: Kategóriákat listázó (index) útvonal létrehozása

Hozzuk létre a CategoryController-t (a Laravel névkonvencióit betartva, egyes számban legyen a neve):

```
php artisan make:controller CategoryController
```

A CategoryController osztályban hozzuk létre az index() metódust és kérjük le benne az összes kategóriát, viszont még ne a nézetnek adjuk át őket, csak JSON formátumban jelenítsük meg a böngészőnkben:

```
public function index()
{
  $categories = Category::get(); // Adatbázis lekérdezés: SELECT * FROM
categories;
  return $categories;
}
```

7–2. kódrészlet: Kategóriák kilistázása JSON formátumban

Ez így rendben is van, de a lehető legritkábban kell az összes kategóriát ilyen formában visszaadnunk a felhasználónak, ezért a get() függvény előtt lekérhető a take() és paramétereként, hogy hány kategóriát szeretnénk megkapni:

$categories = Category::take(10)->get();

Így nem az összes, hanem csak (maximum) 10 darab kategória fog megjelenni a böngészőben a /categories útvonal lekérésével. Ha pedig túl sok adatsor van a táblánk, akkor lapozhatóvá is tudjuk tenni a gyűjteményt, például ötösével:

$categories = Category::paginate(5);

De a szűrés (WHERE) vagy éppen a rendezés (ORDER BY) alkalmazása is hasonlóképpen nagyon könnyen megvalósítható, hiszen minden, amit a 6.2. alfejezetben tanultunk az Eloquent Model-ekről, azok itt is nagyon hatékonyan használhatók, nem csak a Tinker-ben.

A CategoryController.php fájl elején importáljuk az App\Models\Category osztályt, majd utána valósítsuk meg az index metódust:

```
public function index()
{
  return view('categories.index', [
    'categories' => Category::orderBy('name')->get()
  ]);
}
```

7–3. kódrészlet: Kategóriákat listázó index metódus és a rendezett lista átadása a megfelelő nézetnek

Adjuk a kategóriákat név szerint rendezett formában a **resources / views** mappában létrehozott **categories** mappán belüli **index** nézetnek (a pont elválasztás a mappa és a nézet fájl neve között használatos).

A **resources / views / categories** mappa még nem létezik, úgyhogy hozzuk ezt létre, majd abba az **index.blade.php** fájlt is.

Mivel az egyszerűségre törekszem, hogy a lényeg látható legyen, ezért a nézet fájlban csak egy táblázatos megjelenítést definiálok a kategóriáknak.

```
<h1>Kategóriák</h1>
<table>
  <thead>
   <tr>
      <th>Név</th>
    </tr>
  </thead>
  <tbody>
    @foreach ($categories as $category)
      <tr>
        <td><a href="/categories/{{ $category->id }}">{{ $category->name
}}</a></td>
      </tr>
    @endforeach
  </tbody>
</table>
```

7–4. kódrészlet: categories.index nézet fájl kezdeti tartalma

A táblázat celláiban már úgy definiáltam a kategóriák neveinél a linkeket, hogy azok egyből a **Show** útvonalhoz vezessenek.

7–1. ábra: Kategóriákat listázó nézet megjelenítése a böngészőben

7.2.2. Egy adatsor kiolvasása (show, 1-n)

Ha jelen állapotban rákattintunk a böngészőben valamelyik kategória nevére, akkor még a 404-es hibaoldal jön be. Hozzuk létre ennek is az útvonalat a **web.php** fájlban:

```
Route::get('/categories/{id}', [CategoryController::class, 'show']);
```

7–5. kódrészlet: Egy konkrét kategória részleteit megjelenítő útvonal létrehozása

Az útvonal erőforrás azonosítójánál látszódik, hogy ez már tartalmazni fog egy paramétert is (**{id}**), amelyet a Controller **show()** metódusának paramétereinél is jelezni kell (**$id**). Hozzuk létre ezután a **show()** metódust a **CategoryController** osztályban! *Megjegyzés:* VSCode-ban ha beírjuk a *„fun”*-t, majd TAB-ot nyomunk, akkor legenerálja nekünk a metódus sablonját, így TAB-bal tudunk lépkedni a metódus neve, paraméterei és magja között.

```
public function show($id)
{
  $category = Category::findOrFail($id);
  return view('categories.show', compact('category'));
}
```

7–6. kódrészlet: Egy kategória részleteit lekérő és adattovábbító show metódus

A **categories** nézeteket tartalmazó mappában még nem létezik a show, úgyhogy hozzuk oda létre a **show.blade.php** fájlt. A tartalma pedig legyen a következő:

```
<h1>{{ $category->name }}</h1>
<p>Létrehozása: {{ $category->created_at }}</p>
<p>Módosítása: {{ $category->updated_at }}</p>
<a href="/categories">Vissza a listára</a>
```

7–7. kódrészlet: categories.show nézet fájl kezdeti tartalma

Így tehát már látható a kategória részletezése és egy linket is elhelyeztem a végén, amivel vissza lehet lépni a kategóriákat kilistázó index oldalra.

7–2. ábra: Egy kategóriát részletező nézet megjelenítése a böngészőben

7.2.3. Új adatok feltöltése (create, 1-n)

Már most fontos megjegyezni, hogy mivel itt már átlépünk arra a veszélyes területre, ahol a *„felhasználókkal kommunikálunk”*, emiatt gondoljunk a biztonságra! Tekintsünk úgy minden bemeneti értékre, mintha az ördögtől érkezett volna... ez persze nem azt jelenti, hogy hagyjuk figyelmen kívül őket,

hanem csak azt, hogy járjunk el nagyon körültekintően az érkező bemeneti értékekkel kapcsolatban, törekedve a biztonságra. Az ilyen bemeneti adatokat mindenképpen ellenőrizni kell kliens (HTML5 és JavaScript eszközeink lesznek erre) és szerver oldalon is (itt majd a Laravel segít nekünk a validációban, ellenőrzésben). A validáció témakörét a későbbiekben (8.5. fejezet) fogom ismertetni, most egyelőre elég annyit tudnunk róla, hogy majd kliens és szerver oldalon is ellenőrizni fogunk.

Hozzuk létre azt az útvonalat, amelyen keresztül megjelenítjük majd a felhasználóinknak azt az űrlapot, ahol meg tudják adni például a kategóriák nevét.

```
Route::get('/categories/create', [CategoryController::class, 'create']);
```

7–8. kódrészlet: Új kategóriát definiáló űrlaphoz vezető útvonal létrehozása

Tipp: Ha a **show** útvonal regisztrációja hamarabb (előrébb) történik meg a **web.php**-ban, mint a **create** útvonalé, akkor az általunk gondolt **create** útvonal (például: **/tags/create** elérésére a böngésző 404-es hibakódot fog adni, mivel a **create** azonosítójú (id) adatsort keresi az adattáblában, viszont ilyen természetesen nincsen, úgyhogy a **create** útvonalat helyezzük át a **show** útvonal regisztrációja elé.

A **CategoryController**-ben hozzuk létre a **create()** metódust:

```
public function create()
{
  return view('categories.create');
}
```

7–9. kódrészlet: Kategória létrehozásához szükséges űrlap lekérése a create metódusban

A **categories** nézet mappában hozzuk létre a **create.blade.php**-t és adjuk hozzá ezt a tartalmat:

```
<form action="/categories" method="post">
  <p>
    <label for="nev">Kategória neve:</label>
    <input type="text" name="name" id="nev">
  </p>
  <input type="submit" value="Mentés">
</form>
```

7–10. kódrészlet: categories.create nézet fájl kezdeti tartalma

Megjegyzés: az űrlap elkészítésénél most még tekintsünk el a kliens- és szerveroldali validációtól.

Az űrlap a POST HTTP metódus szerinti **/categories** erőforráshoz fogja elküldeni az űrlapban kitöltött bemeneti mezők értékeit a mentés gomb megnyomásával. Ennek a tesztelését azonban még nem tudjuk megtenni, mivel az útvonal még nem lett beregisztrálva hozzá, illetve még a **store()** metódus sem létezik a **CategoryController** osztályban. Bővítsük ki a **categories** mappán belüli index nézetet a fájl elején a h1-es címsor után egy új sorral, ami a **create** nézethez tudja vezetni majd a felhasználóinkat:

```
<a href="categories/create">Létrehozás</a>
```

7–11. kódrészlet: Létrehozó link az index nézetben

7.2.4. Új adatok elmentése (store, 1-n)

Regisztráljuk be először a **web.php**-ban a **store** útvonalát:

```
Route::post('/categories', [CategoryController::class, 'store']);
```

7–12. kódrészlet: Egy új kategóriát adattáblába elmentő útvonal létrehozása

Majd hozzuk létre a **CategoryController**-ben a **store()** metódust. A felhasználói űrlap kitöltésének eredményei a **request()** segédmetódussal hozzáférhetővé válik, paraméterül kell neki adni az űrlap adott input mezőjének **name** attribútum értékét, ez most a mi példánkban ugyanúgy **name** volt, hiszen a **categories** adattáblában a kategória neve a **name** mezőbe kerül be. Először azonban írassunk csak ki minden értéket, ami az űrlaptól érkezik.

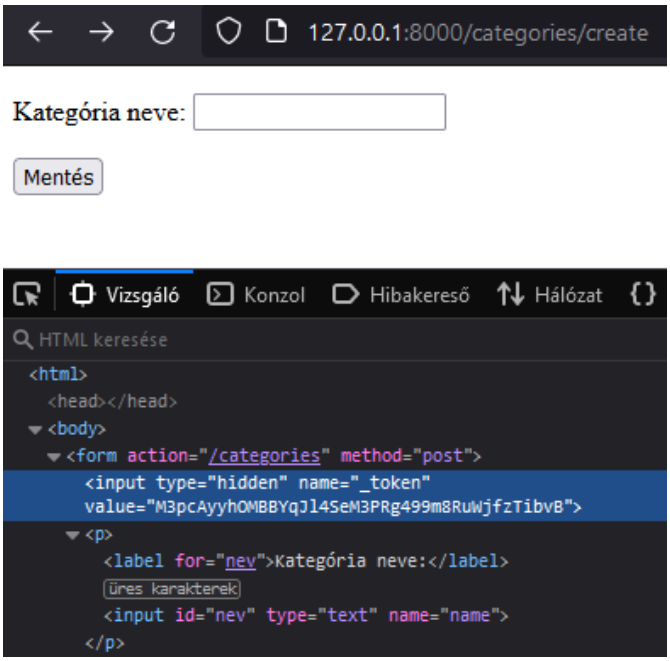
```
public function store()
{
  dd(request()->all());
}
```

7–13. kódrészlet: Kategória létrehozásához szükséges metódusban az űrlap mezők értékeinek kiíratása

Hajtsunk végre egy tesztelést, az űrlap segítségével hozzunk létre egy kategóriát és vizsgáljuk meg, hogy mi történik!

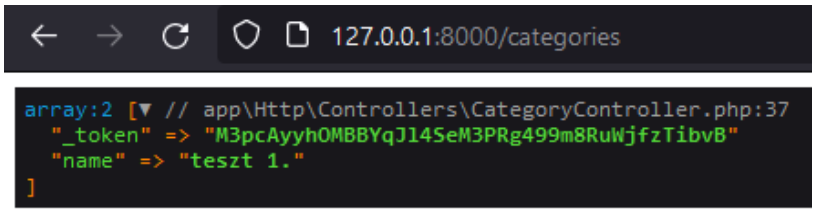
419-es HTTP állapotkódot (hibakódot) kapunk. Ez szintén egy Laravel-es biztonsági mechanizmus miatt van. A rendszer védi az oldalt a **Cross-Site Request Forgery (CSRF)** támadástól. További infók róla elérhetők itt és itt. A támadás elleni védelemhez nekünk egy token-t kell generálnunk, ami segítséget nyújt a védekezésben. Adjuk hozzá a **create** nézet űrlapban a form nyitó tag-je után ezt: **@csrf**

A **create** nézet mentése és az útvonal frissített lekérése után a böngészőben látszólag nem történt semmilyen változás, de ha megvizsgáljuk az űrlap DOM (Document Object Model) fa szerkezetét a Vizsgáló (*„Inspector”*) lapfülön, akkor láthatjuk, hogy bekerült egy rejtett (**hidden**) bemeneti mező az űrlapba, értékül pedig egy véletlenszerűen generált karaktersorozatot kapott. Ennek az értékét fogja a Laravel megvizsgálni és a kategória elmentése előtt ellenőrizni, hogy ugyanaz-e a kapott érték a szerver oldalon, ami az űrlap létrehozásakor a kliens oldalon szerepelt.



7–3. ábra: CSRF token mező bekerült az űrlap bemeneti elemei közé rejtetten

Írjuk be a kategória nevéhez, hogy *„teszt 1.”*, és kattintsunk a Mentés gombra. Ezután már sikeresen megtörténik a felhasználói kérés kiszolgálása és a **store()** metódusra visz rá, ahol az űrlap kitöltésének mezőértékei kerülnek átadásra és kiíratásra, benne a **csrf token**-nel.



7–4. ábra: Felhasználói űrlap kitöltésének eredménye és kiíratása

Ahogy azt már megismertük, a **Category** Model osztálynak használható a **create()** metódusa a létrehozáshoz, amihez mindössze arra van csak szükség, hogy a **$fillable** attribútumában a kitölthető mezők fel legyenek sorolva. A helyesen működő **store()** metódus tehát így néz ki:

```
public function store()
{
  // dd(request()->all());
  Category::create([
    'name' => request('name')
  ]);

  return redirect('categories');
}
```

7–14. kódrészlet: Új kategória elmentését (beszúrását) elvégző store() metódus

A **create()** metódus magjában lévő tömb lecserélhető a **request()->all()** utasításra, hiszen úgyis csak az a mező tölthető fel az adatbázisban, amelyet korábban a Model osztály **$fillable** attribútumában

engedélyeztünk, azonban ez kicsit veszélyes lehet, főleg, hogy ha ott könnyelműek voltunk és esetleg a **$guarded = [];** attribútum értéket használtuk helyette. Így biztonsági szempontból mégiscsak célravezetőbb, ha felsoroljuk a mezők nevét az asszociatív tömb kulcs elemeinél és az értékeknél pedig a **request()** segédmetódussal a megfelelő űrlap bemeneti mező értékét rendeljük hozzá.

Végül a **redirect()** segédmetódus segítségével átirányítjuk a felhasználót a **/categories** (GET-es) útvonalra, ami az index nézetet hozza be a böngészőjében, az ottani listában pedig már látható is lesz az újonnan létrehozott kategória.

7.2.5. Meglévő adatok szerkesztése (edit, 1-n)

Ez az útvonal és vezérlő metódus hasonlít a **create()**-hez, mivel egy űrlapot fog ez is megjeleníteni a felhasználónak, de nem üresen, hanem a bemeneti mezők ki lesznek töltve az űrlapon. De kezdjük az útvonal létrehozásával a **web.php** fájlban:

```
Route::get('/categories/{id}/edit', [CategoryController::class, 'edit']);
```

7–15. kódrészlet: Egy meglévő kategória frissítéséhez szükséges űrlapot megjelenítő útvonal létrehozása

Hasonlít továbbá a **show()** vezérlő metódus magjára is, hiszen az **$id** alapján kikeressük a kategóriát és átadjuk az **edit** nézetnek, ami az űrlapjában megjeleníti az értékeit.

```
public function edit($id)
{
  $category = Category::findOrFail($id);
  return view('categories.edit', compact('category'));
}
```

7–16. kódrészlet: Kategória szerkesztéséhez szükséges űrlap lekérése az edit metódusban

A hasonlóságok miatt az **edit** nézet fájl tartalmát másolhatjuk, és beilleszthetjük ide a **create** nézet fájlból, itt pedig kiemelem a lényeges módosításokat, amelyeket végre kell hajtanunk benne.

```
<form action="/categories/{{ $category->id }}" method="post">
  @csrf
  @method('put')
  <p>
    <label for="nev">Kategória neve:</label>
    <input type="text" name="name" id="nev" value="{{ $category->name }}">
  </p>
  <input type="submit" value="Frissítés">
</form>
```

7–17. kódrészlet: categories.edit nézet fájl kezdeti tartalma

Négy módosításra van szükség, amelyeket a legegyszerűbben érthetőtől kezdve magyarázok el:

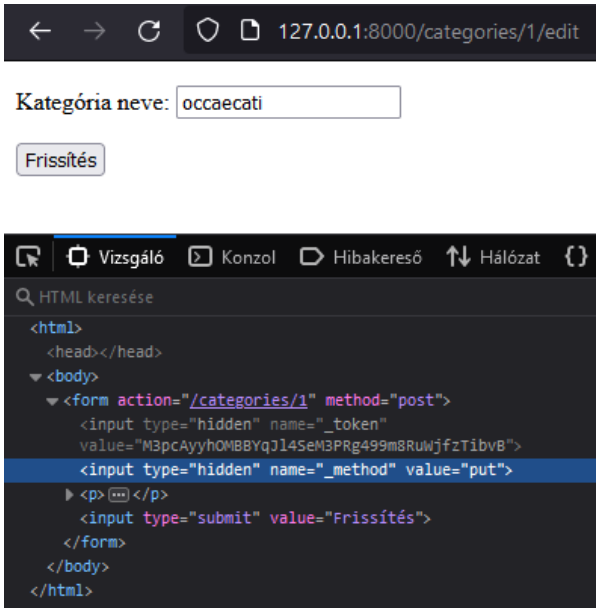
1. Írjuk át a gomb feliratát *„Mentés”-ről „Frissítés”-re,* így biztosan tudni fogjuk, hogy éppen melyik oldalon vagyunk, akkor is, ha a bemeneti mező(k) értéke üres.
2. A bemeneti mező(k) **value** attribútum értékeit állítsuk be a kapott objektum mezőjére, így az fog megjelenni az űrlapon, ami az adatbázis táblában is szerepel.

3. Az űrlap nyitó tag-jében az **action** attribútumot írjuk át a 7–2. táblázatnak megfelelő útvonalra és a **method** attribútum értékét hagyjuk meg **post**-nak, mivel az űrlapot csak GET vagy POST metódusokkal lehet elküldeni.
4. Ahhoz viszont, hogy a Laravel érzékelje, hogy ez nem egy POST-os küldés lesz, tehát nem egy beszúrást kell majd végrehajtania az adattáblában, ehhez szükségünk van egy **@method** direktívára, amelyben jelezzük, hogy ez igazából egy PUT HTTP metódus szerinti küldés, útvonal elérés történik.

Az edit útvonal eléréséhez bővítsük az index nézetünket és a <thead>-ben hozzunk létre egy új <th> elemet a sorban és adjuk azt meg a belsejében, hogy *„Funkciók”.* A <tbody>-ban is hozzunk létre egy új <td> elemet a sorban és adjuk azt meg a belsejében, hogy **id }}/edit">Szerkesztés**

Így tehát egy linkkel elérhető a szerkesztési oldal a kategóriák kilistázásának felületéről.

Vizsgáljuk meg a betöltés után, hogy az űrlapba bekerült így még egy rejtett mező, ami a put metódust rejti.



7–5. ábra: Edit nézet szerkesztési űrlapja a két rejtett mezővel

A Frissítés gombra most még hiába kattintanánk, mert még nem hoztuk létre az **update** útvonalat és vezérlő metódust. Folytassuk is ezzel!

7.2.6. Meglévő adatok frissítése (update, 1-n)

Ez az útvonal hasonlít a **show** útvonalhoz. Maga az útvonal ez lesz:

```
Route::put('/categories/{id}', [CategoryController::class, 'update']);
```

7–18. kódrészlet: Kategóriát adatbázisban frissítő útvonal létrehozása

Az **update()** metódus a **CategoryController**-ben hasonlít a **store()**-hoz, csak a paraméterül megkapott azonosító erőforrását frissítjük itt, nem pedig újat hozunk létre:

```php
public function update($id)
{
  $category = Category::findOrFail($id);
  $category->update([
    'name' => request('name')
  ]);

  return redirect('categories');
}
```

7–19. kódrészlet: Meglévő kategória frissítését elvégző update() metódus

Így most már működni fog a szerkesztés és frissítés, ha például a *„teszt 1.”* kategória számát átírjuk *„2”*-re.

Adatkapcsolat nélküli táblánál nincs teendőnk a frissítéssel, mivel nem érint más táblát. A **categories** tábla azonban adatkapcsolatban van a **posts** táblával, viszont, ha a kiválasztott kategória nevét frissítjük, az adatkapcsolat szempontjából nem lesz hatással a blogbejegyzésekre, mivel az **id** mezőn keresztül a kapcsolat továbbra is fenn fog állni, csak más lesz a kategória neve. Ellenben, ha töröljük a kategóriát…

7.2.7. Meglévő adatok törlése (destroy, 1-n)

Végül a törlés művelet maradt hátra, amelyet a frissítéshez hasonlóan fogunk végrehajtani, de külön ehhez már nem szükségeltetik nézet fájl és ahhoz vezető útvonal. A törlést a következő útvonalon keresztül fogjuk elérni (az útvonal URI része itt is ugyanaz lesz, mint a **show** és **update** útvonalak esetén, csak a HTTP metódusban különböznek):

```php
Route::delete('/categories/{id}', [CategoryController::class, 'destroy']);
```

7–20. kódrészlet: Kategóriát törlő útvonal létrehozása

Az útvonal elérését az **edit** nézetből valósítjuk meg, és ott hozunk létre egy új űrlapot, aminek elküldése hatására meghívásra kerül ez az útvonal, majd rajta keresztül a **destroy()** vezérlő metódus.

```php
<form action="/categories/{{ $category->id }}" method="post">
  @csrf
  @method('delete')
  <input type="submit" value="Törlés">
</form>
```

7–21. kódrészlet: Törlést indító űrlap és gomb az edit nézetben

Az útvonalhoz tartozó **destroy()** metódus pedig legyen ilyen:

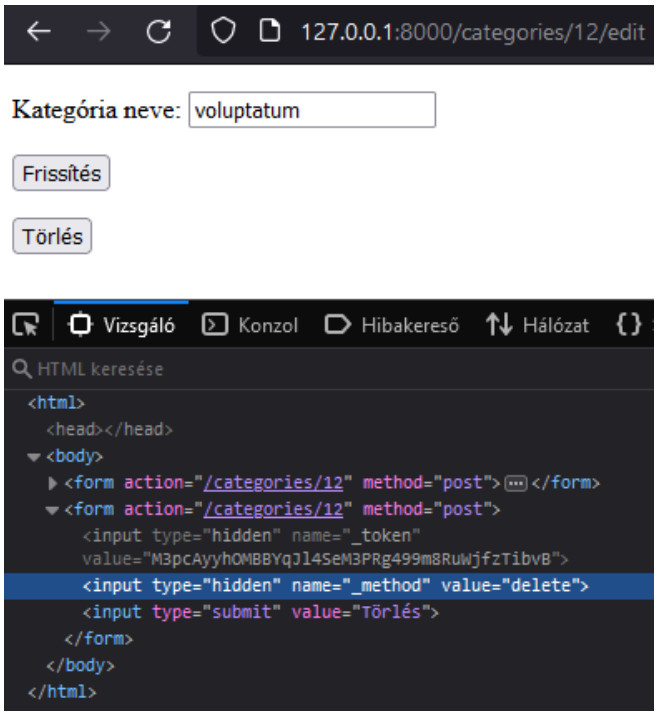
```php
public function destroy($id)
{
  $category = Category::findOrFail($id);
  $category->delete();

  return redirect('categories');
}
```

7–22. kódrészlet: Kategória törlését végző destroy() vezérlő metódus

7.2.7.1. Kategória törlésének hatása a kapcsolódó elemekre (tovább gyűrűzés)

Bár azt írtam, hogy a **categories** táblában nincsen idegen kulcs, emiatt a kezelése könnyű, ami valóban igaz is volt. Azonban erre a táblára hivatkozik idegen kulcs a kapcsolat másik táblájában, és ha olyan sort törlünk a **categories** táblából, amelyre hivatkozik sor a **posts** táblából, akkor a *„cascade"* idegen kulcs kényszer (lásd a 6–43. kódrészletet) miatt a **posts** táblából is törölni fogja az adott kategóriához tartozó blogbejegyzéseket (ahol a **category_id** megegyezik a törölt **category id** értékével). Ezt a következő 7.3. alfejezetben részletesebben megvizsgáljuk, most csak arra figyeljünk, hogy olyan kategóriát töröljünk csak, amihez nem tartozik blogbejegyzés. Nálam ilyen például a 12-es azonosítójú kategória.

A szerkesztési **edit** nézet betöltésekor megvizsgálva a második űrlapot, be is került rejtett metódust tartalmazó **delete** értékű mező, a törlés pedig működik a *„Törlés"* gomb megnyomásával. Arra vigyázzunk, hogy ez *nem „soft delete"-es* adattábla, így a **categories** táblából ténylegesen törlődni fognak azok az adatsorok, amelyeket így törlünk.



7–6. ábra: Edit nézet az új törlés űrlappal és annak rejtett mezőivel

Ha olyan kategóriát törlünk, amelyhez kapcsolódik blogbejegyzés (adatsor), akkor a **posts** táblából azok az adatsorok is törlődnének, amelyek az adott kategóriához tartoznak. Viszont mivel a **Post** Model osztálynál definiáltuk a *„soft delete"* tulajdonságot, így ténylegesen az adott törölt kategória blogbejegyzései nem kerülnek fizikailag törlésre a **posts** adattáblában, hanem csak a **deleted_at** mezőikbe fog bekerülni az aktuális időbélyeg (lásd a 6.2.2.5. alfejezetet ezzel kapcsolatban).

7.2.8. Összefoglalás és egyszerűsítés (kód újraszervezés)

Az alfejezet összefoglalásaként kijelenthetjük, hogy már ismerjük és tudjuk használni a 7 RESTful útvonalat és a rajtuk keresztül elérhető vezérlő metódusokat. A fejezet további részében főleg ezek finomítására, optimalizálására, egyszerűsítésére helyezem a hangsúlyt, amellett, hogy megvizsgáljuk,

hogy a kapcsolódó adattáblák működésére milyen hatással vannak az itt bemutatott műveletsorok. Végül egy egyszerűsítést nézzünk még meg az útvonalak regisztrációja kapcsán.

Listázzuk ki az útvonalakat:

php artisan route:list

```
POST      categories ..................................................... CategoryController@store
GET|HEAD  categories ..................................................... CategoryController@index
GET|HEAD  categories/create .............................................. CategoryController@create
PUT       categories/{id} ................................................ CategoryController@update
DELETE    categories/{id} ............................................... CategoryController@destroy
GET|HEAD  categories/{id} ................................................ CategoryController@show
GET|HEAD  categories/{id}/edit ........................................... CategoryController@edit
```

7–7. ábra: Kategóriákhoz tartozó 7 RESTful útvonal (bal oldalon) és vezérlő metódusok (jobb oldalon)

Mivel ezt a 7 útvonalat majdnem minden erőforráshoz létre kell hozni, ezért létezik hozzá egy egyszerűsítés, amellyel összesítetten, mindössze egyetlen útvonal regisztrációjával kiváltható mind a 7 útvonal létrehozása. Az eddig meglévő útvonalainkat kommentezzük ki és adjuk hozzá ezt az új, összesítő útvonalat:

```
// Route::get('/categories/create', [CategoryController::class, 'create']);
// Route::post('/categories', [CategoryController::class, 'store']);

// Route::get('/categories/{id}/edit', [CategoryController::class, 'edit']);
// Route::put('/categories/{id}', [CategoryController::class, 'update']);

// Route::delete('/categories/{id}', [CategoryController::class,
'destroy']);

// Route::get('/categories', [CategoryController::class, 'index']);
// Route::get('/categories/{id}', [CategoryController::class, 'show']);

Route::resource('categories', CategoryController::class);
```

7–23. kódrészlet: Új, összesítő útvonal regisztrációja a meglévő 7 különálló helyett

Ha mentés után lekérjük újra az útvonalainkat a php artisan route:list utasítással, akkor ugyanúgy (7–7. ábra) meg kell kapnunk a már az imént is bemutatott 7 RESTful útvonalat és a hozzájuk tartozó vezérlő metódusokat.

Az útvonalak ilyen összesítő módon történő regisztrációján túl a Controller metódusainak létrehozása is meg tud történni egyszerűsített formában, de ezt csak a következő alfejezetben fogjuk áttekinteni és kipróbálni.

A 7.2. alfejezetben végrehajtott programkód-módosítások ebben a GitHub commit-ben érhetők el.

7.3. Egy-több (1-n) adatkapcsolat kezelése a „több” (n) oldalon

A **Category-Post** kapcsolat kezelését folytatjuk az 1-n-es kapcsolat *„érdekesebb”* oldalával, ahol a külső kulcs mező és kapcsolat definiálása szerepel, a **posts** adattáblával és vele együttesen a **Post** Model

osztállyal. Az ismert, hogy a **Post** Model résztvevő egy másik, több-többes kapcsolatban is a **Tag** Model osztállyal, de azt a kapcsolattípust a következő, 7.4. alfejezetben vesszük végig.

7.3.1. Bevezetés és az útvonalak elnevezése

Az előző alfejezetben megismert útvonal összegzést a **post**-okat kezelő útvonalaknál már alkalmazhatjuk:

```
Route::resource('posts', PostController::class);
```

7–24. kódrészlet: 7 RESTful útvonal létrehozása

Importáljuk is be a **web.php** elején a **PostController**-t, de az még nem létezik, úgyhogy hozzuk létre és a következő egyszerűsítést a **PostController** létrehozó parancsánál tudjuk végrehajtani:

```
php artisan make:controller PostController -r
```

Az **-r** kapcsoló a *„resource”* szóra utal, amellyel a 7 RESTful útvonalhoz létrehozza a Controller-ben a 7 vezérlő metódust is a keretrendszer. Ha most lekérjük az útvonalakat (route:list), akkor vizsgáljuk meg a lista eredményét:

```
GET|HEAD        posts ..................... posts.index › PostController@index
POST            posts ..................... posts.store › PostController@store
GET|HEAD        posts/create .............. posts.create › PostController@create
GET|HEAD        posts/{post} .............. posts.show › PostController@show
PUT|PATCH       posts/{post} .............. posts.update › PostController@update
DELETE          posts/{post} .............. posts.destroy › PostController@destroy
GET|HEAD        posts/{post}/edit ......... posts.edit › PostController@edit
```

7–8. ábra: Útvonalak automatikus elnevezése

Sárgával kiemeltem az automatikusan létrehozott útvonalak *elnevezéseit*. Az útvonalakat a **name('útvonal_neve')** segédmetódussal tudjuk elnevezni, és hozzáfűzni a nevet az útvonalhoz. Mi magunk is el tudunk nevezni útvonalat, például a kezdőoldal nézetünket így:

```
Route::get('/', function () {
  return view('welcome');
})->name('homepage');
```

7–25. kódrészlet: Kezdőoldal elnevezése

A célja az útvonalak elnevezésének az, hogy magát az útvonalat az *alias* nevével használhatjuk a webalkalmazás bármely részén és ha később valamilyen oknál fogva úgy döntünk, hogy magát az URI részt megváltoztatjuk, akkor az alkalmazás még ugyanúgy működni fog, mivel az alias nevével hivatkoztunk magára az útvonalra. Az adott útvonal nevére a **route('útvonal_neve')** segédmetódussal tudunk hivatkozni, amelyet a további alfejezetekben gyakorlati példák egész sorával fogok bemutatni.

Az útvonalak elnevezése nélkül, ha meggondoljuk magunkat a korábbi terveinkhez képest, és más útvonalon szeretnénk elérni valamilyen erőforrást, akkor azt minden olyan helyen meg kell változtatni, ahol hivatkoztunk rá: például a weboldalunk menüstruktúrájában, az űrlapunk (**form**) **action** attribútumában, az oldalon elhelyezett egyéb linkeken stb. Ez, azontúl, hogy rettentő kényelmetlen (mindenhol egyesével megváltoztatni), meglehetősen nagy hibalehetőséget is rejt magában, amit pedig mi el szeretnénk kerülni. Hiszen bárhol megfeledkezhetünk róla, hogy ott, azon a helyen is meg kellett volna változtatni a *„beégetett”* URI-t és emiatt a weboldalunk hibás működést eredményezhetne. Az alias

nevek használatával csak egyetlen helyen kell megváltoztatnunk az URI-t és minden ugyanúgy működni fog továbbra is. Az így elnevezett útvonalak neveinek mindenképpen egyedinek kell lennie, tehát például nem regisztrálhatunk két **homepage** nevű útvonalat. Helyezzük is el a **homepage** útvonal linkjét a sablonunk bal felső sarkában lévő *„Future Imperfect"* logó linkjében. Ezt a **resources / views / includes / _header.blade.php** h1-es címsorában (2. sorban) találhatjuk meg. A **href** attribútum értékét kell megváltoztatni az **index.html**-ről erre: **{{ route('homepage') }}**

Megjegyzés: A metódus első kötelező paramétere az útvonal neve, a második és a harmadik paraméter opcionális. A második paraméter az útvonalhoz tartozó paraméter átadására szolgál (erre nézünk majd számos példát). A metódus harmadik paraméterével az adható meg, hogy abszolút vagy relatív legyen a legenerált útvonal. Alapértelmezetten a Laravel **route()** segédmetódusa abszolút útvonalakat generál (ennek az oka a keresőoptimalizálás és a gyorsítótárazás területein keresendő). Ha azt szeretnénk, hogy relatív útvonalat generáljon nekünk a **route()** metódus, akkor ezt külön meg kell határozni a számára egy paraméterrel:

route('homepage');

=> http://127.0.0.1:8000

route('homepage', ['id' => 1], false);

=> /1

route('homepage', absolute: false); // nincs második paraméter, ezért jelöljük a harmadik címkéjét

=> /

7.3.2. Több adatsor kiolvasása (index, n-1)

Az **index()** blogbejegyzéseket listázó metódusban nincsen paraméterünk, egyszerűen a lekért **post**-okat átadjuk és megjeleníthetjük őket. *Megjegyzés:* ha bizonytalanok lennénk a metódusok magjának összeállításában, akkor nyugodtan vegyük elő a **CategoryController** osztály ide vonatkozó részeit és ugyanez vonatkozik majd a nézetekre is, mert kiindulási alapként nyugodtan használhatjuk a **categories** nézet mappában lévő Blade fájlokat, amelyeket aztán tovább bővíthetünk vagy módosíthatunk.

A metódusban adjuk át a **posts.index** nézetnek a blogbejegyzéseink listáját publikálás dátuma szerint csökkenő sorrendben, tehát amit legutoljára publikáltunk, az kerüljön legfelülre.

```
public function index()
{
  return view('posts.index', [
    'posts' => Post::orderBy('published_at', 'desc')->get()
  ]);
}
```

7–26. kódrészlet: PostController osztály index() metódusa

A helyes működéshez importálnunk kell a **PostController** fájl elején a **Post** Model osztályt. A nézet és a mappája még nem létezik, úgyhogy hozzuk létre a **posts** mappát és benne az **index.blade.php**-t. A fájl

tartalmát átmásolhatom a **categories** mappa **index** nézetéből és kisebb módosításokkal már használható is:

```
<h1>Blogbejegyzések</h1>
<a href="{{ route('posts.create') }}">Létrehozás</a>
<table>
  <thead>
    <tr>
      <th>Cím</th>
      <th>Funkciók</th>
    </tr>
  </thead>
  <tbody>
  @foreach ($posts as $post)
    <tr>
      <td><a href="{{ route('posts.show', $post->id) }}">{{ $post->title
}}</a></td>
      <td><a href="{{ route('posts.edit', $post->id)
}}">Szerkesztés</a></td>
    </tr>
  @endforeach
  </tbody>
</table>
```

7–27. kódrészlet: posts.index nézet kezdeti tartalma

Vastagítva kiemeltem, hogy milyen elnevezésekben különbözik minimálisan ez az **index** fájl a **categories** mappában lévőtől. Így már működni is fog a nézetünk, ha betöltjük a **/posts** útvonalat a böngészőben. A *„Funkciók”* oszlopban lévő weblinkek majd a **show** és **edit** nézetekhez fognak elvezetni minket, míg a felső címsor alatti link a **create** nézethez irányíthat minket.

7.3.3. Egy adatsor kiolvasása (show, n-1)

Következik egy blogbejegyzés megjelenítése a részleteivel együtt. De előbb vizsgáljuk még meg a **posts** tábla az URI-kat tartalmazó oszlopát. Korábban az **id**-t használtuk az útvonalak wildcard részében, itt pedig (7–8. ábra), amikor már összesítetten hoztuk létre az útvonalakat, akkor a Model neve szerepel kis kezdőbetűvel (**post**). Ez annyiban segít minket, hogy a paraméteres útvonalak vezérlő metódusainál paraméterként használhatjuk jelen esetben a **Post $post**-ot, tehát a **Post** osztály **$post** nevű objektumát. Ezzel annyit tudunk *„megspórolni”*, hogy az ilyen metódusok magjának első utasításaiban nem kell a **findOrFail($id)**-lel kikeresni az adattábla megfelelő sorának mezőit.

Módosítsuk és bővítsük ennek megfelelően a show metódust a **PostController**-ben:

```
public function show(Post $post)
{
  return view('posts.show', compact('post'));
}
```

7–28. kódrészlet: PostController show() metódusa

Mivel ez a **show** nézetünk még nem létezik a **posts** mappában, hozzuk létre, vagy használjuk fel hozzá a **categories** mappában lévő **show** nézet fájlt, és másoljuk, majd illesszük be a **posts** mappába, és utána következhet egy kis átalakítás:

```
<h1>{{ $post->title }}</h1>
<h2>{{ $post->slug }}</h2>
<h3>Létrehozása: {{ $post->created_at }} | Módosítása: {{ $post->updated_at
}}</h3>
<p>{{ $post->body }}</p>
<a href="{{ route('posts.index') }}">Vissza a listára</a>
```

7–29. kódrészlet: posts.show nézet kezdeti tartalma

Egy dolgot fűzzünk még hozzá, felhasználva a **Post** és **Category** Model osztályok közötti kapcsolatot. Írjuk ki az aktuálisan kiválasztott blogbejegyzés kategóriáját, kattinthatóan úgy, hogy a kategória **show** nézete jöjjön be.

```
<h3>Kategória:</h3>
<a href="{{ route('categories.show', $post->category_id) }}">{{ $post-
>category->name }}</a>
```

7–30. kódrészlet: Blogbejegyzés kategóriája és a hozzá vezető link

Utána természetesen a Model osztályok közötti kapcsolatok *„megfordítása”* szerint a kategória **show** nézetében is kilistázhatjuk a kategóriához tartozó blogbejegyzéseket:

```
<p>Kategóriához tartozó blogbejegyzések:</p>
<ul>
  @forelse ($category->posts as $post)
    <li><a href="{{ route('posts.show', $post->id) }}">{{ $post->title
}}</a></li>
  @empty
    <li>Nem tartozik még a kategóriához blogbejegyzés.</li>
  @endforelse
</ul>
```

7–31. kódrészlet: Kategóriához tartozó blogbejegyzések címeinek kilistázása a categories.show nézetben

A **@forelse** Blade direktívával lehetőségünk van arra, hogy ne csak végig menjünk a gyűjtemény elemein, hanem ha esetleg üres a gyűjtemény, akkor belül az **@empty** ágában lekezelni ezt az eshetőséget is.

Érdekes lenne még megjeleníteni a másik kapcsolatot ebben a részben, ez pedig a **Post** és **Rating** osztályok közötti adatlekérésen alapulhat: adjuk meg a blogbejegyzés oldalán, hogy mekkora értékelést kapott.

Az értékelés megjelenítése a nézetben így történhet meg:

```
<h3>Értékelése: {{ $post->rating->score }}</h3>
```

7–32. kódrészlet: Értékelés megjelenítése a posts.show nézetben

Az **index()** és **show()** vezérlő metódusokat és kapcsolódó nézeteiket érintő változtatások ebben a GitHub commit-ben megtalálhatók.

Tipp: Ha nincs is szükségünk mind a 7 RESTful útvonalra, hanem például csak az index-et és a show-t akarjuk regisztrálni, akkor is érdemes lehet használni a „resource" útvonal összesítőt, mivel két útvonal regisztrálása helyett:

```
Route::get('/categories', [CategoryController::class, 'index'])
   ->name('categories.index');
Route::get('/categories/{id}', [CategoryController::class, 'show'])
   ->name('categories.show');
```

… elég lehet csak egyetlen sorban megadni őket így az **only()** segédmetódussal:

```
Route::resource('categories', CategoryController::class)
   ->only('index', 'show');
```

Ennek pedig az ellentéte is működik (amelyeket ne regisztráljuk a 7-ből), az **except()** segédmetódussal így:

```
Route::resource('categories', CategoryController::class)
   ->except('create', 'store', 'edit', 'update', 'destroy');
```

A fenti útvonal regisztrációkkal ugyanazt érjük el, ugyanazt jelentik, és az útvonal elnevezések (**categories.index**, **categories.show**) is ugyanúgy működnek mindegyiknél.

7.3.4. Új adatok feltöltése (create, n-1)

Maradunk a blogbejegyzéseket kezelő résznél, mivel az 1-n-es adatkapcsolatban a kategóriákkal, a blogbejegyzések állnak a *„többes", „n-es"* oldalon, így a létrehozási felületen (**create** nézet) is meg kell majd jeleníteni az egyszerűbb, általános bemeneti mezőkön túl, a kategória kiválasztó mezőt is.

Maga az útvonal, ami a **posts.create** nézethez vezet, már beregisztrálásra került, ahogyan a **PostController**-ben a **create** vezérlő metódus váza is létezik, viszont tartalma még nincsen, úgyhogy ezzel kezdjük a megvalósítást.

```
return view('posts.create', [
  'categories' => Category::orderBy('name')->get(),
]);
```

7–33. kódrészlet: Adatátadás a posts.create nézetnek

Ne felejtsük el a **Category** Model osztály importálását a fájl elején!

A **posts / create.blade.php** nézet létrehozásához felhasználhatjuk a **categories / create.blade.php** tartalmát, és kibővíthetjük először az egyszerűbb beviteli mezőkkel (**title** és **slug** mezők legyenek **text** típusú **input** beviteli mezők, a **body** pedig **textarea**, ha nem menne, akkor érdemes megtekinteni az alfejezet végén található GitHub commit tartalmát a 7.3.5. alfejezet végén). Majd az űrlapba szúrjuk még be a kapcsolatért felelős kategóriaválasztót, az értékelési pontszámot és a publikálás dátumát:

```
<div>
  <label for="category">Kategória:</label>
  <select name="category_id" id="category">
```

```
    @foreach ($categories as $category)
        <option value="{{ $category->id }}">{{ $category->name }}</option>
    @endforeach
  </select>
</div>
<div>
  <label for="score">Értékelés:</label>
  <input type="number" id="score" name="score" min="1.0" max="10.0"
step="0.1" value="1.0" />
</div>
<div>
  <label for="published_at">Publikálás:</label>
  <input type="datetime-local" name="published_at" id="published_at"
value="{{ now()->format('Y-m-d H:i') }}">
</div>
```

7–34. kódrészlet: A posts.create űrlap újdonság (bonyolultabb?) elemei

A **<select>** HTML tag a megkapott **$categories** gyűjteményt alapul véve, egy **@foreach** direktívával végig tud lépkedni az elemeken és feltöltjük így **<option>** tag-ekkel a lenyíló listát. A szám típusú bemeneti mezőnél meg tudjuk határozni a minimum, maximum és alapértelmezett értékeket, valamint, hogy mennyivel csökkenjen/növekedjen a beállított érték. A publikálás dátumánál az alapértelmezett értéket az aktuális napra és időre állítottuk be így.

Kiegészítés: ha úgy érzékeljük, hogy az adatbázisban nem megfelelőek az adatok, akkor lehetőségünk van akár újra generálni az adatszerkezeteinket (php artisan migrate:fresh) és újra feltölteni a táblákat példa adatokkal (php artisan db:seed), vagy egyben használjuk őket: php artisan migrate:fresh --seed

7.3.5. Új adatok elmentése (store, n-1)

Legelőször adjuk hozzá a **category_id** mezőt a **Post** Model osztály **$fillable** adattömbjéhez! A blogbejegyzés alap elemeinek hozzáadás után következhet az értékelés létrehozása és benne a **post_id**-n keresztül a kapcsolat definiálása. Végül irányítsuk vissza a felhasználót a blogbejegyzéseket listázó (**index**) oldalra, hogy meggyőződhessen róla, elkészült a legújabb blogbejegyzése. Mindez programkódokkal így néz ki a **store()** metódus magjában:

```
$post = Post::create([
  'title' => request('title'),
  'slug' => request('slug'),
  'body' => request('body'),
  'category_id' => request('category_id'),
  'published_at' => request('published_at'),
]);

Rating::create([
  'score' => request('score'),
  'post_id' => $post->id,
]);
```

```
return redirect(route('posts.index'));
```

7–35. kódrészlet: PostController store() metódusának magja

A **create()** és **store()** vezérlő metódusokat és a kapcsolódó **create** nézetet érintő változtatások ebben a GitHub commit-ben megtalálhatók.

7.3.6. Meglévő adatok szerkesztése (edit, n-1)

Maga az útvonal, ami a **posts.edit** nézethez vezet, már beregisztrálásra került, ahogyan a **PostController**-ben az **edit()** vezérlő metódus váza is létezik, viszont tartalma még nincsen, úgyhogy ezzel kezdjük a megvalósítást.

```
public function edit(Post $post)
{
  $categories = Category::orderBy('name')->get();
  return view('posts.edit', compact('post','categories'));
}
```

7–36. kódrészlet: PostController edit() vezérlő metódusa

A nézethez az adatátadás nem csak a **$post**-ra korlátozódik, hanem a kategóriákat (**$categories**) is átadjuk neki.

Az **edit()** vezérlő metódus visszaadja nekünk a **posts / edit.blade.php** nézet fájlt, ami még nem létezik, de az előállításához vegyük alapul az ugyanebben a mappában lévő **create.blade.php**-t, és utána majd azt alakítsuk át:

1. Az űrlap **action** attribútumának értékét.
2. Adjuk hozzá a **put** paraméterű **@method** direktívát.
3. Az egyszerűbb (a 7.2.5. alfejezetben megismert) input mezőknek állítsuk be az alapértelmezett értékét főleg a **value** attribútumokban, mivel egy meglévő blogbejegyzést szeretnénk szerkeszteni.
4. Az eltárolt kapcsolat (kategória) beállítását az **<option>** HTML tag-ben tudjuk meghatározni.
5. A *„Mentés”* feliratú gombot írjuk át *„Frissítés”*-re.

```
<form action="{{ route('posts.update', $post->id) }}" method="post">
  @csrf
  @method('put')
  <div>
    <label for="title">Blogbejegyzés címe:</label>
    <input type="text" name="title" id="title" value="{{ $post->title }}"/>
  </div>
  <div>
    <label for="slug">Útvonal elérése:</label>
    <input type="text" name="slug" id="slug" value="{{ $post->slug }}" />
  </div>
  <div>
    <label for="body">Tartalma:</label>
    <textarea name="body" id="body" cols="30" rows="10">{{ $post->body
}}</textarea>
```

```
  </div>
  <div>
    <label for="category">Kategória:</label>
    <select name="category_id" id="category">
    @foreach ($categories as $category)
      <option value="{{ $category->id }}" @if($post->category_id ==
$category->id) selected @endif>{{ $category->name }}</option>
    @endforeach
    </select>
  </div>
  <div>
    <label for="score">Értékelés:</label>
    <input type="number" id="score" name="score" min="1.0" max="10.0"
step="0.1" value="{{ $post->rating->score }}" />
  </div>
  <div>
    <label for="published_at">Publikálás:</label>
    <input type="datetime-local" name="published_at" id="published_at"
value="{{ $post->published_at }}">
  </div>
  <input type="submit" value="Frissítés">
</form>
```

7–37. kódrészlet: A posts.edit nézet teljes tartalma

Az **<option>** HTML tag-ben a **selected** attribútum az, ami meghatározza az alapértelmezett értéket, ezt pedig egy **@if** direktíva segítségével tudjuk megvizsgálni, hogy hova szúrjuk be ezt az attribútumot: oda, ahol a **$category->id** megegyezik a **$post->category_id** külső kulcs értékével (egy ilyen lesz csak, a kapcsolat típusából adódóan).

7.3.7. Kitérő: dátum- és időkezelés (formázás a Model segítségével)

Érdekesség: a HTML űrlap elemben a **datetime-local** típus alapértelmezetten az idő szegmensben csak órát és percet mutat, ezt láthattuk a **create** nézetben. Viszont az adatbázisban a **timestamp** típusú mező eltárolja a – felhasználó által nem beállított – másodpercet is (00 lesz). Ez a **create** űrlapon keresztül történő létrehozásnál mindig 00 lesz, viszont, ha a **PostFactory**-t használtuk a létrehozásra, akkor ettől eltérő másodpercek is eltárolásra kerülhettek. Majd, amikor az **edit** nézetben lévő űrlap **datetime-local** típusú bemeneti mezője megkapja az adatbázisban lévő nem csak 00 másodpercet tartalmazó értéket, már mutatni fogja a másodperceket is az órán és percen kívül.

A másodpercek módosítását nem fogja engedélyezni a kliens oldali validáció elküldéskor!

Az iménti *„Érdekesség”* részben leírt probléma megoldása lehet az, hogy a PHP DateTime típusú mezőit automatikusan átalakítjuk olyan formátumra, amelyet szeretnénk használni meghíváskor (**accessor**, további információk erről itt). Ezt a kapcsolódó Model osztály – jelen esetben a **Post.php**-ban – tudjuk megtenni. Egy védett metódushívással tehető ez meg, a metódus neve pedig úgynevezett *„camel case”*

formátumban kell, hogy definiálásra kerüljön: kis kezdőbetűvel, ha pedig több szóból áll, akkor minden újabb szó nagy kezdőbetűvel kezdődjön, az aláhúzásokat figyelmen kívül hagyva. Így lett az alábbi példában a **published_at** mező hívója **publishedAt**. Az attribútum elkészítésekor nekünk most csak a **get** (lekérést biztosító) hozzáférésre van szükségünk, de a **set** is ugyanígy implementálható lenne.

```
protected function publishedAt(): Attribute
{
  return Attribute::make(
      get: fn (string $value) => \Carbon\Carbon::parse($value)->format('Y-m-
d H:i'),
  );
}
```

7–38. kódrészlet: published_at érték lekérésekor alkalmazott formátum a dátumon kívül órát és percet tartalmaz

Az **Attribute** használatához ne felejtsük el importálni az osztályt (**Illuminate \ Database \ Eloquent \ Casts \ Attribute**).

A kipróbálást megtehetjük a Tinker-ben is, és megnézhetjük a kiválasztott blogbejegyzés **published_at** mező értékét a fenti *„accessor"* élesítése előtt és után is.

```
PS C:\xampp\htdocs\l10-components> php artisan tinker
Psy Shell v0.11.12 (PHP 8.1.2 — cli) by Justin Hileman
> App\Models\Post::first()->published_at;
= "2023-06-18 14:16:52"

>
PS C:\xampp\htdocs\l10-components> php artisan tinker
Psy Shell v0.11.12 (PHP 8.1.2 — cli) by Justin Hileman
> App\Models\Post::first()->published_at;
= "2023-06-18 14:16"
```

7–9. ábra: published_at mező hozzáférésének megváltoztatása előtt és után (másodpercek eltűntek)

Az **edit** nézetben lévő űrlap segítségével is tudjuk tesztelni, hogy jól működik-e, csak annyit kell tennünk, hogy olyan blogbejegyzést választunk szerkesztésre, amelynél a **published_at** mező értéke *„00"*-tól eltérő másodperc értéket tartalmaz.

7.3.8. Meglévő adatok frissítése (update, n-1)

A szerkesztési űrlap elküldése után az **update()** vezérlő metódus kapja meg az elküldött adatokat az útvonalán keresztül. Ezekután a blogbejegyzés és az értékelésének frissítése már nem is okozhat problémát.

```
public function update(Post $post)
{
  $post->update([
    'title' => request('title'),
    'slug' => request('slug'),
    'body' => request('body'),
    'category_id' => request('category_id'),
```

```
    'published_at' => request('published_at'),
  ]);

  Rating::where('post_id', $post->id)->update(['score' =>
request('score')]);

  return redirect(route('posts.index'));
}
```

7–39. kódrészlet: Blogbejegyzés részleteinek és kapcsolódó elemeinek frissítése

Az adatok frissítése sikeresen végbemegy, nincs hatással a kapcsolódó táblák (**categories**, **ratings**) adatkapcsolati mezőire (alapértelmezetten csak olyan kategória választható ki frissítéskor, ami létezik és itt még nem feltételezzük – majd csak a validálás fejezetben –, hogy *„meghackelhetjük"* a mező értékét).

Az **edit()** és **update()** vezérlő metódusokat és a kapcsolódó **edit** nézetet érintő változtatások ebben a GitHub commit-ben megtalálhatók.

7.3.9. Meglévő adatok és az adatkapcsolat törlése (destroy, n-1)

A 7.2.7. alfejezethez hasonlóan, itt is az **edit** nézetbe helyezzük el azt az űrlapot, amellyel a törlést tudjuk végrehajtani.

```
<form action="{{ route('posts.destroy', $post->id) }}" method="post">
  @csrf
  @method('delete')
  <input type="submit" value="Törlés">
</form>
```

7–40. kódrészlet: Blogbejegyzés törlését indító űrlap a posts.edit nézetben

Az űrlapban lévő útvonalat megcélzó vezérlő metódus pedig így néz ki a **PostController**-ben:

```
public function destroy(Post $post)
{
  $post->delete();

  return redirect(route('posts.index'));
}
```

7–41. kódrészlet: A PostController destroy() vezérlő metódusa

A törlés működik is, ha kipróbáljuk, eltűnik a listából a törölt blogbejegyzés.

7.3.9.1. Blogbejegyzés törlésének hatása a kapcsolódó elemekre

A **Post** Model-ünk fel van készítve a *„soft delete"* tulajdonság lekezelésére, tehát a törlés hatására nem fog ténylegesen törlődni az adatsor a **posts** táblában, csak a **deleted_at** mező kapja meg a törlés időbélyegét, emiatt pedig nem kerül majd ténylegesen kilistázásra a *„törölt"* blogbejegyzés (lásd a 6.2.2.5. alfejezetet ezzel kapcsolatban).

De mi lenne a kapcsolódó elemkkel, ha nem definiáltuk volna a „soft delete"-et a Post Model-hez?

- A blogbejegyzés törlése a kategóriájára nincsen hatással (1-n kapcsolat, n elemből egy kerül törlésre).
- A **post_tag** kapcsolótábla több-többes (n-n) kapcsolatokat tárol, de ezek törlését majd a 7.4.7. alfejezetben fogjuk áttekinteni.
- A **ratings-posts** táblák közötti adatkapcsolat miatt a **ratings** táblából is törlődni fog a blogbejegyzés értékelése (kapcsolódó adatsora), elvileg… de ez nem fog így működni!

Vizsgáljuk meg a legutóbbi pontot a Tinker segítségével. Indítás után válasszunk ki egy olyan Post-ot, amely rendelkezik értékeléssel és próbáljuk meg ténylegesen törölni azt:

```
App\Models\Post::find(5)->forceDelete();
```

Az utasítás kiadásának hatására a rendszer kivételt dob nekünk, ugyanis nem engedélyezi ennek az adatsornak a törlését a **posts** táblában, egészen addig, ameddig tartozik hozzá értékelés (sor) a **ratings** táblában. Ez a megszorítás a **ratings-posts** táblák közötti kapcsolat tulajdonsága miatt van így, amit a **create_ratings_table** migrációs fájlban definiáltunk mezőként és kapcsolatként (lásd 6.4.2.2. alfejezetben tárgyalt, alapértelmezetten használt *„restrict"* kapcsolat beállítási típust). Ha mégis úgy döntenénk, hogy a blogbejegyzés (adatsor) tényleges törlése hatására szeretnénk automatikusan törölni a hozzá tartozó értékelést is a **ratings** táblában, akkor változtassuk meg a kapcsolat típusát egy új migrációs fájl segítségével:

```
php artisan make:migration change_post_id_fk_in_ratings_table
```

```php
public function up(): void
{
  Schema::table('ratings', function (Blueprint $table) {
    $table->dropForeign('ratings_post_id_foreign');
    $table->foreign('post_id')->references('id')->on('posts')
->cascadeOnUpdate()->cascadeOnDelete();
  });
}

public function down(): void
{
  Schema::table('ratings', function (Blueprint $table) {
    $table->dropForeign('ratings_post_id_foreign');
    $table->foreign('post_id')->references('id')->on('posts')
->restrictOnUpdate()->restrictOnDelete();
  });
}
```

7–42. kódrészlet: A ratings tábla idegen kulcs kapcsolat típusának megváltoztatása

Mivel módosítani nem lehet a kapcsolat típusát, ezért először az **up()** és **down()** metódusokban is törölni fogjuk a kapcsolatot magát, majd újra definiáljuk őket. Itt már – a korábban bemutatotthoz képest – alternatív módon definiáltuk a kapcsolat típusát frissítés és törlés esetén, illetve a **down()** metódusban ténylegesen megadtuk, hogy a kapcsolat típusa *„restrict"*, ezáltal elkerülhetjük azt problémát, hogy ha valaki nem tudná, mi az alapértelmezett beállítás a kapcsolatra vonatkozóan.

Ezután már futtathatjuk a migrálást és meg fog változni a kapcsolat típusa **cascade**-re, vagyis tovább fog gyűrűzni a hivatkozott táblában (**posts**) való adatsor törlés a hivatkozó táblára (**ratings**) is, tehát ott is törlődni fognak az adatsorok (ha a Tinker-ben újra a **forceDelete()**-et hívjuk meg, ahogy azt tettük az alfejezet korábbi részében, akkor az 5-ös számú blogbejegyzés és az értékelése is törlődni fog).

A **destroy()** vezérlő metódust és a kapcsolódó **edit** nézetet érintő változtatások ebben a GitHub commit-ben megtalálhatók az itt hozzáadott migrációs fájllal együtt.

Feladat: mielőtt belevágnánk a következő alfejezetbe, ahol a **posts**, **tags** és a kapcsolótáblájuk (**post_tag**) kezelése lesz a fő téma, érdemes megvalósítani a korábbiak és ezen fejezet alapján a **tags** tábla CRUD műveleteinek lekezelését: a **Tag** útvonalaival, **TagController**-rel és a **views** mappában a **tags** mappán belüli nézetek (**index, show, create, edit**) segítségével.

Ellenőrzésként én is létrehozom ezeket a fájlokat és egy GitHub commit-ben itt elérhetővé teszem.

7.4. Több-többes (n-n) adatkapcsolat résztvevőinek kezelése

A több-többes adatkapcsolat kezelését a **posts** és **tags** táblák között a **post_tag** adattábla valósítja meg a benne lévő külső kulcsok segítségével. Ebben az alfejezetben áttekintjük azokat a CRUD műveleteket, amelyek ezt a kapcsolótáblát és az ő adatainak lekérdezését, manipulálását jelentik.

Gyakorlás szempontjából a blogbejegyzések műveleteit az Eloquent ORM, míg kezdetben (**index, show**) a **tag**-ek műveleteit a Query Builder segítségével fogjuk szemléltetni. A **create(), store(), edit(), update(), destroy()** vezérlő metódusoknál (és ahol létezik, ott a nézeteknél is) az Eloquent ORM-et alkalmazhatjuk, mivel az ilyen egyszerűbb szituációkban (ahol nincs szükség bonyolult lekérdezések megfogalmazására) sokkal kevesebb kódolással jár.

Tipp: Bonyolult lekérdezések építésekor hasznos lehet például ennek a csomagnak a használata: https://github.com/tpetry/laravel-mysql-explain

Nem csak a tényleges lekérdezést mutatja meg nekünk, hanem optimalizáció és teljesítmény javítás céljából láthatjuk a lekérdezés részleteit.

7.4.1. Több adatsor kiolvasása aggregáltan (index, n-n)

A listázó vezérlő metódusoknál és nézeteknél elegendő lehet egy összegzést adni arról, hogy az adott blogbejegyzéshez mennyi címke (tag) és az ellenkező oldalon az adott **tag**-hez mennyi blogbejegyzés (**post**) tartozik.

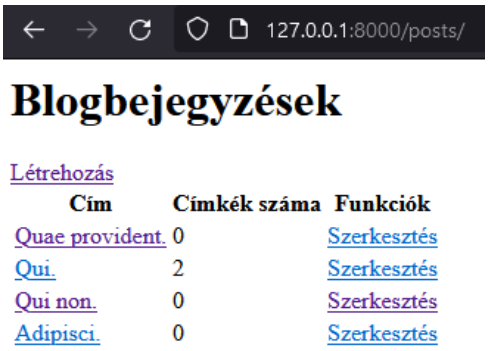
7.4.1.1. Post: index

A **PostController index()** vezérlő metódusában nincs is teendőnk ahhoz, hogy megszámoljuk a blogbejegyzéshez tartozó tag-ek számát. A hozzá tartozó **index** nézetben egy táblázatunk van, amelybe vegyünk fel egy új oszlop nevet (**<th>**) *„Címkék száma”* felirattal és a táblázat törzsében írjuk ki a **tag**-ek

számát. Az érintett sort emelem ki az alábbi kódrészletben (amelyet a blogbejegyzés címe és a funkciót tartalmazó szerkesztés link közé érdemes tenni):

```
<td>{{ $post->tags->count() }}</td>
```

7–43. kódrészlet: Blogbejegyzéshez tartozó tag-ek számának kiírása a posts.index nézetben



7–10. ábra: Blogbejegyzések kilistázása a címkék számával együtt (részlet)

7.4.1.2. Tag: index

Amikor a Query Builder-rel építünk lekérdezéseket, akkor az már olyan funkcionalitás (üzleti logika), hogy semmiképpen sem való a nézetbe elhelyezni. Sőt, igazából az is megfontolandó, hogy a Controller vezérlő metódusába helyezzük-e el, hiszen ezek olyan általános lekérdezések is lehetnek, amelyeket aztán újra és újra felhasználhatunk, emiatt pedig érdemes lehet az elhelyezése a kapcsolódó Model osztályban. Gondoljunk csak arra, hogy a **tag**-ek kilistázása a blog oldalunk több részén is megjelenhet különböző kinézetekkel, ekkor a címkékhez tartozó bejegyzések számosságának megjelenítésére is szükség lesz majd.

Szerkesszük a fentiek miatt a **Tag** Model osztályt és adjunk hozzá egy új metódust:

```
public function numberOfPosts() : int {
  return DB::table('posts')
    ->join('post_tag', 'post_tag.post_id', '=', 'posts.id')
    ->join('tags', 'post_tag.tag_id', '=', 'tags.id')
    ->where('tags.id', $this->id)
    ->count();
}
```

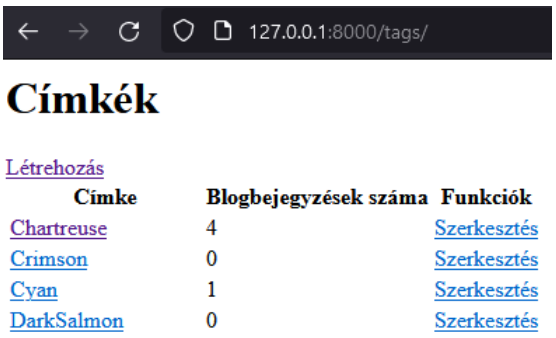
7–44. kódrészlet: Blogbejegyzések számának lekérése a Tag-hez Query Builder-rel

A kód helyes működéséhez a **Tag** Model osztály tetején importáljuk az **Illuminate\Support\Facades\DB** osztályt. A metódus egy egész számot ad visszatérési értékként a dupla természetes összekapcsolás (**join()**) és szűrés (**where()**) után. Látható, hogy ami az Eloquent ORM-ben egy sima tulajdonságon keresztüli metódushívás volt (ezért azt nem is érdemes kiszervezni az üzleti logikába), a Query Builder-nél egy kapcsolótáblán keresztüli összekapcsolás volt egy kicsit hosszabban, de talán mégis átláthatóbban egy SQL nyelvhez szokott fejlesztő számára.

Ezután már hasonlóan, mint a **posts.index** nézetben tettük, a táblázatot bővítsük a **tags.index** nézetben is (a fejlécben az oszlopnév megadásával, alul a táblázat törzsében pedig az alábbi kódsorral):

```
<td>{{ $tag->numberOfPosts() }}</td>
```

7–45. kódrészlet: Tag-hez tartozó blogbejegyzések számának kiírása a tags.index nézetben



7–11. ábra: Címkék kilistázása a blogbejegyzések számával együtt (részlet)

Az alfejezethez tartozó programkód változásokat ebben a GitHub commit-ben lehet megtalálni.

7.4.2. Egy adatsor kiolvasása (show, n-n)

Az adott blogbejegyzés és címke részletezésénél is ki tudjuk listázni (egy felsorolással), hogy milyen elemek tartoznak hozzájuk a másik fél részéről. Ezt fogjuk megtenni a már az index részben látható megoldással (**Post** – Eloquent ORM, **Tag** – Query Builder).

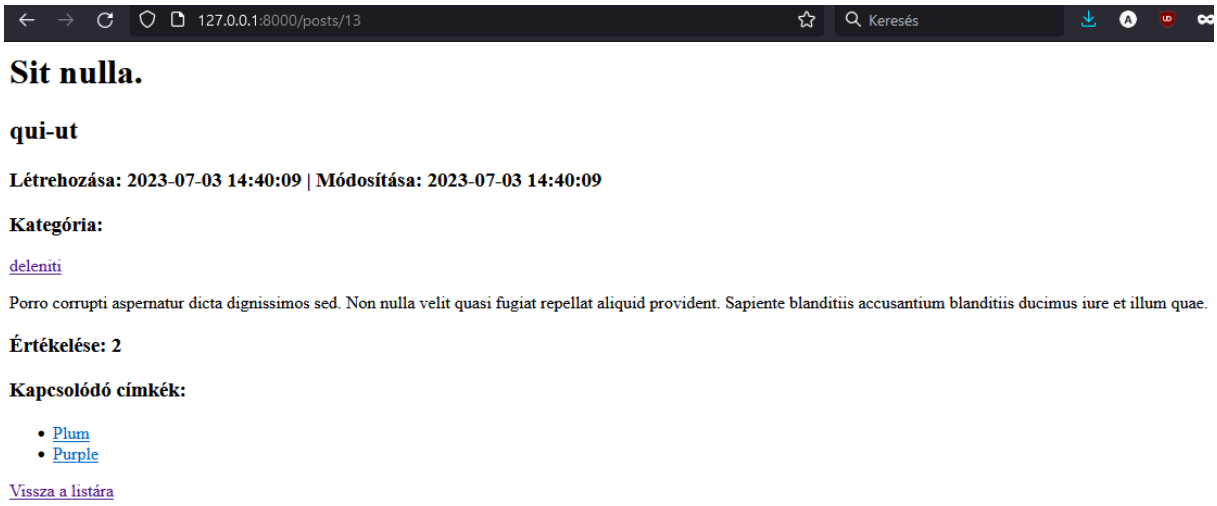
7.4.2.1. Post: show

Az Eloquent kapcsolatot használhatjuk a nézet fájlban is, így a **PostController show()** metódusát nem kell átalakítanunk, enélkül is tud működni a posts.show nézetben a következő kódrészlet:

```
<h3>Kapcsolódó címkék:</h3>
<ul>
  @forelse ($post->tags as $tag)
  <li><a href="{{ route('tags.show', $tag->id) }}">{{ $tag->name }}</a></li>
  @empty
  <li>Nincs hozzárendelve címke a blogbejegyzéshez</li>
  @endforelse
</ul>
```

7–46. kódrészlet: Blogbejegyzéshez kapcsolódó címkék kilistázása a posts.show nézetben

Így egy felsorolásban láthatjuk a blogbejegyzéshez tartozó címkék nevét weblinkkel ellátva, vagy azt, hogy a bejegyzéshez nem tartozik címke.



7–12. ábra: A posts.show nézet megjelenítése

7.4.2.2. Tag: show

A Query Builder segítségével épített lekérdezést szintén helyezzük el a **Tag** Model osztályában egy metódusban, amit utána bárhol (másik Model-ben, Controller-ben és még a nézetben is) meghívhatunk.

```
public function listOfPosts() : Collection {
  return DB::table('posts')
    ->select('posts.id', 'posts.title')
    ->join('post_tag', 'post_tag.post_id', '=', 'posts.id')
    ->join('tags', 'post_tag.tag_id', '=', 'tags.id')
    ->where('tags.id', $this->id)
    ->get();
}
```

7–47. kódrészlet: Címkéhez tartozó blogbejegyzések listájának lekérése

Ahhoz, hogy ez a metódus működjön, a visszatérési típusát (**Collection**) az osztály előtt importálni kell:

use Illuminate\Support\Collection;

Utána rátérhetünk a **tags.show** nézetre, ahol meghívjuk a $tag objektum **listOfPosts()** metódusát, és felsoroljuk azokat a blogbejegyzéseket, amelyek a címkéhez tartoznak. Ha nem tartozik a címkéhez blogbejegyzés, azt is jelezzük a látogatónak.

```
<h3>Kapcsolódó blogbejegyzések:</h3>
<ul>
  @forelse ($tag->listOfPosts() as $post)
  <li><a href="{{ route('posts.show', $post->id) }}">{{ $post->title
}}</a></li>
  @empty
  <li>Nincs hozzárendelve blogbejegyzés a címkéhez.</li>
  @endforelse
</ul>
```

7–48. kódrészlet: Címkéhez kapcsolódó blogbejegyzések kilistázása a tags.show nézetben

7–13. ábra: A tags.show nézet megjelenítése

Az alfejezethez tartozó programkód változásokat ebben a GitHub commit-ben lehet megtalálni.

7.4.3. Új adatok feltöltése (create, n-n)

A létrehozási űrlapokat (**Post, Tag**) a több-többes kapcsolatok hozzáadási részével kell bővítenünk. Mind a két Controller-ben bővítenünk kell a nézeteknek való adatátadást a címkékkel, illetve blogbejegyzésekkel (Ahol esetleg hiányozna, ott a Controller-ben importáljuk is az ellentétes osztályt.). Mindkét űrlapon a *„Mentés”* gomb előtt hozzuk létre ezt a szekciót.

7.4.3.1. Post: create

A **posts.create** nézetnek való adatátadás bővítése a **PostController create()** metódusában így bővült a címkékkel:

```
return view('posts.create', [
  'categories' => Category::orderBy('name')->get(),
  'tags' => Tag::orderBy('name')->get(),
]);
```

7–49. kódrészlet: Címkékkel bővített adatátadás a blogbejegyzéseket létrehozó űrlapnak

A **posts.create** nézet bővítése a címke-kapcsolat létrehozásához:

```
<label for="tags">Címkéi:</label>
<select name="tags[]" id="tags" multiple>
  @forelse ($tags as $tag)
    <option value="{{ $tag->id }}">{{ $tag->name }}</option>
  @empty
    <option value="0">Nincs még címke a blogon</option>
  @endforelse
</select>
```

7–50. kódrészlet: Blogbejegyzés létrehozó űrlap bővítése a címkés kapcsolat hozzáadásához

A többes kijelölés kulcsa itt a **<select>**-ben megtalálható **multiple** kulcsszó, ez engedi, hogy több címkét is ki tudjunk majd jelölni Ctrl vagy Shift gombok lenyomása mellett. A lehetséges több adat átadása miatt a <select> **name** attribútumában egy **tags[]** tömböt adunk majd át a **store()** metódusnak. Amíg a **store()** metódus magjában nem kezeljük le és mentjük el az újonnan érkező **tags[]** tömb értékeit, addig itt a

Mentés gombbal való tesztelésnek még nincs sok értelme, elég, ha azt kipróbáljuk, hogy tudunk-e több címkét kijelölni a listában.

7.4.3.2. Tag: create

Gyakorlásként a címkéket létrehozó űrlapot is bővítsük az alapján, amit a blogbejegyzést létrehozó űrlapon megcsináltunk. Ha esetleg nem sikerülne, akkor érdemes megnézni ezt a GitHub commit-et, amelyben elhelyeztem a megoldást.

7.4.4. Új adatok elmentése (store, n-n)

Mindkét esetben a **store()** metódusok kerülnek átalakításra a Controller-ekben. Az új elem létrehozási űrlapjáról most már a **tags[]**, illetve a **posts[]** tömbök is megérkeznek. Az új kapcsolatok definiálása mindkét esetben mehet az **attach()** metódus használatával.

7.4.4.1. Post: store

A **PostController store()** metódusában, miután létrehoztuk az új blogbejegyzést (és magát az objektumot el is tároltuk a **$post** változóban), következhet a kapott címkék alapján a **Post-Tag** kapcsolatok létrehozása:

```
$post->tags()->attach(request('tags'));
```

7–51. kódrészlet: Új blogbejegyzéshez a címke-kapcsolatok hozzáadása

A jobb átláthatóság miatt szedtem így szét, mert ha már rutinosabbak vagyunk és a **Post::create()** metódushívás eredményét nem használnánk fel, akkor a **->tags()->attach(request('tags'))** metódusokat hozzá is fűzhetnénk a **Post::create()** metódus végére, és akkor rögtön az új blogbejegyzés létrehozásakor hozzáadódnának a kapcsolatai is. *Figyelem!* Mi most felhasználjuk a **Post::create()** visszatérési eredményét, hiszen a **Rating** létrehozásához erre szükségünk van, úgyhogy ne tegyük meg ezt az összevonást itt, csak majd a **TagController**-ben esetleg.

7.4.4.2. Tag: store

A TagController store() metódusában már megtehetjük az összevonást az alábbiak szerint:

```
Tag::create([
  'name' => request('name')
])->posts()->attach(request('posts'));
```

7–52. kódrészlet: Új címkéhez a blogbejegyzés-kapcsolatok hozzáadása

Az alfejezet programkód változásait ebben a GitHub commit-ben lehet megtalálni.

7.4.5. Meglévő adatok szerkesztése (edit, n-n)

A Controller fájlok **edit()** metódusaiban (a **create()**-hez hasonlóan) szintén át kell adni a címkéket, illetve a blogbejegyzéseket a nézeteknek, hogy meg tudják jeleníteni majd őket. A **create** nézetekből átmásolásra kerülhet a kapcsolatok meglétét jelző **<select>** és utána már csak azt kell ellenőriznünk, hogy melyik kapcsolatok léteztek, amelyeket automatikusan be kell jelölnünk a **<select> <option>** listájában.

7.4.5.1. Post: edit

A **PostController edit()** metódusában a kategóriákon kívül adjuk át a nézetnek a címkéket is:

```
$tags = Tag::orderBy('name')->get();
return view('posts.edit', compact('post','categories', 'tags'));
```

7–53. kódrészlet: Címkelista átadása a blogbejegyzést szerkesztő nézetnek

A **posts.edit** nézetbe a **posts.create** nézetből átmásolható a címkéket felsoroló <select>, az <option> tag-en belül pedig egy hasonló vizsgálatot kell elvégezni, amelyet a felette látható kategóriákat tartalmazó **<option>**-nél már végrehajtottunk. Itt is a **selected** attribútum lesz az, amit el kell helyeznünk a megfelelő **<option>** tag-en belül. Viszont most a már ott használt **@if** Blade direktívát cseréljük le és helyette a **@selected** direktívát használjuk: ez megvizsgálja, hogy a paraméterében kapott feltétel igaz-e, és ha igen, akkor automatikusan elhelyezi majd a HTML kódban a **selected** attribútumot.

```
<option value="{{ $tag->id }}"
@selected($post->tags->pluck('id')->contains($tag->id)) >{{ $tag->name
}}</option>
```

7–54. kódrészlet: Címkéket felsoroló opciók selected attribútum hozzáadása programozottan

Ezt egy ábrával is szeretném szemléltetni, ahol kiemeltem az elmondott részeket:

7–14. ábra: Címkéket felsoroló opció selected attribútuma a kategóriás selected-hez képest

Már maga a **@selected** utáni feltételvizsgálat sem feltétlenül könnyen érthető, de vegyük végig: az adott **$post tag**-jei közül a **pluck()** metódussal kinyerjük az összes id attribútumot, amit visszaad egy tömbbe, ezután pedig a **contains()** metódussal megvizsgálja, hogy az adott **$tag id** szerepel-e benne, ha igen, akkor oda fogja tenni a **selected** attribútumot, ha nem, akkor nem.

7.4.5.2. Tag: edit

Gyakorlásként a címkéket módosító űrlapot is bővítsük az alapján, amit a blogbejegyzést módosító űrlapon megcsináltunk. Ha esetleg nem sikerülne, akkor érdemes megnézni ezt a GitHub commit-et, amelyben elhelyeztem a megoldást. *Megjegyzés:* a **tags.edit** nézetben egy elírást kijavítottam egy újabb GitHub commit-ben, így már megfelelő a listában a kiválasztás.

Tesztelés szempontjából az edit nézetekben lévő űrlapoknál csak azt tudjuk ellenőrizni, hogy tényleg azok a kapcsolatok vannak-e kijelölve a **<select>** listájában, amelyeket az adatbázis is tartalmaz.

7.4.6. Meglévő adatok frissítése (update, n-n)

A Controller fájlok **update()** metódusaiban érdemes újra áttekintenünk, hogy milyen lehetőségeink vannak a kapcsolatok kezelésére: az **attach()** és **detach()** páros itt nehezen lenne használható, viszont lehetőségünk van szinkronizálni a kapcsolatok módosításait (lásd a 6.4.3.7. alfejezetet).

7.4.6.1. Post: update

Az update() metódust mindössze egyetlen utasítással kell bővíteni:

```
$post->tags()->sync(request('tags'));
```

7–55. kódrészlet: Blogbejegyzés címkéinek szinkronizálása a frissítéskor

7.4.6.2. Tag: update

A címkék oldalán is csak ugyanerre van szükség és már működni is fog a frissítés.

Az alfejezetben végrehajtott programkód módosítások ebben a GitHub commit-ben elérhetők.

7.4.7. Meglévő adatok és az adatkapcsolat törlése (destroy, n-n)

A blogbejegyzéseknél aktiváltuk a *„soft delete"* funkcionalitást, így azok törlése nem lesz hatással a kapcsolat törlésére, hiába tűnik el a blogbejegyzések listájából maga a *„törölt"* post, a címkéknél továbbra is úgy fogjuk látni, mintha az a kapcsolat még valóban meg volna (hiszen meg is van, csak a túloldalon a blogbejegyzés lett törlésre kijelölve).

Itt egy útelágazáshoz érkeztünk, ahol el kell döntenünk, hogy mit szeretnénk. A lehetőségeink:

1. A **PostController destroy()** metódusában a $post-nak nem a **delete()**, hanem a **forceDelete()** metódusát hívjuk meg, ekkor ténylegesen törlődni fognak a kapcsolatai is, mivel a **post_tag** táblában lévő **post_id** mezőhöz a kapcsolat **onDelete('cascade')** tulajdonsággal lett definiálva.
2. Ha meg akarjuk hagyni a Post *„soft delete"* tulajdonságát, de a kapcsolatait törölni akarjuk (hogy ne látszódjon a címkéknél hamis blogbejegyzés szám), akkor a **destroy()** metódust kell bővítenünk. Később, ha úgy döntünk, hogy a **restore()** visszaállítási metódust is implementáljuk, akkor sem lenne már esély a volt kapcsolatokat helyreállítani, hiszen a **post_tag** tábla nem rendelkezik jelenleg a *„soft delete"*-hez tartozó **deleted_at** mezővel és ezzel a tulajdonsággal.
3. Ha meg akarjuk hagyni a Post *„soft delete"* tulajdonságát és az ő kapcsolataira is ki akarjuk terjeszteni ezt (cascade), akkor több lehetőségünk is van:
    a. A Laravel 8-as verziója előtt – és kényelmi szempontból a legfrissebb verziókban is – többnyire csomagokat használtak/használnak erre:
        i. https://github.com/michaeldyrynda/laravel-cascade-soft-deletes
        ii. https://github.com/shiftonelabs/laravel-cascade-deletes
        iii. … de ne feledjük, ami a kényelmünket szolgálja a programozásban, azt általában másképp fizetjük meg (például lassabb lesz tőle a program, vagy nem annyira hatékony a működésben stb.).

b. Azonban, ha nem szeretnénk csomagokkal *„túlterhelni"* a Laravel projektünket, akkor meg lehet oldani ezt eseményekkel (lásd majd a 12.4. alfejezetet) és úgynevezett *„megfigyelőkkel"* (lásd majd a 12.4.3. alfejezetet) is.

Én a második pont mellett döntöttem és aszerint a következő utasítással kell bővíteni a **PostController destroy()** metódusát:

```
$post->tags()->sync([]);
```

7–56. kódrészlet: Blogbejegyzés kapcsolatainak törlése (üresre állítása a sync() metódussal)

Az utasítás hatására a **post_tag** tábla sorai úgy szinkronizálódnak, hogy a *„soft delete"*-re kijelölt blogbejegyzés kapcsolatai törlésre kerülnek (*„üres kapcsolat"* kerül szinkronizálásra). Ezzel megegyező utasítás lenne a következő is:

```
$post->tags()->detach($post->tags);
```

7–57. kódrészlet: Blogbejegyzés kapcsolatainak törlése (detach() metódussal)

Az iménti kettő közül azt érdemes választani, amelyik számunkra érthetőbb, logikusabb.

A kapcsolatok törlése így együttesen fog működni a blogbejegyzés vagy a címke törlésével, mivel a **post_tag** kapcsolótábla migrációjánál a **tag_id**-s kapcsolatot is *„cascade"* paraméterrel láttuk el az **onDelete()** metódusban, tehát a címke törlése is tovább gyűrűzik a kapcsolataira, bármiféle további fejlesztés nélkül.

Az alfejezethez tartozó minimális kódváltoztatások ebben a GitHub commit-ben találhatók meg.

7.5. Erőforrás REST API tesztelése Postman-nel

Erőforrások lekérdezéseit, hozzáféréseit (létrehozás, frissítés, törlés) a Laravel API útvonalakon keresztül vezérlők metódusaival is végre lehet hajtani, ki lehet szolgálni. Ekkor természetesen nem lesz szükség a Controller-ekben **create** és az **edit** útvonalakra, hiszen ezek csak az űrlapokat jelenítették meg egy-egy nézeten keresztül a felhasználónak.

7.5.1. Előkészítés: útvonalak és a Controller

A tesztelés végrehajtásához válasszunk ki egy erőforrást, például a kategóriákat és először hozzuk létre nekik a **Api/V1/CategoryController** fájlt (de előtte az **app / Http / Controllers** mappán belül az **Api** mappát, majd azon belül a **V1** mappát):

```
php artisan make:controller Api/V1/CategoryController --api --model=Category
```

A vezérlő metódusait majd ezután fogjuk csak implementálni, most az útvonalakat regisztráljuk a **routes / api.php** fájlban.

```
Route::prefix('v1')->group(function () {
  Route::apiResource('categories', CategoryController::class);
});
```

7–58. kódrészlet: REST API útvonalak létrehozása a kategóriákhoz

Ne felejtsük el importálni az **Api / V1 / CategoryController** osztályt az **api.php** fájl elején.

Ezután meg is jelennek már az útvonalaink között a frissek, de érdemes egy szűrést is hozzáadnunk a lekéréshez, hogy rögtön az API-os útvonalakat kapjuk meg:

php artisan route:list --path=api/

```
PS C:\xampp\htdocs\l10-components> php artisan route:list --path=api/

  GET|HEAD        api/user ..... 
  GET|HEAD        api/v1/categories ..... categories.index › Api\V1\CategoryController@index
  POST            api/v1/categories ..... categories.store › Api\V1\CategoryController@store
  GET|HEAD        api/v1/categories/{category} ..... categories.show › Api\V1\CategoryController@show
  PUT|PATCH       api/v1/categories/{category} ..... categories.update › Api\V1\CategoryController@update
  DELETE          api/v1/categories/{category} ..... categories.destroy › Api\V1\CategoryController@destroy

                                                                                  Showing [6] routes
```

7–15. ábra: A projektünk API-os útvonalai

A 7–2. táblázat eszerint alakul út erre:

HTTP metódus	Erőforrás azonosító (URI)	Vezérlő metódus (Controller action)	Működés leírása *Példa*
GET	**/api/v1/tags**	*Index*	Lekérjük az összes tag-et és megjelenítjük őket
GET	**/api/v1/tags/{id}**	*Show*	Lekérünk egy konkrét tag-et (id alapján) és megjelenítjük
POST	**/api/v1/tags**	*Store*	Eltárolunk egy tag-et a tags adattáblában (INSERT)
PUT	**/api/v1/tags/{id}**	*Update*	Frissítünk egy konkrét tag-et (id alapján) (UPDATE)
DELETE	**/api/v1/tags/{id}**	*Destroy*	Törlünk egy konkrét tag-et (id alapján) (DELETE)

7–3. táblázat: REST API CRUD útvonalak és (API) vezérlő metódusok működése példával

HTML űrlap metódus nem tartozik már ezekhez, hiszen nincs is olyan útvonalunk és metódusunk, amely egy űrlaphoz vezetne.

7.5.2. API verziókezelés

Az alkalmazásfejlesztés során az API felület is fejlesztésre kerülhet. A verziókezeléssel ezt a fejlesztést lehet lekövetni, hiszen a változás az API fejlődésének is természetes része. Néha a fejlesztőknek frissíteniük kell az API kódját, hogy kijavítsanak benne bizonyos működéseket, befoltozzanak biztonsági réseket, vagy éppen új funkcionalitásokat helyezzenek el az API felületen. Bizonyos változások nem érintik az API felületünk felhasználóit, de lesznek mindig olyanok is, amelyek számukra is érzékeny változtatásokat (*„breaking changes”*) tartalmaznak és esetleg nem is fogunk (vagy akarunk) figyelni a visszafelé való kompatibilitásra. Ez szintén problémákhoz (váratlanul hibás működés, adatvesztés stb.) vezethet a továbbiakban. Az API verziószámozásával lehetőségünk van arra, hogy biztosítsuk a változásainak nyomon követését, így megőrizhetjük az API-unk felhasználóinak bizalmát, miközben

biztonságosan és hibamentesen tudunk nagy teljesítményű lekérdezéseket végrehajtani. Hiszen általában ez a cél egy API működtetése során.

Új verziót kell létrehoznunk minden olyan változtatás esetén, amely miatt az API felhasználóinak módosítaniuk kell a saját alkalmazásuk kódbázisán. Leggyakoribb példák az ilyen változtatásokra:

- Egy attribútum vagy API végpont (útvonal) átnevezésekor;
- Egy korábban opcionális paraméter kötelezővé válásakor;
- Adat formátum vagy típus megváltozásakor (például: egy szöveges típus JSON formátumú lesz);
- Egy attribútum tulajdonságainak megváltozásakor (például: új **maxLength** érték egy mezőnél).

A Laravel-ben az útvonal verziója az **api/** után és az erőforrás neve elé kerül be többnyire. Ez persze nincs kőbe vésve, de általában így épülnek fel az erőforrások CRUD műveleteihez létrehozott API végpontok. Ezért is hoztuk mi létre *„V1"*-es (vagyis 1. verziós) útvonalakat és Controller-t a neki megfelelő mappában (**app / Http / Controllers / Api / V1**).

7–16. ábra: Postman által javasolt verzió léptetési stratégia (Forrás: [6])

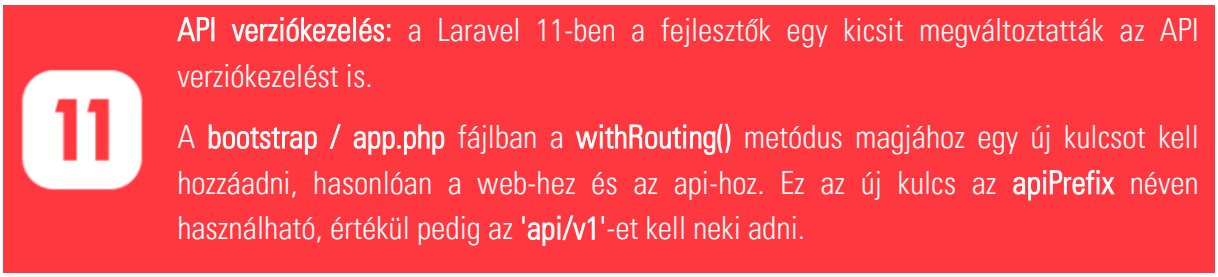
A verziókezelési stratégiát minél hamarabb érdemes kiválasztani, így a jövőben rugalmas tervezési mintákat követve tudjuk elkerülni a *„breaking change"* jellegű változások előfordulását. Ezáltal egy reális ütemterv is kialakítható ahhoz, hogy előre meghatározzuk az API fejlesztésének irányvonalait.

Ezután végig kell gondolnunk, hogy egy-egy változás igényli-e a teljes API új fő verziójának változtatását, esetleg elegendő egy új funkcionalitás vagy hibajavítás esetén az alverziószámot növelni.

Az nagyon fontos, hogy a változásoknak megfelelően mindig frissítsük az API elérésének és jellemzőinek leírását a dokumentációban. A változáskövetési dokumentációban érdemes megindokolni a változás szükségességét és azt is, hogy milyen hatással lesz ez az API felületünk felhasználóira. Emellett az új verzió elérésének paramétereit is meg kell határozni benne. Ha új funkciót készítettünk egy régi helyett, akkor azt is ki kell fejteni, hogy hogyan hat ez a felhasználók API hívásaira, mit kell másképp megadniuk, vagy másképp kezelniük a saját kódbázisukban.

Az új API verziót először *„házon belül”*, saját fejlesztőinkkel, saját tesztelőinkkel, esetleg külső tesztelő munkatársakkal kell tesztelni, mielőtt elérhetővé tennénk a felhasználóink számára. Így talán minimális esély lehet arra, hogy hibás működésű API felületet nyitunk az értékes ügyfeleinknek.

Miután elérhetővé tettük a hibamentesen és stabilan működő új verziót, a régi verziót ki kell vonnunk a működésből a későbbi félreértések elkerülése végett. Azonban a régi verzió elérését sem egyik pillanatról a másikra kell megszüntetni, hanem egy ütemterv szerint érdemes végrehajtani a kivonást, közben pedig folyamatos támogatást nyújtani a meglévő régi ügyfeleinknek, felhasználóinknak a zökkenőmentes átálláshoz.



7–1. újdonság: API verziókezelés

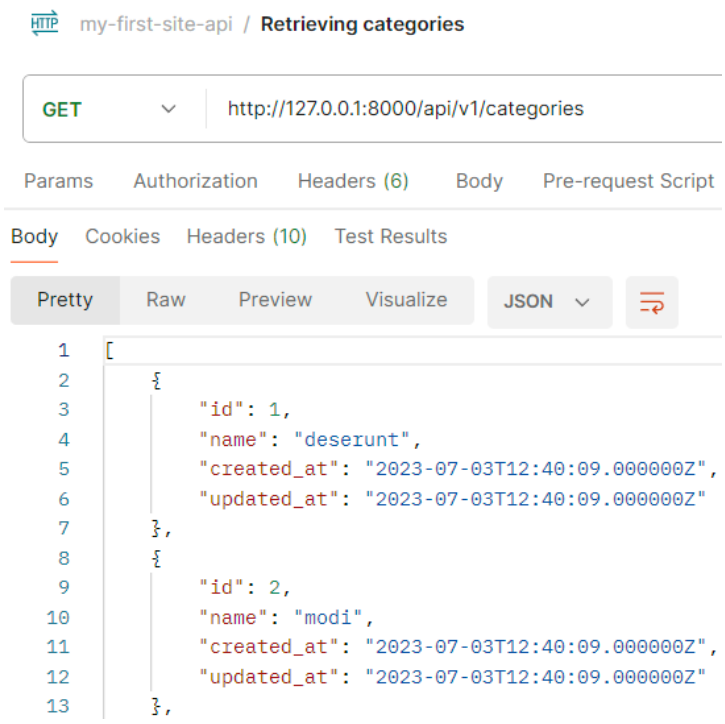
7.5.3. Erőforrások lekérdezése (index, show)

Megtehetnénk azt, hogy az 5 REST API útvonalhívásokra Eloquent Model osztályok példányait adjuk vissza, például így:

```
public function index()
{
  return Category::all();
}

public function show(Category $category)
{
  return $category;
}
```

7–59. kódrészlet: Category Model példányok visszaadása az API kérésekre

Ekkor az API hívás eredménye így néz ki például az index útvonal elérésekor a Postman-ben:



7–17. ábra: Kategóriák lekérése Eloquent Model osztály segítségével a Postman-ben

De alapvetően a Laravel-ben nem Eloquent osztály példányokat (egy kategóriát, egy blogbejegyzést stb.) küldünk vissza egy API kérésre válaszként, hanem erőforrásokat (**resources**). Így például nem kell visszaadni válaszként az Eloquent példányt jelképező adattábla sor időbélyegeit (**created_at**, **updated_at**), vagy bármilyen olyan mezőt, amelyet nem szeretnénk elérhetővé tenni az API felületen keresztül. Hozzuk létre a kategóriákat jelképező erőforrást:

```
php artisan make:resource V1/CategoryResource
```

A verziószámot itt is adjuk hozzá, hogy nyomon követhető lehessen a jövőben. Az utasítás kiadásának hatására létre is jön az új osztály és fájl az **app / Http / Resources** mappában. A **toArray()** metódus visszatérési tömbjében tudjuk definiálni azokat a mezőket, amelyeket át szeretnénk majd küldeni a felhasználói kérésekre válaszként:

```php
public function toArray(Request $request): array
{
  return [
    'id' => $this->id,
    'name' => $this->name
  ];
}
```

7–60. kódrészlet: CategoryResource osztály toArray() metódusa

Megjegyzés: bár itt még nem látszódik, az attribútumnevek megadásánál a *„camel case”* átírási szabályt követjük, tehát a **created_at**-ből **createdAt** lesz.

A **CategoryResource** kódjában a **$this** magát a **categories** adattábla egy sorát jelképezi.

Alakítsuk át az erőforrásnak megfelelően a Controller **index()** és **show()** metódusait:

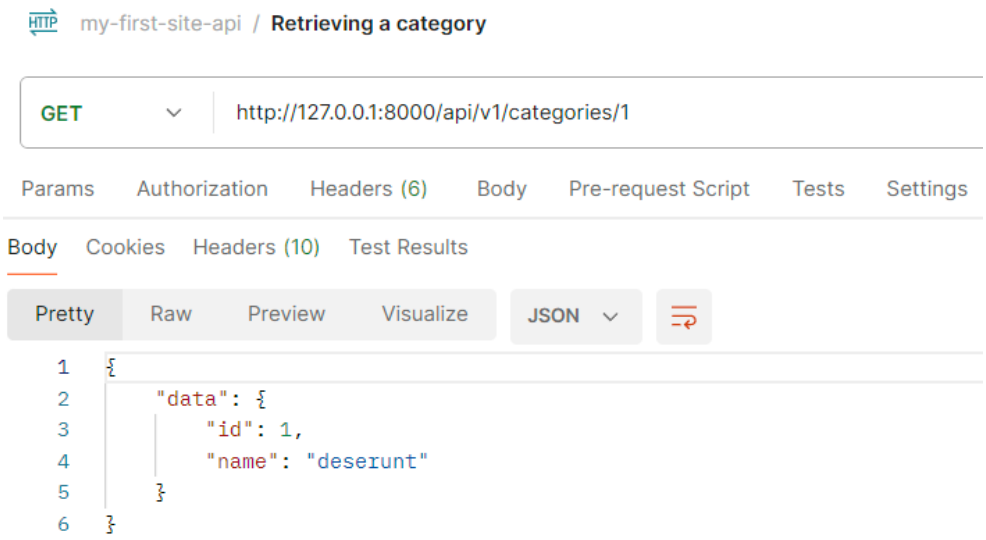
```
public function index()
{
  return CategoryResource::collection(Category::all());
}

public function show(Category $category)
{
  return CategoryResource::make($category);
}
```

7–61. kódrészlet: CategoryResource gyűjtemény és elem visszaadása az index() és show() vezérlő metódusokban

Ne felejtsük el importálni a Controller fájlban a **Category** Model-t, a **Controller** ősosztályát, amelyet kiterjeszt a **V1/CategoryController**-ünk, a **CategoryResource** osztályt, illetve az **Illuminate / Http / Response** ősosztályt is.

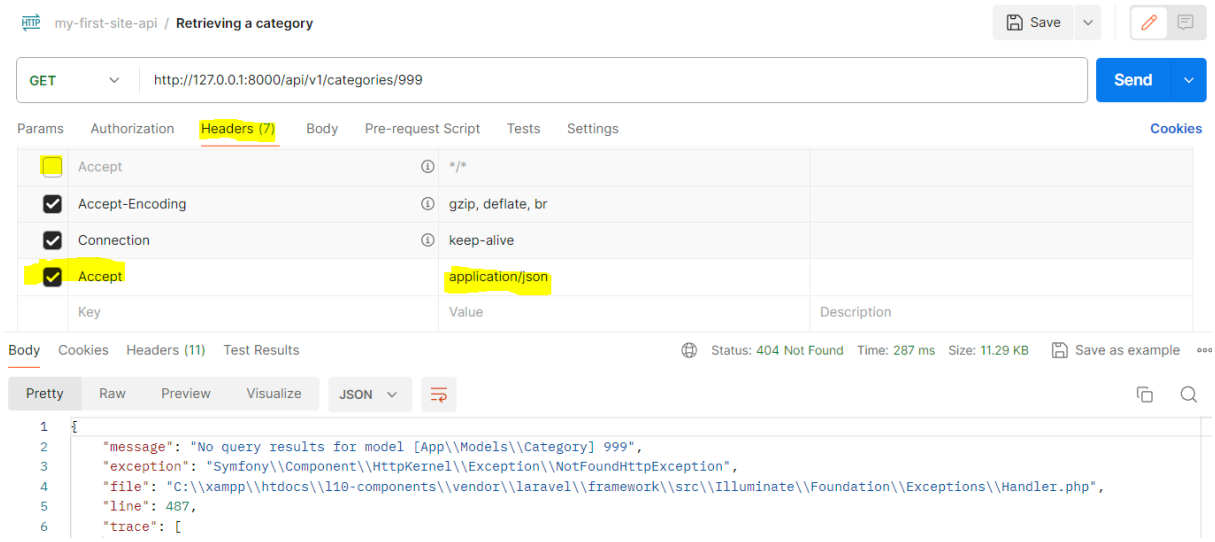
Ha most újra lekérjük a végpontokat a Postman-ben, akkor látható, hogy csak két mező (**id, name**) került visszaadásra és föléjük egy új *„data"* réteg került be. Ez a legjobb gyakorlat (*„best practice"*) a JSON alapú API-k használatakor, hiszen így az adatok mellett bármilyen más információt, üzenetet is át tudunk adni a felhasználónak, például azt, hogy sikeresen vagy sikertelenül próbált meg végrehajtani valamit.

Megjegyzés: ha az **index()** metódusban a **Category::all()** helyett **Category::paginate()**-et használunk, akkor automatikusan lapozhatóvá teszi a kategóriáinkat és hozzáadja a visszaadott gyűjteményhez a lapozáshoz szükséges paramétereket (első / utolsó oldal linkje, oldalanként megkapott kategóriák száma stb.). Így ezeket a lapozáshoz használt API linkeket is lekérhetjük a Postman segítségével.



7–18. ábra: Egy adott kategória erőforrás lekérése

Előfordulhat, hogy olyan kategóriát szeretnénk lekérni, amely nem létezik (például nálam a 999.-et). Ekkor nem JSON válasz érkezik, hanem HTML kód, amelyet a Postman a *„Pretty"* lapfülön teljes egészében megmutat és a *„Preview"* lapfülön meg is tud szépen jeleníteni, de ez egy API működésénél biztosan nem megfelelő, mert ekkor egy egész HTML fájlt kellene feldolgoznia a hívó félnek, amely nem reális.



7–19. ábra: A kérésre JSON válasz kikényszerítése a Postman-ben

A Postman egy kényszert próbál ránk erőltetni, ha a *„Headers”* lapfülre rámegyünk a kérés alatt. Az *„Accept”* fejléc attribútum beállítása mindenképpen ***/*** és nem is szerkeszthető, úgyhogy egy kicsit trükközni kell, és ki kell venni a sor elején lévő checkbox-ból a pipát, így az már nem lesz érvényes. Ezután fel kell venni újként az *„Accept”* kulcsot **application/json** értékkel.

Ezáltal kikényszerítettük a szervertől, hogy a válasznak JSON-ban kell érkeznie, és így kinyerhető belőle például a **message** attribútum több más attribútum mellett. Most már könnyedén feldolgozható a kérésre kapott válasz, hogy nincsen eredménye a 999-es kategória lekérdezésének. (*Megjegyzés:* ezt a 10.1.2.3. alfejezetben automatikussá tesszük.)

Feladat: most gyakorlásként érdemes megcsinálni a blogbejegyzések (**Post**), címkék (**Tag**) és a kommentek (**Comment**) modellekhez tartozó API útvonalakat, erőforrásokat, vezérlőket és azok **index()**, **show()** metódusait.

A Post erőforrásnál figyeljünk a *„camel case”* átírásra, tehát a **toArray()** metódus visszatérési tömbjénél a **published_at** mező esetén legyen a kulcs **publishedAt**, az érték pedig **$this->published_at**

A helyes működés teszteléséhez duplikáljuk a már meglévő request-jeinket a Postman-ben, és állítsuk át az útvonalakat az erőforrásnak megfelelően.

Miután elkészültek az iménti feladatban kért dolgok. Vizsgáljuk meg a kapcsolatok kérdéskörét!

7.5.3.1. Erőforrás kapcsolatok lekérdezése

Egy kategóriához több blogbejegyzés is tartozhat, ezért a kategóriák (vagy egy konkrét kategória) lekérésénél jelenítsük meg a hozzájuk / hozzá tartozó blogbejegyzéseit is. A cél tehát az, hogy ha a felhasználó a következő kérést indítja az alkalmazásunk felé, akkor a kategóriákat blogbejegyzésekkel együtt visszaadjuk:

http://127.0.0.1:8000/api/v1/categories?includePosts=true

Ha ezt a linket hozzáadjuk egy új kéréshez (*„Retrieving categories with posts”* néven) a Postman-ben, akkor egyelőre csak a kategóriákat adja vissza nekünk, és nem foglalkozik azzal, hogy mi milyen paramétert adtunk még hozzá a linkhez.

A módosítást kezdjük az **index()** metódussal és alakítsuk át a magját!

```
public function index()
{
  $categories = Category::query();

  if( request('includePosts') )
  {
    $categories = $categories->with('posts');
  }

  return CategoryResource::collection($categories->paginate());
}
```

7–62. kódrészlet: A kategóriákat kérés esetén blogbejegyzésekkel együtt adjuk vissza

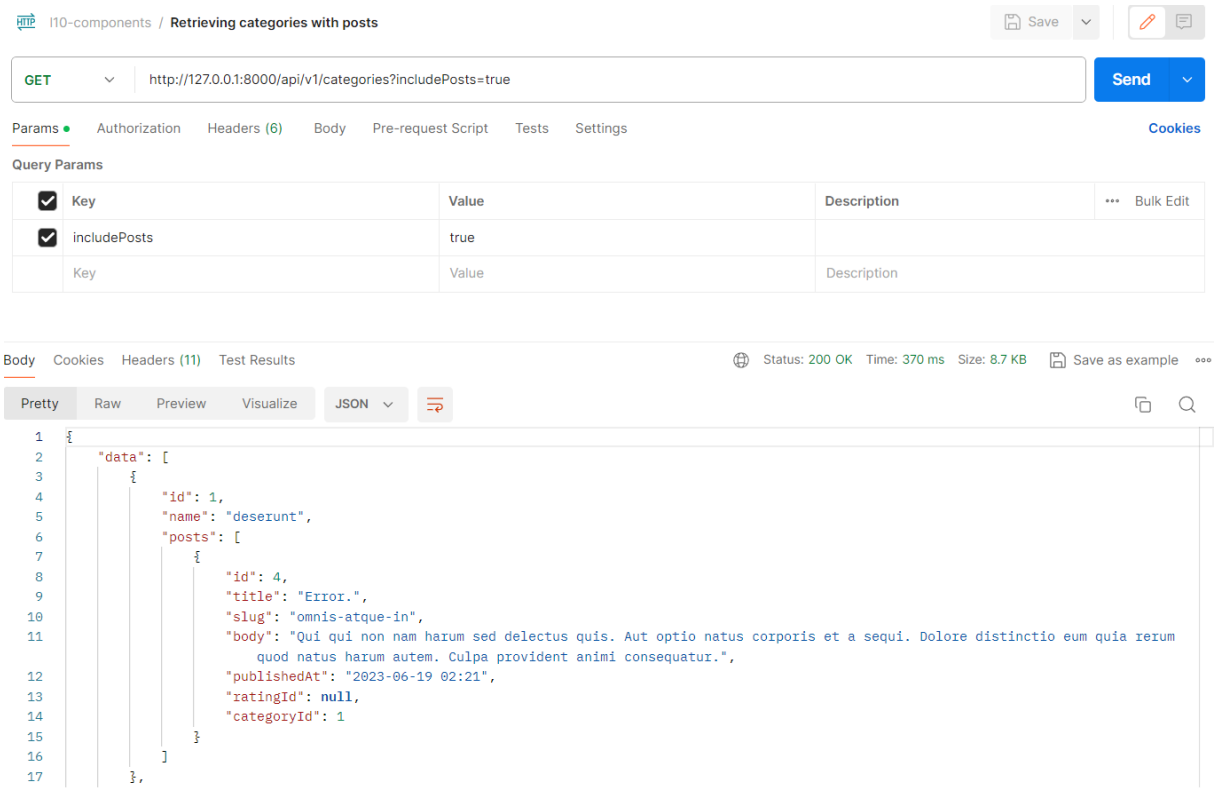
Először a kategóriákhoz egy lekérdezést rendelünk, amelyet bővítünk, ha a felhasználói kérésben igaz értéket kap az *„includePosts”* paraméter. Azonban ez így még nem adja vissza nekünk a blogbejegyzéseket is a kategóriáknál. Ehhez még bővítenünk kell a **CategoryResource toArray()** visszatérési tömbjét:

```
return [
  'id' => $this->id,
  'name' => $this->name,
  'posts' => PostResource::collection($this->whenLoaded('posts'))
];
```

7–63. kódrészlet: Kapcsolat feltételes betöltődése a whenLoaded() metódussal

Ez lehetővé teszi a Controller számára, hogy eldöntse, mely kapcsolatokat kell betölteni a modellbe, és az erőforrás egyszerűen csak akkor tartalmazza őket, ha már ténylegesen betöltődött. Végső soron ez megkönnyíti az *„N+1”* lekérdezési problémák elkerülését az erőforrásokon belül. A **whenLoaded()** metódus használható egy kapcsolat feltételes betöltésére. A kapcsolatok szükségtelen betöltésének elkerülése érdekében ez a metódus a kapcsolat nevét fogadja csak el paraméteréül a konkrét kapcsolat helyett.

Az eredmény itt látható:



7–20. ábra: Kategóriák blogbejegyzésekkel együtt történő lekérése

A show esetében egy kicsit módosítani kell a kódon ahhoz, hogy az adott kategóriához is betöltődjenek a blogbejegyzések:

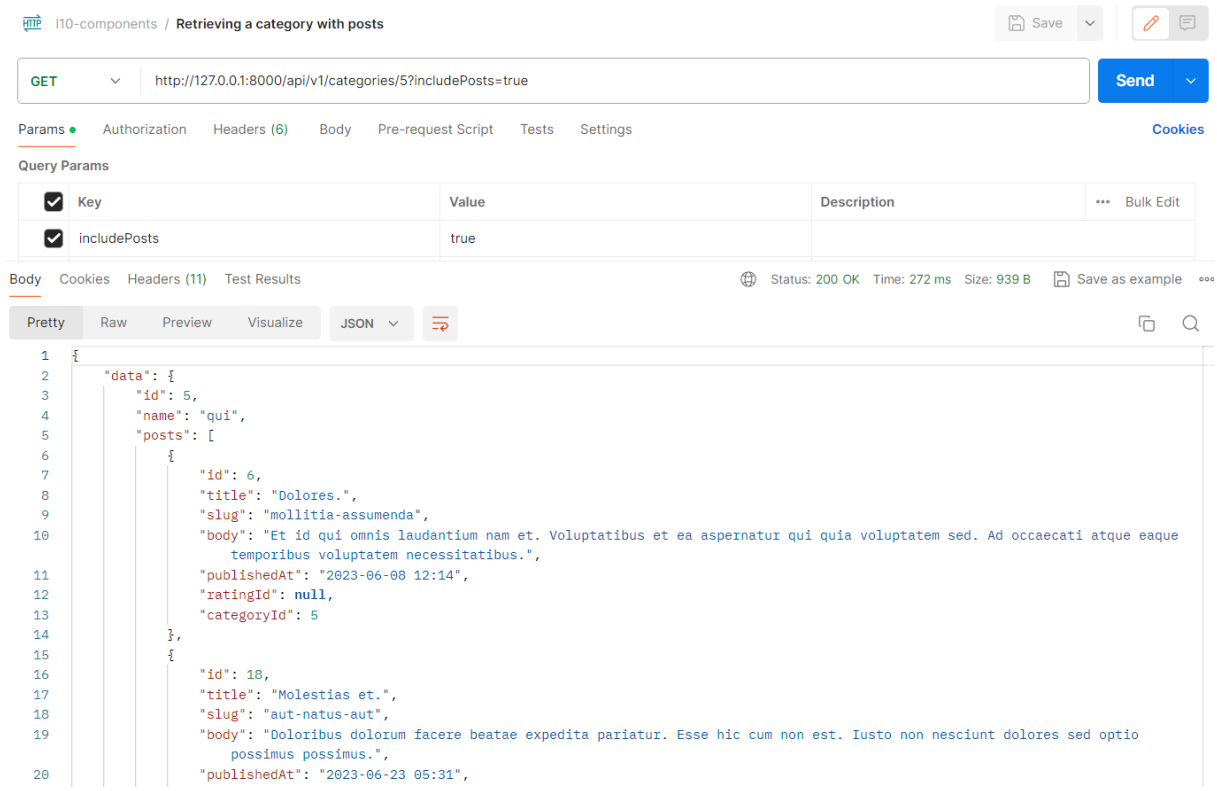
```
public function show(Category $category)
{
  if( request('includePosts') )
  {
    return CategoryResource::make($category->loadMissing('posts'));
  }
  return CategoryResource::make($category);
}
```

7–64. kódrészlet: Adott kategóriához szükséges blogbejegyzések betöltése

A **loadMissing()** metódus betölti az (egy elemű) Eloquent gyűjteményhez a kapcsolatot, ha még nem lenne betöltve hozzá.

Az alábbi linkkel kinyerhető az adott kategória erőforrás a blogbejegyzéseivel együtt:

http://127.0.0.1:8000/api/v1/categories/5?includePosts=true



7–21. ábra: Adott kategória lekérése a kapcsolódó blogbejegyzéseivel együtt

Ez alapján a többi erőforrásnál is meg lehet csinálni – igény szerint – kapcsolatok betöltését is az adott erőforrások mellé.

Az alfejezetben található programkód módosítások (a Feladat megoldásával együtt, de további kapcsolatok definiálása nélkül) megtalálhatók ebben a GitHub commit-ben.

7.5.4. Erőforrások létrehozása, módosítása, törlése (store, update, destroy)

Az eddigi linkek elérése működik a böngészők segítségével is, hiszen GET-es felhasználói kérések voltak. Innentől viszont már óriási segítségünkre lesz a Postman, mivel nem kell például JavaScript nyelven (vagy valamilyen egyéb erre a nyelvre épülő keretrendszerrel) API hívásokat szimulálnunk, hiszen ezeket a műveleteket űrlapok elküldésével kellene végrehajtanunk.

7.5.4.1. Létrehozás (store) – POST kérés

Az eltároláshoz az útvonal már regisztrálásra került, de ezt kell majd definiálni a Postman-be is:

POST http://127.0.0.1:8000/api/v1/categories

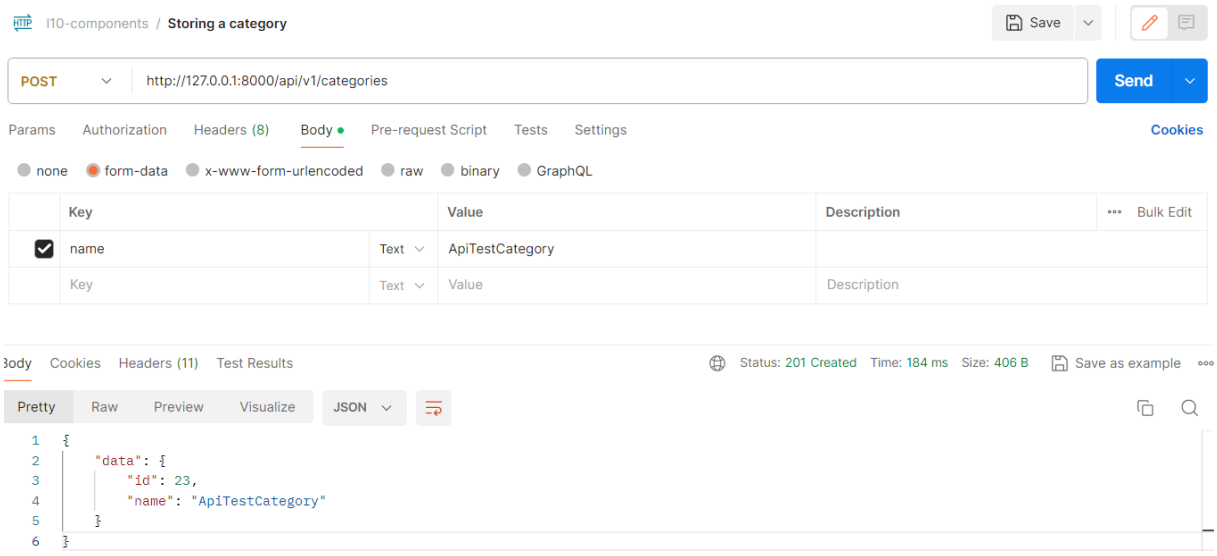
Ezzel kell létrehoznunk majd egy új kategóriát. A **V1/CategoryController**-ben a **store()** metódus szolgál az erőforrás létrehozására.

```
public function store()
{
  return CategoryResource::make(
    Category::create(
      request()->all()
    )
```

```
    );
}
```

7–65. kódrészlet: CategoryController store() metódusa API híváskor

Ez mindössze egyetlen utasítás, csak a jobb átláthatóság miatt több sorba tördelve. A **request()** segédmetóduson keresztül az összes felhasználótól érkező paramétert fogadjuk, a létrehozáskor a **Category** Model osztály **$fillable** attribútuma miatt úgyis csak a **name** mezőt engedi át. Ezt a mezőt a Postman-ben az új kérés a *„Body"* lapfülön, azon belül pedig például a *„form-data"* fülön adhatjuk meg (key) egy példa értékkel (value), majd el is indíthatjuk a kérést.



7–22. ábra: Kategória létrehozása (kérés és válasz)

A kérés eredményeként visszakapjuk a létrejövő erőforrást, illetve a válasz felett láthatjuk a státuszkódot: *„201 Created"*, amely jelzi, hogy sikeresen elkészült. Kétszer nem tudnánk ugyanezt a kérést elindítani, mert a **Category name** attribútumának egyedinek kell lennie a **categories** adattáblában. Válaszul egy HTML oldalt kapunk és *„500 Internal Server Error"* státuszkódot. API szempontjából a kérés *„Accept"* fejléc értéke átállíthatjuk **application/json**-ra, hogy JSON választ kapjunk, amely könnyebben feldolgozható, van benne **message** és egy **exception** attribútum a hibáról.

7.5.4.2. Frissítés (update) – PUT kérés

Az erőforrás módosítását, frissítését szintén egy már létező útvonalon keresztül tudjuk elérni:

PUT http://127.0.0.1:8000/api/v1/categories/23

A *„Body"* lapfülön most az *„x-www-form-urlencoded"* fület kell választani a paraméterek átküldéséhez. Ez ugyanazt a kódolást tartalmazza majd, mint amikor az URL-be írtunk be paramétereket és értékeiket **=** (egyenlőségjellel) összekötve és **&** (és) karakterekkel összefűzve több paraméter esetén. Ha ezt a fület választjuk, akkor a Postman elküldés előtt lekódolja (RFC 1738 kódolás[9]) a paraméter adatokat. Így

[9] A kódolási szabványról részletesebben itt olvashatunk: https://datatracker.ietf.org/doc/rfc1738/

azonban csak szöveges formátumú adatokat küldhetünk, míg a *„form-data”* esetében küldhettünk volna fájlokat is akár.

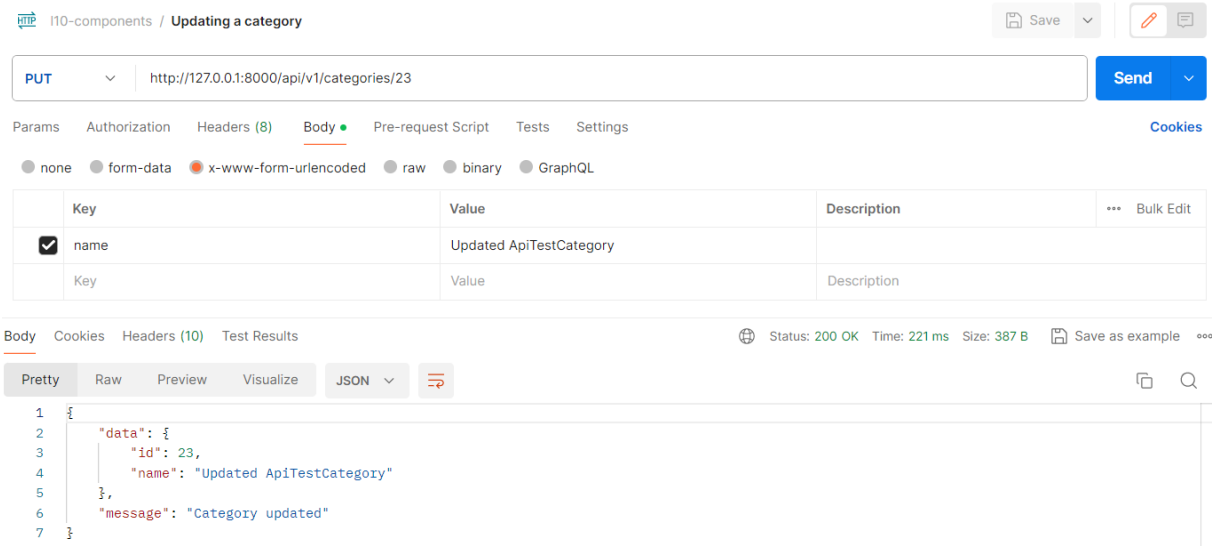
Ezeken kívül küldhetünk még nyers (*„raw”*) adatokat is: mi magunk is megfogalmazhatjuk például a JSON szerkezetet, amit továbbítani szeretnénk a szervernek, de ezen kívül HTML, XML jelölő nyelven is megfogalmazhatjuk és a kérés magjában elhelyezhetjük adatainkat, sőt még JavaScript kódot is küldhetünk ugyanezen a módon, de most a kódbeszúrásos támadást ne teszteljük itt külön.

Maradjunk a példánál és folytassuk a munkát, a **V1/CategoryController update()** metódusa a következő szerint alakul:

```php
public function update(Category $category)
{
  $category->update([
    'name' => request('name')
  ]);

  return response()->json([
    'data' => CategoryResource::make($category),
    'message' => 'Category updated'
  ], Response::HTTP_OK);
}
```

7–66. kódrészlet: CategoryController update() metódusa API híváskor

A paraméterül kapott **$category** objektumon hívott **update()** metódus frissíti a kategória nevét. Utána a visszatérési utasításban mi magunk rakhatjuk össze a JSON választ, amely egy tömb két eleméből tevődik össze (a kategória erőforrást tartalmazza adatként, míg üzenetként a kategória frissítésről adunk visszajelzést), és mindezeket kiegészítjük egy HTTP 200-as OK státuszkóddal. Importáljuk hozzá a fájl elején az **Illuminate\Http\Response** osztályt. A kérés elküldése és a válasz így látható a Postman-ben:



7–23. ábra: Kategória módosítása (kérés és válasz)

Eredményként visszakapjuk a definiált válasz adatait.

7.5.4.3. Törlés (destroy) – DELETE kérés

A törlés is hasonlóan működik, mint a módosítás. Az útvonal itt is regisztrálásra került már:

DELETE http://127.0.0.1:8000/api/v1/categories/21

Ezután nincs más hátra, mint definiálni a **V1/CategoryController destroy()** metódusának magját:

```php
public function destroy(Category $category)
{
  $category->delete();

  return response()->json([
    'message' => 'Category deleted'
  ], Response::HTTP_NO_CONTENT);
}
```

7–67. kódrészlet: CategoryController destroy() metódusa API híváskor

Alapértelmezetten a válasz egy törlés esetén lehet többféle státuszkódú is:

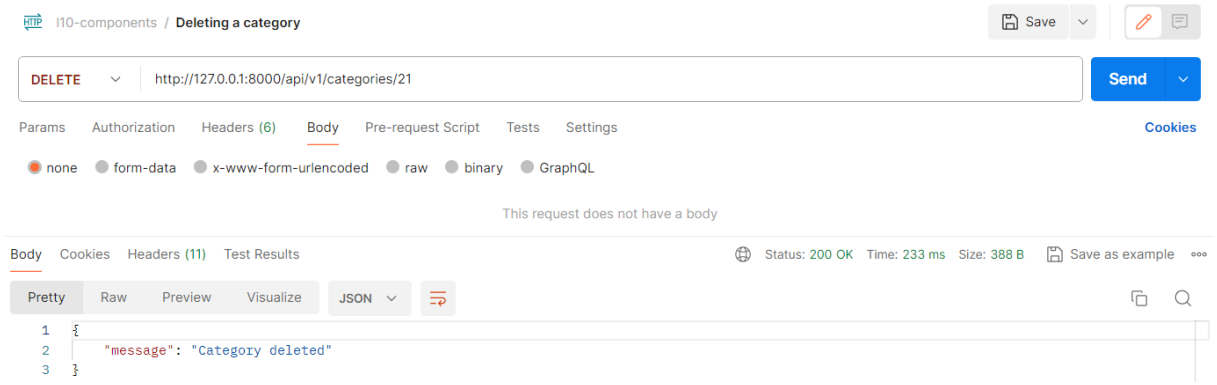
1. *„200 OK”,* ha a sikeres törlés után a válasz tartalmaz egy erőforrást is.
2. *„202 Accepted”,* ha a művelet még nem került végrehajtásra.
3. *„204 No content”,* ha a művelet végrehajtásra került, de a válasz nem tartalmazza az erőforrást.

Ez utóbbit a Postman nem értelmezi jól, hiába hajtódott végre a törlés. Megtekinthetjük a Console-t a Postman-ben (bal alul az alkalmazásban), és a következőt mutatja:

```
DELETE http://127.0.0.1:8000/api/v1/categories/23
Error: Parse Error: Expected HTTP/

Request Headers
  User-Agent: "PostmanRuntime/7.36.3"
  Accept: "*/*"
  Postman-Token: "55dab199-8733-4f9d-a7c2-9004af82b152"
  Host: "127.0.0.1:8000"
  Accept-Encoding: "gzip, deflate, br"
  Connection: "keep-alive"
```

7–24. ábra: Postman Console hiba „204 No Content” válasz esetén

Ez egy ismert Postman bug évek óta, ezért – mert bár nincsen erőforrás, amit visszaadnánk –, de 200-as **HTTP_OK** választ írjunk be a **destroy()** visszatérési utasításába (ahogy az **update()**-ben is tettük).



7–25. ábra: Kategória törlése (kérés és válasz)

Gyakorlásként érdemes megvalósítani a többi erőforrás létrehozó, módosító és törlő vezérlő metódusait is, és ellenőrzésként kipróbálni a helyes működésüket a Postman-ben. Az alfejezetben megtalálható programkód módosítások ebben a GitHub commit-ben találhatók meg.

7.6. Automatikus tesztelés (CRUD útvonalak, műveletek és a kapcsolatok egy elszeparált adatbázisban)

A releváns tesztesetek létrehozása előtt érdemes futtatni a tesztelési utasítást a terminal-ban:

```
php artisan test
```

Mindkét **ExampleTest** osztálynak és a bennük lévő 1-1 tesztesetnek (**assert** műveletnek) le kell futnia sikeresen.

Hiba és figyelmeztetés megoldásai

Ha esetleg hibát kapnánk, akkor győződjünk meg róla, hogy fut-e az alkalmazásunk a böngészőben (ha nem, akkor indítsuk el: `php artisan serve` és `npm run dev` futtatása történjen meg egy-egy terminal-ban).

Ha esetleg figyelmeztetést *(„Your XML configuration validates against a deprecated schema. Migrate your XML configuration using "--migrate-configuration"!")* kapnánk a tesztelést indító utasítás kiadásának hatására, akkor egy újabb utasítást kell kiadnunk, amely a PHPUnit beállítását (**phpunit.xml** fájl a projekt gyökerében) módosítja nekünk:

```
vendor/bin/phpunit --migrate-configuration
```

Az utasítás hatására létrejön egy backup (mentett, archivált) fájl **phpunit.xml.bak** névvel, ami az eddigi **phpunit.xml** fájl tartalmát menti el nekünk, míg az új beállítások megtörténtek a **phpunit.xml** fájlban. Utána már figyelmeztetés nélkül fog lefutni a tesztelési utasítás.

Ha sikeresen lefutottak, akkor törölhető a két **ExampleTest.php** fájl a **Feature** és a **Unit** könyvtárból is. A továbbiakban adatbázistáblákat érintő teszteseteket fogunk futtatni.

A CRUD műveletek tesztelésénél már érdemes figyelni arra, hogy ne az éles (production) vagy a fejlesztési (dev) környezetben használatos adatbázist *„szemeteljük tele”,* hanem egy kimondottan tesztelésre használt adatbázist, amelynek a szerkezete (táblák, mezőik, kapcsolataik stb.) ugyanolyanok lesznek, mint az éles adatbázisé.

Vigyázat! Ha nem végezzük el az alfejezet további részében található beállítások közül valamelyiket (**phpunit.xml** vagy **.env.testing**), akkor az éles adatbázisba fognak lefutni a tesztjeink, és ez nagy valószínűséggel az éles adataink elvesztésével járhat!

7.6.1. Tesztelés a phpunit.xml beállításai alapján

Nyissuk meg a projekt gyökerében található **phpunit.xml** fájlt. Ebben szintén megtalálhatók azok a környezeti változók, amelyek a teszteléshez hasznosak lesznek, de egyelőre ki vannak kommentelve: **DB_CONNECTION** és **DB_DATABASE** sorai.

Alapértelmezetten ez egy SQLite adatstruktúrában menedzseli az erőforrások adatait és nem is egy külön fájlban, hanem a memóriában fogja eltárolni az adatbázist. Definiáljuk az első tesztesetet, ami az adatbázisunkat érinti!

7.6.1.1. Kategórialista megjelenítése (index) teszteset

Kezdésnek hozzuk létre az új teszt osztályt:

```
php artisan make:test CategoryTest
```

A teszteset és aztán annak az implementálása:

1. Hozzunk létre egy kategóriát!
2. Próbáljuk elérni a kategóriák index útvonalát! Ellenőrizzük, hogy
    a. sikerült-e elérni,
    b. látható-e a létrehozott kategória neve.

```
public function test_category_index_contains_non_empty_table()
{
  $category = Category::create([
    'name' => 'Category 1',
  ]);
  $response = $this->get('/categories');
  $response->assertStatus(200);
  $response->assertSee('Category 1');
}
```

7–68. kódrészlet: Kategória index funkciójának tesztelése nem üres adattábla esetén

A teszteset osztály előtt importáljuk a **Category** Model osztályt is, természetesen. Futtassuk a tesztesetet!

```
php artisan test
```

De rögtön elbukik (FAILED), a kapott hibaüzenet: *SQLSTATE[HY000]: General error: 1 no such table: categories (Connection: sqlite, SQL: insert into "categories" ("name", "updated_at", "created_at") values (Category 1, 2023-08-14 21:34:05, 2023-08-14 21:34:05))*

A **categories** adattábla tehát nem található az SQLite adatbázisban, így a metódus legelső sora sem tud már végrehajtódni (kategória létrehozása).

7.6.1.2. RefreshDatabase trait használata

Használjuk a Laravel keretrendszer **RefreshDatabase** trait-jét: ezzel gyakorlatilag a Laravel keretrendszer előre megírt osztályait és azok funkcionalitásait tudjuk használni. Itt konkrétan azt, amellyel előkészíti az adatbázis szerkezetet a teszteléshez, mintha lefuttatnánk a terminal-ban egy `php artisan migrate:fresh` parancsot. Ezért is nagyon veszélyes, hogy ha véletlenül a phpunit.xml megfelelő részeinek (vagy majd a 7.6.3. alfejezetben látható megoldás) beállítása nélkül futtatjuk a tesztelést, akkor könnyedén elveszíthetjük az éles adatbázisban lévő adatainkat (*„rákérdezés nélkül”* fogja végrehajtani az adatbázis újraépítését a rendszer).

Adjuk hozzá a trait-et a **CategoryTest** osztály magjának legelejére (figyelem, nem az osztályon kívülre, hanem azon belülre):

```
use RefreshDatabase;
```

7–69. kódrészlet: RefreshDatabase trait használata

Ha most futtatjuk a tesztelést, akkor újra hibaüzenetet kapunk, de most már más a probléma: *SQLite doesn't support dropping foreign keys (you would need to re-create the table).*

Mivel a háttérben a **RefreshDatabase** trait használata miatt lefut a `php artisan migrate:fresh` utasítás, ezért először a **phpunit.xml** alapján eldobja az adatbázisban lévő táblákat a rendszer, majd újra fel akarja építeni a táblák szerkezeteit a migrációs fájlok alapján. Ez azonban problémát jelent, mivel az SQLite nem támogatja az idegen kulcs kényszer megszüntetését (ahogy a hibaüzenet is írja). A **change_post_id_fk_in_ratings_table** migrációs fájl **up()** metódusában pontosan ez van, ami a problémát okozza (**dropForeign()**) nekünk ebben a szituációban.

7.6.1.3. SQLite probléma feloldása vagy elkerülése

Két lehetőségünk van:

1. Maradunk a tesztelésnél az SQLite adatbázis-kezelőnél, de akkor módosítanunk kell a migrációs fájljainkat.
2. Áttérünk a tesztelésnél is a MySQL adatbázis-kezelőre és akkor a **phpunit.xml**-t kell módosítanunk.

Vegyük át mindkét lehetséges megoldást, aztán válasszuk majd a nekünk megfelelőbbet!

Az első esetben, ahogy írtam az imént az SQLite adatbázis miatt a migrációs fájlok **dropForeign(…)** utasításai okozzák a problémát. Ha úgy érezzük, hogy nincs olyan sok migrációs fájlunk, amiken módosítani kellene, akkor a legegyszerűbb, ha egy feltételvizsgálatot teszünk be ezek elé az utasítások elé, mint például az **add_rating_id_to_posts_table** migrációs fájl **down()** metódusában található utasításnál:

```
if (DB::getDriverName() !== 'sqlite') {
  $table->dropForeign('posts_rating_id_foreign');
}
```

7–70. kódrészlet: dropForeign probléma megoldása SQLite adatbázis használatakor

A VSCode bal oldali függőleges menüjében a nagyító ikon segítségével az egész projektben kereshetünk, így ott érdemes a *„dropForeign"* kulcsszóra rákeresni és mindegyik találatnál (öt darab van), elhelyezni ezt a feltételvizsgálatot.

Ezután már működni fog, ha elindítjuk a tesztelést!

A második módszer, hogy SQLite helyett MySQL adatbáziskiszolgálót használunk és a phpunit.xml fájlban elvégezzük a szükséges módosításokat:

```
<env name="DB_CONNECTION" value="mysql"/>
<env name="DB_USERNAME" value="root"/>
<env name="DB_DATABASE" value="l10_blog_test_db"/>
```

7–71. kódrészlet: Teszteléshez a környzeti változók beállítása a phpunit.xml-ben

Megjegyzés: a **DB_PASSWORD** paramétert azért nem kell felvenni, mert a jelszó egy üres szöveg, ami megegyezik az alapértelmezett értékkel, lásd ehhez a **config / database.php** fájl mysql-re vonatkozó részét.

Ehhez persze szükség van az új tesztelési adatbázisra a MySQL kiszolgálón, így ott a tesztelés megkezdése előtt hozzuk létre neki az adatbázist:

```
CREATE DATABASE l10_blog_test_db;
```

Utána indíthatjuk ezzel a módszerrel is a tesztelést és már működni fog!

Ismétlem, válasszuk a nekünk megfelelőbb megoldást a fentiek közül, mindkettőnek megvan az előnye és a hátránya is!

Az alfejezetben található programkód módosítások ebben a GitHub commit-ben találhatók meg. A változtatások mindkét megoldást tartalmazzák, de a második módszert csak kikommentezve a **phpunit.xml** fájlban.

Megjegyzések:

1. A tesztelési adatbázisban a MySQL kiszolgálón hiába keresnénk a **categories** táblában létrehozott egy darab kategóriát tartalmazó sort, az a tesztelés hibátlan lefutása után törlésre is fog kerülni a **RefreshDatase** trait használata miatt.
2. Ha mégis azt szeretnénk, hogy az első megoldásnál maradjunk, tehát az SQLite-ot használnánk a tesztelésre, de sok az idegen kulcs kapcsolatunk, és nem akarjuk mindenhol a feltételvizsgálatot beilleszteni a **dropForeign()** utasítás elé, akkor egy *„gyors javítást"* (*„hotfix"*-et), javítást érdemes alkalmazni a Laravel keretrendszer SQLite driver-éhez, mert ezt még nem támogatja. Részletesebb információért ezt az oldalt érdemes átböngészni.

7.6.2. További CRUD tesztesetek összeállítása sablon szerint

Állítsuk össze és implementáljuk a további CRUD funkcionalitások teszteseteit! A már bevált folyamatot kövessük, a kategória erőforrásra hozzuk létre a teszteseteket: már csak hat funkcionalitást kell tesztelni, **assert()** utasításokkal tudjuk ezt elvégezni a **CategoryTest** osztályban (hiszen a listázást – **index** – már teszteltük a hét RESTful metódus közül).

A **RefreshDatabase** trait használatának köszönhetően, minden egyes teszteset (metódus) végrehajtása előtt és után megtörténik az adatbázisban lévő táblák *„lebontása és felépítése"*.

Bár a továbbiakban *„program-képességeket"* (feature) tesztelünk, ezekre is vonatkoznia kell azon tulajdonságoknak, amely jellemzők egy jó egységtesztnek (unit) is ismertetőjegyei: legegyszerűbben a **FIRST** mozaikszóval jegyezhetjük meg őket (Robert C. Martin - Tiszta kód könyve nyomán):

1. *Gyorsak* (**F**ast): a teszteknek gyorsan kell lefutniuk.
2. *Függetlenek* (**I**ndependent): a teszteknek egymástól függetlennek kell lenniük.
3. *Megismételhetőek* (**R**epeatable): a teszteknek bármennyiszer megismételhetőnek kell lenniük.
4. *Önmagukra nézve érvényesek* (**S**elf-validating): a teszteknek logikai kimenete kell, hogy legyen, ami által meg tudjuk állapítani, hogy átmentek vagy elbuktak.
5. *Jól időzítettek* (**T**imely): a teszteket kellő időben kell implementálnunk, lehetőleg a közvetlenül tesztelendő kód előtt (tesztvezérelt szoftverfejlesztés: előbb a tesztek, aztán a megvalósítás).

Az *automatikus teszteket* ezek után három részre oszthatjuk, amely egy jó mintának *(„receptnek")* tekinthető a tesztek elkészítéséhez:

1. *Előkészítés, elrendezés* (**A**rrange): ez a rész felelős a tesztelt rendszer egy kívánt állapotba állításáért azáltal, hogy bizonyos adatokat hozzáad, definiál, vagy beállít valamit. Ez a rész egy vagy több utasításból is állhat.
    a. Például: létrehoztunk egy új kategóriát.
2. *Cselekvés* (**A**ct): ez a rész szolgál a tesztelt rendszer metódusainak meghívására egységteszt esetén, míg képességteszt esetén általában valamilyen webes útvonal elérése, vagy egy API hívás működése történik meg. Itt szoktuk meghatározni azt is, ha valamilyen felhasználó szerepkörét eljátszva, szimulálva szeretnénk elérni az útvonalakat, mintha azt tényleg egy felhasználó végezné el egy böngésző előtt ülve. Amennyiben lehetőség van rá, akkor a kimeneti értéket eltárolhatjuk és felhasználhatjuk. Ez a rész általában egyetlen utasításból áll.
    a. Például: megpróbáltuk elérni a kategóriákat tartalmazó lista oldal útvonalát.
3. *Állítás, kijelentés* (**A**ssert): ez a rész szolgál az iménti szakasz (cselekvés) kimenetének az ellenőrzésére. A tesztelt kimenetel lehet egy visszatérési érték vagy a tesztelt rendszer végső állapota. Az assert utasításoknak mindig igaznak kell lenniük, különben a teszt elbukik. Ez a rész is állhat több utasításból, de ne feltétlenül akarjunk túl sok mindent ellenőrizni, hanem akkor már ezeket próbáljuk meg elszeparálni különböző tesztesetekbe.
    a. Például: ellenőriztük, hogy 200-as HTTP státuszkóddal tért-e vissza az útvonal lekérésére a válasz.

Az iménti felsorolást tehát nagyon jól alkalmazhatjuk, mint egy receptet a tesztek elkészítéséhez. Ezt a módszertan (3A: Arrange-Act-Assert) Bill Wake-től ered, amely módszertan manapság már egyre inkább helyettesítheti, leválthatja a felhasználói történeteknél (user stories) alkalmazott Given-When-Then sablont (lásd a 3.7.3. alfejezetet).

7.6.2.1. Adott erőforrás megjelenítése (show)

A 3A sablon használatára nézzünk meg egy példát, ami a show útvonal tesztelésére szolgál.

Arrange	Hozzunk létre egy kategóriát.
Act	Próbáljuk meg elérni az újonnan létrehozott kategória útvonalát.
Assert	Ellenőrizzük, hogy sikeres volt-e az elérés és látható-e az új kategória neve.

7–4. táblázat: 3A sablon használatára példa: show

A három rész implementálása pedig így nézhet ki egy tesztesetben, tesztelő metódusban:

```
public function test_show_category_ successful()
{
  // Arrange
  $category = Category::create([
    'name' => 'Category 1',
  ]);

  // Act
  $response = $this->get('/categories/' . $category->id);

  // Assert
  $response->assertStatus(200);
  $response->assertSee('Category 1');
}
```

7–72. kódrészlet: Kategória show funkciójának tesztelése

Ez így jó is lehetne, de az **assertSee()** nem mindig ad korrekt eredményt, mert mi van akkor, ha például az oldal más részén látható ez a *„Category 1"* felirat és nem feltétlenül a kategóriákat tartalmazó táblázatban vagy esetleg van *„Category 10"* az oldalon? Emiatt érdemesebb inkább azt tesztelni, hogy az *adat* megvan-e a nézetben, amely *adatnak* ott kell lennie. Az utolsó **assertSee()** utasítás után szúrjuk be ezt a kódrészletet:

```
$response->assertViewHas('category', function ($cat) {
  return $cat->name === 'Category 1';
});
```

7–73. kódrészlet: assertViewHas() használata show metódus tesztelésénél

Itt a **$category** változó értékét vizsgáljuk meg, amit a **categories.show** nézetben használunk. Az objektum name attribútumának nyilván annak kell lennie, mint amit előtte a metódusban beállítottunk, így itt most már az adatot ellenőrizzük, nem csak azt, hogy maga a szöveg látható-e az adott nézetben.

Ezek alapján az index nézet tesztelésére (test_category_index_contains_non_empty_table) is visszatérhetünk, és ellenőrizhetjük ott is (az assertSee() utasítás után), hogy a categories gyűjtemény tartalmazza-e az éppen létrehozott új kategóriát (ezt ott a use után kell átadnunk a függvénynek).

```
$response->assertViewHas('categories', function ($collection) use
($category) {
  return $collection->contains($category);
});
```

7–74. kódrészlet: assertViewHas() használata index metódus tesztelésénél

Mindkét teszteset lefutásánál helyes, zöld eredményt kell kapnunk.

7.6.2.2. Létrehozási útvonal elérése (create)

A létrehozási útvonal elérése egy aránylag egyszerű teszttel ellenőrizhető, hiszen ilyet már korábban (3.7.3. alfejezet) csináltunk többször.

```
public function test_visitor_can_access_category_create_page()
{
  $response = $this->get('/categories/create');
  $response->assertStatus(200);
}
```

7–75. kódrészlet: Egyszerű útvonal teszt a create nézet eléréséhez

7.6.2.3. Adatok eltárolása az adattáblában (store)

A store() metódus tesztelésénél már sokkal jobban megfigyelhető újra a 3A által kínált recept.

```
public function test_create_category_successful(): void
{
  $category = [
    'name' => 'Category 123',
  ];

  $response = $this->post('/categories', $category);

  $response->assertStatus(302);
  $response->assertRedirect('categories');

  $this->assertDatabaseHas('categories', $category);
}
```

7–76. kódrészlet: Kategória helyes eltárolásának tesztelése

Az Arrange (új kategória létrehozása) és Act (útvonal felkeresése, post metódussal) rész már ismerős lehet, úgyhogy térjünk rá az Assert-ekre:

- a 302-es HTTP státuszkód a *„Found”*, egy átirányítás, amely azt jelzi, hogy a kért erőforrás ideiglenesen egy új helyre költözött. Amikor egy szerver 302-es státuszkódot küld egy kérésre, az általában tartalmaz egy *„Location”* fejlécet, amely meghatározza az erőforrás új helyének URL-jét, erre a címre lesz átirányítva a böngésző.

- A HTTP állapot kódokról bővebben itt olvashatunk: https://developer.mozilla.org/en-US/docs/Web/HTTP/Status

- Az átirányítást az **assertRedirect()** utasítással ellenőrizzük is, hogy sikeresen megtörtént-e.
- Végül az **assertDatabaseHas()** metódussal ellenőrizzük, hogy a **categories** adattábla tartalmazza-e a **$category** adatait tartalmazó adatsort.

Utóbbi azonban nem biztos, hogy a legjobb megoldás, mert esetleg már tartalmazta korábban ezeket az adatokat az adattábla, úgyhogy van egy még hasznosabb megoldás az újonnan létrejövő ellenőrzésére. Szúrjuk be a tesztelő metódus végére az alábbi két sort:

```
$lastCategory = Category::latest()->first();
$this->assertEquals($category['name'], $lastCategory->name);
```

7–77. kódrészlet: Tesztelés: az utoljára beszúrt sor tényleg az-e, amit beszúrtunk

Így a **categories** adattáblába *legutóbb beszúrt elemet* hasonlítja össze az itt létrehozott **$category name** attribútumával.

Mindkét teszteset lefutásánál helyes, zöld eredményt kell kapnunk.

7.6.2.4. Szerkesztési űrlap megjelenítése (edit) a meglévő adatokkal

Itt most az **Arrange** részben nem konkrétan mi határozunk meg egy új kategóriát, hanem a **factory**-ja segítségével hozunk létre egyet, így gyakorlatilag egy véletlenszerű névvel jön létre a kategória.

Az **Act** részben az útvonalat manuálisan építjük fel, itt *nem működne* a **route()** segédmetódussal az útvonal meghatározása.

```
public function test_category_edit_with_correct_value(): void
{
  $category = Category::factory()->create();

  $response = $this->get('/categories/' . $category->id . '/edit');

  $response->assertStatus(200);
  $response->assertSee('value="' . $category->name . '"', false);

  $response->assertViewHas('category', $category);
}
```

7–78. kódrészlet: Szerkesztési űrlap mezője megkapja-e a helyes adat értéket

Az **assertSee()** metódus második paramétere (bool) az " (idézőjel) és általában a *speciális karakterek* miatt kell, ennek **false**-ra állítása nélkül ugyanis *"* HTML elemre lesz átalakítva az idézőjel, és el fog bukni a teszteset.

Az **assertViewHas()**-t pedig azért adjuk hozzá, hogy teljesen megbizonyosodjunk arról, nem csak *„véletlenül”* látjuk a kategória nevét egy bemeneti mezőben, hanem ténylegesen tartalmazza a nézet az adatot. Az **edit()** metódus a **PostController**-ben is a **$category**-t kapja meg paraméterül, itt is emiatt erre van szükség.

7.6.2.5. Frissített adatok eltárolása az adattáblában (update)

A frissítő eljárásunk működik, de azért teszteljük le ezt is, hogy a későbbiekben a helyes működés bizonyosságához csak az automatikus tesztet kelljen lefuttatni.

```
public function test_update_category_successful()
{
  $category = Category::factory()->create();

  $category->name = 'Updated Name';

  $response = $this->put('categories/'.$category->id, $category->toArray());
  $this->assertDatabaseHas('categories', [
    'id' => $category->id,
    'name' => 'Updated Name'
  ]);
}
```

7–79. kódrészlet: Szerkesztett adat eltárolásának tesztelés az adattáblában

Arrange	Az új kategória létrehozása után megváltoztatjuk a nevét.
Act	A megfelelő metódussal (**put**) próbáljuk elérni a frissítés útvonalát (1. paraméter) és átadjuk neki tömbként a kategóriát (2. paraméter)
Assert	Ellenőrizzük, hogy sikeres frissült-e a **categories** adattáblában a kategória azonosítójához tartozó kategória név.

7–5. táblázat: 3A sablon használatára példa: update

7.6.2.6. Adatok törlése az adattáblából (destroy)

A törlési eljárásunk is működik, de készítünk tesztet ehhez is.

```
public function test_category_delete_successful()
{
  $category = Category::factory()->create();

  $response = $this->delete('categories/' . $category->id);

  $response->assertStatus(302);
  $response->assertRedirect('categories');

  $this->assertDatabaseMissing('categories', $category->toArray());
  $this->assertDatabaseCount('categories', 0);
}
```

7–80. kódrészlet: Tesztelés: törölt adat ténylegesen hiányzik az adattáblájából

Arrange	Az új kategória létrehozása.
Act	A megfelelő metódussal (**delete**) próbáljuk elérni a törlés útvonalát.

Assert	Ellenőrizzük, hogy átirányításra kerültünk-e a kategóriák index oldalára. Majd azt is megvizsgáljuk, hogy az adatbázisból *„hiányzik-e”,* tehát nincs benne az adott kategória. Végül a **categories** adattábla sorainak számát ellenőrizzük és nullának kell lennie.

7–6. táblázat: 3A sablon használatára példa: delete

7.6.2.7. Konkrét tesztesetek végrehajtása

Ha valamilyen konkrét teszteset osztályát vagy metódusát szeretnénk csak lefuttatni, akkor azt a terminal-ban így tehetjük meg például (egy konkrét metódussal):

```
php artisan test --filter=test_category_delete_successful
```

A --filter kapcsoló után írhatunk osztálynevet és metódusnevet is!

7.6.3. Tesztelés az .env.testing beállításai alapján

Ha nem a **phpunit.xml** beállításai alapján szeretnénk tesztelni, akkor kikommentezhetjük a **phpunit.xml** érintett sorait (**DB_CONNECTION**, **DB_DATABASE** és ha van, akkor a további **DB_** előtaggal rendelkező beállításokat).

A szerkezete az adatstruktúráknak ugyanaz lesz továbbra is, mint a fejlesztési / éles adatbázisé. Másoljuk le az **.env** fájlt és hozzunk létre egy új **.env.testing** nevű fájlt a következő módosításokkal:

1. a **DB_CONNECTION**-t írjuk át **sqlite**-ra
2. a **DB_HOST** legyen **127.0.0.1**
3. a **DB_DATABASE**-t változtassuk meg erre: **:memory:**

A további **DB_** előtaggal rendelkező beállítás törölhető, ha vannak, mert azok **mysql** specifikus beállítások.

Ha most elindítjuk a tesztelést, akkor már az **.env.testing** fájl alapján fogja tesztelni az alkalmazásunkat.

Az alfejezetben történt programkód módosításokat ebben a GitHub commit-ben lehet megtalálni.

7.7. Összegzés

A fejezet során áttekintettük a CRUD műveleteket és megnéztük, hogy hogyan támogatja a Laravel keretrendszer egy-egy erőforrás kapcsán a kiolvasást, listázást, létrehozást, szerkesztést és törlést. Megismerkedtünk a CRUD műveletekhez szükséges útvonalakkal és vezérlő metódusokkal, továbbá a szükséges nézeteket is létrehoztuk hozzájuk. Mindezeket úgy, hogy közben a Model-eket nem önmagukban kezeltük, hanem a legfontosabb kapcsolattípusokkal együttesen.

Az erőforrásainkat nem csak webes oldalról kezeltük, hanem API felületen keresztül is lekérdeztük őket, létrehoztunk belőlük példányokat, módosítottuk és töröltük is őket. Ezen feladatok megoldása közben megismerkedtünk mélyebben a Postman alkalmazással, amellyel kiválóan tudunk szimulálni POST, PUT/PATCH, DELETE HTTP metódus szerinti kéréseket anélkül, hogy űrlapot kellett volna létrehoznunk hozzájuk.

A CRUD műveleteket teszteltük: először a phpunit.xml beállításai alapján SQLite-ban és MySQL-ben is, majd végül az **.env.testing** környezeti fájl beállításai alapján.

Feladat: ennek az összetett fejezetnek a végén gyakorlásként érdemes megcsinálni például egy blogbejegyzések kommentjeit tartalmazó **Comment** Model osztályt és minden kapcsolódó elemét (útvonalak, vezérlő metódusok, gyár, seeder, nézetek stb.) úgy, hogy egy **Post**-hoz több **Comment** is tartozhat (1-n).

Tipp az induláshoz: `php artisan make:model Comment -mfcr`

Ellenőrzésként én is létrehozom ezeket a fájlokat és egy GitHub commit-ben itt elérhetővé teszem.

Sőt! Utána érdemes hozzákötni a kommenteket a blogbejegyzésekhez, és azoknál is megjeleníteni őket valamilyen formában (index -> kommentek száma, show -> kommentek tartalma). Ennek is elkészítem a változtatásait, amelyeket ebben a GitHub commit-ben teszek elérhetővé.

8. CRUD útvonalak, funkciók és nézetek integrálása (Integration of CRUD routes, actions and views)

A fejezet során folytatjuk az ismerkedést a CRUD útvonalakkal, funkcionalitásokkal és a nézeteivel. Először az útvonalak regisztrálása során néhány egyedi technikát fogunk alkalmazni. Utána a kissé puritán nézeteinket, amelyeket csak a legminimálisabb design-nal láttunk el az előző fejezetben, most integrálni fogjuk a meglévő sablonunkba (résznézetek és komponensek segítségével). Aztán megvizsgáljuk, hogy a CRUD műveletek milyen automatikusan végrehajtódó eseményeket idéznek elő, mire jók ezek és hogyan tudjuk őket programozottan kezelni. Végül a tesztelési tudásunkat fogjuk mélyíteni úgy, hogy közben még egy új eszközt is használatba veszünk.

8.1. Kód újraszervezés (Refactoring): az útvonalak

Az útvonalak témájának vizsgálatára mindig vissza-visszatérünk, most sem lesz ez másként, ismét bővítjük tudásunkat, amelyeket az útvonalakkal kapcsolatban érdemes ismerni. Ebben az alfejezetben megvizsgáljuk, hogy milyen lehetőségeink vannak még az útvonalak paraméterezésére, illetve a teljesen egyedi, akár több paraméterrel ellátott útvonalak regisztrációjára és a működésük támogatására.

8.1.1. Útvonalak paraméterezése

Több kérdés is felmerülhet bennünk:

- *Milyen paraméterek lehetségesek a wildcard helyeken?*
- *Javasolt-e a használatuk?*
- *Mire kell még figyelnünk, ha használjuk őket?*

A válaszok összegzése pedig irányt mutat nekünk a hatékonyság felé:

- Amit *ne* használjunk: {id}
    - A 7.2. alfejezetben még az id-t használtuk paraméterként, azonban ez magával vonz egy olyan utasítást a vezérlő metódus elején, hogy vagy a find($id) vagy még inkább a findOrFail($id) metódussal le kellett kérnünk az aktuális erőforrás adattáblájának a megfelelő azonosítójú (id) sorát. Például, ha a GET HTTP metódus szerinti /categories/{id} akkor ez a show vezérlő metódust azonosítja, és ennek a legelején rögtön le kellene kérnünk a categories táblából a releváns sort, ami egy felesleges kódsor lenne, hiszen használhatnánk ehelyett a következőt...
- Amikor a 7.3. alfejezetre tértünk át, akkor a modell (Model) nevét angolul egyes számban, kis kezdőbetűvel használtuk az útvonalaknál: {post}, {tag} stb.
    - Ha ezeket használjuk az útvonalaknál, akkor a vezérlő metódusok paraméterében már rögtön írhatjuk is az erőforrás (Model osztály) neve után a példány nevét is (pl.: Post $post), amit utána a metódus magjában már úgy használhatunk, hogy az adott azonosítójú példány adataival tudunk dolgozni. A háttérben rejtetten ez is az id-t használja azonosításra (kikeresésre), hiszen az útvonalban például azt adhatjuk át, hogy 'post/12', akkor ez a 12-es azonosítójú blogbejegyzést fogja jelenteni a program számára.

- Lehetőségünk van még arra is, hogy ne az **id** alapján azonosítsuk az erőforrást, például egy felhasználót a neve vagy az e-mail címe alapján azonosíthatunk *egyedien*, például: **{user:email}**, **{tag:name}**, **{category:name}** stb. De fontos itt hangsúlyozni az utolsó szót, hogy egyedien. Ugyanis, ha olyan paraméter szerint szeretnénk lekérni például egy felhasználót, ami nem egy egyedi értéket tartalmaz, akkor a működés inkonzisztenssé válhat. Adatbázistábla szinten tehát ennek a mezőnek (**users** tábla email attribútuma, vagy az **categories** tábla **name** attribútuma) kulcsnak kell lennie. Ez a táblakényszer biztosítja számunkra, hogy az oszlopban lévő minden érték egyedi lesz, ami által minden adatsor a táblában egyedien azonosítható lesz. *Megjegyzés:* a mi blogos példánkban a **posts** tábla **slug** attribútuma nem kulcs mező, úgyhogy arra nem is feltétlenül igaz az, hogy egyedileg azonosítana minden egyes sort a táblában. Próbáljuk is ki emiatt, hogy miként jelentkezik a probléma, amit említettem, utána pedig orvosoljuk ezt úgy, hogy kulccsá alakítjuk a **slug** mezőt.

Gyakorlati szempontból, próbáljuk ki a legutóbbit, mivel erre még nem láttunk konkrét példát.

A **web.php**-ban a posts erőforrás útvonal után szúrjuk be ezt:

```
Route::get('/posts/{post:slug}',[PostController::class, 'show'])
->name('posts.show');
```

8–1. kódrészlet: Blogbejegyzések azonosítása slug alapján

Mindössze ennyi változtatás kell ahhoz, hogy most már **slug** alapján kerüljenek azonosításra a blogbejegyzések, például így: http://127.0.0.1:8000/posts/sit-et (nálam a **posts** adattáblában van *„sit-et”* **slug**-gal rendelkező blogbejegyzés). A **PostController show()** metódusában semmilyen módosításra nincs szükség ahhoz, hogy működjön a blogbejegyzés megtekintése.

De mi volna akkor, ha lenne ugyanolyan slug-gal rendelkező több blogbejegyzés is a posts táblában? Hibát nem fogunk kapni, de ekkor az útvonal lekérése a böngészőben az első találatot adja vissza és az ahhoz tartozó blogbejegyzést tudjuk megtekinteni.

Miután visszaváltoztattuk az azonos **slug**-gal rendelkezőket különbözőkre, akkor következhet az, hogy magát a **slug** mezőt egyedivé tesszük egy új migrációs fájl segítségével:

```
php artisan make:migration add_unique_slug_to_posts_table
```

A migrációs fájl lényegi sorai alább láthatók:

```
// up: kulcs kényszer hozzáadása
$table->string('slug')->unique()->change();
// down: kulcs kényszer elvétele
$table->dropUnique('posts_slug_unique');
```

8–2. kódrészlet: Egyedi kulcs kényszer hozzáadása/elvétele a posts tábla slug mezőjénél

Migrálás után így már alkalmazható a **slug** mező a blogbejegyzések azonosítására. Ha hibát kapnánk a migráláskor, az főleg azért lehet, mert van a **posts** táblában több olyan sor, amelyekben a **slug** mező értéke nem egyedi, tehát azonos. Töröljük ki vagy módosítsuk azokat a sorokat, amelyekben azonos a **slug** mező értéke.

Miért érdemes slug alapján azonosítani a blogbejegyzéseket? A válasz a SEO[10] témakörét érinti, mivel egy **slug** mindig sokkal beszédesebb, mint egy sima szám és a keresőoptimalizálásnál a keresők előrébb rangsorolják a találati listában a blogbejegyzéseinket akkor, ha **slug**-ot használunk az azonosításra.

A teljes logikát viszont most ne alakítsuk át a **slug**-os azonosítás szerint, úgyhogy kommentezzük ki a **web.php**-ban ezt az újonnan hozzáadott útvonalat. Az alfejezetben bemutatott változások ebben a GitHub commit-ben érhetők el.

8.1.2. Teljesen egyedi útvonalak regisztrálása

Sose feledjük el, hogy a Laravel, hatalmas szabadságot ad nekünk. Így nem csak a CRUD-os útvonalakat tudjuk használni, hanem gyakorlatilag bármilyen útvonalat tudunk regisztrálni a rendszerbe. Ebben az alfejezetben összegyűjtöttem néhány példát arra vonatkozóan, hogy a blogos projektünkben milyen egyedi útvonalakra lehet szükségünk és azoknak az üzleti logikáját, nézetét hogyan valósítsuk meg.

Ezeket az egyedi útvonalakat a szűrés témájára fogjuk felépíteni. A blogbejegyzéseket fogjuk szűrni kategóriák szerint és publikálás dátumát tekintve egy kezdő- és végdátumra.

Ezzel az útvonallal lehetne lekérni az adott kategóriába tartozó blogbejegyzéseket egy adott publikálási dátum intervallumon belül.

```
Route::get('/posts/{category:name}/{from}/{to}', [PostController::class,
'getCategoryPostsFromTo']);
```

8–3. kódrészlet: Blogbejegyzéseket szűrő útvonal regisztrációja

Ekkor a felhasználó így hívhatná meg ezt az útvonalat manuálisan (esetleg egy form input mezőiből felparaméterezve):

http://127.0.0.1:8000/posts/deleniti/2023-06-01/2023-06-14

Így a felhasználó visszakapná a *„deleniti”* (random szöveg) kategóriába tartozó blogbejegyzéseket, amelyeket 2023. június 1. és 14. között publikáltak (az alsó korlát még beletartozik, a felső már nem).

Az útvonal lekérésének feldolgozási oldalán a **PostController** felé irányítjuk a felhasználót, ahogy azt tettük a RESTful metódusok esetében is. Viszont ne konkrétan ott oldjuk meg a kérés feldolgozását a **getCategorysPostsFromTo()** metódusban, hanem hívjuk segítségül az **üzleti logikát** biztosító Model osztályt, ami nem csak az adatkapcsolatok menedzseléséért felel, hanem ide érdemes definiálni azokat a metódusokat is, amelyek a **Post** osztállyal, objektumaival kapcsolatosak. A **getCategorysPostsFromTo()** metódus például így nézhet ki a **PostController**-ben:

```
public function getCategoryPostsFromTo($category, $from, $to)
{
  return Post::getFilteredPosts($category, $from, $to);
}
```

8–4. kódrészlet: Szűrt blogbejegyzések lekérése a PostController-ben

[10] Search Engine Optimization

A **Post** Model osztályban pedig egy újrafelhasználható és jól paraméterezhető metódussal tudjuk lekérni a szűrt blogbejegyzéseket. Itt már sokkal jobban alkalmazható (vagyis számomra átláthatóbb) a Query Builder-ben írt lekérdezés, főleg a **join()** miatt. A Query Builder megvéd minket a kódbeszúrásos támadások ellen is, emiatt itt most nem kell aggódnunk.

```
public static function getFilteredPosts($category, $from, $to)
{
  return DB::table('posts')
    ->select('title', 'slug', 'body', 'published_at', 'name')
    ->join('categories', 'posts.category_id', 'categories.id')
    ->where('name', $category)
    ->whereBetween('published_at', [$from, $to])
    ->get();
}
```

8–5. kódrészlet: Paraméterezett szűrés a blogbejegyzések eredménylistájára

(Ne felejtsük el importálni hozzá a DB facade-ot!)

A böngészőben a lekért útvonal (http://127.0.0.1:8000/posts/deleniti/2023-06-01/2023-06-14) eredményül ad egy JSON listát, amelyben a lekért blogbejegyzések mezői vannak. *Megjegyzés:* nálatok biztos, hogy más kategória nevet és dátum intervallumot kell megadni, hogy ne egy üres eredményhalmazt kapjatok vissza.

8.1.2.1. Opcionális útvonal paraméter

Előfordulhat, hogy az útvonalnál szeretnénk használni opcionális paramétert, például arra, ha a példánknál maradunk, hogy a publikálás dátumára szűrünk, de csak a kezdődátumot tudjuk megadni, a végdátumot nem (ekkor a végdátum valószínűsíthetően a mai nap dátuma lehetne). Módosítsuk a 8–3. kódrészletet úgy, hogy a **{to}** paramétert opcionálissá tesszük egy kérdőjel segítségével: **{to?}**

Ezután a **PostController getCategorysPostsFromTo()** metódusának harmadik paraméteréhez is hozzá kell adnunk kezdőértéket (tegyük egyenlővé null-lal), hiszen nem biztos, hogy a felhasználó meg fogja adni ennek értékét, mivel opcionálissá tettük.

Ezután a **Post** Model **getFilteredPosts()** metódusának paraméterlistájában is a harmadik **$to** paramétert opcionálissá kell tennünk, ugyanúgy, ahogy a **PostController**-ben tettük, illetve egy feltételvizsgálattal utána értéket adunk neki, hogy a programlogika utána is ugyanúgy működjön, ha a felhasználó nem adott meg végdátumot a publikálás szűrésére.

```
public static function getFilteredPosts($category, $from, $to = null)
{
  if($to == null) $to = Carbon::tomorrow();
  return …
```

8–6. kódrészlet: Post Model getFilteredPosts() metódusának bővítése az opcionális paraméteres működéshez

Ha nem kapott értéket a **$to** változó az útvonalon keresztül a felhasználótól, akkor a holnapi napot állítjuk be a végdátumnak. Azért a holnapot, mert a dátum intervallum felső határa már nem tartozik bele a

szűrésbe a **whereBetween()**-en belül, így a mai nap publikált blogbejegyzéseket még visszaadja ez a szűrés.

A módosítás hatására az útvonalak így már akkor is működnek, ha van vagy ha nincsen végdátum meghatározva bennük. Példák:

- http://127.0.0.1:8000/posts/deleniti/2023-06-01/2023-06-14
- http://127.0.0.1:8000/posts/deleniti/2023-06-11/

Az egyedi útvonalak és az opcionális paramétereket érintő programkód módosítások elérhetők ebben a GitHub commit-ben.

Tipp: a **published_at** (vagy bármilyen más) mezőre építhetünk index fát egy új migrációs fájl segítségével, ha tudjuk, hogy sokszor szeretnénk szűrni rá, ezáltal gyorsabbak lesznek az erre vonatkozó lekérdezéseink.

8.2. CRUD nézetek integrálása a sablonba

Korábbi alfejezetekben (7.2., 7.3., 7.4.) megvalósítottuk kapcsolattípusok szerint csoportosítva a 7 RESTful útvonalat és a hozzájuk kapcsolódó vezérlő metódusokat, kiegészítve a megjelenítéseikkel. Ebben az alfejezetben a nézeteket integráljuk bele a *„Future Imperfect"* sablonunkba, amelyet a 4.5. alfejezetben telepítettük a Laravel projektünkhöz.

8.2.1. Meglévő CRUD nézetek keretbe foglalása

Használjunk egy keretes szerkezetet, ahova betölthető a kezdőoldal fő része (hármasával lapozható blogbejegyzések), a főmenü, és az erőforrások nézetei. Mivel maga a sablon angol, ezért a szükséges módosításokat megtesszük majd a nézetekben, de mindig jelezni fogom magyarul is, hogy mi micsoda.

8.2.1.1. Keretes szerkezet

Egyelőre ne komponens alapon szervezzük meg a webalkalmazásunk kinézetét, hanem keretes szerkezetbe helyezzük majd bele a dinamikusan változó tartalmakat.

Másoljuk át a **components / layout.blade.php** tartalmát egy új **app.blade.php** fájlba. Majd töröljük ki a **main id**-val rendelkező **div**-ben lévő három post osztályú **<article>** tag-et belőle és csak ennyi maradjon az érintett main div-en belül:

```
<!-- Main -->
<div id="main">
  @yield('main')
</div>
```

8–7. kódrészlet: Keretes szerkezetben a dinamikusan változó tartalom helyőrzője

Így tehát a *„main"* helyőrzőbe tudjuk majd betölteni a tartalmakat úgy, mint a kezdőoldalon korábban látható három (statikus) blogbejegyzést (immár dinamikusan majd) és a többi erőforrásunkat (kategóriák, címkék, blogbejegyzések stb.).

8.2.1.2. Kezdőoldal fő része: lapozható blogbejegyzések

A kezdőoldalon három darab blogbejegyzés volt látható, alattuk pedig egy lapozás. Ez gyakorlatilag azt jelenti, hogy a blogbejegyzések listázását meg kell valósítani a lapozással együtt.

Ehhez először a kezdőoldalhoz vezető útvonalat módosítsuk az alábbi szerint:

```
Route::get('/', function () {
  return view('home', [
    'posts' => Post::paginate(3),
  ]);
})->name('home');
```

8–8. kódrészlet: Kezdőoldal módosított útvonala a blogbejegyzések hozzáadásával

Megjegyzés: ha már nagyon sok blogbejegyzésünk van, akkor érdemes lehet használni a **paginate()** helyett a **simplePaginet()** metódust, ez ugyanis nem számolja meg, hogy összesen mennyi blogbejegyzés van (ezt megspórolja), hanem csak legenerálja a *„következő"* és *„előző"* linkeket.

A *„home"* nézet viszont még nem létezik, úgyhogy ezt hozzuk létre. A **home.blade.php** tartalma pedig ez legyen:

```
@extends('app')
@section('main')
@foreach ($posts as $post)
  @include('includes._post')
@endforeach
{!! $posts->links() !!}
@endsection
```

8–9. kódrészlet: Home nézetben kilistázzuk a három blogbejegyzést lapozással együtt

A blogbejegyzések részleteit a kezdőoldalon egy résznézet segítségével jelenítjük meg (itt az útvonaltól mindig három elemű **$posts** gyűjteményt kapunk, amiben 1-1 **$post** objektum részleteit mutatjuk meg). Ezt a résznézet fájlt hozzuk létre az **includes** mappába **_post.blade.php** névvel. A tartalma pedig kezdetben lehet a **components / layout.blade.php** fájlban lévő egy darab post osztályú **<article>** tag és belső tartalma. Utána ezt a statikus tartalmat tegyük dinamikussá úgy, hogy a mi **$post** objektumunk különböző tulajdonságait kérjük le és helyezzük el a nézetben. Értelemszerűen megjeleníthetjük benne a következőket:

- blogbejegyzés címét (linkként, ami a konkrét blogbejegyzés show útvonalára vezet),
- a címkéit (szintén linkként, amelyek az adott **tag show** útvonalára vezetnek),
- a publikálás dátumát (azért, hogy tudjuk dátumként formázni, egy kicsi PHP kódot kell elhelyeznünk a fájlban, így a **$post->published_at** mezőt, ami szöveges, dátummá alakíthatjuk),
- felhasználóhoz még nem kötöttük hozzá a blogbejegyzéseket, de a tartalmi részben látható képet elláthatjuk egy linkkel, ami a blogbejegyzés címéhez hasonlóan a show útvonalra vezeti majd a látogatót,
- helyezzük el a **</header>** és **<footer>** tag-ek közé a blogbejegyzés tartalmát (**body** elemét)
- a **<footer>** részben a *„Continue Reading"* szöveghez ismét adjuk hozzá a linket, ami a blogbejegyzés show útvonalára vezeti a látogatót,

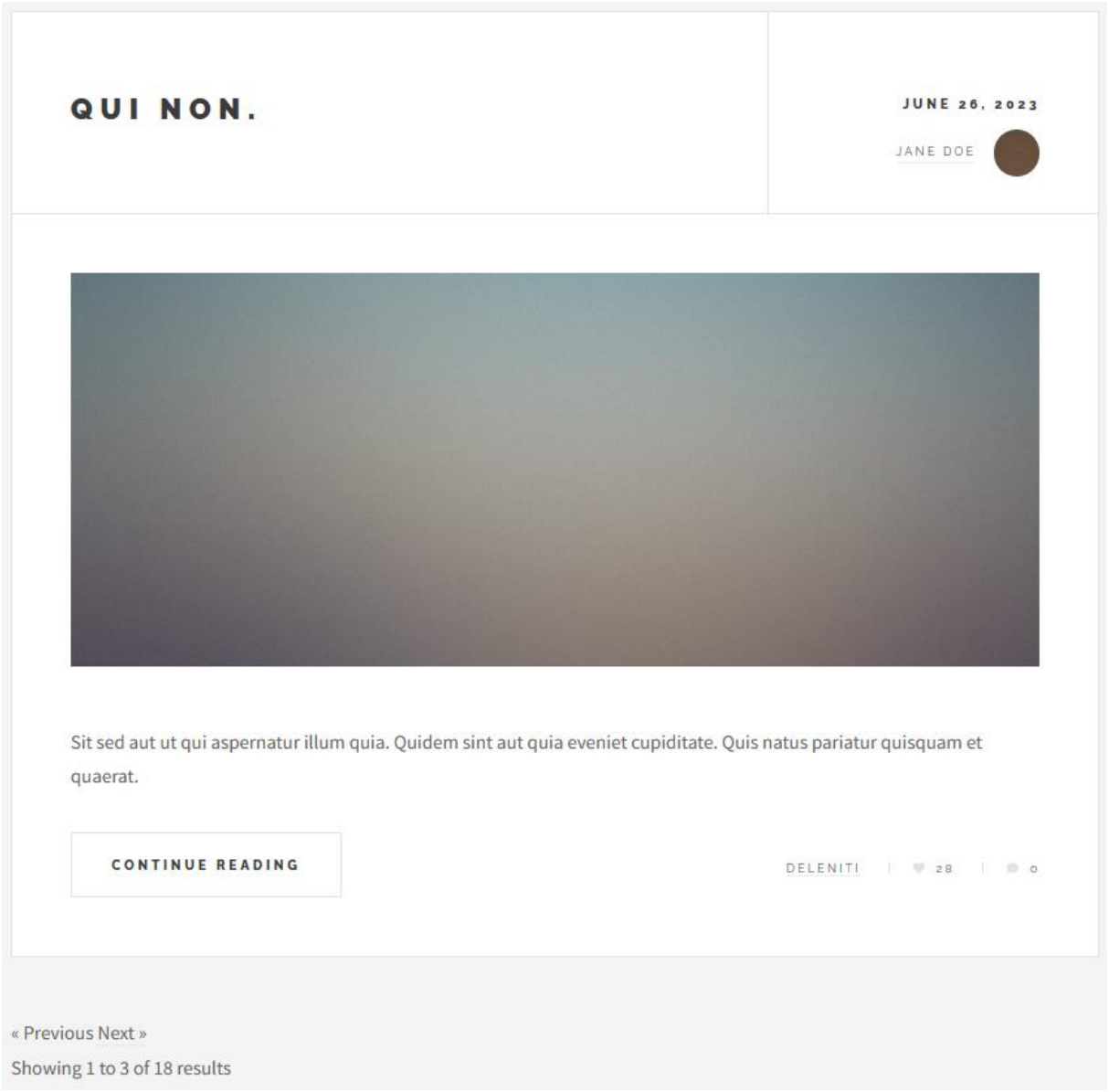
- a lábléc (**footer**) statisztikai részében pedig jelenítsük meg a blogbejegyzés kategóriájának nevét, a kommentek számát, a kedvelések számát (szív melletti számot) hagyhatjuk változatlanul statikusan, mivel arra még nincsen dinamikusan változó tartalmunk.

Az imént felsorolt változtatásokat a leírások alapján lehet megvalósítani, de ha nem menne, akkor az alábbi kódrészletben mutatóm be a változtatásokat:

```
@php
  $published_at = new DateTimeImmutable($post->published_at);
@endphp
<!-- Post -->
<article class="post">
  <header>
    <div class="title">
      <h2><a href="{{ route('posts.show', $post->id) }}">{{ $post->title
}}</a></h2>
      <p>
        @foreach ($post->tags as $tag)
        <a href="{{ route('tags.show', $tag->id) }}">#{{ $tag->name }}</a>
        @endforeach
      </p>
    </div>
    <div class="meta">
      <time class="published" datetime="{{ $published_at->format('Y-m-d')
}}">{{ $published_at->format('F d, Y') }}</time>
      <a href="#" class="author"><span class="name">Jane Doe</span><img
src="{{ Vite::asset('resources/images/avatar.jpg') }}" alt="" /></a>
    </div>
  </header>
  <a href="{{ route('posts.show', $post->id) }}" class="image featured"><img
src="{{ Vite::asset('resources/images/pic01.jpg') }}" alt="" /></a>
  <p>{{ $post->body }}</p>
  <footer>
    <ul class="actions">
      <li><a href="{{ route('posts.show', $post->id) }}" class="button
large">Continue Reading</a></li>
    </ul>
    <ul class="stats">
      <li><a href="#">{{ $post->category->name }}</a></li>
      <li><a href="#" class="icon solid fa-heart">28</a></li>
      <li><a href="#" class="icon solid fa-comment">{{ $post->comments-
>count() }}</a></li>
    </ul>
  </footer>
</article>
```

8–10. kódrészlet: includes / _post.blade.php résznézet tartalma

A blogbejegyzés részleteit dinamikusan kezelő résznézet így elkészült. A kezdőoldal (http://127.0.0.1:8000/) elérésével látható is a tartalma.



8–1. ábra: A harmadik blogbejegyzés a kezdőoldalon (random generált adatokkal: cím, dátum, tartalom, kategória, kommentek száma), alatta az automatikusan generált lapozással

Már a *„résznézetek"* elgondolás is a komponensek használatának irányába indul el, azonban ezek még kevésbé újrafelhasználhatók és testre szabhatók, emiatt érdemesebb lesz majd a későbbiekben (8.2.2. alfejezet) a komponensek használata.

Magát a lapozást a Laravel automatikusan generálja nekünk, nem tudjuk így a stílusát és a szövegezését egyszerűen megváltoztatni, úgyhogy ezt az automatikus generálást le kell cserélnünk. A Laravel keretrendszer dokumentációja hosszan részletezi a lapozásnak a háttérlogikáját, illetve a testre szabási lehetőségeit, úgyhogy ezt fel fogjuk használni ahhoz, hogy a saját (korábban a sablonban látott lapozási kinézetet létrehozzuk a kezdőoldalon látható három blogbejegyzés alatt).

A sablonban mindössze két gomb van a lapozásra: *„Previous Page"* (előző oldal) és *„Next Page"* (következő oldal) feliratokkal. Előbbi nyilván a legelső oldalon legyen inaktív, míg utóbbi a legutolsó oldalon legyen inaktív, hiszen ezekről a helyekről nem tudunk már előrébb, illetve hátrébb lapozni. Az érintett **home.blade.php** nézetben kommentezzük ki az automatikusan generálódó lapozást, és illesszük be a saját logikánk szerinti lapozást!

```
{{-- {!! $posts->links() !!} --}}
<!-- Pagination -->
<ul class="actions pagination">
  @if ($posts->onFirstPage())
    <li><a href="" class="disabled button large previous">Previous
Page</a></li>
  @else
    <li><a href="{{ $posts->previousPageUrl() }}" class="button large
previous">Previous Page</a></li>
  @endif

  @if ($posts->hasMorePages())
    <li><a href="{{ $posts->nextPageUrl() }}" class="button large next">Next
Page</a></li>
  @else
    <li><a href="" class="disabled button large next">Next Page</a></li>
  @endif
</ul>
```

8–11. kódrészlet: Saját, testreszabott lapozás integrálása a kezdőoldalba

A testre szabáshoz a lapozás segédmetódusait használtuk így fel, ami a Laravel dokumentációjában is elérhető. A teljesség igénye nélkül, elérhetők olyan funkcionalitások a lapozáshoz, mint például hány elem van egy oldalon összesen (**count()**), melyik oldalon áll most a lapozás (**currentPage()**), és a 8–11. kódrészletben is látható feltételvizsgálatok (első oldalon állunk-e – **onFirstPage()**, van-e még további oldal – **hasMorePages()**) és az előző/következő oldalra navigáló linkek (**previousPageUrl(), nextPageUrl()**) stb.

Jelenleg a lapozásnál nem látható, hogy összesen hány oldal van és hányadik oldalon állunk éppen, de ezt is könnyedén hozzá tudjuk adni a metódushívások alkalmazásával, az @endsection elé illesszük be az alábbi kódrészletet:

```
<div>Showing {{ $posts->currentPage() }} to {{ $posts->lastPage() }} of {{
$posts->total() }}</div>
```

8–12. kódrészlet: A lapozásban az aktuális oldal száma, az összes oldal száma és az összes elem száma, amiben lapozhatunk

Végül még egy dolgot csináljunk meg a lapozásban: a két (előre és hátra) lapozó gomb közé (**<ul>** tag közepére) soroljuk fel az oldalszámokat linkekkel együtt, hogy egyből oda tudjon navigálni a felhasználó. Az éppen aktuális oldal gombját pedig tegyük inaktívvá:

```
@for ($page = 1; $page <= $posts->lastPage(); $page++)
  <li><a href="{{ $posts->url($page) }}"
class="@if($page == $posts->currentPage()) disabled @endif button large">
{{ $page }}</a></li>
```

```
@endfor
```

8–13. kódrészlet: Oldalszám gombok megjelenítése a lapozásnál (aktuális oldal inaktív)

Az iménti kódok végrehajtásának eredménye itt látható a lapozásnál:

8–2. ábra: Testre szabott lapozás a kezdőlapon a 3 blogbejegyzés alatt

Megjegyzés: a **disabled** HTML attribútum kapcsán is működik már az, amit a **@selected** Blade direktívánál láthattunk a 7.4.5.1. alfejezetben. Így nincs szükség **@if-@endif** párosra, hanem csak egy **@disabled()** direktívában definiáljuk a feltételt, és oda fogja generálni a HTML kódba a **disabled** szöveget az elemhez. Ez ugyanígy működik már a **@required()** kapcsán is a Laravel 10-ben.

8.2.1.3. Fejléc

A fejlécben lévő horizontális menüt az **includes / _header.blade.php** fájlban tudjuk szerkeszteni. A felsorlásban a linkek átírása a meglévő útvonalakban nem okozhat már nekünk problémát. A módosított linkek így alakulnak (a fájl eleje látszódik itt):

```
<header id="header">
  <h1><a href="{{ route('home') }}">Future Imperfect</a></h1>
  <nav class="links">
    <ul>
      <li><a href="{{ route('categories.index') }}">Categories</a></li>
      <li><a href="{{ route('tags.index') }}">Tags</a></li>
      <li><a href="{{ route('posts.index') }}">Posts</a></li>
```

8–14. kódrészlet: Módosított menüpontok a fenti vízszintes menüben

8.2.1.4. Sablon stílusa

A kezdőoldalon látható három blogbejegyzés *„kártyás”* kinézete (fehér felület a bézs háttértől egy vékony kerettel elválasztva, lásd: 8–1. ábra) a többi erőforrás nézeteihez tökéletesen megfelelő lesz. Az említett stílusbeli formázásokat a **resources / css / main.css** fájl tartalmazza (az **article .post** osztálya a **main.css** ~2736 sora környékén található meg). A célunk az lehet, hogy a kategóriák és címkék nézeteit is ilyen módon jelenítsük meg az oldalon. Az viszont nem hatékony (és hibázásra is több lehetőséget ad), ha itt a **main.css** fájlban kezdjük el a **.post** osztályok után felsorolni a **.category, .tags, .comment** osztályokat, mivel ezek több más stílusszabály halmaznál is mindig oda kellene írni (a **post header** része, a **header title** része stb.).

Az egyszerű megoldást a CSS előfeldolgozó adja nekünk, mivel a sablon egy ilyen fájlhalmazt is biztosít a számunkra (**resources / sass** mappában). Ezt már a 4.5.3.4. alfejezetben is működésre bírtuk, viszont akkor utána maradtunk a **resources / css / main.css** fájlnál, most viszont a **resources / sass / main.scss** fájlra lesz szükségünk.

A váltáshoz két dologra van szükségünk: először a **vite.config.js** fájlban az input tömb tartalmában írjuk át a **css / main.css** fájlt **sass / main.scss** fájlra. Az **app.blade.php** fájlban pedig a **@vite** direktívában ugyanígy írjuk át a fájl elérhetőségét.

Ezután már csak egy dologra van szükségünk: a **resources / sass / components** mappában lévő **_post.scss** fájlban a legfőbb (legelső) stílusszabály szelektorját bővítsük ki: **.post, .category, .tag, .comment { … }**

Így az erőforrásoknál már tudjuk alkalmazni ezeket az osztályokat és alosztályaikat. A változások érvényre juttatásához mindenképpen futtatnunk kell a Vite-ot a terminal-ban, ha nem menne: npm run dev

Megjegyzés: mivel így a fájl már nem csak a post osztályról és alrészeiről szól, ezért érdemes lehet a fájlt átnevezni például arra, hogy **_resource.scss**, de akkor a **main.scss**-ben is módosítani kell az importálás elérési útját ennél a fájlnál.

```
@import 'components/resource';
```

8–15. kódrészlet: main.scss-ben az új nevű fájl importálása a post helyett (_ karakterre itt nincs szükség)

8.2.1.5. Kategória, címke, blogbejegyzés, komment index és show nézetei

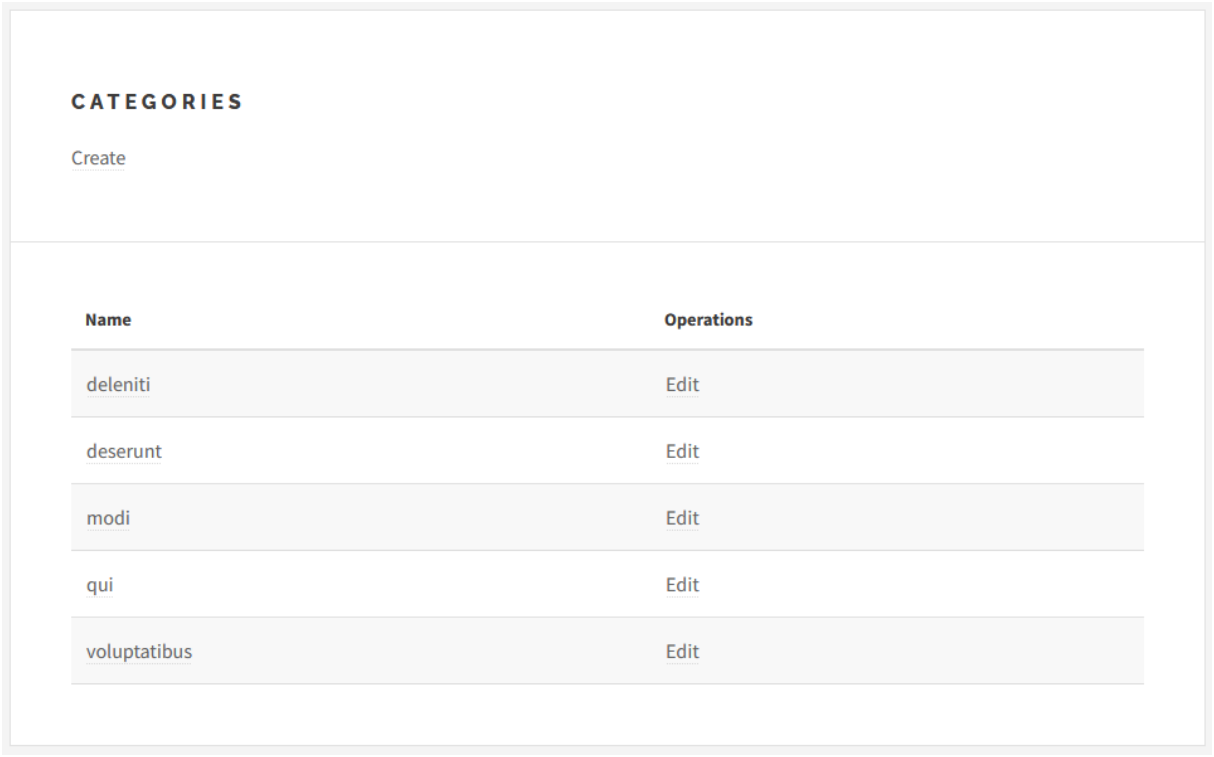
Az erőforrás nézetek átalakítását a kategóriák segítségével mutatom be, de ugyanígy alkalmazható ez majd a címkékre, blogbejegyzésekre, kommentekre és bármely más erőforrás nézeteire is. Kezdjük a kategóriák index (**resources / views / categories / index.blade.php**) nézetével: mindig az **app.blade.php** nézet fájlt kell kiterjeszteni és annak a main szekcióját feltölteni tartalommal.

```
@extends('app')
@section('main')
<article class="category">
  <header>
    <div class="title">
      <h1>Categories</h1>
      <a href="{{ route('categories.create') }}">Create</a>
    </div>
  </header>
  <table>
    <thead>
      <tr>
        <th>Name</th>
        <th>Operations</th>
      </tr>
    </thead>
    <tbody>
    @foreach ($categories as $category)
      <tr>
        <td><a href="{{ route('categories.show', $category->id) }}">{{
$category->name }}</a></td>
        <td><a href="{{ route('categories.edit', $category->id)
}}">Edit</a></td>
      </tr>
    @endforeach
```

```
    </tbody>
  </table>
</article>
@endsection
```

8–16. kódrészlet: Sablonba integrált category.index nézet forráskódja

A *„kártyás”* kinézet pedig így már hozzá is adódik, ha az **<article>** tag-nél beállítottuk a **.category** osztályt. Az eredmény itt látható:



8–3. ábra: Kategóriák index nézete a sablonban

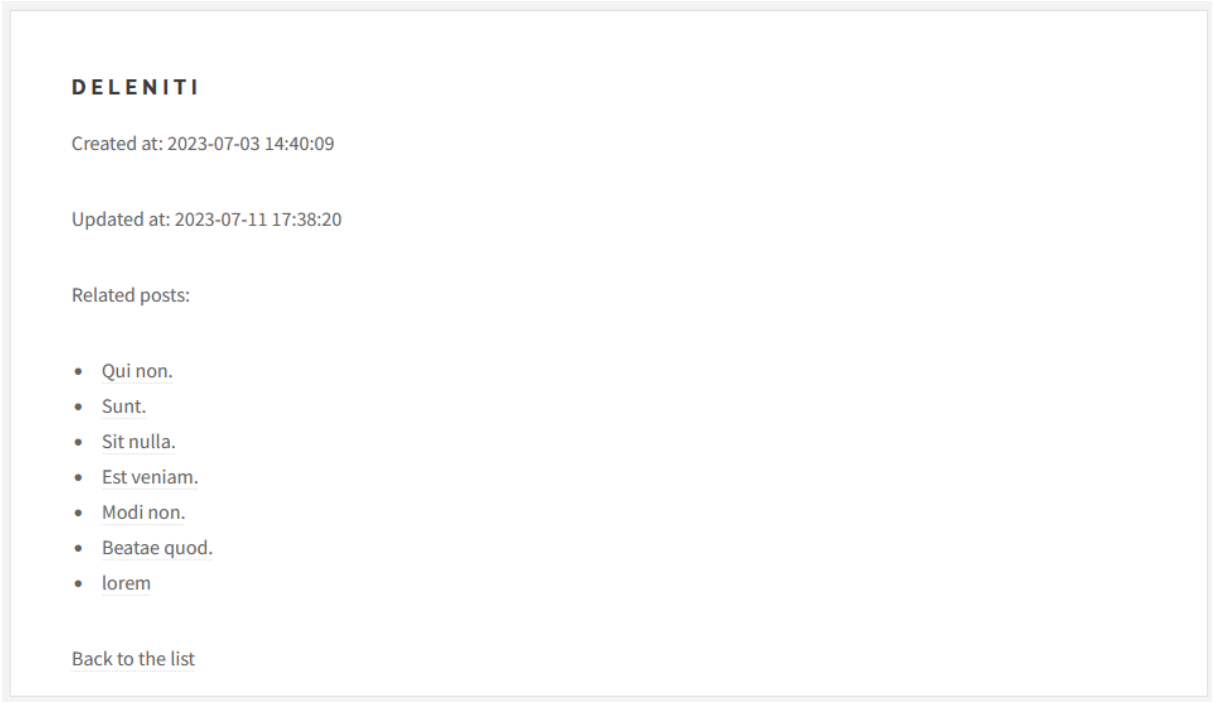
A kategória nevére kattintással a **show** nézetnek kell betöltődnie, amelyen az alábbi átalakítást kell végrehajtani:

```
@extends('app')
@section('main')
<article class="category">
  <h1>{{ $category->name }}</h1>
  <p>Created at: {{ $category->created_at }}</p>
  <p>Updated at: {{ $category->updated_at }}</p>
  <p>Related posts:</p>
  <ul>
  @forelse ($category->posts as $post)
    <li><a href="{{ route('posts.show', $post->id) }}">{{ $post->title
}}</a></li>
  @empty
    <li>There are not any blog posts.</li>
  @endforelse
  </ul>
```

```
  <a href="{{ route('categories.index') }}">Back to the list</a>
</article>
@endsection
```

8–17. kódrészlet: Sablonba integrált categories.show nézet forráskódja

A kiválasztott kategória megtekintésének eredménye itt látható a sablonban:



8–4. ábra: Kategória show nézete a sablonban

Az alfejezet eddig ismertetett programkód módosításait ebben a GitHub commit-ben lehet elérni.

Feladat: ennek az alfejezetnek a végén gyakorlásként érdemes integrálni a sablonba a címkék mellett a további erőforrások (blogbejegyzések, kommentek) **index, show** nézeteit is. Ellenőrzésként én is létrehozom ezeket a fájlokat, és egy GitHub commit-ben itt elérhetővé teszem. Az űrlapokat tartalmazó nézeteket megcsináljuk komponens alapúan a következő alfejezetben.

8.2.2. Komponensek alkalmazása a sablonban

Az alfejezetben a webalkalmazás további részeire koncentrálunk és megvalósítjuk őket anonim komponens alapon (ahogy használtuk őket a 4.4.1. alfejezetben).

8.2.2.1. Menü (hamburger menü)

A jobb felső menü elemei az **includes / _menu.blade.php** fájlban találhatók meg. Komponenseket azért érdemes készíteni, mert néhány paraméterezésen kívül a főbb működés felhúzható egy sablonra. Ha megvizsgáljuk a következő struktúrát (8–5. ábra), akkor az látható, hogy a szekcióban egy felsorolás található és a felsorolás elemei (**<li>**) a leginkább általánosítható menü link elemek.

```
<!-- Links -->
<section>
    <ul class="links">
        <li>
            <a href="#">
                <h3>Lorem ipsum</h3>
                <p>Feugiat tempus veroeros dolor</p>
            </a>
        </li>
        <li>
            <a href="#">
                <h3>Dolor sit amet</h3>
                <p>Sed vitae justo condimentum</p>
            </a>
        </li>
        <li>
            <a href="#">
                <h3>Feugiat veroeros</h3>
                <p>Phasellus sed ultricies mi congue</p>
            </a>
        </li>
        <li>
            <a href="#">
                <h3>Etiam sed consequat</h3>
                <p>Porta lectus amet ultricies</p>
            </a>
        </li>
    </ul>
</section>
```

8–5. ábra: Menüstruktúra forráskódja

A komponens létrehozható a **components** mappában **nav-link.blade.php** névvel. A tartalma legyen ez:

```
<li>
  <a href="{{ $url }}">
    <h3>{{ $title }}</h3>
    <p>{{ $description }}</p>
  </a>
</li>
```

8–18. kódrészlet: Navigációs link komponens sablonja

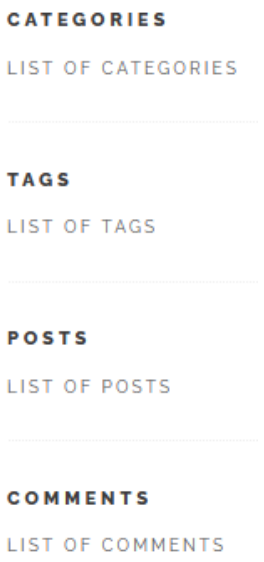
Utána már módosítható a **_menu.blade.php** korábban látott része az alábbiakra:

```
<!-- Links -->
<section>
  <ul class="links">
    <x-nav-link
      url="{{ route('categories.index') }}"
      title="Categories"
      description="List of categories"
```

```
    />
    <x-nav-link
      url="{{ route('tags.index') }}"
      title="Tags"
      description="List of tags"
    />
    <x-nav-link
      url="{{ route('posts.index') }}"
      title="Posts"
      description="List of posts"
    />
    <x-nav-link
      url="{{ route('comments.index') }}"
      title="Comments"
      description="List of comments"
    />
  </ul>
</section>
```

8–19. kódrészlet: Navigációs link komponensek használat a menüben

A menüstruktúra módosított része itt látható:



8–6. ábra: A komponens alapú linkeket tartalmazó menü

Az alfejezetben látható programkód módosítások ebben a GitHub commit-ben érhetők el.

8.2.2.2. Komponens alapú erőforrás űrlapok nézetei

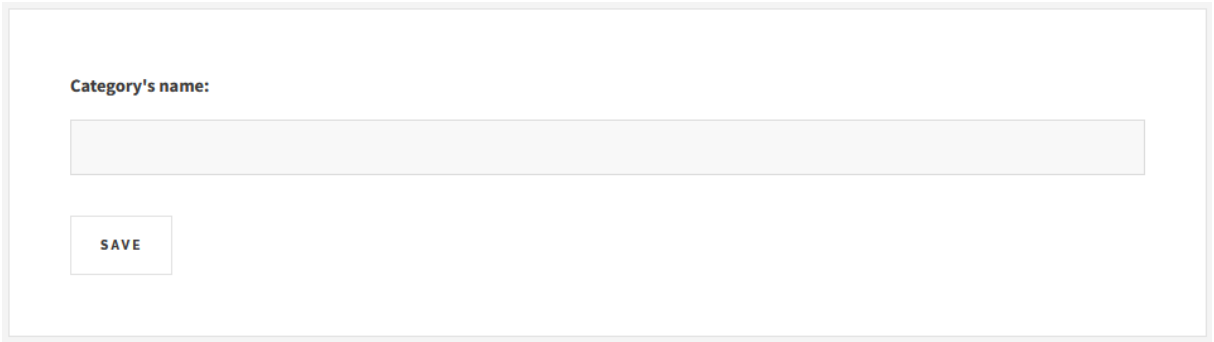
Az erőforrások űrlapokat tartalmazó nézeteinél ugyanúgy maradhatnak a keretes szerkezetet kiterjesztő nézetek, mint az **index** és **show** nézeteknél már használtuk. Maradjunk itt is a kategóriák **create** és **edit** nézeténél és szerkesszük ezeket először! Az első, létrehozást biztosító **create** nézet:

```
@extends('app')
@section('main')
```

```
<article class="category">
  <form action="{{ route('categories.store') }}" method="post">
    @csrf
    <p>
      <label for="nev">Category's name:</label>
      <input type="text" name="name" id="nev">
    </p>
    <input type="submit" value="Save">
  </form>
</article>
@endsection
```

8–20. kódrészlet: Sablonba integrált categories.create nézet forráskódja

A létrehozó űrlap a sablonban itt látható:



8–7. ábra: Új kategóriát létrehozó űrlap a sablonban

A szerkesztési és törlési nézet ehhez képest már csak apróbb módosításokat tartalmaz:

```
@extends('app')
@section('main')
<article class="category">
  <form action="{{ route('categories.update', $category->id) }}"
method="post">
    @csrf
    @method('put')
    <p>
      <label for="nev">Category's name:</label>
      <input type="text" name="name" id="nev" value="{{ $category->name }}">
    </p>
    <input type="submit" value="Update">
  </form>

  <form action="{{ route('categories.destroy', $category->id) }}"
method="post">
    @csrf
    @method('delete')
    <input type="submit" value="Delete">
  </form>
</article>
```

```
@endsection
```

8–21. kódrészlet: Sablonba integrált categories.edit nézet forráskódja

A funkcionalitást (üzleti logikát) nem érintették ezek a változtatások, így azoknak továbbra is működniük kell, de érdemes tesztelni a kategóriáknál ezt (megtekintés – lista, egyéni, létrehozás, módosítás, törlés).

Ennél az állapotnál viszont lépjünk tovább és készítsünk űrlap komponenst is, amely újra és újra felhasználható lesz a számunkra, ehhez pedig vegyük alapul a kategória **create** nézetében lévő űrlapot.

8.2.2.2.1. Űrlap (form) komponens

Hozzunk létre a **resources / views / components** mappában egy **form.blade.php** fájlt. Amikor egy űrlapot szeretnénk általánosítani, akkor tudjuk, hogy a **<form>** nyitó tag-ben kell lennie egy **method** és egy **action** attribútumnak. A **method**-ot alapértelmezetten állíthatjuk POST-ra, hiszen minden olyan esetben, amikor létrehozni, módosítani vagy törölni szeretnénk, akkor a **method**-nak POST-nak kell lennie és most csak az űrlap komponens univerzitása miatt készülünk fel arra az eshetőségre is, amikor esetleg GET metódussal kellene elküldeni az űrlap mezőinek értékeit. A második fontos paraméternek, az **action**-nek az alapértelmezett értékét egy üres string-re ("") állítjuk, mivel ez lesz érvényes bármely esetben, aztán a legtöbb esetben úgyis egy útvonal értéket kap meg ez az attribútum értékül. A két működés szempontjából fontos attribútumon kívül, érdemes még felkészíteni az űrlap komponenst a kinézetet (vagy dinamikus működést) befolyásoló attribútum (**id, class, style** stb.) használatára is. A leírtak alapján a **form.blade.php** komponens első fele így néz ki:

```
@props([
  'method' => 'POST',
  'action' => ''
])

<form
  method="{{ $method === 'GET' ? 'GET' : 'POST' }}"
  action="{{ $action }}"
  {{ $attributes }}
>
```

8–22. kódrészlet: Űrlap komponens első fele

Az űrlap magjában a **@csrf** direktívára a token generálása miatt mindenképpen szükségünk van. A **@method** direktívára pedig akkor van szükség, amikor frissíteni vagy törölni szeretnénk majd, tehát a kapott **$method** változó értéke se nem GET, se nem POST. Végül a komponens a kapott belső tartalmakat (input mezőket) simán elhelyezheti a magjában a záró **</form>** tag előtt.

```
  @csrf
  @if ( ! in_array($method, ['GET', 'POST']))
    @method($method)
  @endif
  {{ $slot }}
</form>
```

8–23. kódrészlet: Űrlap komponens második fele

A komponens felhasználásához a **categories / create.blade.php** nézetet alakítsuk át, főleg a **<form>** nyitó tag-gel lesz itt feladatunk:

```
<x-form
  method="post"
  action="{{ route('categories.store') }}"
  style="background-color: lightgray">
  <p>
    <label for="nev">Category's name:</label>
    <input type="text" name="name" id="nev">
  </p>
  <input type="submit" value="Save">
</x-form>
```

8–24. kódrészlet: Kategóriát létrehozó nézet átalakítása komponens alapúra

A nyitó tag-en kívül a **@csrf token**-t is eltávolítottuk ebből, hiszen azt a komponens adja majd neki. A **style** attribútumot pedig csak azért adtam itt hozzá, hogy teszteljem a működését a komponens ezen részének. A helyes működés megtapasztalása után kivehető a **<form>** nyitó tag-ből a **style** attribútum.

Az **edit** nézetben lévő űrlapok így alakíthatók át **form** komponens alapúra:

```
<x-form action="{{ route('categories.update', $category->id) }}"
method="put">
  <p>
    <label for="nev">Category's name:</label>
    <input type="text" name="name" id="nev" value="{{ $category->name }}">
  </p>
  <input type="submit" value="Update">
</x-form>

<x-form action="{{ route('categories.destroy', $category->id) }}"
method="delete">
  <input type="submit" value="Delete">
</x-form>
```

8–25. kódrészlet: Kategóriát frissítő/törlő űrlapok komponens alapon

8.2.2.2.2. Űrlap és gomb (form, input submit) komponens

Megjegyzés: az alfejezetben a **PostController**-en és az érintett nézetein túl a **sass / components / _form_scss** fájlt is szerkeszteni kell, mivel a **multiple <select> kijelölt opcióinak** a háttere is fehér volt alapból, ami nem segítette a felhasználót egészen addig, amíg rá nem kattintott a mezőre (meg nem kapta a fókuszt). Így inkább csak azoknak az opcióknak a szöveg- és háttérszínét változtassuk meg fehérre, amelyek nincsenek kijelölve (az **option** után a **:not(:checked)** pszeudo-osztály szelektor kiegészítéssel).

```
option:not(:checked) {
  color: _palette(fg-bold);
  background: _palette(bg);
}
```

8–26. kódrészlet: Eltérő háttérszín azoknak a select-ben lévő opcióknak, amelyek nincsenek kijelölve

Ahogy azt láttuk, az **edit** nézetben két űrlapunk is van egy-egy küldést eredményező gombbal. Foglalkozzuk most a törléssel, mivel az az űrlap csak egyetlen gombot tartalmaz. Hozzunk létre egy új komponenst a **components** mappába **form-button.blade.php** névvel. A tartalma összeállításakor szintén a korábban létrehozott űrlap komponenst használjuk, így egymásba ágyazott komponenseink lesznek.

```
<x-form method="{{ $method }}" action="{{ $action }}">
  <input type="submit" value="{{ $slot }}" {{ $attributes }}>
</x-form>
```

8–27. kódrészlet: Egyetlen gombot tartalmazó űrlap komponens alapon

A **form** komponenst felhasználva átadjuk neki a **$method** és **$action** változókat, belül pedig a **$slot** változó segítségével kiírjuk a gomb feliratát, esetlegesen az **$attributes** alkalmazásával stílusinformációkat is átadhatunk a gombnak.

Alkalmazzuk ezt a komponenst a kategória törlésénél:

```
<x-form-button action="{{ route('categories.destroy', $category->id) }}"
method="delete" style="background-color: red">
  Delete
</x-form-button>
```

8–28. kódrészlet: Egymásba ágyazott űrlap és gomb komponens alkalmazása

A **style** attribútum itt is elhagyható, illetve cserélhető **id**-ra vagy **class**-ra, hogy esetleg a felhasznált CSS keretrendszer támogatásával történhessen meg a stílus átalakítása.

8.2.2.2.3. Lenyíló lista (select, option) komponens

A lenyíló lista komponens elkészítéséhez használjuk a blogbejegyzések létrehozó (**posts.create** nézetben lévő) űrlapját. Ezt is, akárcsak a többi nézetet majd, alakítsuk át olyanra, hogy beilleszkedjen a sablonunk main szekciójába, használjuk hozzá továbbá a form komponenst és a feliratokat írjuk át angolra.

A blogbejegyzés kiválasztott kategóriájának lenyíló listáját alakítjuk át komponens alapúra. De előtte optimalizáljunk a kódon, mert a **PostController create()** vezérlő metódusban a kategóriáknál egy kicsit több adat kerül lekérésre az adattáblájából és továbbküldésre a nézetnek, mint amennyire alapból szükség volna, hiszen ott csak a **categories** táblából kiválasztott **id** és **name** mezőkre van szükség, úgyhogy módosítsuk a nézetnek való adatátadást (a címkék kapcsán is):

```
public function create()
{
  return view('posts.create', [
    'categories' => Category::orderBy('name')->get(['id', 'name'])
->pluck('name', 'id'),
    'tags' => Tag::orderBy('name')->get(['id', 'name'])->pluck('name',
'id'),
  ]);
}
```

8–29. kódrészlet: Rendezett, szűrt és transzformált kategóriák átadása a posts.create nézetnek

Ha valamiben nem vagyunk biztosak, használjuk megnyugtatásunkra és a megbizonyosodásra a Tinker-t.

```
App\Models\Category::orderBy('name')->get(['id', 'name']);

App\Models\Category::orderBy('name')->get(['id', 'name'])->pluck('id', 'name');

App\Models\Category::orderBy('name')->get(['id', 'name'])->pluck('name', 'id');
```

8–1. utasítások: Rendezett, szűrt és transzformált kategóriák lekérése

Az első utasítás eredménye: mezőkre szűrt gyűjtemény a modellekkel.

```
> App\Models\Category::orderBy('name')->get(['id', 'name'])
= Illuminate\Database\Eloquent\Collection {#7235
    all: [
      App\Models\Category {#7237
        id: 4,
        name: "deleniti",
      },
      App\Models\Category {#7238
        id: 1,
        name: "deserunt",
      },
      App\Models\Category {#7239
        id: 2,
        name: "modi",
      },
      App\Models\Category {#7240
        id: 6,
        name: "provident",
      },
      App\Models\Category {#7241
        id: 5,
        name: "qui",
      },
      App\Models\Category {#7242
        id: 7,
        name: "sunt",
      },
      App\Models\Category {#7243
        id: 3,
        name: "voluptatibus",
      },
    ],
  }
```

8–8. ábra: Összes kategória mezőinek szűrése egy gyűjteményben

A második utasítás eredménye: gyűjtemény transzformálása asszociatív tömbbé (a kulcsok a kategóriák nevei, az értékek a kategóriák azonosítói).

```
> App\Models\Category::orderBy('name')->get(['id', 'name'])->pluck('id', 'name')
= Illuminate\Support\Collection {#7250
    all: [
      "deleniti" => 4,
      "deserunt" => 1,
      "modi" => 2,
      "provident" => 6,
      "qui" => 5,
      "sunt" => 7,
      "voluptatibus" => 3,
    ],
  }
```

8–9. ábra: Transzformált kategóriák (name => id) gyűjteménye

A harmadik utasítás eredménye: gyűjtemény transzformálása asszociatív tömbbé (kulcsok a kategóriák azonosítói, értékek a kategóriák nevei).

```
> App\Models\Category::orderBy('name')->get(['id', 'name'])->pluck('name', 'id')
= Illuminate\Support\Collection {#7234
    all: [
      4 => "deleniti",
      1 => "deserunt",
      2 => "modi",
      6 => "provident",
      5 => "qui",
      7 => "sunt",
      3 => "voluptatibus",
    ],
  }
```

8–10. ábra: Transzformált kategóriák (id => name) gyűjteménye

A **pluck()** segédmetódussal tudjuk a kategória objektumok mezőit ({id, name}) átalakítani egyetlen tömbbé (**[id => name]**), amelyet így a nézetben a **select option** értékei helyesen tudnak megkapni (a **pluck()** metódus paramétereinek sorrendje fontos, hogy így legyen, mert ez a helyes – lásd a 8–10. ábra részleteit). Ha a **pluck()** metódusban az **id** lenne az első paraméter és a **name** a második, akkor a nevekhez rendelné az azonosítószámokat az új gyűjteményben.

Ezután hozzuk létre a lenyíló lista komponens fájlját a **components** mappában: **select.blade.php**

```
<select name="{{ $name }}" id="{{ $id }}" {{ $attributes }}>
@foreach ($options as $key => $value)
  <option value="{{ $key }}">{{ $value }}</option>
@endforeach
</select>
```

8–30. kódrészlet: Lenyíló lista komponens a select nézet fájlban

Az itteni **<select>** nyitó címkében semmi új nincsen, hiszen egyszerűen felhasználjuk majd a kapott **$name**, **$id** változók értékeit, az **$attributes** változóval pedig a további paramétereket (**class**, **style** stb.). A **@foreach** egy asszociatív tömböt fog kapni (lásd a 8–10. ábra – **id** azonosítókhoz lesznek rendelve a nevek). Így tehát annyira általánossá alakítottuk az **<option>** tag-eket, hogy mindegy lesz a későbbiekben,

hogy az adott adattáblában az **id**-n kívül a másik mezőt **name**-nek vagy **title**-nek neveztük el, ugyanúgy fog működni velük. A **posts.create** nézetben így alakítsuk át a korábbi kategóriákat tartalmazó **<select>**-ünket:

```
<x-select name="category_id" id="category_id" :options="$categories" />
```

8–31. kódrészlet: Kategóriákat listázó komponens a posts.create nézetben

Az **$attributes** változó még egy másik szempontból is nagyon hasznos lesz itt nekünk, mivel általánossá tudjuk tenni a többszörös kiválasztást is. A megoldás emiatt működik a címkéknél is, mivel azt csak egy **multiple** kulcsszóval kell kiegészíteni a **<select>** nyitó tag-jében. Sőt, még a megfelelő kinézetet is átadhatjuk neki, például, ha 10 elemet akarunk megjeleníteni a listában, akkor azt így tehetjük meg (a **style**-ban megfogalmazott stílusszabály is kell hozzá!):

```
<x-select name="tags[]" id="tags" :options="$tags" multiple size="10"
style="height: 100%"/>
```

8–32. kódrészlet: Címkéket listázó komponens a posts.create nézetben

Ha azt szeretnénk, hogy ez az általános **select** komponensünk az **edit** nézetekben is működjön, akkor a kiválasztott értékek átadását kell még megoldanunk, de előbb módosítsuk a **PostController**-ünk **edit()** vezérlő metódusát ugyanúgy ahogy a 8–29. kódrészletben tettük a **create()** metódusnál: csak a kategóriák és címkék azonosítóit és neveit adjuk át a nézetnek egy asszociatív tömbben. *Megjegyzés:* amikor azt gondoljuk (olvassuk), hogy valamilyen kódrészletet ismételjünk meg ugyanúgy, mint egy másik helyen, akkor azt a kódsorozatot rögtön érdemes *„kiszervezni"* egy metódusba, majd azt a metódust meghívni az érintett helyeken. A metódusba való kiszervezés mindig az üzleti logikát érinti, így az érintett Model fájlba helyezzük el a metódust. Az alfejezet végén lévő *„Feladat"* megoldásánál már érdemes így cselekedni.

Ezután nyilvánvaló, hogy a **select** komponensünkben kell elhelyezni a kiválasztott (**selected**) opció(k)nak megfelelő értékeket. A blogbejegyzésnek egy kategóriája van, de több címkéje is lehet, emiatt egy kicsit trükkösen kell megfogalmaznunk, amikor ezt a két dolgot általánosítani szeretnénk:

- Amikor egy kiválasztott érték volt lehetséges a blogbejegyzés kategóriáinál, akkor ott egy egyenlőségvizsgálat elegendő volt a **selected** attribútum hozzáadásához (7.3.6. alfejezet).
- Amikor viszont több kiválasztott érték is lehetett, akkor a gyűjteménynél a **contains()** segédmetódust használtuk a releváns elemek meglétének vizsgálatához (7.4.5. alfejezet).

Mivel utóbbi a nagyobb halmaz, ezért az lehet az általánosabb is, amit a **select** komponens forráskódjában tudunk implementálni:

```
<option value="{{ $key }}" @selected( $selectedValues->contains($key) ) >
{{ $value }}</option>
```

8–33. kódrészlet: A select komponensben az option selected attribútumának hozzáadása

Ez így a címkéknél biztosan használható lesz egyszerűen, hiszen korábban is ott működött ugyanígy:

```
<x-select name="tags[]" id="tags" :options="$tags" multiple size="10"
style="height: 100%" :selectedValues="$post->tags->pluck('id')"/>
```

8–34. kódrészlet: Kiválasztott címkék átadása a select komponensnek

Alapból a kiválasztott kategória azonosítója a **$post->category_id** mezőjével kérhető le, viszont ez így nem gyűjtemény, de könnyedén azzá alakítható és így működni fog vele is a **contains()** metódus feltételvizsgálata:

```
<x-select name="category_id" id="category_id" :options="$categories"
:selectedValues="collect($post->category_id)" />
```

8–35. kódrészlet: Kiválasztott kategória átadása egy elemű gyűjteményként a select komponensnek

Ez így már teljesen általánosan működik az **edit** nézetre vonatkozóan! Azért viszont, hogy a **select** komponensünk a **create** nézeteknél is működőképes maradjon (ahol nyilvánvalóan nincsenek még kiválasztott értékek), emiatt állítsuk be a **selectedValues** alapértelmezett értékét egy üres tömbre.

Megjegyzés: ha csak ennek az egy változónak állítottam be az alapértelmezett értékét, akkor hibát adott a rendszer az oldal betöltődésekor *(Undefined variable $name),* pedig a **$name** változónak mindig volt értéke, emiatt minden szükséges/alkalmazott változónak állítottam be alapértelmezett értéket, így már működött az oldal megfelelően. Szúrjuk be ezt a **select.blade.php** fájl legelejére ezt:

```
@props([
  'name' => '',
  'id' => '',
  'options' => [],
  'selectedValues' => collect()
])
```

8–36. kódrészlet: A select komponens változóinak alapértelmezett értékei

Az alfejezetben található programkód módosítások ebben a GitHub commit-ben elérhetőek.

Természetesen elképzelhető még további komponensek alkalmazása is ezen a ponton, de azok implementálását már az eddig tanultak és a bemutatott példák alapján az Olvasóra bízom. A komponensekre, ugyanúgy, ahogy például az útvonalakra, mindig vissza fogunk térni a későbbiekben is.

Feladat: ennek az alfejezetnek a végén gyakorlásként mindenképpen érdemes integrálni a sablonba komponensekkel a további erőforrások: címkék és kommentek **create**, **edit** nézeteit is (a kapcsolódó egyéb módosításokkal együtt: Controller-ek, Model-ek). Ellenőrzésként én is módosítom ezeket a fájlokat, és itt egy GitHub commit-ben elérhetővé teszem.

8.3. Laravel Folio nézet oldalai és a CRUD funkciók

A Laravel Folio megoldással már találkoztunk (lásd a 3.5. alfejezetet). Ezzel az eszközzel még inkább le tudjuk rövidíteni, kevesebb szereplőssé tudjuk tenni a felhasználói kérések kiszolgálásának útvonalát.

Tegyük fel, hogy van egy új erőforrásunk project néven. Ehhez olyan oldalakat (pages) készítünk, amelyek képesek alkalmazkodni a CRUD-os gondolkodáshoz, útvonalakhoz, vezérlő metódusokhoz.

8.3.1. Erőforrásokat listázó oldal (index)

Hozzunk létre egy új mappát a **resources / views / pages** mappában **projects** néven, és hozzuk oda létre a **projects.blade.php** nézetünket, ami egyszerűen ugyanaz a magja, mint amit a 3–37. kódrészletben látunk.

Ez az oldal majd megfelelő lesz arra, hogy kvázi az *„index()"* Controller metódus visszatérési oldalaként funkcionáljon. Ezért nevezzük is át a **projects.blade.php**-t **index.blade.php**-ra már az új helyén. A Folio még mindig fel fogja ismerni, hogy ez a nézet oldalunk a **/projects** útvonalon lesz elérhető. Ha kipróbáljuk az oldal elérését (http://127.0.0.1:8000/projects), akkor meg is kapjuk a címsoros *„My Projects"* feliratot.

8.3.2. Folio oldalak integrálása a sablonba

Az integrációhoz jó alap, ha vesszük például a **categories** nézeteket tartalmazó mappát és példaként használjuk a **projects** mappa Folio oldalainak át- és kialakításához, de a fő keretet adó sablon fájljainkhoz (például fejléc és menüstruktúra) is hozzá kell nyúlnunk, és el kell helyeznünk bennük a projektek elérési linkjeit.

A Folio-s oldalak útvonalait is képesek vagyunk elnevezni. Ehhez meg kell nyitnunk a nézet fájlokat és például a **projects / index.blade.php**-ban az első sor elé be kell szúrnunk egy PHP kódrészletet.

```
<?php
  use function Laravel\Folio\name;

  name('projects.index');
?>
```

8–37. kódrészlet: Útvonal elnevezése egy Folio oldalnál (index)

A sablonba integrált projekteket listázó oldal *„majdnem"* teljes kódja itt látható (lásd a kódrészlet alatti két problémának a megoldását):

```
<?php
  use function Laravel\Folio\name;

  name('projects.index');
?>

@extends('app')
@section('main')
  <article class="project">
    <header>
      <div class="title">
        <h1>Projects</h1>
        <a href="{{ route('projects.create') }}">Create</a>
      </div>
    </header>
    <table>
      <thead>
        <tr>
          <th>Title</th>
          <th>Operations</th>
        </tr>
      </thead>
      <tbody>
        @foreach ($projects as $project)
```

```
            <tr>
              <td><a href="{{ route('projects.show', ['project' => $project])
}}">{{ $project->title }}</a></td>
              <td><a href="{{ route('projects.edit', ['project' => $project])
}}">Edit</a></td>
            </tr>
          @endforeach
        </tbody>
      </table>
    </article>
@endsection
```

8–38. kódrészlet: A projects.index Folio nézet oldal tartalma

Két probléma is lesz a megoldással:

1. Hiányzik a nézetnek a **$projects** tömb és elemei, ezeket a fájl elején lévő *„php”* szekcióban hozzá tudjuk adni, amelyet így aztán már fel tud használni a nézet: **$projects = \App\Models\Project::all();**
2. A további útvonalak (**create**, **show**, **edit**) még nem léteznek ilyen elnevezésekkel. Ezeket a következő alfejezetekben fogjuk létrehozni a nézeteikben, és utána már működni fog az index oldal is megfelelően.

Megjegyzés: ha azt szeretnénk, hogy már most jól jelenjen meg az **index** oldalunk, akkor ideiglenesen kommenteljük ki a **create**, **show**, **edit** linkeket generáló kódsorokat, amelyek az adott elnevezésű nézet oldal létrehozása után visszaállíthatók (az itt látható kommentezés nélküli formában).

Ahhoz, hogy az index oldalon a projektlista is megkapja a kártyás kinézetet nyissuk meg a **resources / sass / components / _resource.scss** fájlt, és az erőforrás **class**-okat felsoroló sort a fájl elején bővítsük ki a **.project** osztállyal: **.post, .category, .tag, .comment, .project** {

Ez a módosítás a Vite futásának köszönhetően rögtön eredményre is jut, végre is hajtódik.

A két menüstruktúrában is tegyük elérhetővé ezt a projekteket listázó oldalt. Kezdjük a fejlécben lévő menüvel. Nyissuk meg a **resources / views / includes / _header.blade.php** fájlt és a felsoroláshoz adjuk hozzá ezt a sort:

```
<li><a href="{{ route('projects.index')}}">Projects</a></li>
```

8–39. kódrészlet: Fejlécben lévő menü új eleme

A *„hamburger menü”* struktúráját is bővítsük ki egy új menüelem komponens hozzáadásával a **resources / views / includes / _menu.blade.php** fájlban:

```
<x-nav-link
    url="{{ route('projects.index') }}"
    title="Projects"
    description="List of projects"
/>
```

8–40. kódrészlet: Hamburger menüben lévő menü új eleme

Ezután mindegyik oldal helyes működését megvalósítjuk, aztán integráljuk őket a sablonunkba. A kinézetük minden esetben olyan lesz, mint a korábban létrehozott erőforrásoknál is volt.

8.3.3. Egy erőforrás megtekintési oldala (show)

A show nézet oldal megoldása egy kicsit trükkös lesz, mivel, ha most létrehoznánk a **show.blade.php**-t, akkor az a **/projects/show** útvonalon lenne elérhető, ami ugyanolyan statikus, mint az **index** volt, és nem tartalmaz még dinamikus paramétert, *„wildcard"* útvonalrészletet.

```
<div>
    <h1>My specific Project</h1>
</div>
```

8–41. kódrészlet: A pages / show.blade.php tartalma

A Folio-s útvonal listát mindig le tudjuk kérni a php artisan folio:list utasítással.

```
GET      /projects ..................................................... projects/index.blade.php
GET      /projects/show ................................................ projects/show.blade.php
```

Az útvonal lekéréssel megkapjuk a böngészőben a statikus tartalmat, de nekünk most az a célunk, hogy a lekért azonosítóhoz vagy erőforráshoz tartozó adatokat kapjuk majd meg az adatbázisból.

Ehhez hozzunk létre egy **Project** Model osztályt, a kapcsolódó migrációs fájllal, vezérlővel, adatgyárral és Seeder osztállyal együtt, csak nagyon egyszerűen, hogy a példát szemléltetni tudjuk.

php artisan make:model Project -mfcs

A migrációs fájlt egy **string** típusú **title** nevű mezővel bővítsük csak és migrálhatunk is.

```
$table->string('title');
```

8–42. kódrészlet: A projects tábla alapértelmezett mezőin kívüli mezője

A **Project** Model osztályhoz adjuk hozzá a kitölthető mezőt:

```
protected $fillable = ['title'];
```

8–43. kódrészlet: Project Model osztály kitölthető mezője

A **ProjectFactory** osztály **definition()** metódusának visszatérési tömbjét bővítsük ki a következő utasítással:

```
'title' => fake()->sentence()
```

8–44. kódrészlet: Project generálásához a title mező példa értékeinek definíciója

A **ProjectSeeder** osztály **run()** metódusába pedig írjuk be a következő utasítást:

```
\App\Models\Project::factory(20)->create();
```

8–45. kódrészlet: 20 példaprojekt legenerálása seed-eléskor

Utána pedig feltölthetjük tesztadatokkal a **projects** adattáblát:

php artisan db:seed --class=ProjectSeeder

Most már visszatérhetünk az eredeti témánkhoz és átalakíthatjuk a Folio **show** nézetet úgy, hogy dinamikusan meg tudja jeleníteni az adott projektet. Nevezzük át a **show.blade.php** fájlt **[id].blade.php**-ra (így konkrétan szögletes zárójelekkel). Ha most újra lekérjük a Folio útvonal listát, már látni is fogjuk, hogy a jól megszokott paraméteres útvonalat kapjuk meg a statikus **show** helyett. A tartalmát módosítsuk erre:

```
<h1>My specific Project: {{ $id }}</h1>
```

8–46. kódrészlet: Az [id].blade.php Folio oldal dinamikus tartalmának kiíratása

Ha most meghívjuk a következő útvonalakat, akkor azt tapasztaljuk, hogy működni fog, meg fog jelenni az útvonalba beírt szám az oldal tartalmában is:

- http://127.0.0.1:8000/projects/1
- http://127.0.0.1:8000/projects/1000

Viszont működik az 1000-rel is, pedig nekünk csak 20 sorunk van a **projects** adattáblában. Ez eddig nagyjából rendben van, de így még nincsen hozzákötve a **Project** Model osztályon keresztül érkező adathalmaz, ami a **projects** adattáblából jönne.

Ehhez mindössze annyit kell tenni, hogy a Folio nézet oldal nevében az **[id]**-t átírjuk **[project]**-re. A nézet fájl tartalmát pedig módosítsuk úgy, hogy ne az **$id** mező kerüljön kiíratásra, hanem a **$project->title** mezője. Így már működni fog a hozzákötés:

8–11. ábra: Specifikus projekt megtekintése Folio oldal segítségével

Viszont 1000-es projektünk nincsen, úgyhogy arra 404-es HTTP állapotkódot kapunk, helyesen.

A működés így már megfelelő, úgyhogy illesszük be a nézet részeit a sablonba, integráljunk!

```
<?php
  use function Laravel\Folio\name;

  name('projects.show');
?>
@extends('app')
@section('main')
<article class="project">
  <h1>{{ $project->title }}</h1>
  <p>Created at: {{ $project->created_at }}</p>
  <p>Updated at: {{ $project->updated_at }}</p>
  <a href="{{ route('projects.index') }}">Back to the list</a>
</article>
@endsection
```

8–47. kódrészlet: A [project].blade.php Folio nézet oldal tartalma

8.3.4. Erőforrást létrehozó oldal (create)

A helyes működéshez szükségünk lesz az útvonalak és vezérlő metódusok közül a következők regisztrálására, létrehozására: **store**, **update**, **destroy**. Ezek pont azok, amelyekhez nem tartozik nézet oldal. A **routes / web.php**-t bővítsük ezzel:

```
Route::resource('projects', ProjectController::class)->only('store',
'update', 'destroy');
```

8–48. kódrészlet: Nézet nélküli CRUD útvonalak regisztrációja

Importálni ne felejtsük el a **ProjectController** osztályt a fájl elején! A vezérlő érintett metódusainál most tekintsünk el a Request osztályok létrehozásától és a validációtól az egyszerűség kedvéért.

```
public function store()
{
  Project::create(request()->all());
  return redirect(route('projects.index'));
}

public function update(Project $project)
{
  $project->update(request()->all());
  return redirect(route('projects.index'));
}

public function destroy(Project $project)
{
  $project->delete();
  return redirect(route('projects.index'));
}
```

8–49. kódrészlet: Nézet nélküli vezérlő metódusok a ProjectController osztályban

Ezután már létrehozható a **create** oldal a **categories.create** nézet alapján könnyedén, csak az útvonal elnevezését ne hagyjuk le a fájl elejéről! Ha nem sikerülne megoldani, akkor az alfejezet végén található GitHub commit-ben megtalálható ennek az oldalnak is a kódja.

8.3.5. Erőforrást szerkesztő oldal (edit)

Az **edit** nézet oldal elnevezési logikája egy kicsit szintén trükkös, úgyhogy engedjük a Folio-nak, hogy segítsen minket. Általában a nézet oldalakat nem szoktuk paranccsal létrehozni, mert általában mindig egy üres fájllal szoktunk kiindulni (nem úgy, mint mondjuk egy Model vagy Controller osztály esetén). A fájl tartalma kezdetben most is megfelelő úgy, ha üres, viszont a pontos helyének meghatározásában segít az, ha parancsot használunk a létrehozáskor. Így paranccsal hozzuk létre a Folio oldalt:

```
php artisan make:folio "projects/[project]/edit"
```

A fájl nem a többi mellé kerül be, hanem egy **[project]** almappába jön létre az új **edit.blade.php** fájl.

Magára az **edit** oldalra ugyanaz a megállapítás elmondható, hogy jó alapként tud hozzá szolgálni a **categories.edit** nézet, kiegészítve az útvonal elnevezéssel a fájl elején.

Miután elkészült az **edit** oldalba ágyazott útvonal is, visszaillleszthetjük az **index** oldalba a **create**, **show**, **edit** linkeket tartalmazó útvonalakat és tesztelhetjük a projekteket érintő összes funkcionalitás működését.

Az alfejezetben végrehajtott programkód módosítások ebben a GitHub commit-ben találhatók meg.

8–12. ábra: Folio oldalon keresztüli útvonal regisztráció és nézet generálása a felhasználói kérés kiszoglálásához a CRUD műveletek végrehajtásakor

Ismételt megjegyzés: a Laravel Folio használata csak egy lehetőség, akinek ez *„állna jobban kézre”*, az használja nyugodtan ezt, de nem kötelező, mert emellett a már létező megoldásaink tökéletesen működnek továbbra is a Laravel-ben.

8.4. Automatikus tesztelés (Laravel Dusk eszközzel)

A Laravel Dusk egy hatékony, könnyen használható eszköz a tesztek automatizálására. A Laravel Dusk hivatalos dokumentációja itt érhető el: https://laravel.com/docs/10.x/dusk Ezzel a Dusk-kal a fejlesztő képes lesz *„end-to-end” tesztet* csinálni (egy funkcionalitást elejétől a végéig tesztelni, mondjuk egy regisztrációs vagy belépési folyamatot, vagy a mi példánknál maradva: egy blogbejegyzés létrehozását).

Amikre szükségünk van a használatához, azokat sorrendben olvashatjuk alább:

1. Google Chrome böngésző (mivel ehhez van olyan driver, ami szükségeltetik a Dusk-hoz)
2. Telepítsük a Dusk-ot és függőségeit a projektünk terminal-jában ezzel a composer utasítással:

   ```
   composer require --dev laravel/dusk
   ```

3. A következő utasítással pedig a projektünkbe fogjuk telepíteni a Dusk-ot. Az utasítás hatására létrejön a **tests / Browser** mappa a projektünkben és benne egy példa teszteset, egy **ExampleTest.php**, meg még néhány beállítási fájl. Továbbá ezzel fog települni a Chrome driver is az operációs rendszerünkben.

   ```
   php artisan dusk:install
   ```

A létrejövő teszteset egyből azt szeretné ellenőrizni, hogy a főoldalon látható-e a Laravel szöveg, ami igazat is adna, ha egy teljesen új projektbe telepítettük volna a Dusk-ot.

A következő beállítást az **.env** fájlunkban kell elvégeznünk. Az abban lévő **APP_URL**-t kell megváltoztatni olyanra, ami kiszolgálás esetén működik. Ha például a `php artisan serve` paranccsal (tehát a PHP nyújtotta Development Server-rel) szoktuk futtatni a kiszolgálást, akkor ezt az **APP_URL**-t be kell állítani erre: http://127.0.0.1:8000/ címre. De ha például a XAMPP és a benne lévő Apache webszerver segítségével szeretnénk futtatni, akkor pedig ez lehet az **APP_URL** értéke: **localhost/[projekt-neve]/public** ha a felhőben (Azure, Heroku stb.) helyen futtatjuk, akkor aszerint állítsuk be ennek az **APP_URL** változónak az értékét. Mi indítsuk is el a `php artisan serve` paranccsal a kiszolgálást.

A **tests / Browser / ExampleTest.php** fájlunkat nyissuk meg, és nézzük meg, hogy mi van benne: a **testBasicExample()** metódusban láthatunk olyat, amit már a korábbi teszteknél is tapasztaltunk (vagy legalábbis hasonlót). Ezen teszteset is meg szeretné látogatni a webalkalmazásunk főoldalát, és ott ellenőrzi, hogy látható-e (a forráskódjában) a *„Laravel"* szó. Futtassuk a tesztelést:

```
php artisan dusk
```

Tipp: lehetséges, hogy hibát kapunk (**SessionNotCreatedException** névvel). Ekkor mindenképpen:

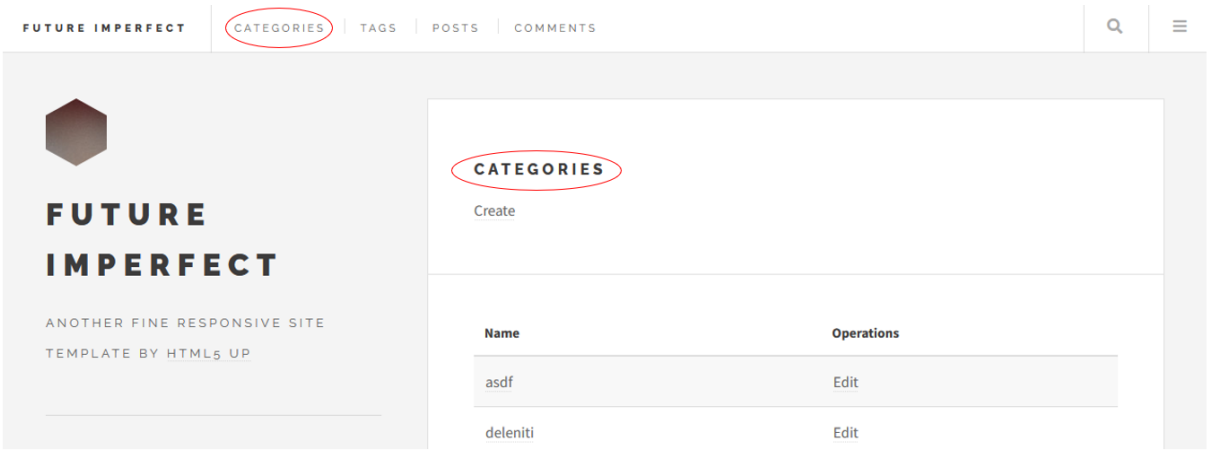
1. frissítsük fel a Chrome böngészőnket a legújabb verzióra,
2. adjuk ki ezt az utasítást: `php artisan dusk:chrome-driver`

Utána, ha újra futtatjuk a fenti parancsot, már azt kell megkapnunk, hogy minden rendben van a telepítéssel és a környezettel, de talán a tesztünk nem fut le pozitívan...

Az aktuális projektünkben nem biztos, hogy a főoldalon látható a *„Laravel"* szó, úgyhogy ezt szabadon módosíthatjuk aszerint, hogy mi is van az alkalmazásunk főoldalán a sikeres teszt lefutáshoz.

8.4.1. Assert utasítások közötti különbség vizsgálata

A Laravel Dusk **assertSee()** és az **assertDontSee()** metódus is egy kicsit másképp működik, mint a Laravel saját, ugyanilyen nevű metódusai. Mutatok erre egy példát, hogy a későbbiekben ne futhassunk bele ebbe a *„bug"*-ba. A **/categories** oldalt teszteljük. Az oldal így néz ki:



8–13. ábra: Kategóriákat listázó oldal kinézete

A *Case sensitive* tulajdonságra fogok koncentrálni, vagyis arra, hogy tesz-e különbséget kis és nagybetű között egyik illetve másik **assertSee()** (és **assertDontSee()**) metódus. Emiatt a *„CATEGORIES"* szöveget be is karikáztam a képen. De nézzük először a már ismert működést: a **Feature / CategoryTest.php**-ben hozzunk létre egy új metódust, amiben ezt a két **assert** metódust teszteljük:

```
function test_categories_page()
{
  $response = $this->get('/categories');

  $response->assertSee('Categories');
  $response->assertDontSee('CATEGORIES');
}
```

8–50. kódrészlet: Kategóriákat listázó oldal tesztelése a CategoryTest osztályban

Az *„assert"* utasítások a HTML kódban keresnek, így meg is találja az **assertSee()** a menüstruktúrában és a címnél is a *„Categories"* szöveget, és nem látja a csupa nagybetűs változatát.

A **Browser / ExampleTest**-ben lévő **testBasicExample()** metódus magját módosítsuk egy kicsit:

```
$this->browse(function (Browser $browser) {
  $browser->visit('/categories')
          ->assertSee('CATEGORIES')
          ->assertDontSee('Categories');
});
```

8–51. kódrészlet: Kategóriákat listázó oldal tesztelése a Dusk-os ExampleTest osztályban

Ez pedig pont ellentétesen működik mert ez is mindkettőre igazat ad vissza a számunkra, tehát mintha nem a HTML forráskódban keresné a szöveget, hanem abban, amit ténylegesen látunk az oldalon (HTML + CSS együtt, hiszen a CSS csinált a *„Categories"* szövegből csupa nagybetűs változatot). Erre az eltérésre, különbségre érdemes figyelni a későbbiekben!

Látható tehát, hogy a Dusk-os Assert utasítások egy kicsit másképp működnek, mint a PHPUnit által használt utasítások. Érdemes őket további vizsgálat alá vetni itt, ha a későbbiekben főleg a Dusk-ot használnánk a teszteléshez: https://laravel.com/docs/10.x/dusk#available-assertions

8.4.2. Laravel Dusk beállítások

Az összes Dusk-os tesztosztály őse a **DuskTestCase** osztály, ami a test mappában található **DuskTestCase.php** fájlban található meg. Ha megvizsgáljuk ezt a szülő osztályt, akkor azt tapasztalhatjuk, hogy számos beállítás található meg benne: van benne képernyőméretre vonatkozó beállítás, és van benne egy **--headless=new** kapcsoló is, ami a böngésző megnyitására vonatkozik teszt közben. Ha kikommentezzük ezt a sort és újra futtatjuk a `php artisan dusk` parancsot, akkor most már meg fog nyitni egy új Google Chrome böngésző ablakot és láthatjuk, hogy mit csinál a tesztelés közben... most persze még ez nem annyira látványos, hiszen csak egy szót keres az oldalon, de majd fogunk vele űrlapot is kitöltetni, ami már sokkal látványosabb lesz. Attól függően, hogy mennyire látványos / kevésbé látványos dolgot szeretnénk tesztelni a Dusk-kal, nyugodtan kapcsolgassuk (kommentezzük) ki/be ezt a **--headless=new** sort.

A Dusk-os tesztkörnyezet létrehozásához hozzuk létre az **.env.dusk.local** fájlt az **.env.testing** alapján a terminal-ban:

```
cp .env.testing .env.dusk.local
```

Arra mindenképpen érdemes figyelni, hogy az **APP_URL** értéke http://127.0.0.1:8000 legyen az **.env.dusk.local** fájlban. Illetve arra, hogy a Dusk nem tudja a teszteléshez használni az SQLite in-memory adatbázist, úgyhogy használjuk a MySQL adatbázis-kezelő rendszerben lévő **l10_blog_test_db** adatbázisunkat a teszteléshez. Az érintett beállítások tehát így néznek ki:

```
APP_URL=http://127.0.0.1:8000

DB_CONNECTION=mysql
DB_HOST=127.0.0.1
DB_PORT=3306
DB_DATABASE=l10_blog_test_db
DB_USERNAME=root
DB_PASSWORD=
```

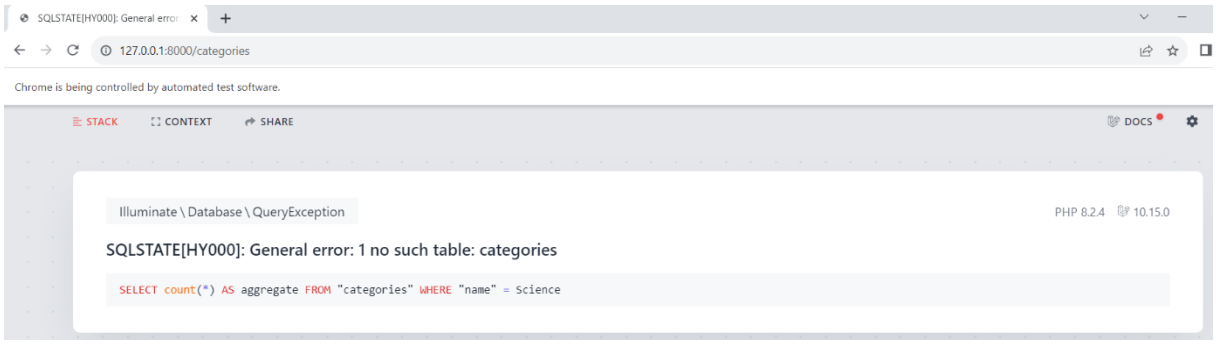
8–52. kódrészlet: Az .env.dusk.local fontos beállításai

Így most már tudunk úgy adatbázisműveleteket végrehajtani, hogy az az éles adatbázisunkat ne érintse. Ehhez majd a tesztosztályunkhoz *(és nem a metódusainkhoz)* hozzá kell adni ezt az utasítást:

use DatabaseMigrations;

Továbbá importáljuk is be a fájl elején az **Illuminate\Foundation\Testing\DatabaseMigrations** trait-et!

Ha esetleg elfelejtenénk átállítani ezeket az **.env.dusk.local**-ban történő paramétereket, akkor a rendszer könnyedén meg tudna tréfálni minket. Például, ha azt szeretnénk tesztelni, hogy létrehozunk egy új kategóriát (*„Science”* névvel) és szeretnénk látni az új kategória nevét, akkor a tesztünk helyes eredménnyel futna le, mivel az alábbi oldal jönne be az új kategória létrehozása művelet végrehajtása során:



8–14. ábra: Tévesen igaz (false positive) eredménnyel fut le a teszteset végrehajtása

Ezen az oldalon pedig látszódik a *„Science"* szó (a fő hibaüzenet alatti SQL utasításban), így tévesen, helyes lefutást mutatna a teszteset végrehajtása után a rendszer.

8.4.3. Egyedi Laravel Dusk tesztosztály létrehozása és működtetése

Azért, hogy lássuk a tesztesetek tényleges lefutását, érdemes most a DuskTestCase osztályban a driver() metódusban a '--headless=new' sort.

A beállítások elvégzése után már létre is hozhatjuk a Dusk lehetőségeire épülő tesztosztályainkat. Kezdjük a legegyszerűbbel, amit az előző alfejezet végén említettem meg, és haladjunk a bonyolultabbak felé: teszteljük a kategória létrehozását -> címke frissítését -> blogbejegyzés létrehozását szépen sorban.

8.4.3.1. Dusk: kategóriát létrehozó teszteset megtervezése és végrehajtása

Hozzuk létre a teszteset osztályát először:

```
php artisan dusk:make CategoryTest
```

Ezzel az utasítással létrejön a tests / Browser mappában a CategoryTest.php (ne keverjük össze a másikkal!). Ez a fájl is tartalmazza a testExample() metódust, ami a *„Laravel"* szöveget keresi a kezdőoldalon, de ezt a metódust törölhetjük az osztályból. A CategoryTest osztály új tartalma itt látható:

```
use DatabaseMigrations;

public function test_a_visitor_can_create_categories()
{
  $this->browse(function (Browser $browser) {
    $browser->visit('/categories/create')
      ->type('name', 'Science') // input mező name attribútum értéke
      ->click('input[type="submit"]')
      ->assertSee('Science');
  });
}
```

8–53. kódrészlet: A Dusk-os CategoryTest osztály tartalma

A kód alapján az automatikusan lefutó teszt meglátogatja (visit()) az adott útvonalat, begépeli (type()) a megadott input mezőbe a Science szót, majd rákattint (click()) a küldés gombra és utána elvárja, hogy látható legyen a *„Science"* szöveg azon az oldalon, ahova átirányításra került (index).

Futtassuk is le a tesztet és a megnyíló új Chrome böngészőben (figyelem, ha hibát kapnánk, akkor frissítsük a böngészőt is és a Dusk driver-ét is, ahogy korábban javasoltam):

```
php artisan dusk tests/Browser/CategoryTest.php
```

A megnyíló Chrome böngészőben végig követhető a folyamat és láthatjuk, ahogy automatikusan beírásra kerül a *„Science"* szó a kategóriát létrehozó űrlapon a **name** szövegmezőbe, majd megnyomásra kerül a *„Save"* gomb és az átirányítás következtében odajutunk a kategóriákat kilistázó **index** oldalra, ahol látható az egyetlen, új kategória. Azért csak egy, mivel a tesztelési adatbázisba hozta ezt létre nekünk. A táblák aztán a teszt lefutása után törlésre is kerülnek a tesztelési adatbázisban.

8.4.3.2. Dusk: címke szerkesztését és frissítését végző teszteset megtervezése és végrehajtása

Hozzuk létre a teszteset osztályát:

```
php artisan dusk:make TagTest
```

Egy kicsit bonyolultabb már ez a műveletsor, mivel bővebben elő is kell készíteni a teszteset végrehajtását, illetve ezen az űrlapon már **select** HTML mező is szerepel.

Arrange	Az új kategória létrehozása után hozzunk létre például nyolc új blogbejegyzést, majd egy új címkét. Ezt a címkét kapcsoljuk össze az 1, 3, 5 azonosítójú blogbejegyzésekkel
Act	A Dusk-os metódussal érjük el az új címke szerkesztésének útvonalát. Az adatgyár által generált nevet írjuk át *„new tag name"*-re. Majd módosítsuk a hozzá kapcsolt blogbejegyzéseket is: a címkét a 4. és 6. blogbejegyzéshez rendeljük hozzá.
Assert	Ellenőrizzük, hogy látható-e az új címke neve a címkéket listázó oldalon, illetve a kapcsolt blogbejegyzések *„darabszám"* értéke látható-e (2).

8–1. táblázat: 3A minta alapján megtervezett címke szerkesztési és frissítő teszteset

A teszteset implementálása (ne felejtsük el az adatbázist migráló utasítást előtte hozzáadni az osztályhoz):

```
public function test_a_visitor_can_update_a_tag()
{
  Category::factory()->create();
  Post::factory(8)->create();

  $tag = Tag::factory()->create();
  $tag->posts()->sync([1,3,5]);

  $this->browse(function (Browser $browser) {
    $browser->visit('/tags/1/edit')
      ->type('name', 'new tag name')
      ->select('posts[]', [4, 6])
      ->click('input[type="submit"]')
      ->assertSee('new tag name')
      ->assertSee('2');
```

```
  });
}
```

8–54. kódrészlet: Címkét szerkesztő és frissítő Dusk-os teszteset

Importáljuk hozzá a fájlban **Category**, **Post** és **Tag** Model osztályokat is a fájl tetején és az osztályon belül a **DatabaseMigrations**-t. Ez a kódrészlet a korábbihoz képest a teszteset kapcsán a kiválasztással (**select()** segédmetódussal) bővült, ahol az első paraméternek a **select** (alapértelmezetten) **name** attribútumában található **posts[]**-ot kellett beírni, de a Dusk a CSS selector-okat is tudja használni, így a **#posts** (**id** attribútumban) lévő azonosítóval is helyesen tudtunk volna rá hivatkozni.

8.4.3.3. Dusk: blogbejegyzést létrehozó teszteset megtervezése és végrehajtása

Hozzuk létre a teszteset osztályát:

```
php artisan dusk:make PostTest
```

Ez a teszteset leginkább az űrlapon található bemeneti mezők sokasága és különbözősége miatt érdekes a számunkra.

```
public function test_a_visitor_can_create_posts()
{
  Category::factory()->create();
  Tag::factory()->create();

  $this->browse(function (Browser $browser) {
    $browser->visit('/posts/create')
      ->type('title', 'My new title')
      ->type('slug', 'my-title')
      ->type('body', 'Content of my new post.')
      ->select('category_id', 1)
      ->type('score', 9.9)
      ->keys('#published_at', '2023', '{tab}', '10', '20', '16', '30')
      ->select('tags[]', 1)
      ->scrollIntoView('#tags')
      ->click('input[type="submit"]')
      ->assertSee('My new title')
      ->assertSee('1')
      ->assertSee('0');
  });
}
```

8–55. kódrészlet: Blogbejegyzést létrehozó Dusk-os teszteset

Importáljuk hozzá a fájlban **Category** és **Tag** Model osztályokat is a fájl tetején és az osztályon belül a **DatabaseMigrations**-t. A teszteset végrehajtásának előkészítése megtörténik azáltal, hogy a **select** mezőkbe kerülnek adatok a két adatgyár (**Category**, **Tag**) futása által.

Nézzük sorban az egyes bemeneti mezőket:

- Amikkel nem volt probléma (vagyis már ismert módon működnek): **title**, **slug**, **body**, **category_id** mezők.
- A **score**-nál csak arra hívom fel a figyelmet, hogy mivel ez egy **number** típusú bemeneti mező, ezért simán számként adjuk meg a példa adatot hozzá.
- A **published_at** mező egy datetime-local típusú dátumot és időt tartalmazó mező, általában a dátumok kezelésével sok probléma szokott lenni, amibe itt is belefuthatunk könnyen: a Laravel Dusk alapértelmezetten angolszász formátumban futtatja a teszteket, így az adatok kitöltése bár helyes lenne a dátumnál, az időnél viszont nem, de mégsem fogadná el az adattáblánk ezt a formátumú bemeneti értéket: „10/20/2023 06:16 PM"
    - a **DuskTestCase.php**-t kell elővenni a beállítás módosításához és ahol a '**--headless=new**'-t kikommenteztük, oda írjuk be utána egy új beállításként ezt: '**--lang=hu_HU',**
    - maga az érték billentyűk megadásával történik meg, mint ha manuálisan csinálnánk és megkapná a fókuszt a bemeneti mező: először a dátumot adjuk meg, utána ütni kell egy TAB-ot, majd sorban jöhet a hónap, nap, óra és perc számai. A **type()** segédmetódus helyett a **keys()**-t használjuk, mivel komplexebb megadásokra is képes ez, lásd például a TAB gombot.
    - ennél a mezőnél továbbá elvárta a Dusk, hogy a selector-t **id**-val (**#published_at**-ként) adjuk meg, nem volt neki megfelelő # nélkül ez az azonosítás.
- A címkék (**tag**-ek) kiválasztásával nem lehet gond.
- Viszont a Dusk csak azt a gombot *„tudja megnyomni",* ami látható a képernyőn. Az űrlap viszont túl hosszú ahhoz, hogy akár egy laptop átlagos méreténél látható legyen, ezért szükség van a **scrollIntoView()** segédmetódusra, amelynek paraméterül a gomb feletti mező azonosítóját adjuk meg: **#tags**
- Utána már működni fog a gombnyomás, mivel látható lesz a képernyőn a gomb.
- Az assert utasítások:
    - A blogbejegyzések lista oldalán látnunk kell az új blogbejegyzés címét,
    - a hozzá kapcsolódó címkék számát: 1
    - és a kapcsolódó kommentek számát: 0

A **click()** metódusban egy CSS szelektort helyeztünk el és így találja meg az oldalon jelen esetben most a *„Save"* (Mentés) gombunkat, ez teljesen jó most, de ha az oldalon esetleg több űrlap is látszódna, amelyeken van küldés gomb, (vagy nem annyira értenénk esetleg a CSS szelektorokhoz), akkor elhelyezhetünk a **resources / views / posts / create.blade.php**-nk elküldés gombjába egy új attribútumot:

dusk="save-button"

Ekkor a tesztben a **click()** metódust így is meghívhatjuk:

->click('@save-button')

A Laravel Dusk telepítési, beállítási és tesztelési fájljai ebben a GitHub commit-ben találhatók meg.

8.5. Összegzés

A fejezet feldolgozása során újra szerveztük (hatékonyabbá, optimálissá, egyszerűbbé, olvashatóbbá) tettük programkódjainkat. Az erőforrások CRUD műveleteit integráltuk a már meglévő webalkalmazásunk sablonjába, annak keretes szerkezetébe. A kliens oldali kódokat, ahol lehetett, átalakítottuk komponens alapúra, így újra felhasználható, paraméterezhető kódokhoz jutottunk.

A Laravel folyamatos fejlődésének köszönhetően, az új Folio eszközt is kipróbáltuk, hogy hogyan működik egy erőforrás (project) összes CRUD funkcionalitásának megvalósításakor.

Végül megismerkedtünk egy új eszközzel, amit a teszteléshez egyszerűen és hatékonyan tudunk használni, ez volt a Laravel Dusk. A segítségével könnyen és látványosan tudunk létrehozni automatikus teszteseteket és tudjuk őket futtatni is vele.

9. Felhasználói adatérvényesítés (Validation)

A validálás egy több rétegű folyamat, amely elengedhetetlen a webalkalmazásunk biztonságos és hibamentes működéséhez. Hogyan tudunk a legegyszerűbben kapcsolatba lépni a weboldalunkra érkező látogatókkal? Űrlapokon keresztül tudunk velük kommunikálni. A kommunikáció legtöbbször egy regisztrációval, bejelentkezéssel, kapcsolatfelvételi űrlappal kezdődhet... aztán attól függően, hogy mivel foglalkozik a webalkalmazásunk, további felületeken is kezdeményezhetünk kommunikációt, például egy webshop-os vásárlási felületen keresztül, ahol el szeretnénk neki adni valamit és lehetőséget akarunk biztosítani, hogy vásároljon tőlünk... vagy éppen csak egy időpontfoglalási rendszert akarunk építeni, ekkor is egy űrlapon keresztül tudunk kapcsolatba kerülni a felhasználóinkkal. Látható, hogy rengetegféle okkal lehet űrlapot építeni, már csak az a kérdés, hogy hogyan tudunk érvényes információt szerezni a felhasználóinktól? Ebben segít nekünk a validálás folyamata.

Onnantól kezdve, hogy a felhasználóinkkal kommunikálunk, már a legelején érdemes tisztázni, hogy ez biztonság szempontjából kritikus folyamatot jelent, és mint tudjuk, tökéletes védelem nem létezik! De minél inkább igyekszünk megnehezíteni a támadó dolgát, annál inkább több időbe kerülne az oldalunk feltörése... feltűnhet, hogy támadót írtam, miközben eddig felhasználókról írtam. Előfordulhat persze, hogy az egyszeri felhasználó csak elrontana valamit az oldalon, aminek kapcsán a validálási folyamatunk segítene neki helyes irányba terelni az űrlap kitöltésének menetét. Én azért úgy szoktam kezelni, hogy minden, ami a felhasználótól bemenetként érkezik az űrlapunkra, arra tekintsünk úgy, mint ha ördögtől való lenne, és próbáljunk ellene védekezni minden lehetséges módon: ellenőrizzük le többször a felhasználótól érkező adatokat, hogy biztosan ne okozhasson problémát a webalkalmazásunk működésében.

Fejlesztőként *hol ellenőrizzünk? Kliens vagy szerver oldalon?* A válasz leginkább az, hogy *mindenhol IS, többször is, akár ugyanazt is*. Azért, hogy megfelelő, érvényes adatokhoz juthassunk egy felhasználótól, ellenőrizzük a kapott értékeket kliens és szerver oldalon egyaránt. Ebben számos dolog segít minket: kliens oldalon a HTML5-ös szabvány (kiegészítve a CSS stílusokkal) valamint a JavaScript kódok (és az összes validálást segítő JavaScript osztálykönyvtár vagy keretrendszer) lehetnek a segítségünkre. Míg szerver oldalon majd a Laravel keretrendszer lesz főleg a segítségünkre.

Kissé laikusként feltehetnénk még azt a kérdést is, hogy ha mindkét oldalon validálunk, akkor megfelelő adatokhoz fogunk-e jutni a felhasználótól? A válasz nem mindig az igen lesz. Ugyanis a validáció *„csak"* abban segít nekünk, hogy megfelelő *formátumban* kapjuk meg az adatokat, például, hogy ahol e-mail címet várunk el, az valóban egy e-mail cím szabályait követő cím lesz-e. De azt nem tudhatjuk (vagy csak nehézkesebben tudjuk ellenőrizni), hogy az valódi e-mail cím-e, létezik-e a valóságban, használja-e azt valaki. Erre a Laravel keretrendszer is egyre kifinomultabb megoldásokkal segít minket (lásd a 9.2. alfejezetet). Vagy ha például a dátum értékekre gondolunk, amelyek rengeteg formátumban léteznek, de általában az adatbázisunk csak egy adott formátumú dátum értéket képes befogadni. Ilyenkor azért, hogy *„ne bukjon el"* az adatbekérési és feltöltési folyamat, már a kliens oldalon ellenőriznünk kell, hogy a felhasználó a nekünk (és az adatbázisunknak) megfelelő formátumban adta-e meg az értéket. Viszont, ha csak a formátumot ellenőrizzük a dátumoknál, a helyesnek elfogadható értéktartományt pedig nem, akkor

könnyedén előfordulhat, hogy a felhasználó egy születési dátum mezőbe beír egy ezer évvel ezelőtti dátumot, ami nyilvánvalóan nem lesz megfelelő nekünk. Még rengeteg példát tudnék hozni az adatérvényesítéssel kapcsolatban, de inkább nézzük meg őket a gyakorlatban.

Mikor történik meg a bemeneti adatok validálása?

- Kliens oldali validáció során némely űrlapelem már a bemeneti adatok megadásakor rászorítja a felhasználót, hogy bizonyos formátumban adhat csak meg adatot, például, ha a mező típusa number, akkor csak számot tudunk beírni, betűket nem. Kliens oldalon a validáció az űrlap adatainak elküldésekor történik meg, tehát amikor a felhasználó megnyom egy küldés gombot (tipikus gombfeliratok lehetnek: mentés, frissítés, regisztráció, belépés stb.). Ekkor újra ellenőriz a böngésző minden mezőt, hogy az adatok megadásakor a formátumok megfelelőek voltak-e és csak akkor engedi tovább a szerver felé a bemeneti adatcsomagot, ha kliens oldalon mindent rendben talált. A kliens oldali validáció a felhasználókat segíti az űrlapok helyes kitöltésében.
- Ha kliens oldalon minden megfelelő volt, akkor az ellenőrző vagy érvényesítő szerepét átveszi a szerver oldalon az ottani kódunk. Onnantól kezdve már az felelős az adatok formátumának megfelelőségéért. Tipikusan szerver oldalon is le szoktunk ellenőrizni minden olyan dolgot, amit kliens oldalon már megtettünk, ugyanis a kliens oldali szabályok könnyedén kijátszhatóak. A szerver oldalon is ellenőrizni kell azt például, hogy minden szükséges adat meg van-e, betartották-e a karakterhosszúsági szabályokat és a többi... rengetegféle szabály betartására kötelezhetjük a felhasználókat. Szerencsére szerver oldalon a Laravel nagyon sok beépített lehetőséget biztosít ahhoz, hogy tudjuk ellenőrizni a beérkező adatokat (9.2.1. alfejezet) és még egyedi validálási folyamatokat és szabályokat is létrehozhatunk (9.2.3. és 9.2.4. alfejezetek). Ha szerver oldali ellenőrzéskor hibát kapunk, akkor visszadobhatjuk azt a kliens oldalnak és jelezhetjük, hogy mi a hiba, mit kellene javítani. Ha mindent hibamentesnek találtunk, akkor pedig elmenthetjük az adatokat az adatbázisba vagy feldolgozhatjuk őket helyben, amiről aztán visszajelezhetünk valamit a felhasználóknak. A szerver oldali validációval ellenőrizhetjük a kevésbé jóindulatú felhasználóktól érkező adatok, a *„kevésbé jóindulatú”* alatt azokat érthetjük, akik esetleg direkt megtörték a kliens oldali validációt, a szerver oldalon viszont az ő támadásaikat is el kell kapnunk.

Szoftverfejlesztőként a programkódjainkat (funkcionalitásokat, nézeteket stb.) úgy kell építenünk, hogy az ellenőrzés (a validációs kódok elhelyezése) már akkor megtörténjen, amikor még ténylegesen a programlogikára koncentrálunk és azt implementáljuk.

9.1. Kliens oldali validáció (Client-side form validation)

Mivel könyv legfőbb témája a Laravel keretrendszer, ezért főleg majd a szerver oldali validációra fókuszálunk, de felvillantok néhány lehetőséget annak kapcsán, hogy kliens oldalon hogyan tudjuk ellenőrizni a felhasználóktól érkező adatokat.

9.1.1. HTML5 lehetőségei a validálásra

A leggyakrabban használt beépített űrlap validációs HTML5 attribútumok (már a nevük is elég beszédes, de rövid magyarázatot azért írok hozzájuk):

- **required**: megadjuk vele, hogy a mező kitöltése kötelező;
- **minlength** és **maxlength**: minimálisan és maximálisan elfogadható bemeneti adat hosszúság;
- **min** és **max**: number típusú bemeneti mezőnél a minimálisan és maximálisan elfogadható érték;
- **type**: bemeneti adatok típusára vonatkozó megszorítás, például: number (szám), email (e-mail), datetime-local (dátum és idő), tel (telefonszám) stb.;
- **pattern**: ennek a használata talán a legnehezebb, de mégis a leginkább testre szabhatóan alkalmazható ellenőrzésre (ha értünk a reguláris kifejezésekhez), mivel egy mintát tudunk meghatározni vele, például ha egy telefonszám mezőnél meghatározzuk, hogy két számmal (körzetszám) kell kezdődnie, majd per jel, majd három szám, majd kötőjel, majd négy szám... akkor ezt egyértelműen ilyen reguláris kifejezéssel tudjuk megadni a **pattern** attribútumnak, a bemeneti mező pedig emiatt el fogja várni, hogy ilyen formátumban adjuk meg a telefonszámot, különben el fog bukni a kliens oldali validáció.
    - Általában gond szokott lenni, ha valamilyen komplexebb mintát várunk el a bemeneti mezőnél és a felhasználó nem tudja kitalálni, hogy ennél a telefonszám bemeneti mezőnél mi a helyes minta, amit követnie kell az adatok megadásakor. Ekkor érdemes használni a **placeholder** attribútumot, amibe tudunk írni egy példát a telefonszám formátumára, amit helyesnek fogadnánk el, ha azt küldené el a felhasználó.

Ezek tehát a HTML űrlapok bemeneti mezőinél validálásra alkalmazott attribútumok, amikkel rá tudjuk bírni a felhasználót bizonyos szabályok, formai követelmények betartására.

Ezek közül a **type** (number és datetime-local), valamint a **min** és **max** attribútumokat is már használtuk, amikor a blogbejegyzésnek pontszám értékelést kellett megadni felvételkor vagy szerkesztéskor.

A példa kedvéért vegyük elő a blogbejegyzések **create** nézetét, majd határozzunk meg néhány kliens oldali validációs elvárást a felhasználó felé:

- A **required** attribútumot gyakorlatilag mindegyik mezőbe beírhatjuk az összes űrlapon (kivétel a **published_at** mezőnél, de annak úgyis be van állítva egy alapértelmezett érték, így, ha azt nem törli a felhasználó akarattal, akkor azzal nem lehet gond), hiszen a legtöbb mezőnél nem engedünk meg null érték beszúrását az adattáblába, így a kitöltésük kötelező kell legyen. Megjegyzés: ez a **required** attribútum **select** komponensnél is működik.
- A **title** mezőnek adjunk meg egy minimális (5) és egy maximális (100) hosszúságot (**minlength** és **maxlength**).
- A **slug** mezőnél várjuk el, hogy legalább két szövegrészből álljon és köztük legyen legalább egy kötőjel. Ezt a **pattern** attribútummal tudjuk kötelezővé tenni a felhasználó számára, segítségként pedig a **placeholder**-t tudjuk használni. *Megjegyzés:* a **PostFactory** osztályban a **slug** generálásánál ugyanezt a szabályt alkalmaztuk.

A felsorolt validációs szabályokra itt vannak a példa implementációk:

```
<textarea name="body" id="body" cols="30" rows="10" required></textarea>
<x-select name="category_id" id="category_id" :options="$categories"
required />
```

```
<input type="text" name="title" id="title" required minlength="5"
maxlength="100" />
<input type="text" name="slug" id="slug" required
pattern="[a-z]+[-]{1}[a-z]+" placeholder="first-post" />
```

9–1. kódrészlet: A posts.create nézet űrlapjában alkalmazott szabályok

A blogbejegyzést létrehozó űrlap ezekután tartalmazni fogja ezeket az elvárásokat, és ha nem teljesítjük őket, akkor a *„Save"* küldő gomb megnyomásakor ellenőrizni fogja a betartásukat a böngésző. Ha valahol, valamelyik mezőnél nem teljesül az adott szabály betartása, akkor az adatok elküldése helyett az alkalmazás odaugrik a böngészőben és a problémás mező megkapja a fókuszt, valamint a böngésző nyelvének megfelelő automatikus hibaüzenet is megjelenik a mező felett/alatt, például magyar Firefox-ban: *„Töltse ki ezt a mezőt."* – a **required** attribútum megsértésénél, vagy „A kért formátumban adja meg az adatot." – a **pattern** minta megsértésénél.

Feladat: gyakorlásként gondoljuk át, hogy az űrlapjainknál milyen kliens oldali HTML validációs attribútumokat kellene elhelyezni az egyes mezőknél, és szükség esetén próbáljuk is ki őket.

A kliens oldali validáció azonban nem elegendő, mivel ezt a felhasználó könnyedén felül tudja írni: ha a böngészőben megvizsgálja a blogbejegyzés **title** bemeneti mezőjét és a DOM fában átszerkeszti (kitörli) a **required** attribútumot, majd úgy küldi el a kódot, akkor egy hibát (Exception) fog kapni, mivel az adattáblába történő beszúráskor nem lesz a **title** mezőnek értéke, pedig az kötelező lenne. Így tehát könnyedén megtámadható a webalkalmazásunk pusztán a HTML5-ös kliens oldali validációval.

9.1.2. CSS lehetőségei a validálásra

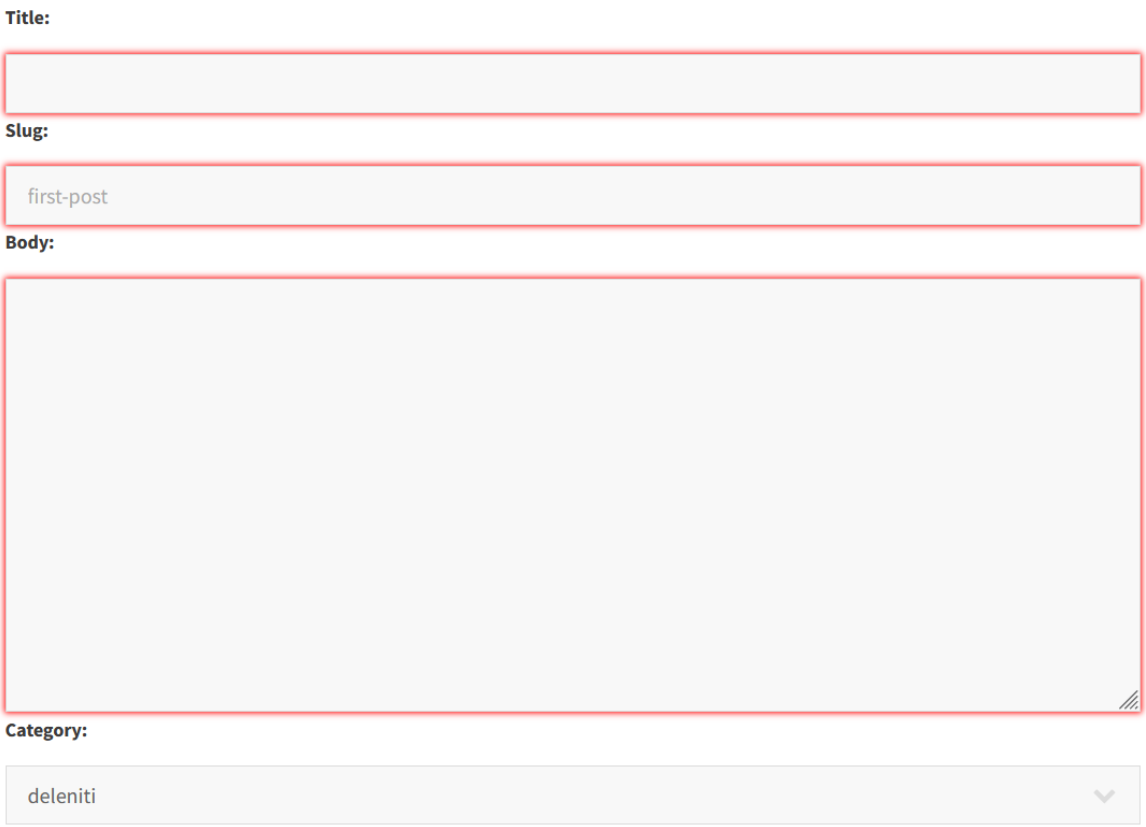
Bár a CSS-t a kinézet formázására tudjuk használni, kevésbé a funkcionalitásokhoz, de mégis egy kicsit a CSS3 már néhány pszeudo osztálya segítségével lehetőséget biztosít a validálás elősegítéséhez:

- A **:valid** és **:invalid** pszeudo-osztályok használatával különböző formázásokat adhatunk a bemeneti mezőknek, ha azok megfelelőek validálás szempontjából, vagy sem. Ha a mező értéke megfelelő, akkor a **:valid** pszeudo-osztály stílusszabályai lesznek rá érvényesek, míg ha nem érvényes, akkor az **:invalid** részben tudunk különböző szabályokat meghatározni rá.
- Az **:out-of-range** pszeudo-osztály és annak szabályai akkor lesznek érvényben, ha az input mezőre meg volt határozva egy **min** és/vagy **max** érték, a felhasználó által pedig ezeken a határokon túllépő értékek kerültek megadásra. Ennek a pszeudo osztálynak az ellentéte az **:in-range**, amelyet akkor *„kap meg"* a bemeneti mező, ha a **min** és **max** határokon belül került megadásra az értéke, tehát rendben van validálás szempontjából.
- Ha valamilyen bemeneti elemet le szeretnénk tiltani, hogy ne lehessen szerkeszteni, mert például egy számítás eredményét akarjuk csak mutatni benne, ami további bemeneti adatoktól függött, akkor a **:disabled** pszeudo-osztállyal tudunk rá hivatkozni a CSS kódunkban és különböző stílusszabályokat beállítani neki.
- Attól függően, hogy a bemeneti mezőnk kötelező kitöltésű volt vagy sem, alkalmazhatók rá a **:required** és az **:optional** pszeudo-osztályok stílus szabályai.

Gyakorlati szempontból maradjunk a blogbejegyzéseket létrehozó űrlapunknál (**posts.create**), hiszen itt már biztosan definiáltuk a bemeneti mezők érvényességi szabályait. Az imént felsorolt pszeudo osztályok közül válasszunk ki néhányat és figyeljük meg, hogyan tudnak segíteni nekünk az érvényes adatok bekérésének kikényszerítésében. A form elemek stílusának módosításához nyissuk meg szerkesztésre ezt a fájlt: **resources / sass / components / _form.scss**

A fájl ~61. sorában már van egy bemeneti mezőkre (input, select, textarea) vonatkozó **:invalid** pszeudo-osztály szelektor és a rá vonatkozó stílus információ, ami a doboz árnyékolásról szól, de *„none"*-ra van állítva, ami azt jelenti, hogy semmilyen árnyékolást nem kap az **invalid** bemeneti mező. Változtassuk meg a **none** értéket erre: **0 0 5px 1px red**

Ha most frissítjük a blogbejegyzést létrehozó nézetünket, akkor minden üres bemeneti mező gyakorlatilag megkapja az **invalid** jelölésnek köszönhetően a piros árnyékolást.



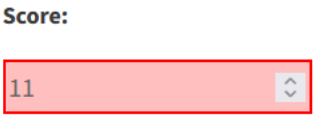
9–1. ábra: Érvénytelen (invalid) mezők árnyékolása a posts.create nézet űrlapján

Az **:out-of-range** pszeudo osztályra még nem vonatkozik semmilyen stílusszabály, így ezt helyezzük el a fájl végén:

```
input:out-of-range {
  background-color: rgba(255, 0, 0, 0.25);
  border: 2px solid red;
}
```

9–2. kódrészlet: Min/Max határátlépés stílusszabálya

Ennek köszönhetően, ha a blogbejegyzés értékelésénél (**score**) olyan számot állítunk be, ami az 1-10 értéktartományon kívül esik, akkor a mező háttérszíne halvány piros lesz, és kap egy piros keretet is a mező. Alapból a léptetéssel nem tudnánk az értéktartományon kívül menni, csak ha manuálisan átírjuk az értéket nagyobbra vagy kisebbre.



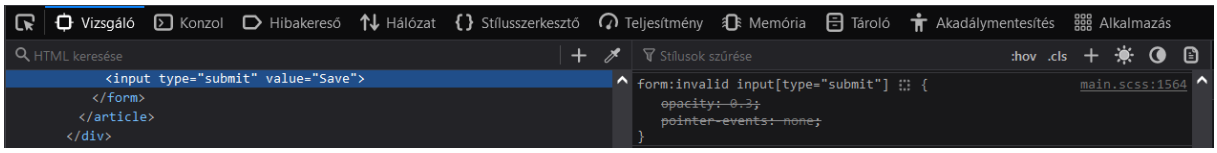
9–2. ábra: Értékhatár átlépés stílusa aktivizálódik

Megtehetjük azt is, hogy *„el sem engedjük"* addig a folyamatot, hogy kiírásra kerüljenek a problémák a szabályok megsértése miatt, hanem egészen addig inaktívvá tesszük a küldés gombot, ameddig az űrlap maga nem válik érvényessé (valid).

```
form:invalid input[type="submit"] {
  opacity: 0.3;
  pointer-events: none;
}
```

9–3. kódrészlet: Érvénytelen űrlap küldés gombja inaktív

De mivel kliens oldalon vagyunk, a böngésző vizsgáló részében ez a szabályozás is könnyen áthágható. A kiválasztott gomb CSS stílusainál kivesszük a pipát és máris kattinthatóvá válik a gomb… ettől persze az űrlap még nem lesz érvényes és elküldhető, de látható belőle, hogy könnyen felülírhatók egy hozzáértőnek a kliens oldali validációs szabályok.



9–3. ábra: Űrlap elküldése kattinthatóvá tehető az érvénytelenség ellenére is

9.1.3. JavaScript lehetőségei a validálásra

A JavaScript alkalmazása az egyik leghatékonyabb megoldás a kliens oldali validáció végrehajtására, a helyes adatok megadásának kikényszerítésére. Én viszont külön JavaScript oktatást nem szeretnék tartani itt, mert nem ez a könyv célja, de több egyetemi kollégám készített a nyelvről és alkalmazásáról anyagokat, amiket szívesen megosztok veletek:

- http://webprogramozas.inf.elte.hu/tananyag/kliens/
- http://webprogramozas.inf.elte.hu/tananyag/weaf1/

A HTML5-ös űrlap bemeneti elemeknél végzett megszorítások segítenek a felhasználóknak megfelelően kitölteni egy mezőt, azonban nem védenek a támadások ellen. Induljunk ki onnan, hogy valaki *„megpróbálja feltörni"* a HTML5-ös validációnkat, tehát mint ahogy az alfejezetekben is írtam, élőben belenyúl a kódba és kitörli például a **required** attribútumot a HTML tag-ünkből, majd úgy küldi el az űrlapot, hogy nem tölti ki a kötelezően kitöltendő mezőt. *Mit tehetünk még ekkor a validációnk sikerességéért?*

JavaScript kódokat fogunk használni, ami persze még közel sem jelent majd számunkra tökéletes védelmet (olyan nincs is), de még egy fokkal nehezebbé tesszük az alkalmazásunk feltörését.

9.1.3.1. Egyszerű példa implementálása

Gyakorlati szempontból még mindig maradunk a blogbejegyzéseknél, de most már az **edit** nézetre térjünk át. Előtte azonban kommenteljük ki az előző alfejezetben a **_form.scss** fájl végére beszúrt, űrlap érvénytelensége esetén a küldés gombot inaktívvá tevő szabályokat. Ezután biztosítsunk helyet az új script-jeinknek az **includes / _scripts.blade.php** fájl végén (ez a fájl **app.blade.php** záró **body** tag-je elé töltődik be):

```
@stack('validation-scripts')
```

9–4. kódrészlet: Validációs script-ek helye (verem)

Nagyon hasonlóan a **@yield** - **@section** Blade direktíva pároshoz, itt a **@stack** - **@push** lesz a segítségünkre abban, hogy JavaScript kódokat helyezzünk el az egyes gyermekoldalakon (az app.blade.php szülő nézetünket így nem kell *„tele szemetelnünk"* oda nem illő script-ekkel). A különbség a két direktíva páros között, hogy amíg egy gyerekoldalon egy **@yield**-be csak egyszer tudunk kódot írni a section-nel, addig egy verembe (stack-be) bármennyiszer szúrhatunk be (push-olhatunk) akár újabb és újabb kódokat, amennyit csak szeretnénk. Míg a **@yield** - **@section** párost inkább a HTML kódokhoz és a CSS/JavaScript csomagok, osztálykönyvtárak importálására használjuk, addig a **@stack** - **@push** párost a változó CSS és főleg JavaScript kódjaink beszúrására alkalmazzuk. A saját JavaScript kódjainknak fenntartott stack-et pedig azért helyezzük el a body záró tag-je elé, mert a kódjainkat gyakran használjuk az oldalon megjelenített (render-elt) HTML kód manipulálására, és ha akkor szeretnénk megváltoztatni valamelyik HTML elemet, amikor még nem került megjelenítésére, akkor hibát kapnánk a JavaScript Console-ban, hogy még nem talált ilyen elemet a DOM fában, ezért nem is működne a JavaScript kódunk.

Megvan tehát a szülő app nézetünkben a JavaScript kódjaink helye, így nincs más hátra, mint az utód nézet fájlba beillesszünk saját kódokat, amivel ellenőrizzük az űrlapunkat. Nyissuk meg a **posts.edit** nézet fájlt, és a szerkesztési űrlap komponens nyitó **x-form** tag-jébe helyezzük el a **name** és az **onsubmit** attribútumokat és értékeket (a jobb átláthatóság miatt a teljes nyitó tag-et megjelenítem itt):

```
<x-form
  action="{{ route('posts.update', $post->id) }}"
  method="put"
  name="post-edit"
  onsubmit="return validateForm();"
>
```

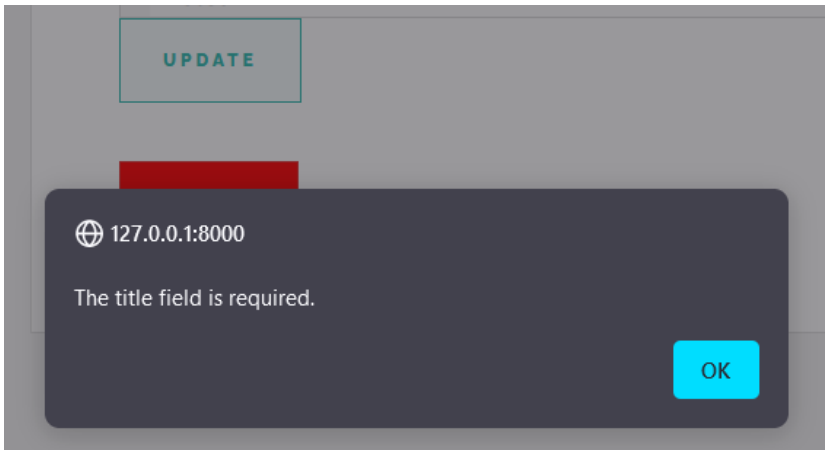
9–5. kódrészlet: A posts.edit nézet blogbejegyzést szerkesztő űrlap komponensének nyitó tag-je

Az egyik attribútummal nevet adunk az űrlapnak, míg az **onsubmit** attribútummal jelezzük majd az űrlapnak, hogy amikor a felhasználó rákattint a *„Update"* (frissítés, küldés) gombra, akkor milyen metódusnak kell végrehajtódni a tényleges adatelküldés előtt. Hozzuk is létre és szúrjuk be a stack-be az új függvényünket (bár nincs jelentősége, én a **@endsection** után helyezem el a kódomat, de megtehetném, hogy a **@section** elé tenném be):

```
@push('validation-scripts')
<script>
function validateForm() {
  const title = document.forms["post-edit"]["title"].value;
  if (title == "") {
    alert("The title field is required.");
    return false;
  }
}
</script>
@endpush
```

9–6. kódrészlet: Űrlapellenőrző metódus a title mező kötelező kitöltéséhez

Tetszőleges blogbejegyzés szerkesztésekor töröljük ki a **title** mező teljes tartalmát, és alul kattintsunk az *„Update”* gombra. Eredményül ezt kell kapnunk:



9–4. ábra: Űrlap érvényesség ellenőrzése JavaScript-tel (figyelmeztető ablak az érvénytelenségről)

A példa kedvéért bővítsük ki még a **validateForm()** függvényt az alábbi feltételvizsgálattal:

```
const tags = document.forms["post-edit"]["tags[]"].selectedOptions;
if(tags.length < 1) {
  alert("Minimum 1 selected tag is required.");
  return false;
}
```

9–7. kódrészlet: Űrlapellenőrzésnél legalább egy címke meglétének ellenőrzése

Tesztelni az ilyen **multiple <select>**-et úgy tudjuk, hogy a Ctrl + bal egérgomb kattintással levesszük a kijelölést az összes már kijelölt elemről, és úgy próbáljuk meg a frissítés gombra kattintani. A figyelmeztetést ugyanúgy megkapjuk, mint az előző esetben is. Ez persze nem a legszebb megoldás, pusztán csak a működést szerettem volna szemléltetni így.

9.1.3.2. Constraint Validation API

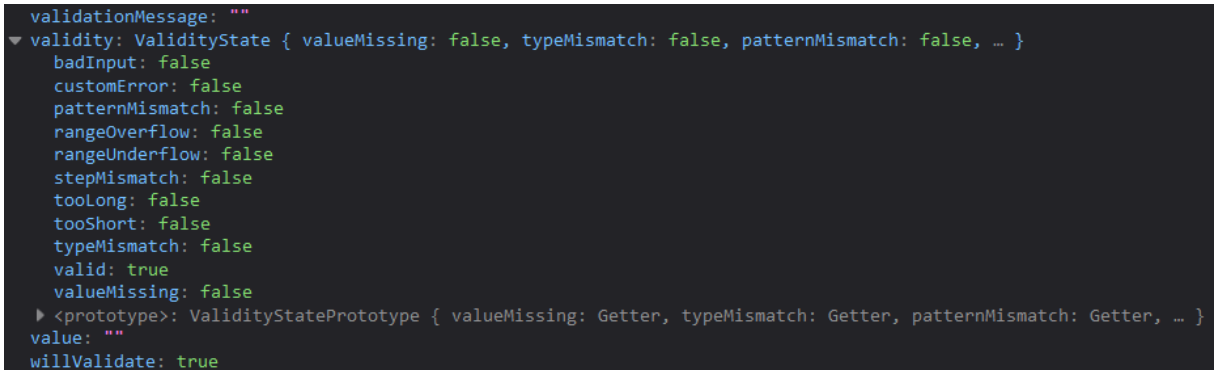
A legtöbb böngésző már támogatja ezt az API-t. Így például a gombokhoz, bemeneti mezőkhöz (input, textarea), select-hez van már egy olyan felület, amit alapértelmezetten használhatunk validációra és nem kell különböző egyéb JavaScript osztályokat importálnunk. Minden imént felsorolt mező automatikusan

rendelkezik azzal a lehetőséggel, hogy validáljuk őket (ellenőrizhetjük a következőket: illeszkedik-e a bemenet egy mintára, túl rövid/hosszú-e a bemenet, túllép/nem ér el valamilyen limitet a szám értéke, típusprobléma e-mailek vagy URL-ek kapcsán, hiányzó bemenet, vagy egyszerűen csak valamilyen egyéb, általunk definiált szabálynak nem felel meg a bemenet). Ezeket tehát mind támogatja az API, illetve ezek ellenőrzését.

Az API még azt is lehetővé teszi, hogy az imént felsorolt mező problémák ellenőrzésénél különböző hibaüzeneteket állíthassunk be a **setCustomValidity(message)** metódus segítségével. Néhányat kipróbálunk a gyakorlatban is és megnézzük, hogy hogyan működnek, de előtte csináljunk egy sima kiíratást abban a böngészőablakban, ahol a blogbejegyzést szerkesztő űrlap látható. Nyissuk meg a JavaScript konzolt (akár valamelyik mező vizsgálatával megnyílik a Vizsgáló és a mellette lévő lapfülön a Konzol van). Írjuk be a következőt oda:

```
console.log(document.forms["post-edit"]["tags[]"])
```

Egy hosszú listát fogunk kapni, de görgessünk le a v kezdőbetűig és ezt kell látnunk:



9–5. ábra: Címkéket tartalmazó bemeneti űrlapelem jelenleg releváns attribútumai

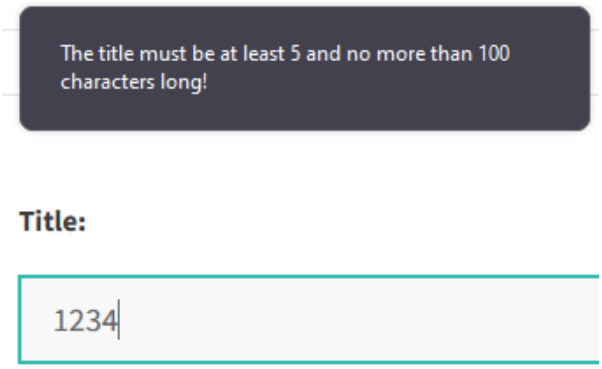
A **validationMessage** tartalmazza a hibaüzenetet, ha nem stimmel valamilyen ellenőrzés. A **validity** attribútum tartalmaz egy **ValidityState** objektumot, aminek láthatók a felsorolásában azok az attribútumok, amiket az imént én is felsoroltam részletesen, hogy milyen problémák fordulhatnak elő egy ilyen **select**-tel. A **willValidate** mező pedig azt jelzi nekünk, hogy ellenőrzésre fog-e kerülni a vizsgált (kiíratott) űrlapelem.

A további gyakorláshoz térjünk vissza a blogbejegyzések létrehozó űrlapjához, mert ott az előző alfejezetekben sok szabályt állítottunk be, amelyeket most a JavaScript és a Constraint Validation API segítségével is ellenőrizni tudunk. Bővítsük a **posts / create.blade.php**-t az **@endsection** után:

```
@push('validation-scripts')
<script>
// Step 1.: required attribute
const myform = document.forms["post-create"];
field = Array.from(myform.elements);
field.forEach(i => {
  if (i.validity.valueMissing) {
    i.setCustomValidity('There must be a value to the field.');
  } else {
```

```
    i.setCustomValidity('');
  }
});


// Step 2.: other specific validation attributes
// -------------- title --------------
const title = document.getElementById("title");

title.addEventListener("input", (event) => {
  if (title.validity.tooLong || title.validity.tooShort) {
    title.setCustomValidity("The title must be at least 5 and no more than
100 characters long!");
  } else {
    title.setCustomValidity("");
  }
});

// -------------- slug --------------
const slug = document.getElementById("slug");

slug.addEventListener("input", (event) => {
  if (slug.validity.patternMismatch) {
    slug.setCustomValidity("Follow the pattern: the slug should consist of
two lower case words separated by a hyphen!");
  } else {
    slug.setCustomValidity("");
  }
});

// -------------- score --------------
const score = document.getElementById("score");

score.addEventListener("input", (event) => {
  if (score.validity.rangeOverflow || score.validity.rangeUnderflow) {
    score.setCustomValidity("The score must be between 1.0 and 10.0!");
  } else {
    score.setCustomValidity("");
  }
});
</script>
@endpush
```

9–8. kódrészlet: Constraint Validation API alkalmazása a posts.create nézetben



9–6. ábra: JavaScript-tel generált validáció és beállított hibaüzenet a Constraint Validation API-val

Megjegyzés: ha még a korábbi (magyar) figyelmeztető üzeneteket kapnánk meg a validáláskor, akkor ezek még a gyorsítótárból (cache) töltődnek be és frissítsük az oldalt úgy, hogy ne vegye figyelembe a böngésző a gyorsítótárat, hanem kérjen le mindent újra szerverről (**Ctrl + F5** funkcióbillentyű megnyomásával). Ezután már a megadott angol nyelvű hibaüzeneteknek kell megjelennie a validálási problémáknál.

9.1.3.3. Támadás a JavaScript-es validáció ellen

A támadó akár az oldal forráskódjában is láthatja, illetve a Vizsgálóban (Inspector) is megtalálhatóak az imént írt JavaScript kódjaink. Így érzékelheti például, hogy a **title** mezőre vonatkozik egy JavaScript-es ellenőrzés. Utána megteheti, hogy mivel a JavaScript kódban az **id** attribútum alapján keresünk rá a mezőre, ezért a Vizsgálóban valós időben átírja a **title** mezőnél az id attribútum értékét például arra, hogy *„new-title”,* átírja még a mezőben a **minlength** értékét 1-re, így – ha csak kliens oldali validációnk van – be is tudja küldeni az 1, 2, 3, 4 karakterhosszúságú címmel rendelkező blogbejegyzést is eltárolásra. Szerver oldalon kivételt fog kapni természetesen, de ezzel a későbbiekben foguk foglalkozni.

9.1.4. JavaScript alapú osztálykönyvtárak a validáláshoz

Ahogy a CSS kapcsán ismert, hogy lehet, sőt érdemes keretrendszereket használni, úgy mint a Bootstrap vagy a Tailwind keretrendszereket, itt is erről van szó. JavaScript-es osztálykönyvtárakkal és keretrendszerekkel rengeteg van, kicsit talán nehéz is eligazodni köztük. Most kigyűjtöttem néhányat, amik űrlapok ellenőrzésére szolgálnak leginkább. Abban a kérdésben, hogy melyik a jobb / rosszabb, én biztosan nem foglalnék állást, érdemes kipróbálni egyet-kettőt közülük, és ami jobban *„kézre áll”,* azt lehet utána alkalmazni. Általában elmondható, hogy az alap funkcionalitásokra, ellenőrzésekre mindegyik alkalmas, ha pedig valamelyik osztálykönyvtár/keretrendszer fejlesztői kitalálnak valami egyedit, jót, akkor azt utána gyorsan lemásolják a többi fejlesztői is.

Néhány szempont, amelyek alapján érdemes megvizsgálni őket, és választani közülük:

1. Mivel egy nagy alkalmazásnál biztosan sok JS csomagot használunk, érdemes figyelni a validációs csomag méretére (minél kisebb legyen).
2. Mikor frissítették utoljára: ha már régebben érkezett hozzá frissítés, akkor nem biztos, hogy érdemes ezt használni, mert esetleg a fejlesztői már nem gondozzák tovább, ami problémás működéseket szülhet a jövőben.

3. Nézzünk meg példákat, próbáljuk ki a *„szimpatikusakat”,* hogy mennyire áll nekünk kézre az alkalmazásuk, mert ha nem logikus számunkra vagy nehezen használható, akkor annak az alkalmazása a későbbiekben csak kellemetlenséggel jár majd.
4. Van-e hozzá hivatalos vagy támogató weboldal: dokumentáció, mintapéldák stb.
5. Mennyi és milyen további függőségeket tartalmaz még. Nyilván, minél kevesebb egyéb függőséget tartalmaz, annál inkább jobb lehet nekünk (kisebb eséllyel lesz problémás a használata egyéb függő csomagok miatt).

Itt egy lista, amiben direkt nem számozott az elemeket, mert érdemes megfontolni az iménti szempontokat, hogy mi szerint érdemes választani ezek vagy éppen más hasonlók közül:

- https://formvalidation.io/
- https://just-validate.dev/
- https://validatejs.org/ itt lehet tesztelni is: https://rickharrison.github.io/validate.js/
- https://pristine.js.org/
- https://github.com/mattkingshott/iodine
- https://bootstrap-validate.js.org/ és ez is biztos nagyon hasznos lesz majd, ha a Bootstrap 5-ös verziójához is lesz már támogatás.

A Laravel kiváló backend-ként tud szolgálni Single Page Application-ökhöz, így meglehetősen jól tud együttműködni a Vue.js és React alkalmazásokkal (akár az Inertia JS-en keresztül). Ezeknek a JavaScript alapú keretrendszereknek jól kiforrott és hatékonyan használható validációs technikái vannak, amelyeket érdemes alkalmazni a fejlesztés során.

Az alfejezet során végrehajtott programkód módosítások ebben a GitHub commit-ben érhetők el.

A felsoroltak közül a Just Validate osztálykönyvtárat én magam is kipróbáltam és több ellenőrzést, tesztelést is futtattam vele. A telepítési, használati tapasztalataimat egy blogbejegyzésemben foglaltam össze: https://attila.gludovatz.hu/posts/komplex-pelda-validalas-3-resz

Ha pedig valaki a Vue.js-t és az Inertia JS-t szeretné együttesen használni, neki javaslom a Laracasts kiváló oktatójának, Jeffrey Way-nek a bemutató videósorozatát, amely itt érhető el ingyenesen: https://laracasts.com/series/build-modern-laravel-apps-using-inertia-js

9.2. Szerver oldali validáció (Server-side validation)

Ha érvényes adatokkal szeretnénk dolgozni, akkor a szerver oldalon is mindenképpen validálnunk kell a felhasználótól érkező adatokat.

Tipp: a szerver oldali validáció könnyű és hatékony beépítéséhez, kezeléséhez, ellenőrzéséhez iktassuk ki (helyezzük el a nyitó **<form>** tag-ben a **novalidate** attribútumot) vagy szüntessük meg a kliens oldali validációt a **submit** input mezőben elhelyezett **formnovalidate** attribútummal. Így ténylegesen a szerver oldali ellenőrzésre tudunk koncentrálni.

Bármelyik erőforrásunkról is legyen szó, a szerver oldali validáció a CRUD folyamatok közül a létrehozást és a frissítést, vagyis a **store()** és **update()** vezérlő metódusainkat fogja érinteni a Controller-ekben. Sőt, általában nagyon hasonlóan (vagy egyformán) kell a két metódusban érvényesíteni az érkező adatokat.

Kezdjük a szerver oldali validáció gyakorlati megvalósítását a kategóriáknál, a létrehozó űrlapnál. Itt kliens oldali validáció hiányában könnyedén el tudjuk menteni az üresen hagyott *„Name"* bemeneti mezővel az űrlapot. Kapunk viszont rögtön egy szerver oldali kivételt, mivel a **name** mezőnek a **categories** adattáblában nincsen alapértelmezett értéke (ez azt eredményezi az alábbi INSERT INTO utasításban, hogy a VALUES részben **?** kerül a **name** mező helyére - üres lesz, nem kap értéket -, ez okozza a hibát).

9–7. ábra: Szerver oldali kivétel - validáció nélkül

9.2.1. Beépített szabályok alkalmazása és a validációs folyamat

De igazából ez is volt az elvárt működés ebben az esetben. Úgyhogy segítsük a felhasználót most szerver oldali validációval! Nyissuk meg a **CategoryController**-t, és koncentráljunk a **store()** metódusára, ahol el kell helyeznünk a szabályokat, amiket be kell tartani. A metódus magjának kezdetén helyezzük el az ellenőrzést!

```
request()->validate([
  'name' => 'required|max:255|unique:categories'
]);
```

9–9. kódrészlet: name mező ellenőrzése a Laravel beépített validációs szabályaival

Az űrlap elküldésével érkező **name** mező értékének több szabályt is be kell tartania, meg kell felelnie nekik. A **required**, ugyanúgy, ahogy a kliens oldali validációnál is tapasztaltuk, a kötelező kitöltésre vonatkozik, a **max:255** a maximális szöveghosszúság értékét szabályozza, míg a **unique:categories** azt ellenőrzi, hogy a **categories** tábla **name** mezőjének értékeiben egyedi-e az újonnan érkező adat. A szabályokat „|" karakterrel kötjük össze és mindegyiknek meg kell felelnie. Ha mégis szükségünk lenne a szabályok megfogalmazásánál a | különleges karakterre, akkor alakítsuk át ezt a string-et egy tömbbé, így:

['required', 'max:255', 'unique:categories']

A két megadási forma ugyanazt a validációs logikát használja, nincs különbség köztük.

A Laravel számos beépített validációs szabály használatát támogatja egyszerűen, ezekről érdemes áttekinteni a listát és a mintapéldákat is itt: https://laravel.com/docs/10.x/validation#available-validation-rules (*Megjegyzés:* természetesen az összes szabályt nem fogja taglalni ez a könyv, csak a számunkra érdekesekre fogunk kitérni itt, de ezek alapján bármelyik Laravel dokumentációban bemutatott szabály használata elsajátítható lesz az olvasók számára.)

Ha most elküldjük a kategóriát létrehozó űrlapot üresen, vagy 255-nél hosszabb névvel, esetleg ugyanolyan kategória névvel, mint ami már létezik a **categories** adattábla **name** mezőjében, akkor az

oldalon *látszólag nem történik semmi, de újra töltődik úgy, hogy a szerver oldali validáció elbukott,* így nem történt meg az új kategória felvétele. A programlogikában, a validáció elbukásának következtében tehát végre sem hajtódott a **Category::create()** utasítás, el sem jutott addig a program futása. A szerver oldali validáció megtörténik, de mivel nem teljesül valamelyik validációs feltétel, ezért visszadobja a kliens oldalnak a keretrendszer, hogy hiba van: jelen esetben nem töltöttük ki a kategória nevét megfelelően. Viszont a kliens oldal (**create** nézet) nincs felkészítve még arra, hogy megmutassuk a visszadobott hibákat. Folytassuk ezzel a munkát, szúrjuk be a **form** komponens nyitó tag-je elé ezt:

```
@if ($errors->any())
  <div>
    <ul>
    @foreach ($errors->all() as $error)
      <li>{{ $error }}</li>
    @endforeach
    </ul>
  </div>
@endif
```

9–10. kódrészlet: Szerver oldali validációs hibák listázása az űrlapoknál

Így az űrlap felett ki tudunk listázni, minden olyan hibát, ami a kitöltésnél validációs szabályba ütközött. Figyeljünk mindig a következetességre, névkonvenciókra, hatékonyságra is. Így ez a feltételvizsgálat kiszervezhető egy olyan résznézetbe, ami aztán majd az összes erőforrással kapcsolatos űrlapunk felett elhelyezhető lesz. Hozzunk létre a **resources / views / includes** mappában egy **_errors.blade.php** nevű nézet fájlt, és illesszük bele a 9–10. kódrészletet. Utána pedig a **categories.create** nézetben a **@if-@endif** szekció helyére illesszük be ezt:

```
@include('includes._errors')
```

9–11. kódrészlet: Szerver oldali validációs hibák listázása az űrlapoknál résznézet beemelésével

Ha átmegy a validáció, megfelelünk a szabályoknak, akkor a 9–9. kódrészletben látható utasítás egy már érvényes tömbbel tér vissza, amit így át is adhatunk a **Category::create()** utasításnak.

```
public function store()
{
  $validated = request()->validate([
    'name' => 'required|max:255|unique:categories',
  ]);

  Category::create($validated);

  return redirect(route('categories.index'));
}
```

9–12. kódrészlet: Érvényes tömbelemek átadása létrehozáshoz

Ahogy jeleztem, ugyanerre a szabályokat tartalmazó csoportra van szükség a frissítés során is, így érdemes kiszervezni a validációt egy külön metódusba. Alább látható a segédmetódus, ami a validációt elvégzi, továbbá a módosított **store()** és **update()** metódusok, amelyek ezt a validációt használják:

```
function validateCategory() : array {
  return request()->validate([
    'name' => 'required|max:255|unique:categories',
  ]);
}

public function store()
{
  Category::create($this->validateCategory());
  return redirect(route('categories.index'));
}

public function update(Category $category)
{
  $category->update($this->validateCategory());
  return redirect(route('categories.index'));
}
```

9–13. kódrészlet: Kiszervezett validálás és kód újraszervezés a CategoryController-ben

Ezen módon végrehajtottuk a *kód újraszervezést (refactoring),* amellyel így továbbra is megfelelően működik az alkalmazásunk, viszont kevesebb kódsorral és a hibákat elkerülve (nincs különböző validáció a metódusokban). *A kód tökéletesen olvasható, így a magyarázó kommentek használata is csökkenthető, hiszen a kód öndokumentáló.*

A validálási metódusnak adhatunk második paramétert is (szintén tömb), így az egyes szabályokhoz saját egyedi hibaüzeneteket határozhatunk meg.

```
request()->validate([
  'name' => 'required|max:255|unique:categories',
], [
  'required' => 'The Category\'s name is required.',
  'max' => 'The Category\'s name could be maximum :max characters.',
  'unique' => 'The Category\'s name must be unique in categories table.',
]);
```

9–14. kódrészlet: Validálási egyéni hibaüzenetek beállítása

A hibaüzeneteknél is működik a testre szabás, akár paraméterezéssel, így a **:max** paraméterrel a konkrét számra (255) tudunk hivatkozni, amit beállítottunk a szabálynál. Sőt, akár többnyelvűsítetten is meg tudunk ide határozni üzeneteket (lásd majd a 15.2. alfejezetet).

A hibaüzeneteket nem csak összesítve tudjuk kiíratni, hanem lehetőségünk van arra is, hogy egy-egy mezőnél közvetlenül írjuk oda, hogy milyen validációs probléma van vele. A **categories.create** nézetben lévő űrlapnál az input mező alá helyezzük el az alábbi kódrészletet:

```
@error('name')
  <span style="color: red;">{{ $message }}</span>
@enderror
```

9–15. kódrészlet: Input mezőhöz rendelt validációs hibaüzenet kiírása

Ezzel egy új **@error** Blade direktívát ismerünk meg, ami a **@selected**-hez hasonlóan valójában egy feltételvizsgálatot rejt magába, és azt vizsgálja meg, hogy az adott mezőnévhez (itt most a **name**) van-e validációs probléma, tehát megfelel ennek a logikának: **@if ($errors->has('name'))**. Ha van probléma, akkor kiírjuk a **$message** alapértelmezetten használható változóval a hibaüzenetet.

Vegyük észre, hogy ez a 9–15. kódrészlet nagyon könnyedén általánosítható, tehát komponens készíthető belőle. Készítsük is ezt el, hogy utána már mindenütt komponenseket használhassunk csak a mezők melletti validációs hibaüzenetek megjelenítésénél. Hozzuk létre a **resources / views / components** mappában az **input-error.blade.php** fájlt ezzel a tartalommal:

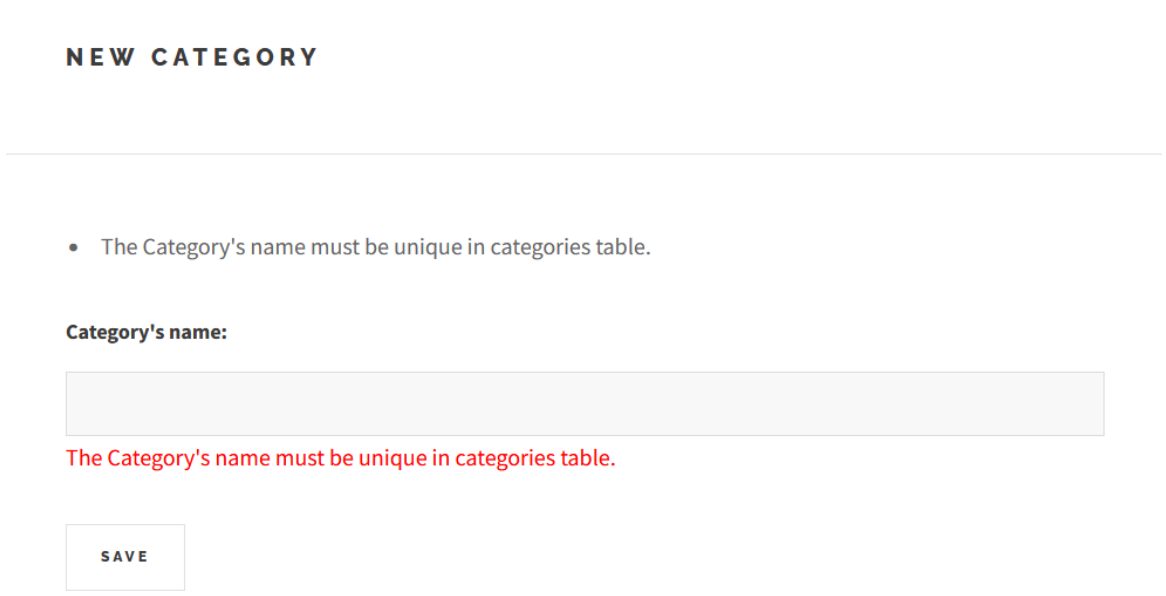
```
@props(['for'])

@error($for)
  <span {{ $attributes->merge(['style' => 'color: red;']) }}>{{ $message
}}</span>
@enderror
```

9–16. kódrészlet: Validációs hibaüzenet komponenssé alakítása

Utána a kategóriát létrehozó **create** nézetben már csak használnunk kell ezt a komponenst, az input mező alatti sorba szúrjuk be ezt:

```
<x-input-error for="name" />
```

9–17. kódrészlet: Input-error komponens felhasználása a categories.create nézetben



9–8. ábra: Validációs hibaüzenet megjelenítése a tesztelés során

Ez a tesztelés rögtön rá is világít egy olyan problémára, hogy ha már létező kategóriát akarunk létrehozni, és megkapjuk a hibaüzentet, akkor eltűnik a beírt kategória név. Erre a Laravel ad egy **old()** segédmetódust nekünk, amelyet az input **value** attribútumába helyezhetünk el.

```
<input type="text" name="name" id="nev" value="{{ old('name') }}">
```

9–18. kódrészlet: Az old() segédmetódussal tudjuk a korábbi - elküldés előtti értéket kinyerni

Így most már akkor is megmarad a bemeneti mező értéke, ha valamilyen szerver oldali validációs hibát ejtettünk. Ez főleg akkor nélkülözhetetlen, ha egy nagy űrlappal dolgozunk, és a felhasználónak elég nagy bosszúsággal járhatna, ha elvesznének a jól beírt adatai, ha valahol hibát vétene, akkor az egész kitöltött űrlapja törlődne újra, egyetlen hiba miatt is akár.

Az elvégzett módosításokat a **categories.edit** nézetben is hajtsuk végre a **create** nézethez hasonlóan (**include, x-input-error**), viszont az imént említett **old()** segédmetódust itt egy kicsit másképp kell használni, hiszen az input bemeneti mezőnek alapból van egy értéke, amit el kell helyezni benne.

```
<input type="text" name="name" id="nev" value="{{ $category->name ??
old('name') }}">
```

9–19. kódrészlet: Bemeneti mező értéke az edit nézetben

Így most már vagy a **??** előtti változó kerül bele a mezőbe, vagy ha nem létezik, akkor a **??** utáni érték. Az pedig egy döntés kérdése, hogy ebben az esetben melyik értéket hova helyezzük a kérdőjelek körül.

Az alfejezetben történt programkód módosításokat ebben a GitHub commit-ben lehet megtalálni.

9.2.2. Validáció a kapcsolatok kezelésében

Még mindig maradunk a beépített szabályok alkalmazásánál, de most kapcsolati szempontból fogjuk megvizsgálni a működésüket: 1-1, 1-n és n-n kapcsolatok lekezelésének szempontjából. Mindegyik kapcsolattípus megtalálható a blogbejegyzéseknél, úgyhogy ezekre fogunk koncentrálni. Ismét a **store()** és **update()** vezérlő metódusok lesznek számunkra a relevánsak a **PostController**-ben. Kezdjük a **store()**-ral, de előtte vegyük ki a kliens oldali validációt a **posts.create** nézetben lévő űrlapból úgy, hogy a **form** komponenshez hozzáadjuk a **novalidate** kulcsszót, így, bár a mezőknél fogja jelezni a piros kerettel, hogy érvénytelenek (**invalid**) az adott mező, de engedni fogja az elküldésüket, a mentést, ami után szerver oldali **exception**-t kapunk (ahogy a 9–7. ábra is mutatja). A **PostController store()** metódusában ilyen sorrendben vannak az utasítások:

1. Blogbejegyzés (**Post**) létrehozása egy kategóriához (**Category**); 1-n kapcsolat.
2. Blogbejegyzéshez tartozó értékelés (**Rating**) létrehozása; 1-1 kapcsolat.
3. Blogbejegyzéshez tartozó címke kapcsolatok létrehozása; n-n kapcsolat.
4. Visszairányítás a blogbejegyzések listájára.

De előtte validáljuk a bemeneti mezők értékeit, ehhez segítségünkre lehet a **posts** adatbázis tábla és mezőinek kényszerei, illetve maga a **posts.create** nézetben lévő létrehozási űrlap is.

```
function validatePost() : array
{
  return request()->validate([
    'title' => 'required|min:5|max:100',
    'slug' => 'required|max:255|regex:/^[a-z]+[-]{1}[a-z]+$/',
    'body' => 'required',
    'published_at' => 'required',
    'score' => 'required|numeric|between:1.0,10.0',
    'category_id' => 'required|exists:categories,id',
    'tags' => 'required|exists:tags,id',
```

```
    ]);
}
```

9–20. kódrészlet: Blogbejegyzés elemeinek validálása

Érdekesség: A validációs szabályok összeállítása történhet az adattábla szerkezet és kényszerek, vagyis a Laravel-ben a migrációs fájl alapján. Ez egy aránylag jól általánosítható folyamat, ha nincsenek egyedi szabályaink (lásd majd a következő alfejezetet). Készítettek ezért egy kiegészítő csomagot a Laravel-hez, ami a migrációs fájlok alapján legenerálja számunkra a validációs szabályokat. A csomag itt érhető el: https://github.com/laracraft-tech/laravel-schema-rules (azt sose feledjük, hogy minden új csomag több függőséggel is jár, illetve magának a projektnek a mérete is megnövekedik a telepítésük által).

A csomag használatát ebben a videóban mutatják be: https://youtu.be/cuSalYyU5lU

Magát a validációt ne szedjük szét külön blogbejegyzésre és értékelésre, hanem együttesen kezeljük őket, emiatt egy kicsit másképp kell majd a **store()** metódusban felhasználni a **validatePost()** visszatérési tömbjét. De előbb koncentráljunk az érdekesebb validációs szabályokra:

- **slug:** egy mintaillesztést, **regex** utasítást használhatunk itt: a **/^** és **$/** jelek közé bemásolhatjuk a create űrlapról származó mintaillesztést, így ez ugyanazt fogja takarni, mint ott: két szöveg, köztük pedig egy kötőjel;
- **score:** ez ugye a **ratings** táblában kerül majd elhelyezésre, viszont egy lebegőpontos szám (double), amit a **numeric** ellenőrzéssel tudunk ellátni, a **between**-nel pedig kiváltható a **min** és **max** külön-külön történő használata;
- **category_id** és **tags:** külső kulcsok ellenőrzése az **exists** validációs szabállyal, ami paraméterként először a kapcsolódó tábla nevét kapja meg, utána pedig a mező nevét, amire hivatkozik. Ez a szabály ugyanúgy működik 1-n és n-n kapcsolatokkal is.

Egyéni hibaüzenetek beállítására itt is van lehetőségünk (a **validate()** metódus második paraméterében), viszont most már nem egy darab mezőnk van, ezért nem elég, ha simán csak arra hivatkozunk a kulcsnál, hogy például **required**, hanem a mező nevét is oda kell írni elé és ponttal elválasztani, például így **title.required** (űrlap mezőnév után pont majd a szabály neve).

Szúrjuk be a **store()** és **update()** metódusaink legelejére a validáló metódus meghívását, a többi részt hagyhatjuk változatlanul:

```
$this->validatePost();
```

9–21. kódrészlet: PostController store() és update() metódusainak elején a validálás

Ha a validálás elbukik, akkor úgysem lépünk tovább a létrehozásokra/frissítésekre.

A **store()** és **update()** metódusokban még egy-egy *kód újraszervezést* végrehajthatunk a validálások után. A **request()** segédmetódusban, ha ugyanazok a mezőnevek szerepelnek, mint amik az érintett adattáblában vannak, akkor nem szükséges kulcs-érték párokkal felsorolni, hogy melyik mezőbe milyen bemenetről érkező értéket töltsön be, hanem simán működik egyszerűbben is. A **store()** esetében így:

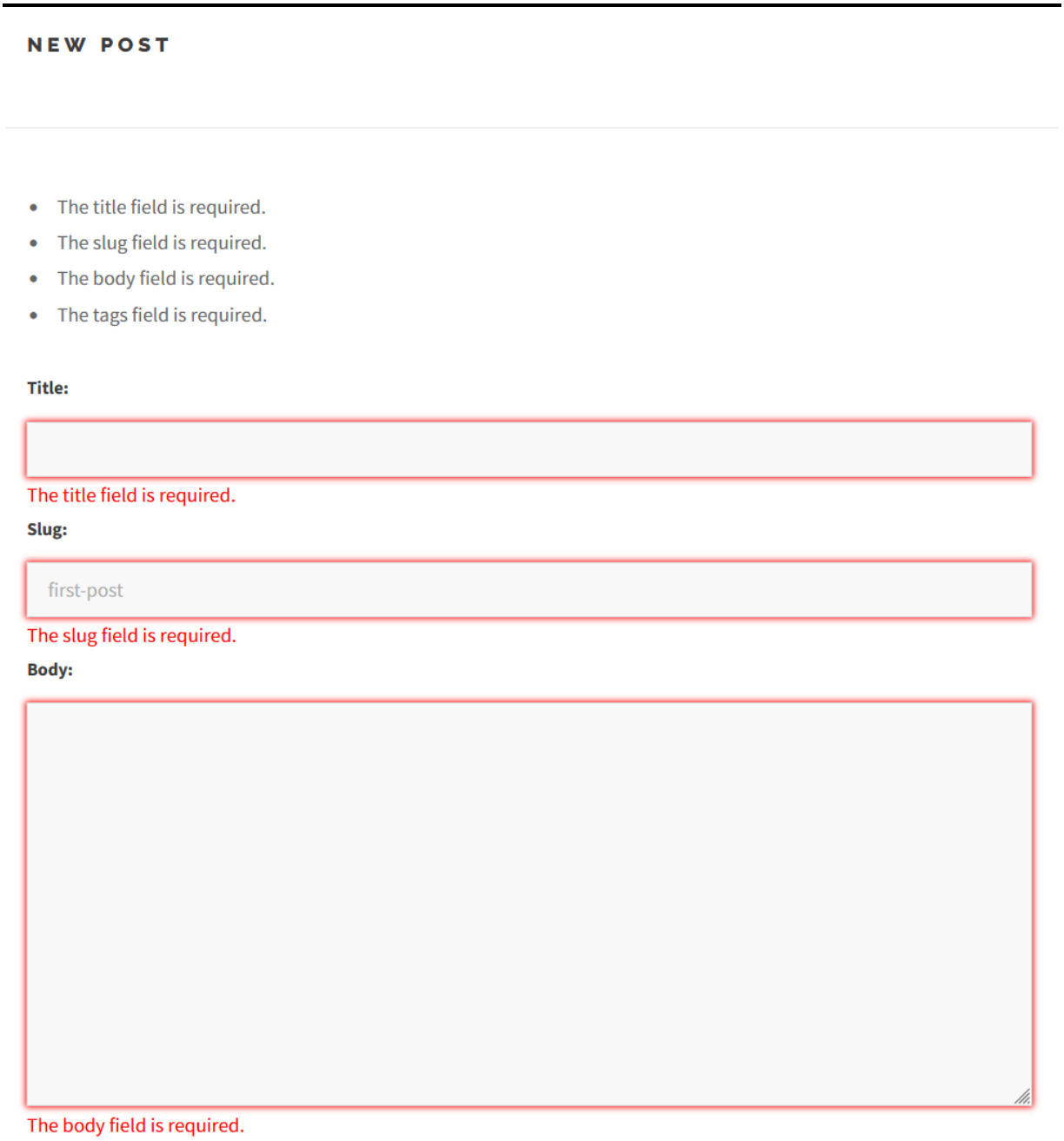
```
$post = Post::create(request(['title', 'slug', 'body', 'category_id',
'published_at']));
```

9–22. kódrészlet: Blogbejegyzés létrehozásának egyszerűsítése a PostController store() metódusában

Az **update()**-nél pedig szintén ugyanígy kell működnie, csak a **Post::create()** helyett az **$post->update()** metódusa kapja meg paraméterül a **request()** több paraméteres hívását majd visszatérését.

A szerver oldali validáció tesztelését a **posts.create** nézetben kezdjük meg, de előtte még helyezzük el (**include**-oljuk) az összes hibát listázó résznézetet az űrlap komponens elé, illetve a mezőhöz tartozó hibát leíró üzenetet (komponenst) a mezők után közvetlenül. A két említett kódsor a 9–11. kódrészlet és 9–17. kódrészlet. Illetve használjuk még fel az **old()** segédmetódust arra, hogy ne vesszenek el a beírt/kiválasztott értékeink akkor sem, ha a szerver oldali validáció elbukik (lásd 9–18. kódrészlet). Eltérő lehet azonban az **old()** segédmetódus felhasználásának a helye és miként:

- Egyszerű szöveges (text, email, number stb.) input mezőknél a **value** attribútumban, például így: 9–18. kódrészlet.
- A **datetime-local** típusú bemeneti mezőnél *„most”*-ra van beállítva az alapértelmezett érték, amit egy kicsit módosítsunk eszerint:
  - **value="{{ old('published_at') ?? now()->format('Y-m-d H:i') }}"**
- A **textarea** mezőnél a nyitó- és zárótag közé, például így:
  - **<textarea name="body">{{ old('body') }}</textarea>**
- A **select** komponenseknél a **:selectedValues** egyedileg meghatározott attribútumába, például így (és ugyanígy működik a **tags** nevű mező kapcsán is):
  - **:selectedValues="collect(old('category_id'))"**

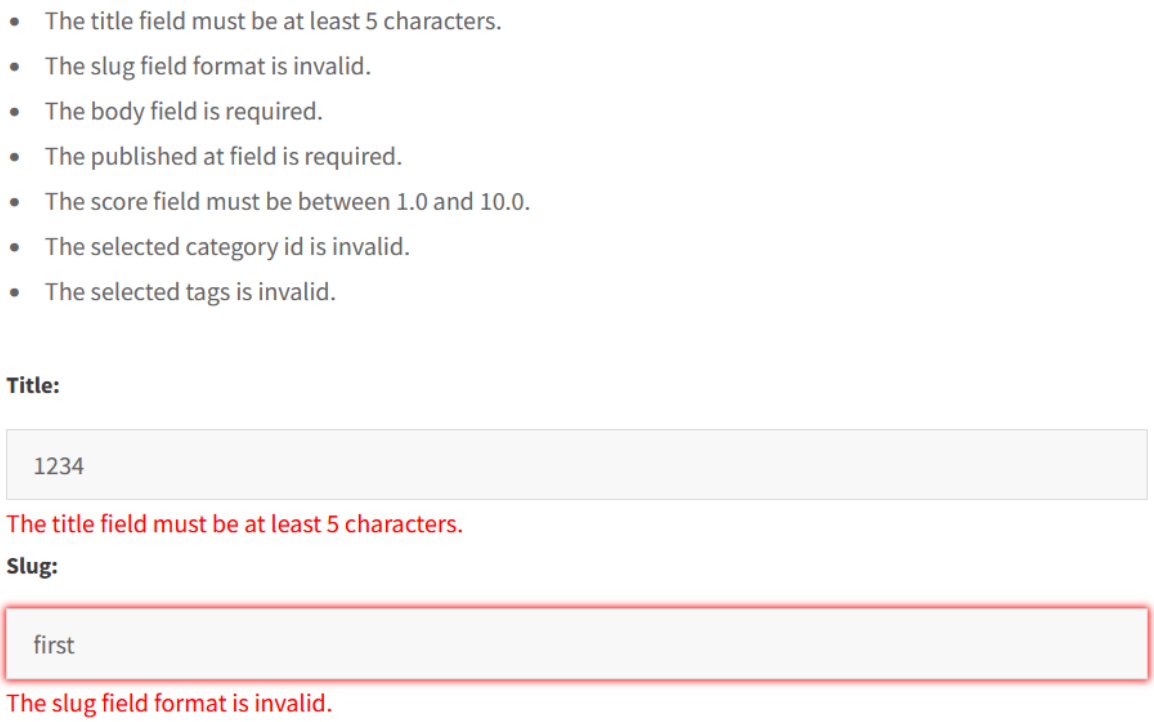


9–9. ábra: Kötelező mezők validációs hibáinak listázása az űrlap felett és a mezőknél

Szabályok megsértésének tesztelését még az alábbiak szerint tudjuk megtenni:

- Adjunk meg egy 4 karakterhosszúságnál rövidebb címet a **title** mezőbe;
- A **slug** mezőbe ne írjunk kötőjelet;
- A **body**-t nem nagyon tudjuk máshogy *„elrontani”;*
- A **category select**-et vizsgáljuk meg a böngészőben, és a kiválasztott **option value** értékét írjuk át egy jó nagy számra, ami biztosan nincsen még a **categories** tábla **id** mező értékei között;
- A **score**-t írjuk át manuálisan például 0.5-re;
- A **published_at** mezőben a másodperc értékre kattintsunk rá és nyomjunk **delete**-t (töröljük ki);
- A **tags select**-ben (a **category**-hoz hasonlóan) válasszunk ki egy értéket, vizsgálóban pedig írjuk át az **option value** értékét egy nagyon nagy számra, ami biztosan nincsen a **tags** táblában.

A hibázásoknak köszönhetően ennyi szerver oldali szabályszegést tudtam kicsikarni a rendszerből:



9–10. ábra: Mezők változatos validációs hibáinak listázása az űrlap felett és a mezőknél (részlet)

Az alfejezet elején lévő számozott felsorolásból is látszik, hogy *tranzakcióba* kellene szervezni a lefutásokat, ugyanis, ha például az értékelés létrehozásakor keletkezik validációs hiba, akkor az jelenleg a blogbejegyzés létrehozása utána lesz már, amihez így egy érvénytelen értékelés kerülne, amit a validáció nem fog majd engedni. *A tranzakcióra itt, mint egy adatbázis műveletsorozatra kell gondolni, amely atomi utasításnak tekinthető, tehát vagy az összes tranzakcióban lévő utasítás végrehajtódik hibamentesen, vagy egyik sem, ha valamelyiknél valamilyen hiba merülne fel.*

```
DB::transaction(function () {
  $post = Post::create(request(['title', 'slug', 'body', 'category_id',
'published_at']));

  Rating::create([
    'score' => request('score'),
    'post_id' => $post->id,
  ]);

  $post->tags()->attach(request('tags'));
});
```

9–23. kódrészlet: Tranzakció definiálása a store() metódusban

```
DB::transaction(function () use ($post) {
  $post->update(request(['title', 'slug', 'body', 'category_id',
'published_at']));

  $post->tags()->sync(request('tags'));
```

```
  Rating::where('post_id', $post->id)->update(['score' =>
request('score')]);
});
```

9–24. kódrészlet: Tranzakció definiálása az update() metódusban

A **DB Facade**-ot importálni kell hozzá a fájl elején.

A blogbejegyzések **edit** nézetét is érdemes kiegészíteni a **create** nézetnél felhasznált elemekkel (HTML5 validálási attribútumok a tag-ekben, validációs hibajelzések, illetve a korábbi értékek visszatöltését validációs hiba esetén). Előbbi kettő könnyebben megvalósítható, utóbbinál egy döntésre van szükség, hogy hogyan jelenjenek meg a korábbi értékek: én amellett tettem le a voksomat, hogy egy validációs hiba miatt a beírt (nem érvényes) értékek maradjanak meg, és ne az adatbázisban addig eltárolt érték, így nálam így néz ki például a **title** mezőben a **value** attribútum: **{{ old('title') ?? $post->title }}**

Így pedig a **select** komponensben: **:selectedValues="collect(old('category_id') ?? $post->category_id)"**

De tovább már nem részletezem, ha valamit nem sikerülne megcsinálni önállóan, akkor érdemes megnézni az alfejezetben található programkód módosításokat ebben a GitHub commit-ben.

Feladat: amennyiben úgy gondoljuk, hogy a beépített validációs szabályokkal megvalósítható a további erőforrások (**Tag, Comment**) validálással kibővített feltöltési és szerkesztési folyamata, akkor érdemes gyakorlásként kibővíteni azok Controller-jeiben a **store()** és **update()** metódusokat, illetve kiegészíteni a **create** és **edit** nézetek űrlapjait is. Az összesített validációs hibalistát tartalmazó résznézet pedig áthelyezhető a **form** komponensbe (így azonban a szerkesztési nézetben a törlési gomb felett is meg fog jelenni a hibalista, hiszen ez is a **form** komponenst használja). A módosításokat tartalmazó GitHub commit itt érhető el.

9.2.3. Egyedi validálási folyamat

Ha a **$request** objektumnak vagy a **request()** segédmetóduson keresztül érkező adatoknak a validálása elbukik, akkor rögtön visszaadja a kérést a kliens oldalnak, és jelezhetjük a nézetben a felhasználónak, hogy valamilyen validációs hiba történt. Előfordulhat azonban, hogy nem szeretnénk itt rögtön befejezni a folyamatot szerver oldalon, hanem még egyéb műveleteket is el szeretnénk végezni, vagy éppen egy másik oldalra irányítani a felhasználót, nem oda, ahonnan érkezett a kérés, például a **create** nézet helyett az **index** oldalra is visszaküldhetjük különböző információkkal ellátva a választ és ezáltal a felhasználót.

Lehetőségünk van arra, hogy a **Validator** (**Facade**-ot) alkalmazzuk. A működés itt is nagyon hasonló lesz, csak itt a **Validator** osztály **make()** metódusát használjuk az ellenőrzésre. Ez a metódus négy paramétert kaphat:

1. paraméter *(kötelező megadni):* a felhasználótól érkező bemeneti adatok, ezeket ugyanúgy mint korábban, most is lehet őket kezelni a **$request** objektummal vagy a **request()** segédmetódussal;
2. paraméter *(kötelező megadni):* a validációs szabályokat tartalmazza;

3. paraméter *(opcionális):* a validációs szabályokra adott alapértelmezett hibaüzeneteket tudjuk itt felüldefiniálni;
4. paraméter *(opcionális):* a validációs hibaüzenetekben attribútumokat helyezhetünk el (mint a **min**, **max** stb. esetekben már láthattuk). Ebben a 4. paraméterben ezeket az attribútumokat szabhatjuk testre, változtathatjuk meg.

Ennek segítségével gyakorlatilag mi magunk tudjuk lekezelni, hogy mi történjen akkor, ha hiba történik a szerver oldali validáció során, így egy fokkal jobban testre szabható megoldáshoz jutunk, ha nem csak arra van szükségünk, hogy visszaadjuk a hibákat a felhasználónak.

A gyakorlati példa bemutatásához a **CommentController update()** metódusát használom, amelyben definiálom a **Validator** Facade **make()** metódusának magját az említett paraméterekkel, majd egy feltételvizsgálattal ellenőrzöm, hogy elbukik-e a validálás, és ha igen, akkor a **comments.index** nézetre irányítom a felhasználót a hibával együtt.

```
public function update(Comment $comment)
{
  $validator = Validator::make(
    request()->all(),
    [
      'username' => 'required|max:255',
      'content' => 'required',
      'post_id' => 'required|exists:posts,id',
    ],
    [
      'post_id.exists' => 'Missing :attribute in posts table'
    ],
    [
      'post_id' => 'post identification'
    ]
  );

  if ($validator->fails()) {
    return redirect(route('comments.index'))
      ->withErrors($validator);
  }

  $comment->update(request()->all());

  return redirect(route('comments.index'));
}
```

9–25. kódrészlet: Validator Facade működése a CommentController update() metódusában

Ehhez persze a **comments.index** nézeten meg kell jeleníteni a hibát, amit a korábban bemutatott: **@include('includes._errors')** utasítással tudunk megtenni a listázó oldalon is.

A tesztelést úgy tudjuk végrehajtani, ha szerkesztésre megnyitunk egy kommentet, a böngésző vizsgálója segítségével átírjuk a kiválasztott blogbejegyzés azonosítóját a **select**-ben, egy olyan **option value** értékre,

ami nem létezik a **posts** adattáblában, majd az Update gombra kattintunk. Így megkapjuk a **comments.index** nézet oldalán a *„Missing post identification in posts table"* egyedileg definiált hibaüzenetet.

Az alfejezetben található programkód módosítások ebben a GitHub commit-ben találhatók meg. További erőforrásoknál egyelőre nem kell végrehajtani a validálási folyamat átalakítását, elegendő a már meglévő működést alkalmazni náluk.

9.2.4. Egyedi szabályok érvényesítése

Bár a Laravel beépített validációs szabályaival a legtöbb lehetőséget le tudjuk fedni (majdnem minden validációs ellenőrzést), amely problémaként előfordulhat a beérkező adatokkal kapcsolatban, mégis megtörténhet, hogy nekünk valamilyen egyedi szabályra lenne szükségünk: például, ha egy megadott születési dátumból kellene kiszámolni, hogy elmúlt-e már 18 éves a weboldalunk látogatója.

Megjegyzés: a 18 éves korra is könnyedén tudunk már beépített validációs szabályt használni. Ha van például egy **date_of_birth** mezőnk, akkor a következő szabályt kellene hozzárendelni: **'before_or_equal:-18 years'** (ha egy év lenne, akkor egyes számba kellene tenni a **years**-t, tehát **year** lenne). A Laravel a már beépített Carbon csomagnak köszönhetően tudja értelmezni az ilyen módon megfogalmazott és futási időben kiszámított dátumokat.

Mivel az alkalmazásunk még nem túlságosan bonyolult, ezért egy olyan gyakorlati példát hozok az egyedi szabályokra, amelyet beépített szabállyal is meg lehetne oldani, viszont itt most áttekintjük a folyamatát annak, hogy hogyan lehet egyedi szabállyal megvalósítani a validálást.

Tehát a szabály, amit megvalósítunk: egy blogbejegyzés publikálási dátuma nem lehet jövőbeli esemény a létrehozás és a frissítés során sem. Beépített validációs szabállyal ezt könnyedén megfogalmazhatjuk a **PostController validatePost()** metódusában a **published_at** mező **required** kényszere után egy **|** karakterrel elválasztva. A publikálási dátum nem lehet jövőbeli dátum, tehát a mai nap vagy korábbi, ezt a **before_or_equal:today** szabállyal definiálhatjuk, vagy pedig a **before:tomorrow** (holnapnál korábban) is működik.

Egy egyedi szabály létrehozásához szükségünk van egy új osztályra, amely magát a szabályt fogja tartalmazni:

```
php artisan make:rule PostPublishedAt
```

Az utasítás kiadásának hatására létre is jött egy új fájl az **app / Rules** mappában (a mappa is új): **PostPublishedAt.php** (nincs konvenció, amely elvárná, hogy a *„Rule"* szót is hozzáfűzzük a név végéhez).

Ha megnézzük a fájlt, akkor egy **validate()** metódus látható benne. A metódus három paramétert kap meg: **$attribute**: ez a beérkező adat nevét szimbolizálja, a **$value** pedig ennek az értékét és a **$fail** callback metódust, amelyet arra használhatunk, hogy ha elbukik a validáció, akkor a neki átadott hibaüzenet paraméterrel tér vissza, az attribútum nevével együtt.

```
public function validate(string $attribute, mixed $value, Closure $fail):
void
{
```

```
  if( date('Y-m-d', strtotime($value)) > now()->format('Y-m-d') ) {
    $fail('The date of :attribute field should be today or earlier than
today');
  }
}
```

9–26. kódrészlet: Egyedi szabály a blogbejegyzés publikációs dátumára vonatkozóan

Ez azonban még nem elég, mivel itt még csak definiáltuk a szabályt és megadtuk a hibaüzenetet, hogy mit küldjön vissza a felhasználónak, ha megsérti. A szabályt hozzá is kell adnunk a **PostController validatePost()** visszatérési tömbjénél a **published_at** mezőhöz:

```
'published_at' => ['required', new PostPublishedAt],
```

9–27. kódrészlet: Egyedi szabály hozzárendelése a published_at mezőhöz

Vegyük észre, hogy most már nem | karakterrel vannak elválasztva a szabályok, hanem egy tömbbe foglaljuk őket, mivel így tudunk hivatkozni egy másik osztályra a tömb egyik eleménél. Ennek helyes működéséhez ne felejtsük el importálni az **App \ Rules \ PostPublishedAt** osztályt a **PostController** fájl elején.

Megjegyzés: a többnyelvűsítés természetesen itt is működik a hibaüzenetnél, ha léteznek a megfelelő lefordított szövegeink, és követjük a lokalizáció alapjait és szabályait (lásd 15.2. alfejezet), akkor a **$fail()** metódusnak a szótár fájljainkban létező kulcsot kell megadni és a következő átalakítással a kulcshoz tartozó választott nyelv szerinti hibaüzenet fog majd megjelenni:

$fail('validation.before_or_equal')->translate();

Sőt, a **translate()** metódus első paraméterében a helyőrzőknek adatot is tudunk átadni, míg a második paraméterben meghatározhatjuk az alapértelmezett nyelvet is (9–30. kódrészlet). A működéshez hozzunk létre két fájlt a projekt gyökerében lévő **lang** mappában lévő **en** és **hu** mappákban (ha a nyelvi mappák nem léteznének, akkor hozzuk létre őket). A két mappában (**en** és **hu**) a fájlok neve legyen azonos: **validation.php**

```
<?php

return [
  'before_or_equal' => 'The date of :attribute field should be today or
earlier than today'
];
```

9–28. kódrészlet: A lang / en / validation.php fájl tartalma

```
<?php

return [
  'before_or_equal' => 'A :attribute mező értéke legkésőbb a mai nap legyen'
];
```

9–29. kódrészlet: A lang / hu / validation.php fájl tartalma

Ezzel el is készült a két szótár, egyelőre csak egy-egy szótárbejegyzéssel (kulcs-érték párral). Ezután már elő tudjuk hívni a szótárban lévő kulcs szerinti hibaüzenetet:

```
$fail('validation.before_or_equal')->translate([
  'attribute' => $attribute
], 'hu');
```

9–30. kódrészlet: Szótár szerinti egyedi validációs hibaüzenet megjelenítése

Teszteljük is le manuálisan az alkalmazásunkat, és próbálunk meg egy jövőbeli publikálási dátumot megadni az új vagy a már létező blogbejegyzésünknek. Meg fogjuk kapni a most *„beégetetten"* magyar hibaüzenetet. Ha azt szeretnénk, hogy az alkalmazásunk nyelve szerinti hibaüzenet töltődjön be automatikusan, akkor itt ezt a **translate()** metódusba beégetett második paramétert vegyük ki, és tanulmányozzuk át a 15.3.2. alfejezetet a megértéshez és a helyes használathoz (ha olyan nyelvet választunk ki, amihez nincsen fordítás – szótár –, akkor az automatikus nyelven fog megjelenni az üzenet).

Az alfejezetben található programkód módosítások ebben a GitHub commit-ben találhatók meg.

Feladat: ellenőrizzük le, hogy a **slug** mező a blogbejegyzéseknél csak kisbetűt tartalmaz-e a kötőjelen kívül, különben adjunk validációs hibaüzenetet róla. (Implicit módon a validálás már a regex-es mintaillesztésnél megtörténik, de a gyakorlás miatt hozzunk létre neki egy új Rule osztályt és kössük össze a **slug** mező szabályaival.) A módosításokat tartalmazó programkód változásokat összevonom a következő alfejezet GitHub commit-jével.

9.2.5. Felhasználói kérések kezelése validálás szempontjából

A Laravel keretrendszer szereti, ha logikusan építjük fel az alkalmazásunkat és az építőkockákból utána mi magunk is hatékonyabban tudunk építkezni. Emiatt válasszuk szét a szerver oldalon a validációt és az eltárolási folyamatot, így a validáció utána újra felhasználhatóvá fog válni, ha ugyanazokat a szabályokat akarjuk használni a **store()** és az **update()** metódusok esetében. Eddig ezt egy külön metódusban valósítottuk meg a Controller fájlokban, osztályokban, de van ennek a validációnak egy jobb helye is: a **Request** osztályokban. *Megjegyzés:* amikor a `php artisan make:controller` parancsot az `-r` kapcsolóval hajtottuk végre, akkor legenerálta nekünk a 7 RESTful vezérlő metódust, és a **store()** valamint az **update()** metódusok paraméterül megkapták a **Request $request** objektumot. Eddig a metódusok magjában nem ezt az objektumot használtuk arra, hogy hozzáférjünk a kliens oldalról érkező adatokhoz, hanem a **request()** segédmetódust, de a működési elv ugyanaz, tehát a **$request->all()** metódushívással is megkaphatjuk ugyanott a felhasználótól érkező adatokat. Ennek a logikáját fogjuk itt most folytatni. Hozzuk létre a **Request** osztályt a **Post** objektumok validálásához:

```
php artisan make:request StorePostRequest
```

A névkonvenció itt azt diktálja, hogy a *„Request"* szót fűzzük hozzá az osztály nevéhez. A parancs kiadásával létrejön az **app** mappában egy új **Requests** nevű mappa, benne pedig az új **StorePostRequest.php** fájlunk. A fájl két metódust tartalmaz, az egyik az **authorize()**, ami a felhasználói hitelesítéssel és engedélyeztetéssel (9. fejezet) lesz kapcsolatban, úgyhogy ennek a visszatérési értékét állítsuk át igazra (**return true;**), hogy ne okozzon 403 HTTP hibakódot *(This action is unauthorized.)*. A másik

metódus a szabályokat tartalmazza, amelyeket a validáció során be kell tartanunk. Másoljuk le, és illesszük be ide annak a tömbnek a tartalmát, ami eddig a **PostController validatePost()** metódusában volt.

```
public function rules(): array
{
  return [
    'title' => 'required|min:5|max:100',
    'slug' => 'required|max:255|regex:/^[a-z]+[-]{1}[a-z]+$/',
    'body' => 'required',
    'published_at' => ['required', new PostPublishedAt],
    'score' => 'required|numeric|between:1.0,10.0',
    'category_id' => 'required|exists:categories,id',
    'tags' => 'required|exists:tags,id',
  ];
}
```

9–31. kódrészlet: StorePostRequest osztály rules() metódusa a validációs tömbbel

Természetesen a **PostPublishedAt** osztály importálásáról itt se feledkezzünk meg!

Bár a validációs szabályok közösek a két metódus (**store()** és **update()**) esetén, így gondolhatnánk, hogy elég lenne egy kérés osztályt definiálni ugyanazzal a szabályrendszerrel, de ha ezt tennénk, akkor megsértenénk az Objektumorientált tervezési alapelvek[11] közül a *„Vonatkozások szétválasztásának"* (Separation of concerns) elvét, illetve a SOLID[12] elvek közül az *„Egy felelősség elvét"* (Single Responsibility Principle) is. Helyette inkább implementáljuk a validációs szabályokat a **StorePostRequest** osztályban, utána pedig hozzuk létre az **UpdatePostRequest** osztályt is, amelynek a magja üres maradhat és ne **FormRequest** osztályt terjessze ki, hanem a **StorePostRequest** osztályt. Így tartalmazni fogja azokat a szabályokat, amelyeket már a **StorePostRequest**-ben is definiáltunk, de itt az **UpdatePostRequest**-ben akár felül is írhatnánk, ha szeretnénk (egyelőre azonban nem szeretnénk).

```
php artisan make:request UpdatePostRequest
```

```
use App\Http\Requests\StorePostRequest;

class UpdatePostRequest extends StorePostRequest
{

}
```

9–32. kódrészlet: UpdatePostRequest osztály kiterjeszti a StorePostRequest osztályt

Ezután törölhetjük a **PostController**-ből a **validatePost()** metódust, és módosítsuk a **store()**, valamint az **update()** metódusokat az alábbiak szerint:

---

[11] Mint már korábban többször, itt is Robert C. Martin - Tiszta kód könyvének ide vonatkozó fejezeteit tudom ajánlani áttekintésre ahhoz, hogy az itt olvasható OOP és SOLID elvet jobban megérthessük.

[12] Részletesebb leírás, PHP programkódon írt példákkal illusztrálva a SOLID elvekről itt található: https://www.digitalocean.com/community/conceptual-articles/s-o-l-i-d-the-first-five-principles-of-object-oriented-design

```
public function store(StorePostRequest $request)
{
  DB::transaction(function () use ($request) {
    $post = Post::create($request->safe()->only(['title', 'slug', 'body',
'category_id', 'published_at']));

    Rating::create([
      'score' => $request->score,
      'post_id' => $post->id,
    ]);

    $post->tags()->attach($request->tags);
  });

  return redirect(route('posts.index'));
}

public function update(UpdatePostRequest $request, Post $post)
{
  DB::transaction(function () use ($post, $request) {
    $post->update($request->safe()->only(['title', 'slug', 'body',
'category_id', 'published_at']));

    $post->tags()->sync($request->tags);

    Rating::where('post_id', $post->id)->update([
      'score' => $request->score
    ]);
  });

  return redirect(route('posts.index'));
}
```

9–33. kódrészlet: PostController egyedi Request objektummal bővített store() és update() metódusa

Így most már bármilyen kérés érkezik a blogbejegyzések eltárolására (**store**) és frissítésére (**update**), először a **StorePostRequest** osztály és a leszármazott **UpdatePostRequest** objektumai miatt azok validációs szabályain kell végig haladnia a kérés kiszolgálásának, és csak akkor történik meg az eltárolás vagy a frissítés, ha nem bukott el valamelyik validációs szabály ellenőrzésekor (ezt ellenőrizzük a **safe()** segédmetódussal). *Megjegyzés:* figyeljünk arra, hogy a tranzakciókon belül nem látszódik alapértelmezetten a **$request** objektum, így azt a **use ($request)** hozzáadásával tehetjük láthatóvá a metóduson belül.

Az alfejezetben található (és az előző alfejezet *„Feladat"* részéhez tartozó) programkód módosítások ebben a GitHub commit-ben érhetők el.

Feladat: Azért, hogy a validálást az API hívásoknál is tudjuk használni, érdemes az összes többi erőforrásunk ellenőrzéseit ugyanígy kiszervezni **Request** osztályokba és azok **rules()** metódusaiba.

Tekintsük meg a **CommentController** osztályt és az abban meglévő egyedi validációs szabályt! Itt használtunk egyedi hibaüzeneteket a validációs szabályokhoz, ott a többdimenziós tömb első elemét a **rules()** metódus visszatérési tömbjébe kell megadni, az egyedi hibaüzenetet tartalmazó második tömbelemet pedig egy **messages()** metódus visszatérési tömbjének kell megadni. Ha van még egyedileg átnevezett attribútumunk is (harmadik tömbelem, például: **['post_id' => 'post identification']**), akkor azt az **attributes()** metódus visszatérési tömbjeként kell megadni.

```
public function rules(): array
{
  return [
    'username' => 'required|max:255',
    'content' => 'required',
    'post_id' => 'required|exists:posts,id',
  ];
}

public function messages() : array
{
  return [
    'post_id.exists' => 'Missing :attribute in posts table'
  ];
}

public function attributes() : array
{
  return [
    'post_id' => 'post identification'
  ];
}
```

Ezeket a módosításokat ebbe a GitHub commit-be helyeztem el.

Érdekesség: a **unique** kényszert tartalmazó mező nem megfelelően működik, amikor nem konkrétan azt a mezőt szerkesztjük, amelyre a **unique** kényszer volt beállítva. Például, ha egy **tag**-et szerkesztünk, de nem a nevét, hanem a hozzá tartozó **post**-okat. Ekkor a frissítéskor a rendszer azt mondja, hogy már létezik ilyen nevű **tag**, ami igaz is, viszont nem azt akartuk frissíteni.

Ekkor a következő kényszer helyett:

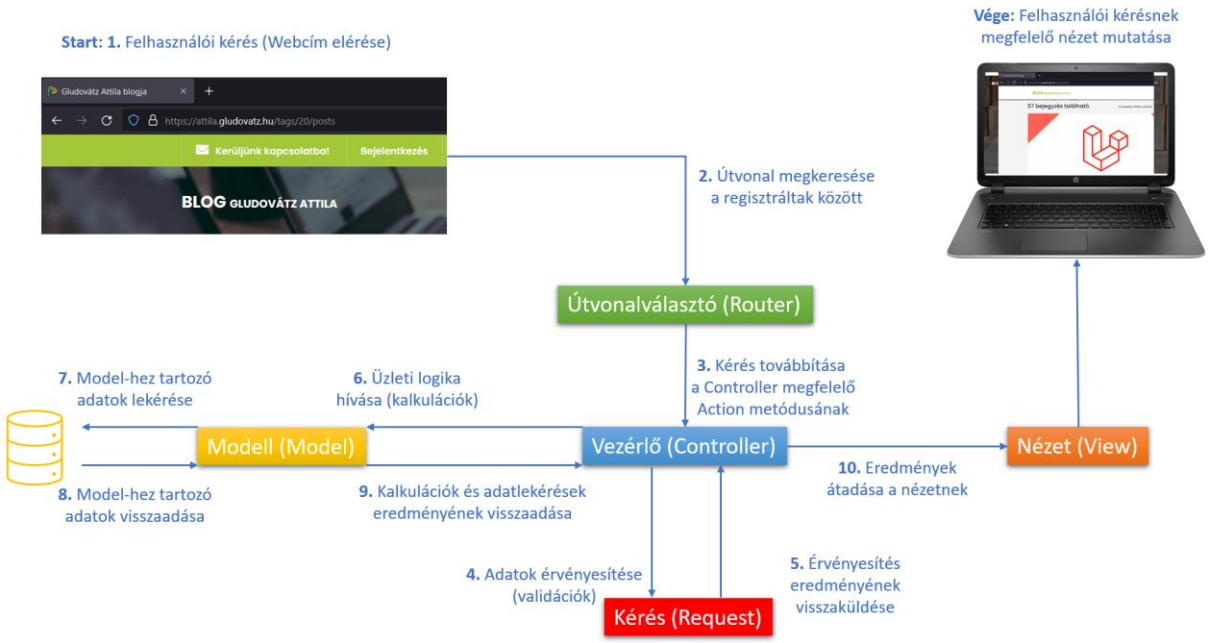
```
'name' => 'required|max:255|unique:tags',
```

Használjuk ezt a megoldást (A **$this** után azt a „mezőnevet” használjuk, amely az útvonalának wildcard helyén szerepelt. Ez leggyakrabban az erőforrás neve vagy az id.):

```
'name' => ['required', 'max:255', Rule::unique('tags')
  ->ignore($this->tag)],
```

Utóbbi helyes működéséhez importálni kell a **Rule** osztályt is:

```
use Illuminate\Validation\Rule;
```



9–11. ábra: Felhasználói kérés kiszolgálásának folyamata az MVC architektúrában a Request osztállyal bővítve

Karcsúbb lett a Laravel 11-es projekt mappa a validálás szempontjából is: az **app / Http / Controllers / Controller** ősosztályból hiányzik a **ValidatesRequests** trait importálása.

Egy logikus méretcsökkentés a Controller-ekben történő validálás kapcsán is megtörtént a Laravel 11-es verziójában: a Controller ősosztályból hiányzik a **ValidatesRequests** trait importálása. Erre igazából nem is nagyon volt szükség, hiszen a validálást a Controller-jeinkben eddig sem úgy hajtottuk végre, hogy **$this->validate(…)**, hanem vagy a paraméterben megkapott **Request $request** objektumon keresztül validáltunk, vagy a **request()** segédmetódus segített minket a validációs procedúra lefolytatásában. A Laravel keretrendszer is ez utóbbi két módszer használatát javasolja, úgyhogy logikus lépés volt kivenni a Controller ősosztályból a **ValidatesRequests** trait használatát.

9–1. újdonság: Laravel projekt méretének csökkenése a validálás kapcsán is

9.2.6. Létrehozási és módosítási API kérések kezelése validálással

A validálás az adatküldést tartalmazó API hívásoknál is fontos, emiatt ebben az alfejezetben áttekintjük, hogy hogyan és miképp tudunk különböző API kéréseket validálni a Request osztályokon keresztül.

9.2.6.1. Létrehozás kérése (Store, Request)

Itt most már elég sok tudást halmoztunk fel, úgyhogy ne a legegyszerűbb megoldáson (kategória létrehozásán) keresztül ismerjük meg az API kérés validálását, hanem egy fokkal bonyolultabb módon, a blogbejegyzések létrehozásán keresztül. Azért így, mert a blogbejegyzés létrehozásánál akkor nem csak egyszerűen a **name** attribútumra kell figyelnünk, mint a kategóriáknál, hanem más speciálisabb attribútumokra is, sőt, még a kapcsolatokkal is kell foglalkoznunk a létrehozás során.

Hozzuk létre az API-hoz a verziószámmal ellátott **StorePostRequest** kérést (így egyből látszani fog az is a **Request** mappában, hogy ezek a most létrehozott kérések az API-hoz tartoznak, mivel van verziójuk):

```
php artisan make:request V1/StorePostRequest
```

Ennek a konkrét kérésnek (és a többi erőforrásnál ugyanúgy majd) vehetjük a tartalmát a már létező web-es kérésekhez készült **StorePostRequest** osztályból, az importálásokkal együtt. De ne másoljunk, hanem terjesszük ki azt az osztályt!

```
use App\Http\Requests\StorePostRequest as WebStorePostRequest;

class StorePostRequest extends WebStorePostRequest
{
  public function prepareForValidation()
  {
    $this->merge([
      'published_at' => $this->publishedAt,
      'category_id' => $this->categoryId,
    ]);
  }
}
```

9–34. kódrészlet: V1/StorePostRequest osztály, tartalma és az importálás

Viszont azokat a kulcsokat **rules()** metódus visszatérési tömbjében, amelyek aláhúzást tartalmaznak, a felhasználói API kérések elküldésekor *„camel case"*-es formában fogjuk megadni, mivel az API működésénél ez az elvárt szabály. Tehát a **published_at** legyen **publishedAt**, míg a **category_id** legyen **categoryId**. Ezzel a **prepareForValidation()** metódussal lehetőségünk van arra, hogy a felhasználói kéréskor beérkező attribútumok itt a **Request**-ben átalakításra, előkészítésre kerüljenek, és mire a **rules()** metódushoz kerülnek az értékek (tehát a konkrét validálás előtt), addigra már az adattáblának megfelelő (aláhúzással elválasztott) mezőnevek lesznek benne.

A **V1/PostController** osztályban pedig használjuk ezt a kérést az eltároláskor, ugyanakkor vegyük alapul hozzá a web-es **PostController** osztály **store()** metódusát, hiszen itt már a kapcsolatokkal is kell foglalkozni:

```
public function store(StorePostRequest $request)
{
  $post = DB::transaction(function () use ($request) {
    $post = Post::create($request->safe()->only(['title', 'slug', 'body',
'category_id', 'published_at']));
```

```
    Rating::create([
      'score' => $request->score,
      'post_id' => $post->id,
    ]);

    $post->tags()->attach($request->tags);

    return $post;
  });

  return PostResource::make($post);
}
```

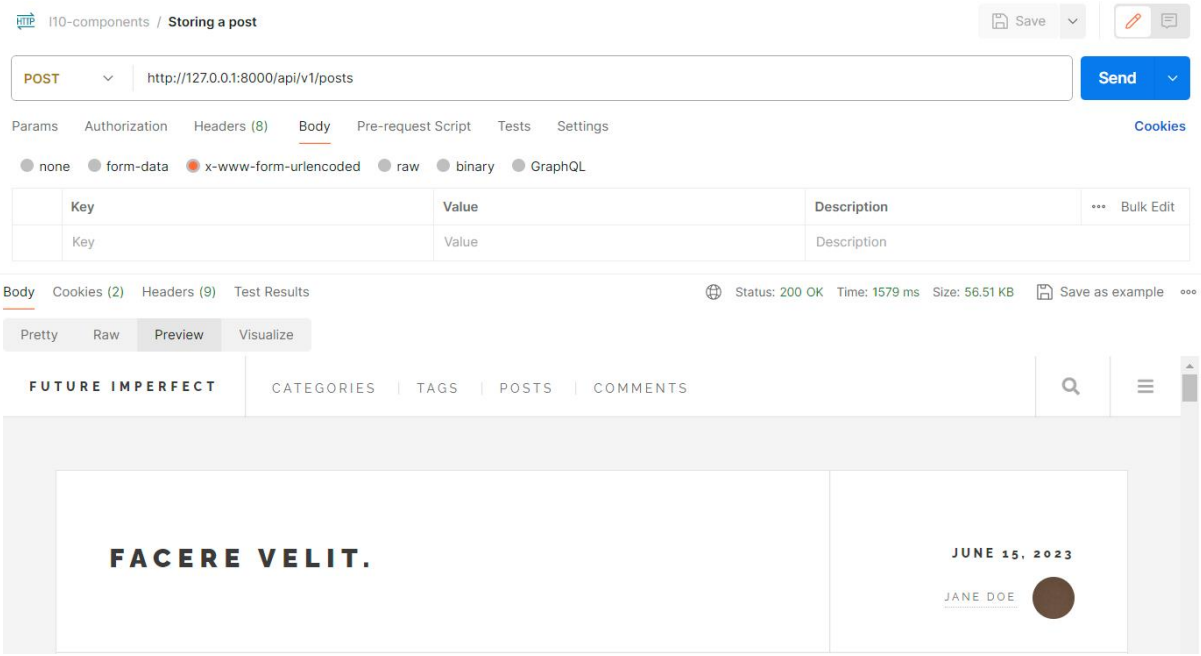
9–35. kódrészlet: Validációval kibővített store() metódus a V1/PostController osztályban

Az osztály elején pedig természetesen importáljuk is be ezt a **V1/StorePostRequest** osztályt.

A Postman-ben hozzuk létre egy új kérést úgy, hogy elrontjuk a validációt!

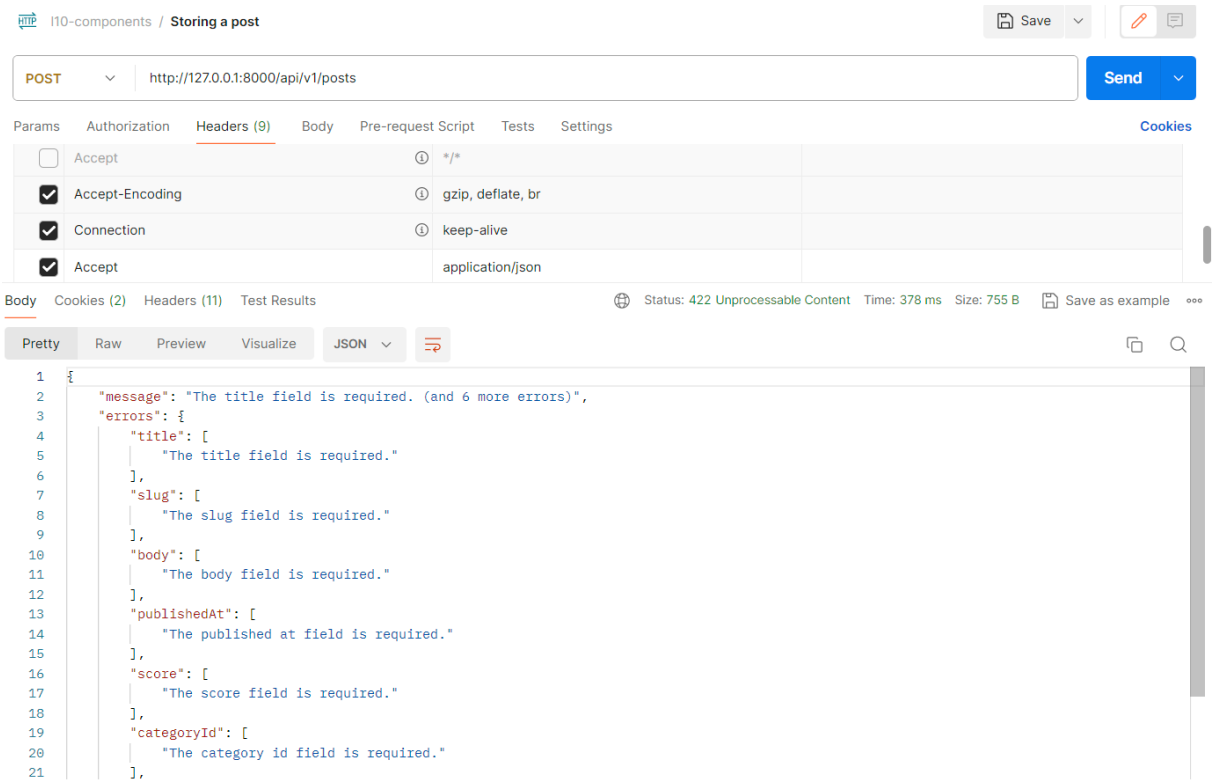
POST http://127.0.0.1:8000/api/v1/posts

Üres *„Body"* résszel elküldve a kérést (ami így nyilvánvalóan nem érvényes kérés), a *„Pretty"* és a *„Preview"* válasz fülön is láthatjuk, hogy ez az alkalmazásunk főoldalának HTML kódját mutatja nekünk.



9–12. ábra: Nem látszódnak a validációs hibaüzenetek az üres blogbejegyzés létrehozásakor

Ez volt az alapértelmezett működés az átirányítással. Viszont mi szeretnénk látni a validációs hibákat! Ehhez tegyük meg azt, amit már korábban többször: a *„Headers"* lapfülön inaktiváljuk az *„Accept"* mezőt, és hozzuk létre újra az *„Accept"* kulcsot **application/json** értékkel, majd futtassuk újra az iménti kérést.



9–13. ábra: JSON válasz kikényszerítése esetén látszódnak a validációs hibaüzenetek üres blogbejegyzés létrehozásakor

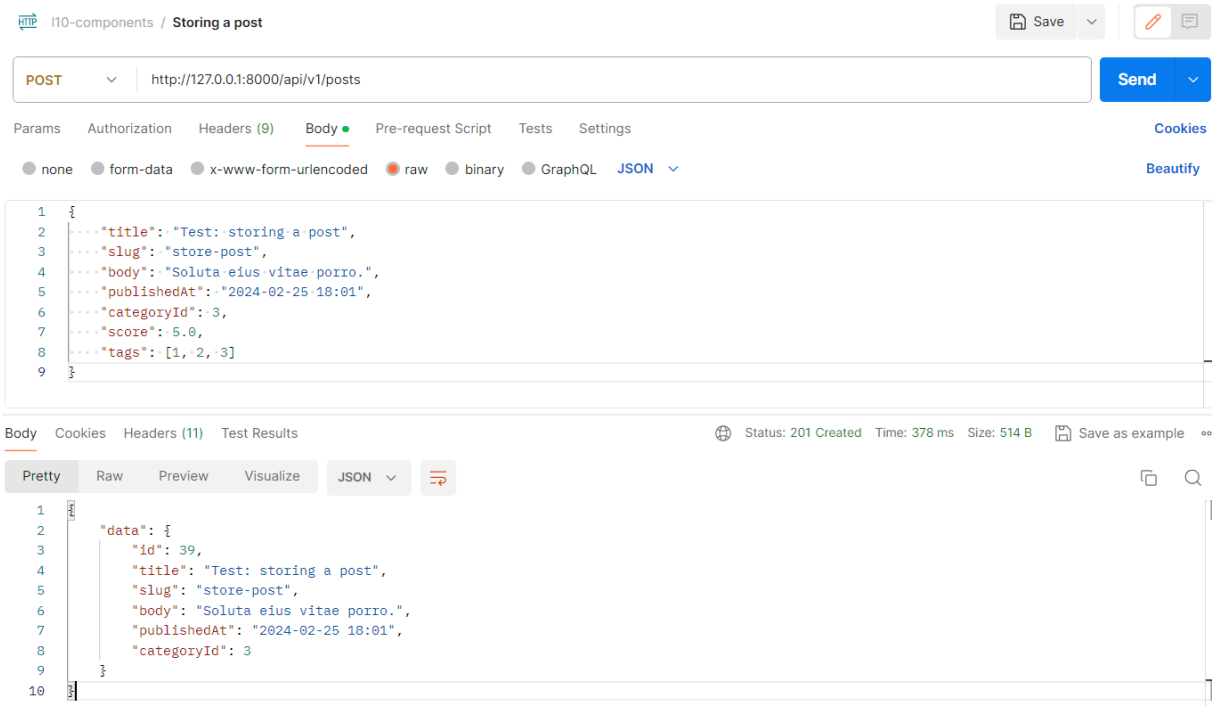
Így most már láthatóak is a validációs hibák és a HTTP státuszkód is *„422 Unprocessable Content”*, vagyis feldolgozhatatlan tartalmat jelöl.

Futtassunk most egy olyan tesztet, aminek át kell mennie a validáción: a kérés *„Body”* lapfülén válasszuk a *„raw”* opciót, a sor szélén pedig a Text helyett legyen *„JSON”* a típus. Majd illesszük be ezt a tartalmat a szerkesztési ablakba:

```
{
    "title": "Test: storing a post",
    "slug": "store-post",
    "body": "Soluta eius vitae porro.",
    "publishedAt": "2024-02-25 18:01",
    "categoryId": 3,
    "score": 5.0,
    "tags": [1, 2, 3]
}
```

9–36. kódrészlet: Blogbejegyzés eltárolása kérés törzse (helyes bemeneti adatokkal)

Válaszként pedig megkapjuk, hogy létrejött az erőforrás és *„201 Created”* státuszkódot jelez a Postman:



9–14. ábra: Blogbejegyzés (és kapcsolatainak) eltárolása API hívás segítségével

A blogbejegyzés és a kapcsolatai (**Rating** -> **score**, **Tag**-ek) is létrejöttek az érintett adattáblákban, tehát a munka sikerrel zárult a létrehozás kapcsán, ezt ellenőrizhetjük a **Post** erőforrás **index** vagy **show** API útvonalainak lekérésével is.

9.2.6.2. Módosítás, frissítés kérése (Update, Request)

Egy RESTful API módosítási, frissítési útvonala egy kicsit másképp működik, mint azt a webes útvonal elérésénél és validációjánál tapasztaltuk. Ott nem különösebben foglalkoztunk még azzal, hogy PUT vagy PATCH HTTP metódus szerint érkeznek be az adatok. A két metódus közti különbség:

- *PUT:* a teljes erőforrás beérkezik, minden mezőjével együtt, azokkal is, amelyek nem változtak meg a módosítás, frissítés során. Kötelező minden mezőnek értéket adni, és úgy átküldeni validálásra.
- *PATCH:* csak azok a mezői érkeznek be a felhasználói kéréssel, amelyek módosításra, frissítésre kerültek.

A leírtakból mindenképpen érezhetjük, hogy a PUT metódussal érkező felhasználói adatcsomag eléggé valószínű, hogy nagyobb lesz, mint a PATCH esetén elküldésre kerülő adathalmaz.

A fentiekkel szemben viszont a Laravel csak egy Controller **update()** metódust biztosít a számunkra, úgyhogy a PUT/PATCH különbséget a Request osztályában kell majd lekezelnünk. Hozzuk létre a hozzá tartozó Request osztályt:

```
php artisan make:request V1/UpdatePostRequest
```

Ahogy a webes kéréseknél is tettük, itt is örököltethetjük az **UpdatePostRequest**-et a **StorePostRequest** osztályból. Így az **authorize()** és a **prepareForValidation()** metódusok már rendelkezésre is fognak állni, a **rules()**-t pedig felüldefiniáljuk:

```
use App\Rules\Lowercase;
use App\Rules\PostPublishedAt;
use App\Http\Requests\V1\StorePostRequest;

class UpdatePostRequest extends StorePostRequest
{
  public function rules(): array
  {
    $method = $this->method;

    if ($method == 'PUT')
    {
      return parent::rules();
    }
    else
    { // 'PATCH'
      return [
        'title' => 'sometimes|required|min:5|max:100',
        'slug' => ['sometimes', 'required', 'max:255', 'regex:/^[a-z]+[-
]{1}[a-z]+$/', new Lowercase],
        'body' => 'sometimes|required',
        'published_at' => ['sometimes', 'required', new PostPublishedAt],
        'score' => 'sometimes|required|numeric|between:1.0,10.0',
        'category_id' => 'sometimes|required|exists:categories,id',
        'tags' => 'sometimes|required|exists:tags,id',
      ];
    }
  }
}
```

9–37. kódrészlet: V1/UpdatePostRequest osztály, tartalma és az importálások

A **rules()** metódus elején lekérjük, hogy a felhasználói kérés PUT vagy PATCH metódus szerint érkezett. Utána pedig megvizsgáljuk a metódus értékét:

- ha PUT, akkor a validációs szabályrendszer ugyanaz, mint a **StorePostRequest** szülőosztályban volt,
- ha PATCH, akkor lemásolható a validációs szabályrendszer a **StorePostRequest** szülőosztályból, de minden mezőre vonatkozó szabályhalmaznál bekerült egy **sometimes** szabály is, ami arra vonatkozik, hogy ha az adott mezőt értéke nem érkezik meg a validálásra, akkor a további szabályokat figyelmen kívül lehet hagyni.

Megjegyzés: ha minden mezőhöz tömbként lenne hozzárendelve a validációs szabályhalmaz, akkor könnyebb lenne mindegyikhez csak hozzáadni az új **sometimes** szabályt, és a kódunk még optimálisabb (rövidebb) lehetne.

A **V1/PostController** osztály **update()** metódusa a következőképpen néz ki a validációs átalakítás után (az osztály felett természetesen importáljuk a **V1/UpdatePostRequest** osztályt):

```
public function update(Post $post, UpdatePostRequest $request)
{
  $post = DB::transaction(function () use ($post, $request) {
    $post->update($request->safe()->only(['title', 'slug', 'body',
'category_id', 'published_at']));

    if(isset($request->tags))
    {
      $post->tags()->sync($request->tags);
    }
    if(isset($request->score))
    {
      Rating::where('post_id', $post->id)->update(['score' => $request-
>score]);
    }
    return $post;
  });

  return PostResource::make($post);
}
```

9–38. kódrészlet: Validációval kibővített update() metódus a V1/PostController osztályban

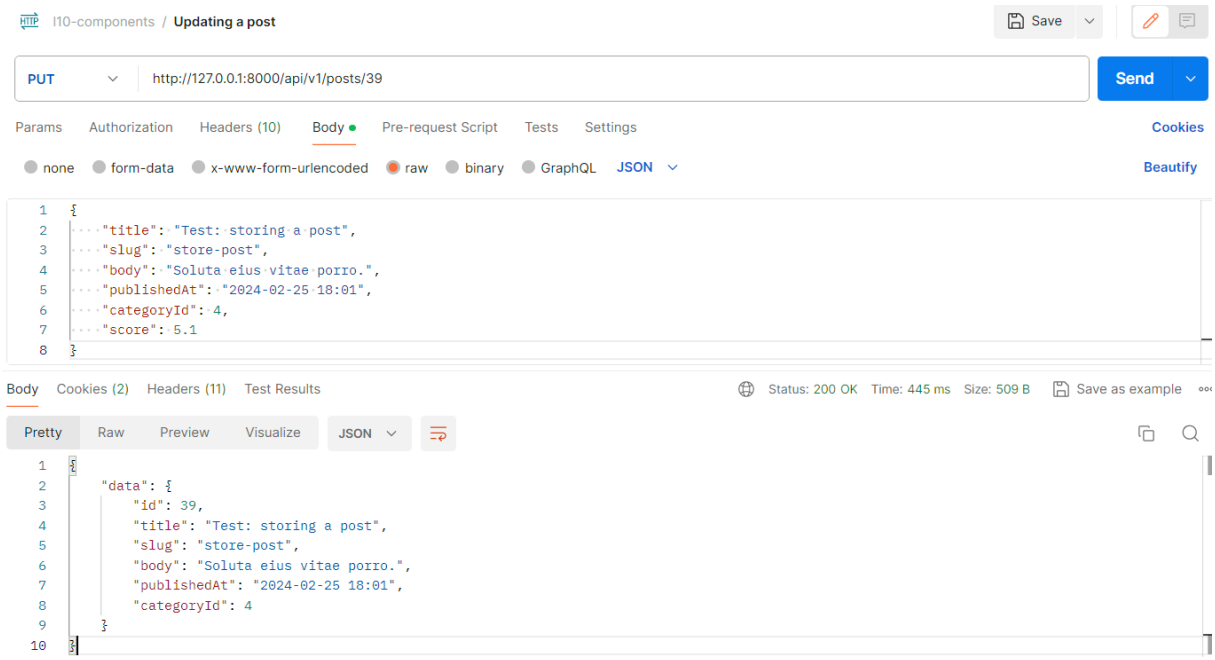
A PATCH-es frissítési kérés csak feltételesen tartalmazza a blogbejegyzés címkéire (**tags**) és értékelésre (**score**) vonatkozó adatokat, így ezeket egy feltételvizsgálathoz kötöttük.

Teszteljük a megoldásunkat a Postman-ben a két kérés futtatásával, kezdjük a PUT kéréssel:

PUT http://127.0.0.1:8000/api/v1/posts/39

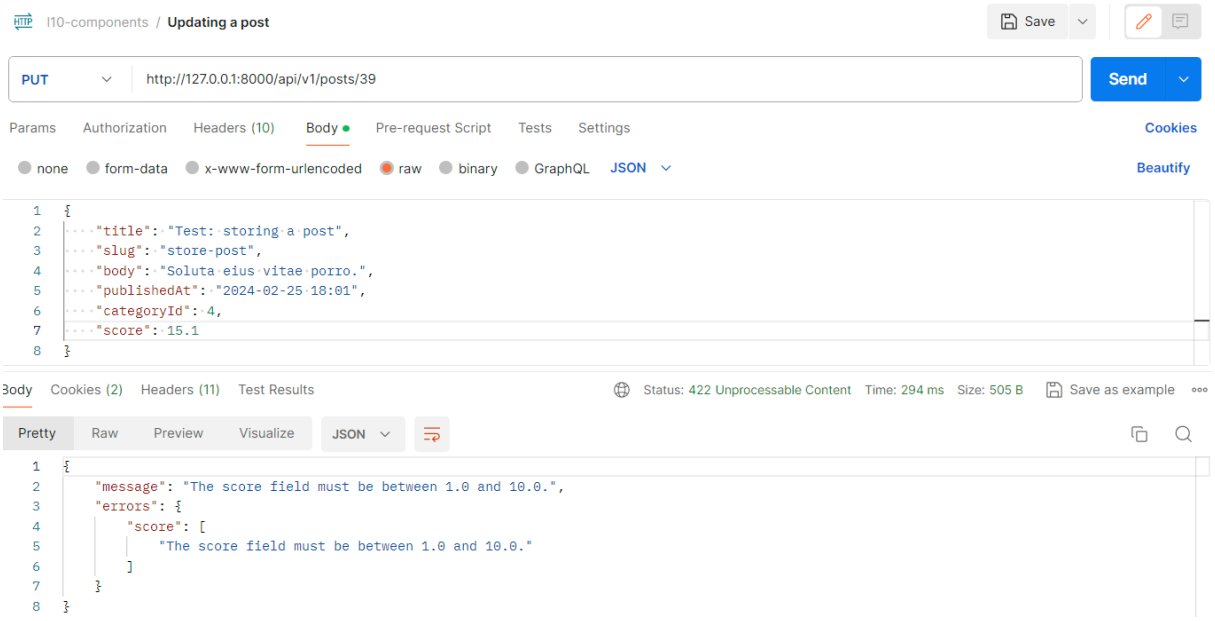
```
{
    "title": "Test: storing a post",
    "slug": "store-post",
    "body": "Soluta eius vitae porro.",
    "publishedAt": "2024-02-25 18:01",
    "categoryId": 4,
    "score": 5.1
}
```

9–39. kódrészlet: Blogbejegyzést módosító kérés törzse (helyes bemeneti adatokkal)



9–15. ábra: Hibátlan PUT frissítési kérés, hibátlan válaszeredménnyel

Ha módosítjuk a kérésben a **score** értékét 15.1-re, akkor meg is kapjuk a validációs hibaüzenetet és a *„422 Unprocessable Content”* választ:



9–16. ábra: Validációs hibát tartalmazó bemenet PUT frissítési kérés esetén, hibás válaszeredménnyel

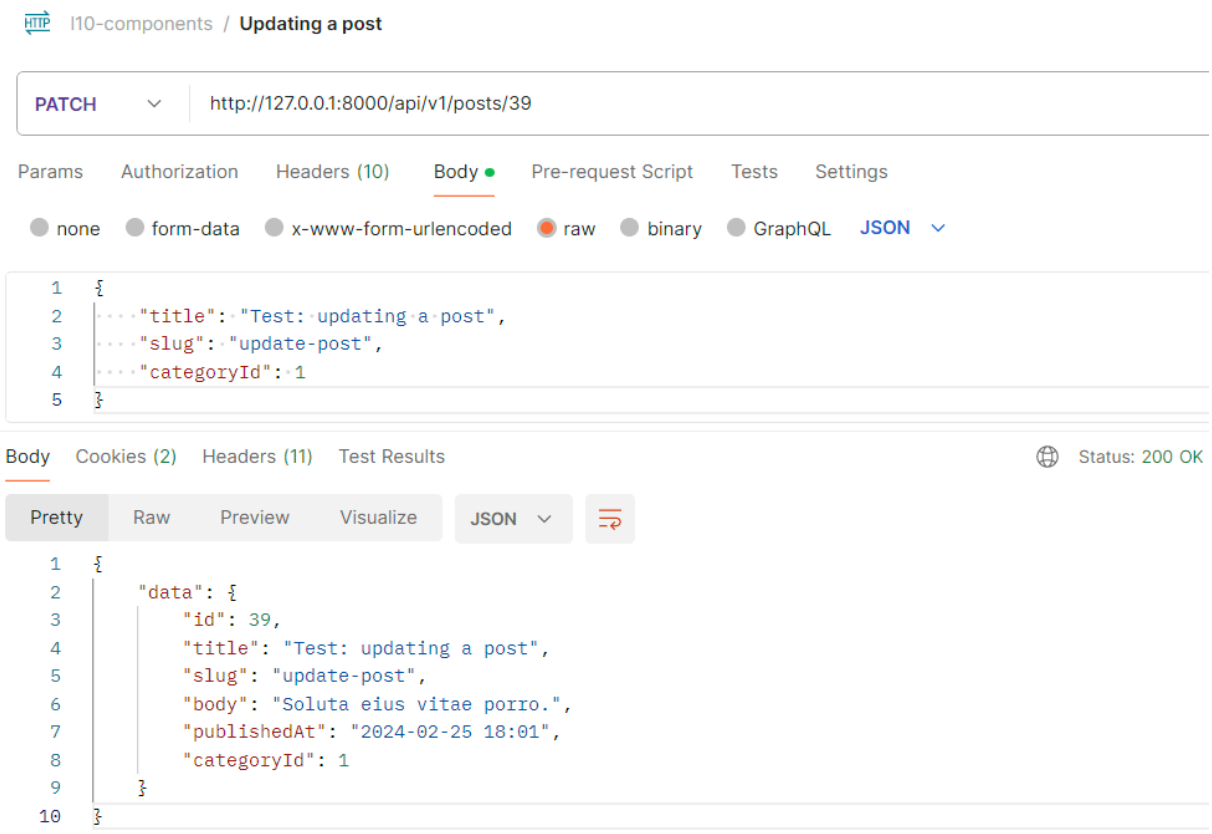
Másoljuk le (*„Duplicate”*) ezt a kérést a Postman-ben, de állítsuk át **PATCH**-re a kérés elküldésének metódusát. Kétféle módon teszteljük ezt is, hibátlan és hibás bemeneti adatokkal, azonban itt már nem kell a teljes csomagot átküldenünk, elegendő például a **title**, **slug**, **categoryId** mezőket meghagyni a *„Body”*-ban. Így viszont a válasz egy hibával érkezne vissza, mert hiányolná a **published_at** mező értékét a rendszer. Ez amiatt van, mert a **prepareForValidation()** a **StorePostRequest** szülőosztályban kötelezően szerepel, ami a PUT frissítési kérés esetén nem is okozott problémát, mivel szerepelt a kérés törzsében az

erre vonatkozó adat. A PATCH kérésnél most viszont kimaradt, így végezzük el ennek a metódusnak is a felüldefiniálását és adjuk hozzá a **V1/UpdatePostRequest** osztályhoz:

```php
public function prepareForValidation()
{
  if($this->publishedAt)
  {
    $this->merge([
      'published_at' => $this->publishedAt
    ]);
  }
  if($this->categoryId)
  {
    $this->merge([
      'category_id' => $this->categoryId,
    ]);
  }
}
```

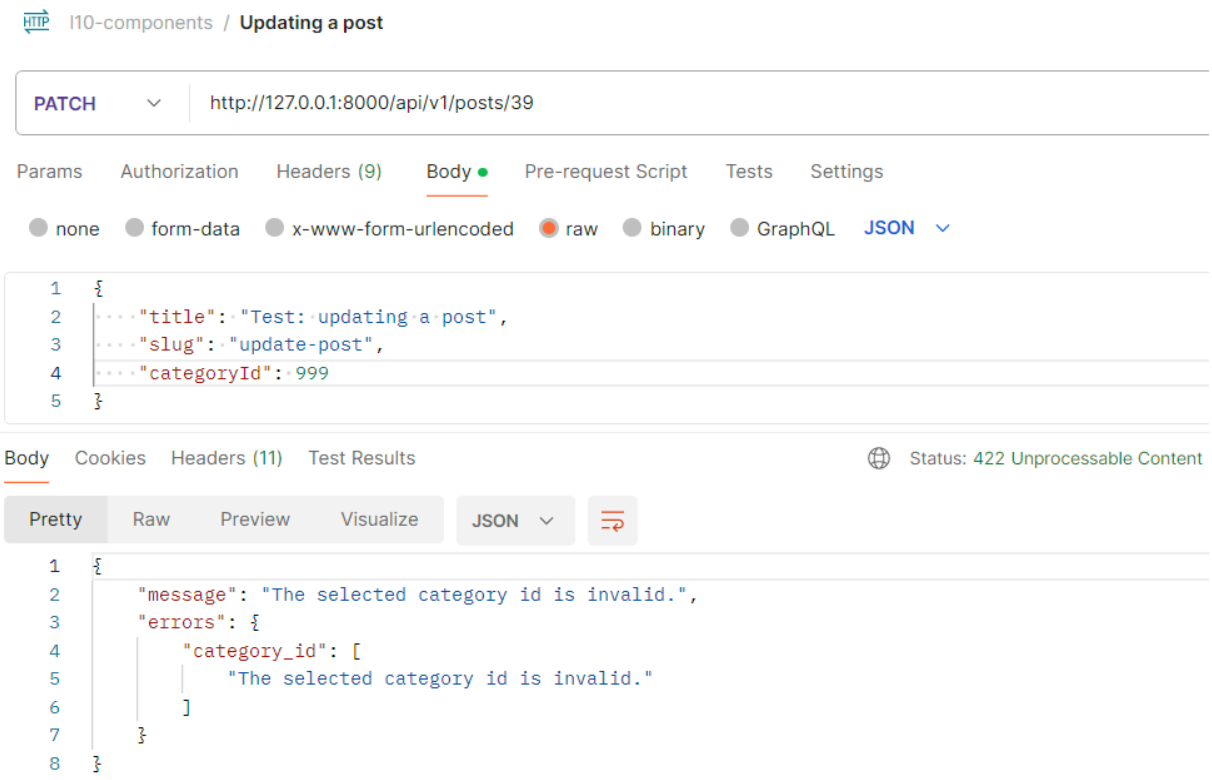
9–40. kódrészlet: Feltételesen érkező felhasználói adatok előkészítése validálásra

Így most már akkor is frissülni fog a blogbejegyzésünk, ha csak a **title**, **slug**, **categoryId** mezőket adtuk meg a kérés törzsében:



9–17. ábra: Hibátlan PATCH frissítési kérés, hibátlan válaszeredménnyel

A validáció viszont továbbra is működik itt is, tehát ha például olyan kategória azonosítót akarunk megadni, ami nem létezik (például 999), akkor arra hibát fog adni válaszként a rendszer:



9–18. ábra: Validációs hibát tartalmazó bemenet PATCH frissítési kérés esetén, hibás válaszeredménnyel

Az alfejezetben megtalálható programkód módosítások megtalálhatók ebben a GitHub commit-ben.

Feladat: gyakorlásként érdemes megcsinálni a kategóriák (**Category**), címkék (**Tag**) és a kommentek (**Comment**) modellekhez tartozó vezérlők API-n keresztül történő létrehozási, módosítási metódusokat, valamint a hozzájuk tartozó felhasználói kérések validációit.

A helyes működés teszteléséhez duplikáljuk a már meglévő request-jeinket a Postman-ben, és állítsuk át az útvonalakat az erőforrásnak megfelelően.

9.3. Automatikus tesztelés (validációs szabályok)

A validációs fejezet végén szeretnénk olyan automatikus teszteket létrehozni, amelyek utána mindig biztosítják, hogy a már megvalósított érvényesítési (validációs) eljárásaink továbbra is jól működjenek az űrlapjainknál. Így az alkalmazásunk *minősége* megmaradhat olyannak, amilyet mi elterveztünk és megvalósítottunk neki.

9.3.1. Tesztelés PHPUnit eszközzel

A PHPUnit egy kiváló eszköz a tesztelésre, amely a validációs szabályok ellenőrzésénél is nagy hasznunkra lehet. Gyakorlatban próbáljuk ki, hogy mi lenne, ha úgy szeretnénk kategóriát frissíteni, hogy nem adjuk meg a nevét (ekkor nyilván egy szerver oldali validációs hibát kell kapnunk).

```
function test_category_update_validation_error_redirects_back_to_form()
{
  $category = Category::factory()->create();
```

```
  $category->name = '';

  $response = $this->put('categories/' . $category->id, $category-
>toArray());

  $response->assertStatus(302); // redirect back to the form
  $response->assertSessionHasErrors(['name']);
}
```

9–41. kódrészlet: Szerver oldali validációs hiba automatikus tesztelése

A teszt sikeresen lefut! Üres név mező megadásakor visszairányításra kerül a felhasználó az űrlaphoz és ott megkapja a munkamenetben a **name**-hez tartozó hibát. Talán még beszédesebb lehet, ha az **assertSessionHasErrors()** metódus helyett az **assertInvalid()** metódust használjuk, ugyanezzel a paraméterrel. Az olvasóra bízom, hogy melyiket választja, melyiket tartja logikusabbnak, olvashatóbbnak, érthetőbbnek.

9.3.2. Tesztesetek megtervezése és végrehajtása a validálás során

Szoftvertechnológiai szempontból a tesztelésnek mindig kiemelt szerepe van a szoftver életciklusa során: a különböző fázisokat külön-külön és egészében is tesztelhetjük, illetve elvárás is, hogy teszteljük azokat. Mi most az űrlapjaink kitöltésének *ellenőrzésére* fogunk koncentrálni.

Informatikusként, programozóként, adatbázis tervezőként, adatelemzőként, először kicsit nehéznek tűnhet teszteseteket megtervezni és írni, mivel ennek magas szintű megvalósítása is egy művészet, külön szakma is épül erre, különböző vizsgázási lehetőségekkel. Én most a tesztelés teljes mélységébe nem mennék bele, de itt van egy kiváló dokumentum, amit szívesen ajánlok, mert elmagyarázza a tesztelés alapjait, technikáit, menedzselését és minden egyéb fontos további tényezőjét. Ha pedig ennél is mélyebben érdeklődik valaki a téma iránt, akkor feltétlenül olvassa el Robert C. Martin - Tiszta kód című alapvetését, amit amúgy minden szoftverfejlesztőnek is ajánlok.

De nem kanyarodjunk el túlságosan a témánktól, és próbáljuk meg tartani a fókuszt az űrlap validáció tesztelésén. Nézzük meg a bemenetek generálásának folyamatát, amit szeretnék tesztelni:

1. Kliens oldali validációs szabályok ellenőrzése (pozitív, sikeres teszt és negatív, sikertelen teszt szempontból is) űrlap elküldésekor;
2. Szerver oldali validációs szabályok ellenőrzése (pozitív és negatív szempontból is) űrlap elküldése után.

Ezekhez először érdemes felmérni az űrlapon leggyakrabban előforduló bemenettípusokat, érvényes és érvénytelen bemeneteket azért, hogy tisztában legyünk vele, milyen módokon is lehet majd tesztelni őket. Azért is fontos ezeket már ilyenkor áttekinteni, mert ha véletlenül valamit a szoftverfejlesztő rosszul implementált vagy elnézett valamilyen kombinációt, amely bekövetkezése miatt a felhasználói elfogadási teszt (User Acceptance Testing) vagy a rendszerteszt (System Testing) során hibát kapunk, akkor az nagyon *kellemetlen* tud lenni, úgyhogy jobb elkerülni az ilyen szituációkat.

Bemenet típusok	Érvényes bemenetek	Érvénytelen bemenetek
Numerikus érték (egész vagy lebegőpontos szám)	• csak szám típusú értékek • kisebb, mint a megadott maximum limit ÉS nagyobb, mint a megadott minimum limit	• számok ÉS (betűk VAGY különleges karakterek) • nagyobb szám, mint a megadott maximum limit • kisebb szám, mint a megadott minimum limit • esetleg a negatív számok • esetleg a 0 • lebegőpontos számoknál a „tizedesvessző” (tizedespont)
Szöveg (karakter vagy karakterlánc)	• csak betűk • csak számok • csak speciális karakterek • az iménti három tetszőleges kombinációja • kisebb hosszúságú (rövidebb), mint a maximális karakter limit ÉS nagyobb hosszúságú (hosszabb), mint a minimális karakter limit	• minimális VAGY maximális karakterlimiteken túllépő
Dátum és/vagy idő	• dátum/időválasztó segéd látható, ha megkapja a fókuszt a mező • megfelelően működik az év / hónap / nap / óra / perc léptetése • az iméntiekkel egyidejűleg a manuális szerkesztését a mezőnek letiltani (csak a segéddel - picker - lehessen módosítani az értékeken) • megfelelő formátumú dátum/idő (webes alkalmazásoknál böngésző nyelvi vagy lokalizációs beállítás függő) • szökőévben február 29. ellenőrzése • nem szökőévben február csak 28 napos, ennek ellenőrzése	• dátum/idő kapcsán nagyon sok mindent lehet tesztelni, de ha az érvénytelen bemenetekre akarunk koncentrálni, akkor az itt felsorolt érvényesek ellentettjét ellenőrizzük

	• óraátállításos napok időpontjainak ellenőrzése • téli/nyári időszámítás ellenőrzése • időzónák ellenőrzése	

9–1. táblázat: Leggyakoribb bemenet típusok érvényes és érvénytelen tesztesetei

Bemenet típus lehet még a listából (**select** options, **radio** buttons, **checkbox** lists) választás, de itt általában elég megkövetelni azt, hogy kötelező-e kiválasztani valamit (esetleg többet) vagy sem.

9.3.3. Szerver és kliens oldali validáció elbukásának tesztelése Laravel Dusk eszközzel

Ebben az alfejezetben a validációs fejezetet szeretném lezárni a gyakorlatban, és olyan automatikus teszteket létrehozni, amelyek utána mindig biztosítják, hogy a már megvalósított érvényesítési (validációs) eljárásaink továbbra is jól működnek az űrlapjainknál is. Így az alkalmazásunk *minősége* megmaradhat olyannak, amilyet mi elterveztünk és megvalósítottunk neki.

Tipp: előfordulhat, hogy a Chrome böngészőnek nem a legfrissebb verzióját használjuk. Ezt mindenképpen érdemes frissíteni, mielőtt belevágnánk ismét a Laravel Dusk-os feladatok megoldásba. A böngésző frissítése után pedig a VSCode-ban lévő terminal-ban adjuk ki ezt a parancsot ahhoz, hogy frissítsük a Dusk-hoz tartozó Chrome driver-t is.

```
php artisan dusk:chrome-driver
```

A frissítések után pedig futtassunk le egy olyan tesztesetet, amiről biztosan tudjuk, hogy működik a projektünkben és hibamentesen lefut, például a **CategoryTest**-et:

```
php artisan dusk tests/Browser/CategoryTest.php
```

Így azt már tudjuk, hogy az érvényes (valid) tesztesetek lefutnak, de ebben a fejezetben pont az érvénytelenség ellenőrzését fogjuk tesztelni a Dusk eszközzel.

Készítsünk egy olyan tesztesetet, ami elbukik a validáció *„eltörése”*[13] miatt. Ehhez persze jó, ha tisztában vagyunk azzal, hogy maga a Laravel Dusk milyen lehetőségeket biztosít az **assert** utasításokra, ezeket érdemes áttekinteni: https://laravel.com/docs/10.x/dusk#available-assertions

Ha mondjuk az előző **CategoryTest**-ben lévő tesztesetet vesszük alapul, és a bemenet kitöltését kikommentezzük, akkor már el is fog bukni a tesztesetünk a teszt végrehajtásakor.

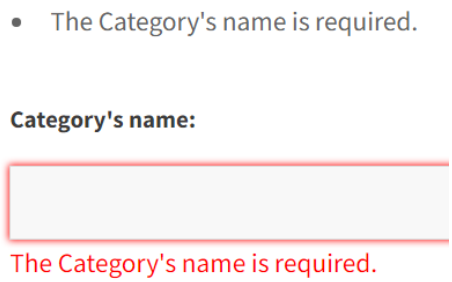
```
public function test_a_visitor_can_create_categories()
{
  $this->browse(function (Browser $browser) {
    $browser->visit('/categories/create')
      // ->type('name', 'Science') // input mező name attribútum értéke
```

[13] IT szleng kifejezés arra, ha valami elromlik.

```
        ->click('input[type="submit"]')
        ->assertSee('Science');
    });
}
```

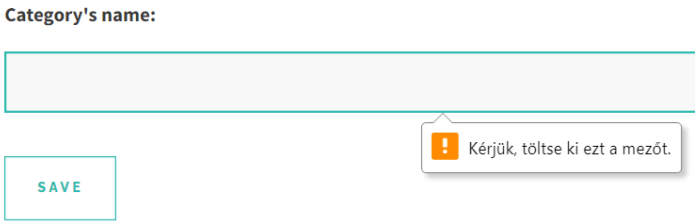
9–42. kódrészlet: CategoryTest osztályban lévő teszteset az input megadása nélkül

Az eredmény alább látható:



9–19. ábra: Szerver oldali validáció elbukása a teszteset futtatása során

Az alkalmazás jelenlegi állapotában a szerver oldali validáció fog elbukni, mivel a **categories.create** nézetben az űrlap komponensben (**x-form**) szerepel a **novalidate** kulcsszó, amely miatt figyelmen kívül hagyja az alkalmazás a **name** input mezőben lévő **required** attribútumot. Ha a **novalidate**-et kivesszük, akkor már a kliens oldali validáció fog elbukni.



9–20. ábra: Kliens oldali validáció elbukása a teszteset futtatása során

Ha nem látnánk, hogy melyik oldali validáció bukik el a teszteset futtatásakor, akkor a **--headless=new** kapcsolót kommenteljük ki a **DuskTestCase.php** fájlban.

Megjegyzés: ha angol vagy bármely más nyelven használjuk a böngészőnket, akkor érdemes először kipróbálni, hogy milyen üzenet jelenik meg a kliens oldali validáció elbukásakor egy-egy mezőnél, és utána azt ellenőrizni az automatikus tesztesetünkben leírt kóddal.

A kliens oldali validáció elbukásakor tehát a nálam futó környezetben a *„Kérjük, töltse ki ezt a mezőt."* szöveget kellene keresni és ellenőriznie a tesztesetnek. Állítsuk vissza a **test_a_visitor_can_create_categories()** metódust eredeti állapotába, hiszen ott mást szerettünk volna tesztelni és annak pozitív eredménnyel kell zárulnia. Lemásolhatjuk ugyanakkor ezt a függvényt és végezhetünk rajta módosítást, hogy a kliens oldali validáció elbukását tudjuk tesztelni.

```
public function test_a_required_input_field_must_be_filled()
{
    $this->browse(function (Browser $browser) {
```

```
    $browser->visit('/categories/create')
      ->click('input[type="submit"]')
      ->assertSee('Kérjük, töltse ki ezt a mezőt.');
  });
}
```

9–43. kódrészlet: Új teszteset a kliens oldali validáció elbukásának ellenőrzésére

Teljesen *„jogosan”* merülhetne fel bennünk, hogy ez a kis figyelmeztető üzenet, amit látunk az oldalon (9–20. ábra), ezt a tesztesetben is ellenőrizhetjük az **assertSee()** metódussal, azonban ez nem fog működni, el fog bukni ez a teszteset.

```
× a required input field must be filled

 FAILED  Tests\Browser\CategoryTest > a required input field must be filled
Did not see expected text [Kérjük, töltse ki ezt a mezőt.] within element [body].
```

9–21. ábra: Értesítés a teszteset elbukásáról

A Dusk keresi a megadott szöveget az oldal forráskódjában, azonban tényleg nincs benne, úgyhogy nem is látja így, jogos tehát a hibajelzés.

A kliens oldali validációs hibaüzeneteket a böngésző JavaScript API-ja segítségével kapjuk meg, ezért ezt kell ellenőrizni a tesztesetnél is.

```
public function test_a_required_input_field_must_be_filled()
{
  $this->browse(function (Browser $browser) {
    $messages = $browser
      ->visit('/categories/create')
      ->script("return document.querySelector('#nev').validationMessage");

    $this->assertEquals('Kérjük, töltse ki ezt a mezőt.', $messages[0]);
  });
}
```

9–44. kódrészlet: Kliens oldali validáció elbukását ellenőrző teszteset

Így a teszteset már eredményesen, hibamentesen fog lefutni. A Laravel Dusk-nak nincs **assertEquals()** metódusa, ezt a PHPUnit segítségével tudjuk használni a tesztesetben.

Az alfejezetben végrehajtott programkód módosítások ebben a GitHub commit-ben érhetők el.

Tipp: Teszteld a tudásod!

Léteznek nyilvános repo-k, amelyekkel a validáció során felhasznált tesztelési ismereteinket is próbára tudjuk tenni, például:

https://github.com/LaravelDaily/Test-Laravel-Validation

Ha klónozzuk a projektet és elindítjuk a tesztelést, akkor kezdetben minden tesztünk elbukik. Ezt a kódbázist kell úgy átalakítani, hogy hiba nélkül lefussanak, átmenjenek a teszteseteink.

További útmutatások, javaslatok a GitHub projekt leírásában érhetők el a könyvtár- és fájlstruktúra alatt.

9.4. Összegzés

Ezen a ponton, ha elégedettek vagyunk a szerver oldali validáció működésével, akkor vissza lehet állítani a kliens oldali validációt is mindegyik implementált űrlapnál.

A fejezet során először a kliens oldali validációval foglalkoztunk. Megnéztük, hogy milyen lehetőségeket nyújt az ellenőrzésre a HTML5, a CSS és a JavaScript (sőt még a JavaScript alapú, leginkább validációra szolgáló keretrendszereket is listáztam, némelyikre a blogomban mutattam példákat). Ez az alfejezet is rávilágított arra, hogy a WEB-es világban minden összefügg mindennel, így nem tekinthetünk el arról, hogy ismerni kell a PHP-n kívül más, szorosan a WEB alapjait képező nyelvet: a HTML-t, a CSS-t, JavaScript-et leginkább, és persze az újdonságaikat, keretrendszereiket.

A szerver oldali validációnál már koncentrálhattunk a Laravel-re, annak beépített szabályaira a felhasználói kérések kiszolgálásánál. De egyedileg megtervezett és megvalósított szabályokat is alkalmaztunk abban az alfejezetben.

A fejezet utolsó részében automatikus teszteket írtunk a PHPUnit és a Laravel Dusk eszközökkel. Automatikus tesztet megtervezni és megvalósítani is művészet. Meg kell tanulni, de rá is kell érezni, hogy hol vannak, lehetnek a hibák. Egy rossz teszt, rossz kódot is eredményezhet, úgyhogy ilyen esetben a tesztet is felül kell vizsgálni, és lehet, hogy újra meg kell írni.

10. A köztes rétegek és a felhasználói hitelesítés (Middleware, Authentication)

Számos dolgot elsajátítottunk már a Laravel keretrendszer használata és fejlesztése során. Ebben a fejezetben elkezdünk a rendszer magjával ismerkedni. Ehhez először áttekintjük a köztes rétegek, vagyis a Middleware-ek szerepét, valamint a helyüket a felhasználói kérések kiszolgálásában.

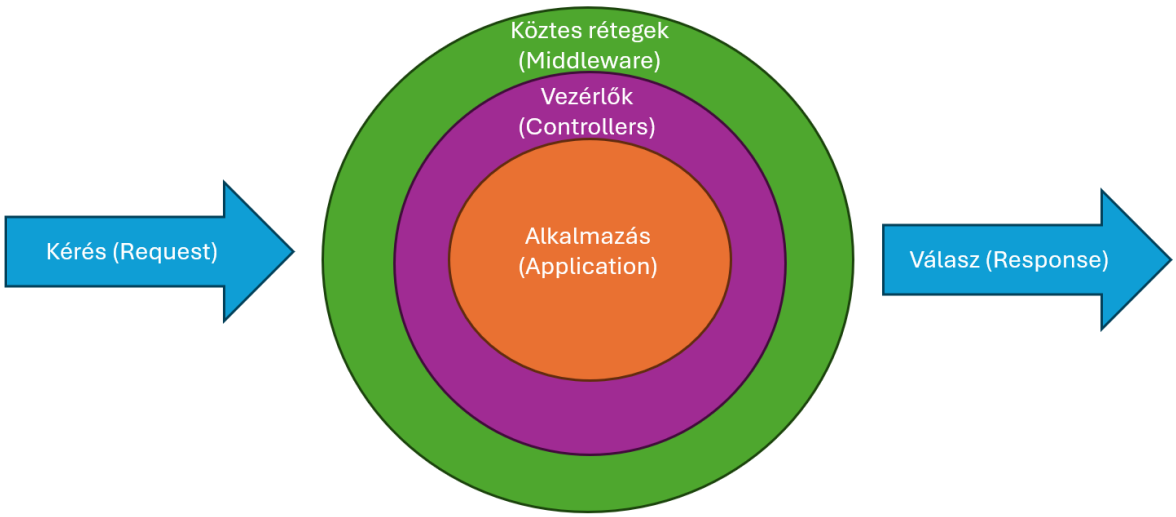
A felhasználói hitelesítés a legtöbb webes alkalmazásban alapvetően szükséges. Emiatt a Laravel is többféle lehetőséget biztosít a számunkra ahhoz, hogy védett tartalmakat készítsünk a regisztrált felhasználóink számára, és azokat csak hitelesítés után tegyük elérhetővé a regisztrált és bejelentkezett felhasználóinknak. A Breeze, Jetstream, Fortify (és a Sanctum csomag) mind olyan *„kezdő készletek",* amelyek egyszerű telepítésével, testre szabásával és használatával a felhasználói hitelesítés könnyedén végrehajtható lesz az alkalmazásainkban. A tesztelés itt is mindig kiemelt szerepet kap az alkalmazásunk helyes működésének ellenőrzése miatt.

10.1. A köztes réteg (Middleware)

Ebben az alfejezetben egy központi elemet, a rendszer magját érintő területet fogunk körüljárni: a Middleware-ek, vagy más néven, köztes rétegek szerepét vizsgáljuk meg közelebbről. A Laravel 10-ben és 11-ben a köztes rétegek kezelése merőben eltérő, úgyhogy az alkalmazásukat mindkét verzióban részletezzük, és példákon keresztül elsajátítjuk az alkalmazásukat.

10.1.1. Mik azok a köztes rétegek?

Ezek vizsgálják meg és szűrik az alkalmazásunk felé érkező HTTP kéréseket, amelyek az alkalmazásunkhoz akarnak hozzáférni. Például, ha vannak regisztrált felhasználóink, akkor ennek segítségével könnyedén fogjuk tudni ellenőrizni, hogy adott látogató *(még csak látogató!)* bejelentkezett-e már. Ha olyan tartalomhoz szeretne hozzáférni, amihez bejelentkezés (hitelesítés) szükséges, és még nincsen bejelentkezve a látogató, akkor tovább irányítjuk a bejelentkezési felületre. Ha a Middleware azt érzékeli a vizsgálat során, hogy a felhasználó már bejelentkezett, akkor engedni fogja neki a hozzáférést az adott tartalmakhoz, és a kérésére válaszul meg is kapja a tartalmakat a böngészőjében a bejelentkezett felhasználó. Egy másik, alapértelmezetten létező és működő Middleware, amit már korábban használtunk, az az űrlapoknál használt `@csrf` Blade direktíva által generált CSRF token ellenőrzése aszerint, hogy az űrlap elküldése és adatainak fogadása után megegyezik-e a token tartalma. Ez utóbbiba már többször belefuthattunk, amikor HTTP 419 állapotkódú hibát kaptunk, az gyakorlatilag ennek a köztes rétegben lévő ellenőrzésnek az elbukása miatt volt.



10–1. ábra: Felhasználói kérések kiszolgálása a köztes rétegekkel

Úgy is fogalmazhatnánk, hogy a Middleware-ek gyakorlatilag hidat képeznek a HTTP kérések és válaszok között, miközben megvizsgálják, és megszűrik a kéréseket.

Léteznek alapértelmezetten a Laravel projekttel együtt létrejövő Middleware-ek, mint az imént említett kettő is, de természetesen mi magunk is tudunk definiálni sajátokat, egyedieket.

Laravel 11: megújult struktúra és fájlszerkezet (*„slimmer application skeleton"* ötlet).

A Laravel 11-ben meglehetősen nagy változások történtek a Middleware-ekkel kapcsolatban. Ezért emiatt áttekintjük, hogy hogyan működnek a Laravel 10-ben, aztán pedig megnézzük, hogy miként változik meg (ha nem is a működésük, de a kezelésük mindenképpen) a Laravel 11-ben. Így azt is át tudjuk majd látni, hogy amit elrejt előlünk a keretrendszer, az igazából mit csinál a háttérben.

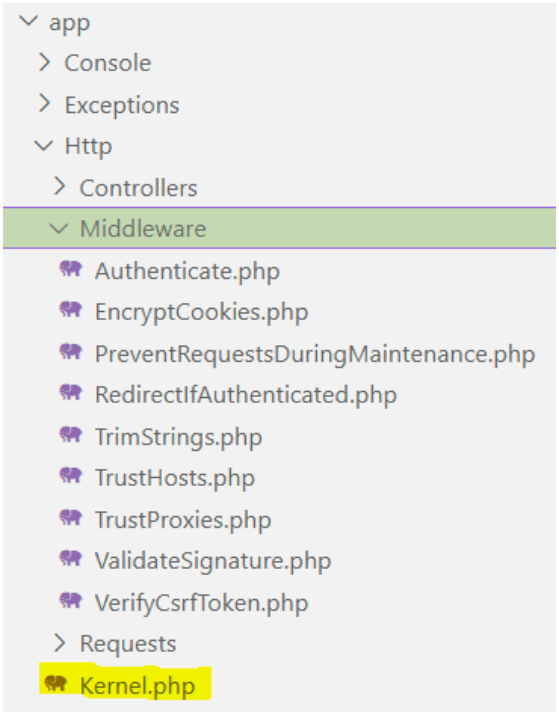
10–1. újdonság: Laravel projekt struktúra csökkenés a Middleware-ek kapcsán is: új logika, új helyen

10.1.2. Köztes rétegek a Laravel 10-ben

Az alapértelmezetten meglévő Middleware-ek, köztes rétegek védik az alkalmazásunkat a nem megfelelő kérésekkel szemben. Ebben az alfejezetben áttekintjük a Middleware-ek típusait és azt, hogy mikor és hol lépnek be a felhasználói kérések kiszolgálásának folyamatába.

10.1.2.1. Globálisan (Global) létező köztes rétegek

A globális Middleware-ek helye az **app / Http / Middleware** mappában van, osztályok és funkcionalitásaik megtalálhatók ezekben a **.php** fájlokban.



10–2. ábra: Middleware-ek helye a mappastruktúrában és a Kernel.php

A globális szó ebben a kontextusban arra vonatkozik, hogy ezek azok a Middleware-ek, amelyek minden egyes HTTP kérés érkezésekor lefutnak. Ehhez azonban nem elég, hogy bekerültek ebbe a Middleware mappába, hanem regisztrálnunk is kell őket (hasonlóan, mint egy útvonalakat a **web.php** fájlban). Ehhez a regisztrációhoz az **app / Http / Kernel.php** fájlra van szükségünk, abban pedig a **protected $middleware** tömbhöz kell hozzáadnunk a globálisan lefuttatni kívánt Middleware-eket. Alapértelmezetten ezek a Middleware-ek futnak le globálisan, tehát minden egyes HTTP kérés esetén, érkezzen az akár egy webes böngészésből, vagy akár egy API hívásból. A **Kernel.php** (ez a **Http** mappában van, nem a **Middleware**-ben), ahogy a neve is mutatja, a rendszer magját jelképezi a HTTP kérések szempontjából.

```
protected $middleware = [
  // \App\Http\Middleware\TrustHosts::class,
  \App\Http\Middleware\TrustProxies::class,
  \Illuminate\Http\Middleware\HandleCors::class,
  \App\Http\Middleware\PreventRequestsDuringMaintenance::class,
  \Illuminate\Foundation\Http\Middleware\ValidatePostSize::class,
  \App\Http\Middleware\TrimStrings::class,
  \Illuminate\Foundation\Http\Middleware\ConvertEmptyStringsToNull::class,
];
```

10–1. kódrészlet: Globális Middleware-ek regisztrációja a Kernel.php-ban

Amelyik Middleware itt ki van kommentezve (**TrustHosts**), az nincs is regisztrálva, a fejlesztők csak amiatt tehették be, hogy ha subdomain-eket használunk majd az alkalmazásunk kitelepítése *(deployment)* során, akkor ennek a Middleware-nek a regisztrációjára szükség lehet.

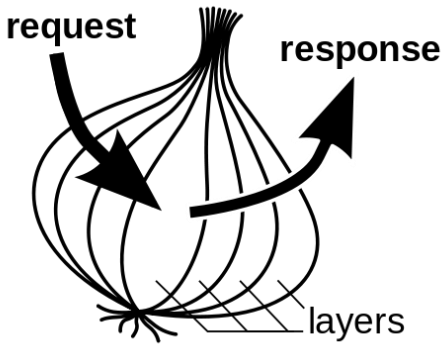
A **$middleware** tömbben lévő globálisan létező Middleware-ek reprezentálnak egy-egy *réteget*, amelyek lehetőséget biztosítanak arra például, hogy cache-eljünk, felhasználói engedélyeztetést vizsgáljunk, vagy

éppen átirányítsuk valahova a felhasználót. Tehát elég sok mindent megtehetünk, amit csak akarunk a felhasználóktól érkező kérések és válaszok kezelésénél is.

Ha megnézzük az ebben a mappában lévő **Authenticate.php** Middleware-t, akkor azt láthatjuk, hogy ebben mindössze egy metódus van, ami tovább irányítja a kérést a bejelentkezési (login) útvonalra, ha nincs még hitelesítve a felhasználó. Érdemes megnézni azt az osztályt is, amit kiterjeszt ez az utódosztály (**Illuminate\Auth\Middleware\Authenticate**). Ebből azt szűrhetjük le, hogy az utódosztály (**Authenticate**) azért nem csak ilyen egyszerű, mint elsőre tűnik, hanem ez a rendszer magjában lévő felhasználói azonosítást egészíti ki a tovább irányítást végző metódussal. Akit érdekelnek a rendszer mélységei, az könnyen *„le tud fúrni"* benne, ha rákattint az ősosztály útvonalára, majd F12-t nyom a VSCode-ban, akkor rögtön látja is ennek a hivatkozott osztálynak a forráskódját, és mivel a Laravel keretrendszer magja nyílt forráskódú, ezért bármilyen mélységig lefúrhatunk a részletekért.

Minden Middleware tartalmaz egy metódust, aminek a neve: **handle()**. Ezen belül bármilyen feladatot megcsinálhatunk (mint például, amiket az imént felsoroltunk), vagy éppen csak tovább dobjuk a következő *rétegnek* a kérést (**return $next($request);**). Ha megvizsgáljuk az ebben az ősosztályban lévő **authenticate()** metódust, akkor abból könnyen világossá válhat számunkra, hogy egy azonosítást hajt végre, és ha az nem sikeres, akkor egy **AuthenticationException**-nel (*Unauthenticated* szövegezéssel) tér vissza nekünk (*megjegyzés:* nem muszáj a forráskód összes többi sorát értenünk most, mert nem azok a lényegesek a számunkra).

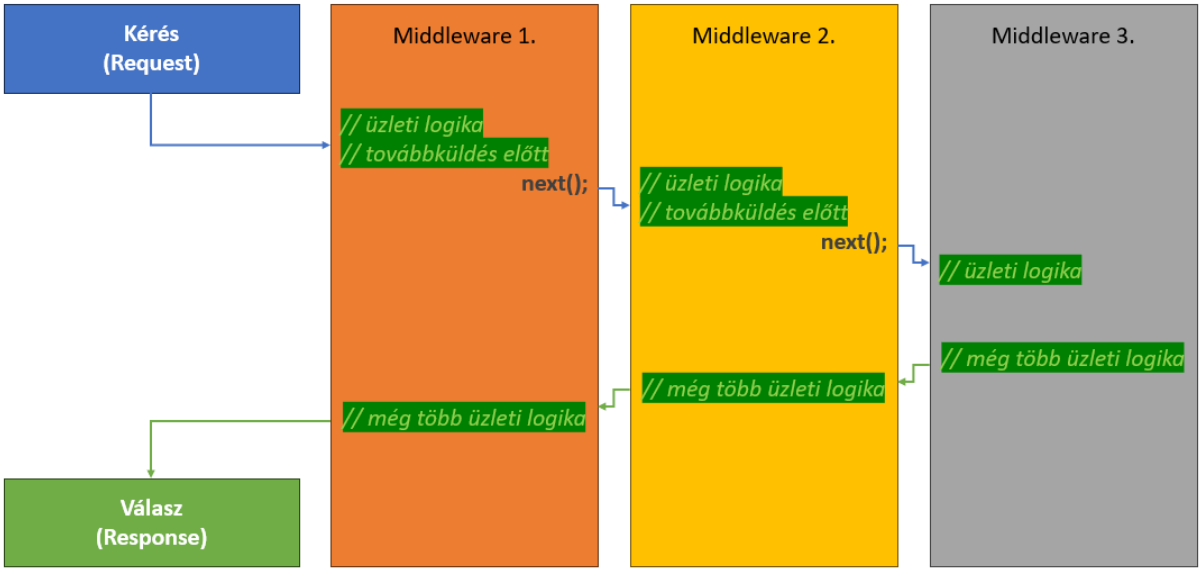
Az imént a *réteg* szót használtam és ez nem véletlen: ugyanis a **Kernel.php**-ban lévő **$middleware** tömbelemek gyakorlatilag egy-egy réteget képviselnek, amelyeken át kell jutnia a felhasználótól/látogatótól érkező HTTP kéréseknek és a válaszok is ezen az úton jutnak vissza a felhasználóhoz/látogatóhoz. Ha valamelyik rétegen nem jut át a kérés, mert például nincsen bejelentkezve a látogató, akkor nem kerül tovább a kérés, hanem elirányítjuk őt egy bejelentkezési oldalra, tehát nem fog olyan választ kapni, aminek a segítségével ő hozzáférhetne bizonyos webalkalmazás tartalmakhoz. Erre egy nagyon jó szemléltetés a hagyma ábrája:



10–3. ábra: A HTTP kérések beérkezése a rétegeken keresztül az alkalmazás magjához, a HTTP válasz visszaküldése is a rétegeken megy keresztül

10.1.2.2. Köztes réteggel történő ellenőrzés ideje

A Middleware-ek segítségével lefutásuk (ellenőrzésük) *előtt* és *után* is végrehajthatunk különböző feladatokat. Ebből a szempontból megkülönböztetünk *„before"* és *„after"* Middleware-eket. Ezt jól szemlélteti a következő ábra:



10–4. ábra: Felhasználói kérés Middleware-beli továbbküldés előtti és utáni üzleti logikai funkciók ábrázolása

A *„before"* Middleware-rel a kérés tovább engedése, vagyis a következő réteghez irányítása előtt lehet végrehajtani műveleteket, például ellenőrizni azt, hogy rendelkezik-e valamilyen jogosultsággal (vagy tartozik-e valamilyen szerepkörhöz) a felhasználó, és csak akkor engedi tovább, ha az ellenőrzés pozitív eredménnyel zárult.

```
public function handle(Request $request, Closure $next): Response
{
  // Feladatok végrehajtása a továbbküldés előtt

  return $next($request);
}
```

10–2. kódrészlet: Middleware-en belüli kérés továbbítása előtti ellenőrzés vagy feladat végrehajtás ("before")

Ezzel szemben az *„after"* Middleware-rel a kérés tovább engedése után hajtunk még végre a Middleware-en belül valamilyen üzleti logikát, például naplózást.

```
public function handle(Request $request, Closure $next): Response
{
  $response = $next($request);

  // Feladatok végrehajtása a továbbküldés után

  return $response;
}
```

10–3. kódrészlet: Middleware-en belüli kérés továbbítása utáni feladat végrehajtás ("after")

10.1.2.3. Útvonalakhoz (Route) tartozó köztes réteg csoportok

Az alkalmazásainkhoz alapvetően két irányból érkezhetnek felhasználói kérések:

1. WEB-ről (egyszerű böngészések által),
2. API-ról (valamilyen saját kliens oldali keretrendszer vagy más alkalmazások által).

Ha most az API oldalra koncentrálunk, akkor egy problémát mindenképpen meg tudunk oldani Middleware-rel. Korábban, ha nem létezett az erőforrás egy lekérésnél, akkor 404-es hibaoldalt kaptunk HTML szerkezetben, hacsak át nem állítottuk az *„Accept"* fejlécet **application/json**-ra manuálisan a Postman-ben (0. alfejezet). Azonban ennek a beállításnak automatikusnak kellene lennie. Ilyenkor adja magát a lehetőség, hogy egy új köztes réteget építsünk a felhasználói kérések kiszolgálásakor, és kikényszerítsük a válasz elfogadható formátumát. Hozzunk létre egy új köztes réteget:

```
php artisan make:middleware JsonResponseApiMiddleware
```

A **handle()** metódus tartalma legyen a következő:

```php
public function handle(Request $request, Closure $next): Response
{
  $request->headers->set('Accept', 'application/json', true);

  return $next($request);
}
```

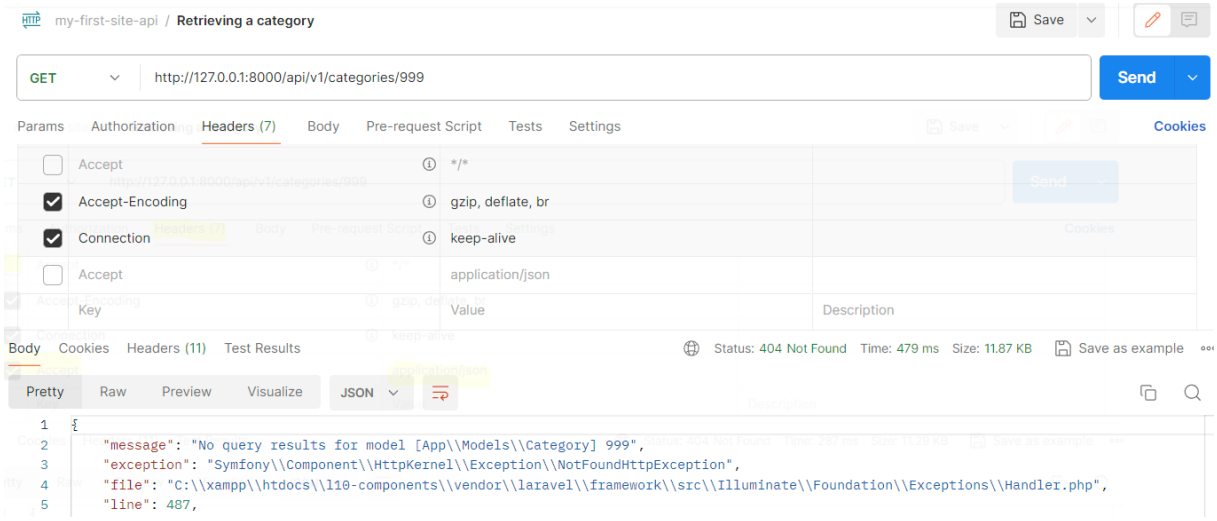
10–4. kódrészlet: JsonResponseApiMiddleware handle() metódusának tartalma

A végrehajtás idejének szempontjából ez egy *„before"* Middleware. Az **app / Http / Kernel.php**-ban pedig adjuk hozzá az **'api' $middlewareGroups** tömbjének legelejére az új köztes réteget:

```php
'api' => [
    \App\Http\Middleware\JsonResponseApiMiddleware::class,
    // …
   ],
```

10–5. kódrészlet: Kernel.php-ban az api MiddlewareGroups új eleme hozzáadásra került

A megoldás tesztelését a Postman-ben elvégezhetjük az egy kategóriát lekérő **show** útvonalnál. Vegyük ki az *„Accept"* fejléceknél a pipát a sor elején és futtassuk újra a kérést a 999. kategória lekérésével.



10–5. ábra: Manuális Accept fejléc érték beállítása nélkül is JSON eredményt kapunk vissza

Az alfejezetben végrehajtott programkód módosítások ebben a GitHub commit-ben találhatók meg.

10.1.2.4. Útvonalakhoz tartozó köztes rétegek

Lehetőségünk van arra is, hogy bizonyos útvonalakhoz, vagy útvonalak egy saját magunk által definiált csoportjához rendeljünk hozzá Middleware-eket. Ennek az lehet a célja, hogy bizonyos útvonalakat csak regisztrált és bejelentkezett felhasználóknak engedünk elérni, vagy épp fordítva, csak egyszerű látogatóknak engedhetünk elérni. A Middleware-ek segítségével tudunk az útvonalakhoz különböző engedélyeztetéseket is definiálni a későbbiekben (lásd 10. fejezet). *Megjegyzés:* a Laravel 11-ben egy útvonalhoz tartozó Middleware-t a 11.3. alfejezetben adunk hozzá a rendszerhez példaként.

Egy már létező példa is van erre az útvonal Middleware-re az alkalmazásunkban, mégpedig az **api.php**-ban lévő **/user** útvonal, amely felhasználói hitelesítéssel (Sanctum csomag segítségével) érhető csak el az API felületen keresztül.

```
Route::middleware('auth:sanctum')->get('/user', function (Request $request)
{
  return $request->user();
});
```

10–6. kódrészlet: Az api/user útvonal eléréséhez Sanctum általi felhasználói hitelesítés szükséges

A **laravel/sanctum** csomagot a legutóbbi Laravel verziók már alapértelmezetten tartalmazzák.

Az útvonalakhoz vagy útvonalak egy csoportjához rendelhető Middleware-ek az **app / Http / Kernel.php**-ben találhatók meg, azon belül pedig a **$middlewareAliases** tömbben (korábban ezt **$routeMiddleware** tömbnek hívták). Az asszociatív tömb kulcsait hozzárendelhetjük útvonalakhoz, utána pedig a tömbelem értékeiben meghatározott osztályok rétegein kell túljutnia az útvonalhoz érkező felhasználói kérésnek.

Ha létrehozunk egy új útvonalat a **web.php**-ban, akkor hozzá tudjuk rendelni például az **auth** Middleware-t azért, hogy az adott útvonalat csak felhasználói hitelesítés után lehessen elérni:

```
Route::get('/profile', function ()
{
  return "My profile";
```

```
})->middleware('auth');
```

10–7. kódrészlet: Új /profile útvonal elérése felhasználói hitelesítéshez kötött

Ha most megpróbáljuk elérni a **/profile** útvonalat, akkor azt kapjuk, hogy a login útvonal nincsen definiálva a rendszerben. Ezt pedig azért kapjuk, mert a Laravel **auth** Middleware-je megpróbál hitelesíteni minket, ami sikertelenül végződik, ezért az **app / Http / Middleware / Authenticate.php** osztályban lévő metódus megpróbál átirányítani minket egy bejelentkezési (**login**) útvonalra, de ez nem létezik még a rendszerben.

Middleware-eket útvonal csoportokhoz is hozzá lehet rendelni, például egy **resource** útvonal csoporthoz: kössük a **comments** útvonalak elérését felhasználói hitelesítéshez:

```
Route::resource('comments', CommentController::class)->middleware('auth');
```

10–8. kódrészlet: 7 erőforráshoz kötődő RESTful útvonal elérése is felhasználói hitelesítést igényel

Az iménti megoldással megegyező eredményre jutunk, ha nem az erőforrás útvonalánál, hanem a vezérlő metódusához adunk hozzá egy konstruktort, amelyben jelöljük, hogy az osztály bármely metódusának elérése felhasználói hitelesítést igényel.

Ha a **CategoryController**-hez hozzáadunk egy konstruktort, akkor ugyanazt fogjuk tapasztalni a **/categories** és az azon belüli útvonalak elérésnél is (ekkor a további Controller metódusok végrehajtása előtt mindenképpen végrehajtódik a konstruktor, így a felhasználói hitelesítés ellenőrző köztes réteg):

```
public function __construct()
{
  $this->middleware('auth');
}
```

10–9. kódrészlet: "Útvonal" Middleware alkalmazása a CategoryController osztályon belül

Ezt persze lehet finomítani, hogy csak bizonyos útvonalakra legyen érvényes a csoporton belül, vagy pedig pont bizonyos útvonalakra ne legyen érvényes. Nézzünk is ezt meg, például a létrehozást, a frissítést és a törlést csak felhasználói hitelesítés után lehet megtenni, míg a listázást és egy elem megtekintését enélkül is megtehetjük:

```
public function __construct()
{
  $this->middleware('auth')->only('create', 'store', 'edit', 'update',
'delete');
}
```

10–10. kódrészlet: Csak bizonyos Controller metódusok hozzáférését korlátozzuk köztes réteggel

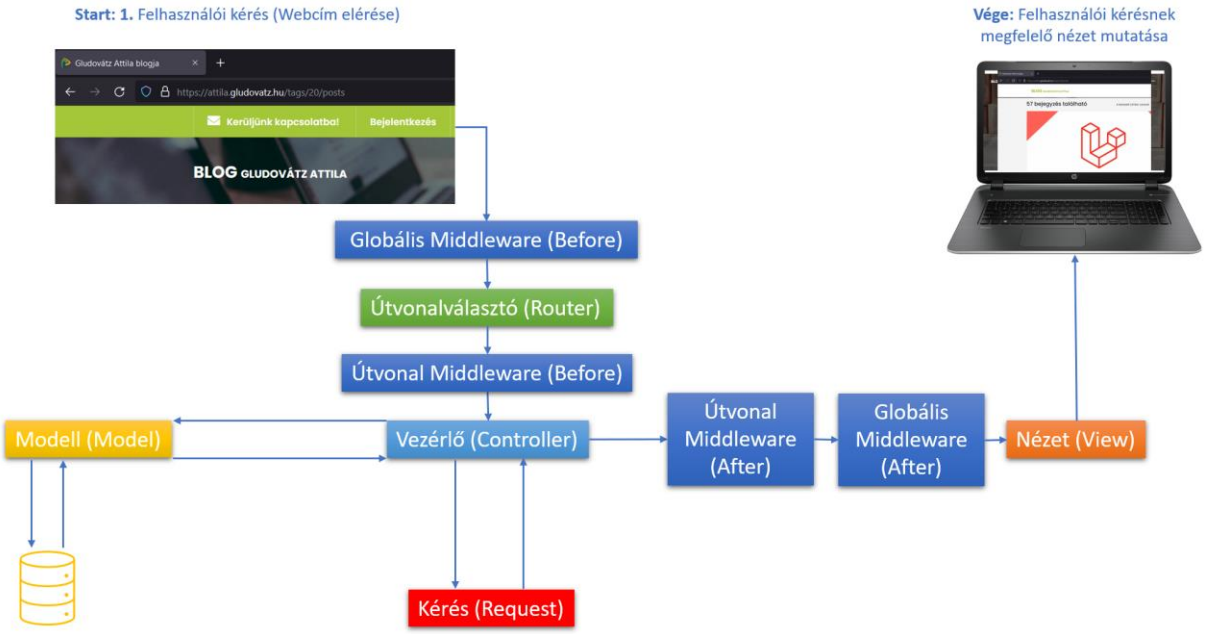
Ennek ellentéte is működik, ha az **only()** helyett az **except()**-et használjuk az **index** és **show** metódusok nevével.

Ha meg szeretnénk látogatni azokat az útvonalakat, amelyek a köztes réteg által védettek lettek, akkor a Laravel átirányítaná a felhasználót a **login** útvonalra, hogy be tudjon jelentkezni, azonban ez még nem létezik, így hibát kapunk. Érdemes lenne az ilyen linkeket, amelyek nem elérhetőek a nem hitelesített látogató számára (kategória létrehozása például) eltüntetni a nézet oldalról, és csak azoknak

megjeleníteni, akik képesek lehetnek elérni azt. Ezt a funkcionalitást a későbbiekben (10.2.4. alfejezet) fogjuk hozzáilleszteni az alkalmazásunkhoz.

Megjegyzés: ez a módosítás itt a manuális tesztelés során hibát eredményezett, de ugyanígy hibát eredményez az automatikus tesztelés során is, amikor a **Feature / CategoryTest.php** teszteseteit ellenőrzi a rendszer.

A köztes rétegek az alább látható helyeken játszanak szerepet a felhasználói kérések kiszolgálásának folyamatában:



10–6. ábra: Felhasználói kérések kiszolgálása során szerepet játszó Middleware-ek helye és szerepe

Megjegyzés: Ennél az ábránál nem a Request és Model osztályokon vagy az adatbázison van a fő hangsúly, hanem a Middleware-ek elhelyezkedésén a felhasználói kérések kiszolgálása során. Az API kérés esetén természetesen nem a nézeten *„keresztül"* fog megtörténni a felhasználói kérés kiszolgálása.

10.1.3. Köztes rétegek a Laravel 11-ben

A legnagyobb változás talán itt is az, amit korábban már többször feljegyeztünk (amikor a Laravel 10 és 11 közötti különbségeket tárgyaltuk), hogy vékonyabb lett a könyvtár- és fájlstruktúra, eltűntek elemek, amik korábban megvoltak (**app / Http / Middleware** mappa és teljes tartalma). Ennek *„köszönhetően"* eltűnt a fő Middleware beállítási (regisztrálási) fájl a **Kernel.php** az **app / Http** mappából.

Új köztes réteget a már megismert paranccsal tudunk létrehozni:

```
php artisan make:middleware MyFirstMiddleware
```

Megjegyzés: az alfejezetben található példák miatt hozzuk létre ugyanígy a **MySecondMiddleware**-t is. Mindkét új Middleware-t minimális tartalommal töltsük fel, mivel itt a regisztrálásukra, hozzáadásukra, eltávolításukra koncentrálunk.

Az utasítások hatására az új Middleware-ek bekerültek a *„szokásos”* helyére: **app / Http / Middleware** mappába. Tartalmuk legyen ez (számozásban eltérőek: a **MySecondMiddleware**-ben a **#** utáni számot cseréljük ki 2-re 1 helyett):

```php
public function handle(Request $request, Closure $next): Response
{
  Log::info("#1 " . $request->url());

  return $next($request);
}
```

10–11. kódrészlet: MyFirstMiddleware handle() metódusa

Az osztályok elején pedig importáljuk ezt a Facade osztályt: **Illuminate\Support\Facades\Log**

A Middleware-ek regisztrálása viszont a korábbiakkal ellentétben átkerült a központi **bootstrap / app.php** fájlba. Ez az alkalmazás beállításainak a helye, a Middleware-eket pedig a **withMiddleware()** metódusban tudjuk regisztrálni. Viszont, ha megnézzük ennek a **withMiddleware()** metódusnak a definícióját az ősosztályában, akkor már rögtön olyan dolgokat találunk, amelyek ismerősek lehetnek.

```php
/**
 * Register the global middleware, middleware groups, and middleware aliases for the application.
 *
 * @param  callable|null  $callback
 * @return $this
 */
public function withMiddleware(?callable $callback = null)
{
    $this->app->afterResolving(HttpKernel::class, function ($kernel) use ($callback) {
        $middleware = (new Middleware)
            ->redirectGuestsTo(fn () => route('login'));

        if (! is_null($callback)) {
            $callback($middleware);
        }

        $this->pageMiddleware = $middleware->getPageMiddleware();
        $kernel->setGlobalMiddleware($middleware->getGlobalMiddleware());
        $kernel->setMiddlewareGroups($middleware->getMiddlewareGroups());
        $kernel->setMiddlewareAliases($middleware->getMiddlewareAliases());

        if ($priorities = $middleware->getMiddlewarePriority()) {
            $kernel->setMiddlewarePriority($priorities);
        }
    });

    return $this;
}
```

10–7. ábra: A withMiddleware() metódus definíciója

Itt már láthatók azok dolgok, amelyeket korábban az **app / Http / Kernel.php**-ban regisztráltunk: **getGlobalMiddleware(), getMiddlewareGroups(), getMiddlewareAliases().** Ha pedig ezeknek a metódusoknak is megnézzük a definiálását (például a **getGlobalMiddleare()**-ét), akkor ténylegesen azokhoz

a tömbökhöz juthatunk, amiket a Laravel korábbi verzióiban láttunk, lásd például az alábbi ábrát összevetve a 10–1. kódrészlettel.

```php
/**
 * Get the global middleware.
 *
 * @return array
 */
public function getGlobalMiddleware()
{
    $middleware = $this->global ?: array_values(array_filter([
        $this->trustHosts ? \Illuminate\Http\Middleware\TrustHosts::class : null,
        \Illuminate\Http\Middleware\TrustProxies::class,
        \Illuminate\Http\Middleware\HandleCors::class,
        \Illuminate\Foundation\Http\Middleware\PreventRequestsDuringMaintenance::class,
        \Illuminate\Http\Middleware\ValidatePostSize::class,
        \Illuminate\Foundation\Http\Middleware\TrimStrings::class,
        \Illuminate\Foundation\Http\Middleware\ConvertEmptyStringsToNull::class,
    ]));

    $middleware = array_map(function ($middleware) {
        return isset($this->replacements[$middleware])
            ? $this->replacements[$middleware]
            : $middleware;
    }, $middleware);

    return array_values(array_filter(
        array_diff(
            array_unique(array_merge($this->prepends, $middleware, $this->appends)),
            $this->removals
        )
    ));
}
```

10–8. ábra: A getGlobalMiddleware() metódus globális köztes rétegeit tartalmazó metódus

Itt tehát megint leginkább arról van szó, hogy a Laravel 11 elrejt előlünk dolgokat, de utánuk tudunk nézni (a nyílt forráskódjának köszönhetően), engedélyezni tudjuk a használatukat akár.

Ha ezeken az alapértelmezett köztes rétegeken kívüli új köztes réteget akarunk regisztrálni a rendszerünkben, akkor a **bootstrap / app.php**-ban lévő **withMiddleware()** metóduson belül tudjuk ezt megtenni. Globális Middleware-t az **append()** metódussal tudunk hozzáadni, míg útvonalakhoz tartozó Middleware-eket a **web()** vagy az **api()** metódussal tudunk hozzáadni a köztes rétegek listájának végéhez.

Néhány példát érdemes megvizsgálni a **bootstrap / app.php**-ban ahhoz, hogy utána könnyedén tudjuk majd alkalmazni a saját Middleware-jeinket (a forráskódban megjegyzésekkel magyarázom, hogy az utána következő kódsorok mit csinálnak):

```php
// Alias név regisztrálása, hogy könnyebben hozzá lehessen adni
útvonalakhoz/útvonal csoportokhoz a köztes réteget
$middleware->alias([
  'some_key' => MyFirstMiddleware::class,
```

```
]);

// Hozzáadás a rétegek végére egyesével
$middleware->append(MyFirstMiddleware::class);

// Beillesztések a meglévő rétegek elé
$middleware->prepend([
  MyFirstMiddleware::class,
  MySecondMiddleware::class,
]);

// Regisztrált réteg eltávolítása egyesével
$middleware->remove(\Illuminate\Http\Middleware\ValidatePostSize::class);
// Több regisztrált réteg eltávolítása
$middleware->remove([
  \Illuminate\Http\Middleware\TrustProxies::class,
  \Illuminate\Http\Middleware\HandleCors::class,
]);

// Egy konkrét réteg hozzáadása a web Middleware csoporthoz
$middleware->appendToGroup('web', MyFirstMiddleware::class);

// web-es Middleware csoporthoz való hozzáadás (append) így:
$middleware->web([MyFirstMiddleware::class, MySecondMiddleware::class]);
```

10–12. kódrészlet: Middleware-ek regisztrálása és eltávolítása egyesével és csoportosan a withMiddleware()-ben

A működés helyességének ellenőrzésére töltsük be az alkalmazásunk kezdőoldalát a böngészőben, és utána ellenőrizzük az alkalmazásunk log-ját a **storage / logs / laravel.log** fájlban (mindig a fájl végére kerülnek a legfrissebb bejegyzések). A meglévő kódsoroknak köszönhetően egy oldalletöltés következtében a log fájlunk ezekkel a kódsorokkal bővült:

```
[2024-03-30 17:41:33] local.INFO: #1 http://127.0.0.1:8000
[2024-03-30 17:41:33] local.INFO: #2 http://127.0.0.1:8000
[2024-03-30 17:41:33] local.INFO: #1 http://127.0.0.1:8000
[2024-03-30 17:41:33] local.INFO: #2 http://127.0.0.1:8000
```

10–13. kódrészlet: Köztes rétegeken való áthaladás naplózása

Mert bár többször fűztük be ugyanazt a két Middleware-t, amikor egymás után ugyanazt fűztük be a rétegekhez, akkor annak nem volt hatása. Hiszen, ha egyszer sikeresen átmegy a kérés adott Middleware-en, akkor újra nem kell ugyanúgy átmennie. Ellenben, ha egy másik Middleware jön utána, akkor újra van értelme befűzni az elsőt. Ezért látszódik így a log-ban lévő felsorolásban, hogy egymás után felváltva jönnek a rétegek.

Megjegyzés: ez a naplózó fájl sokszor segít a hibakeresés folyamatában, akkor főleg, amikor már élesben (**production**) futtatjuk az alkalmazásunkat. Ilyenkor a környezeti **.env** fájlban az **APP_ENV** beállítás **production** és az **APP_DEBUG** beállítás **false**-ra van állítva, ha pedig szerver oldali hiba történik, akkor a

böngészőben 500-as HTTP státuszkódot kapunk. Ekkor a **laravel.log** fájlhoz fűzi hozzá a tényleges problémát a keretrendszer, és így tudjuk a hiba okát megtalálni.

Adott speciális útvonal Middleware-t a Controller osztály konstruktorában is alkalmazhatunk, ahogy azt korábban már láttuk a 10–9. kódrészletben. Hozzunk létre egy példa Controller osztályt, amelyben használjuk az **auth** útvonal Middleware-t:

```
php artisan make:controller HomeController
```

```php
use Illuminate\Routing\Controllers\HasMiddleware;
use Illuminate\Routing\Controllers\Middleware;

class HomeController extends Controller implements HasMiddleware
{
  public static function middleware(): array
  {
    return [
      // minden Controller action-höz rendelünk hitelesítést
      'auth',
      // adott Controller action-ökhöz rendelünk hitelesítést
      new Middleware('auth', only: ['dashboard', 'profile']),
      // bizonyos Controller action-ökhöz nem rendelünk hitelesítést, de a
többihez igen
      new Middleware('auth', except: ['teams']),
    ];
  }
}
```

10–14. kódrészlet: ""Útvonal" Middleware alkalmazása a Controller-ben Laravel 11 esetén

A teljes osztályt, importálásokkal és kiterjesztéssel együtt beillesztettem ide, mivel a beszúrt megjegyzések eléggé jól dokumentálják az egyes funkcionalitásokat, amelyeket alkalmazhatunk itt.

Az alfejezetben végrehajtott programkód módosítások ebben a GitHub commit-ben találhatók meg.

10.2. Felhasználói hitelesítés (Authentication, starter kits)

Webfejlesztőként először arra kell felkészítenünk a webes alkalmazásunkat, hogy látogatók érkeznek, akik csak böngészni szeretnék a tartalmainkat. Azonban nagyon hamar eljön az az állapot, amikor már bizonyos tartalmakat le szeretnénk védeni, azt szeretnénk elérni, hogy csak az oldalunkra regisztrált felhasználók tudjanak megnézni bizonyos speciális tartalmakat. Ekkor már mindenképpen foglalkoznunk kell a felhasználói hitelesítés témakörével.

A felhasználói hitelesítés folyamata egy potenciálisan kockázatos tevékenység, így fokozott figyelemmel kell eljárnunk, amikor megválasztjuk és megvalósítjuk a webes alkalmazásunkban a felhasználói hitelesítés módját.

A Laravel hitelesítési eszközei között megtalálhatóak a *guard* és *provider* elemek. A guard-ok határozzák meg, hogy a felhasználók hitelesítése hogyan történik meg minden egyes kérésnél. A Laravel például a

munkamenetet (session-t) is ilyen guard-dal védi és ennek segítségével végzi el a munkamenetek eltárolását és a sütik (cookie-k) segítségével tartja fenn ezt az állapotot. A provider-ek határozzák meg, hogy a felhasználók hogyan kerülnek lekérése a tartós tárolókból, például az adatbázisokból. Ehhez az Eloquent-et és a Query Builder-t is tudjuk használni alapértelmezetten, de mi is szabadon bővíthetjük ezeknek a körét.

Alapértelmezetten a Laravel a felhasználói adatok eltárolásához az adatbázisban lévő **users** adattáblát használja, amelyhez az **app / Models / User** Eloquent Model osztályon keresztül fér hozzá, ez a **config / auth.php** beállítási fájlban van definiálva és inicializálva. Ha ebben a fájlban átállítanánk a **user providers** szekciót **eloquent**-ről **database**-re, akkor a Query Builder-rel kellene a felhasználó adatait kezelni az adatbázisban.

Webböngésző használata esetén a felhasználó egy bejelentkezési űrlapon keresztül adja meg például a felhasználónevét és jelszavát. Ha ezek a hitelesítő adatok helyesek, az alkalmazás a hitelesített felhasználó adatait a felhasználói munkamenetben (session) tárolja. A későbbi kérésekhez a webböngésző eltárolja a munkamenet azonosítót és ha ez egyezik az alkalmazás által eltárolt azonosítóval, akkor a felhasználót *„hitelesítettnek”* tekinti az alkalmazásunk.

A Laravel és kezdő készletei a felhasználói hitelesítés megvalósítására többféle opciót támogatnak. Képes működni a web-es kérések esetén leginkább (űrlapon megadott) jelszó ellenőrzési alapon, API kérések esetén pedig token egyezőségi alapon, de a kétfaktoros hitelesítést és az egyéb, 3. fél által (Google, Facebook, GitHub stb.) támogatott regisztrációs/bejelentkezési eljárásokat is támogatja.

API kérések esetén nem munkameneteket vagy sütiket használ a hitelesítésre a rendszer, hanem minden egyes kérés mellé küld az alkalmazásnak egy token-t, amely ellenőrzésre kerül, és ha érvényes, akkor kiszolgálásra kerül a kérés.

A Laravel az **Auth** és **Session** Facade-okat biztosítja az üzleti logika felhasználói hitelesítési funkcionalitásainak támogatására. Ugyanez a két segéd a nézeteknél alkalmazható Blade direktívák között is elérhető (**@auth**, **@session**) ellenőrzés céljából.

Az említett hitelesítési eljárásokon kívül a Laravel biztosít a számunkra úgynevezett kezdő készleteket (starter kits), amelyek nem csak a háttérben (szerver oldalon) működő funkcionalitásokat biztosítják a felhasználók hitelesítéséhez, hanem kliens oldali megoldásokat is a rendelkezésünkre bocsát. Ezek a kezdőkészletek tartalmazzák a hitelesítést végző Controller-eket, útvonalakat, nézeteket és automatikus teszteket is. Ha pedig valamit másképp szeretnénk, mint ahogy alapból megkapjuk őket, akkor lehetőségünk van az igényeink szerinti testre szabásra is. Mindezek mellett (vagy inkább helyett) használhatjuk a saját magunk által definiált hitelesítési réteget is, ezek csak lehetőséget nyújtanak számunkra a fejlesztés meggyorsítására, mivel számos elemet készen kapunk általuk.

10.2.1. Breeze hitelesítési csomag

Ez a felhasználói hitelesítést támogató eszköz egy (a többihez képest) kissé egyszerű megoldást nyújt nekünk. Már a Laravel korábbi verzióiban is használható volt, de folyamatosan fejlesztik a funkcionalitásait és a megjelenítési rétegeit is. A Laravel 7-es verziótól nevezik Breeze-nek, mivel innentől már több ilyen

hitelesítési eszköz állt a rendelkezésünkre, és meg kellett különböztetni őket egymástól. De nézzük át, hogy mire is képes ez az eszköz. Lehet vele:

- felhasználókat regisztrálni,
- felhasználókat be- és kijelentkeztetni,
- jelszóemlékeztetőt lehet vele kezelni
- e-mail-es megerősítést lehet beilleszteni a regisztráció folyamatába.

A nézet fájlokban a Blade sablon szerinti utasításokkal, direktívákkal tudjuk majd kezelni, irányítani a felhasználóinkat. De ha valakinek már több gyakorlata van a Livewire vagy az Inertia (Vue.js vagy React alapon) JavaScript alapú keretrendszerekkel, akkor a Blade sablon motor használata helyett alkalmazhatja ezeket is. A nézet oldalak kinézetéért a Tailwind CSS keretrendszer lesz a felelős alapértelmezetten, így annak a használatával tudjuk hatékonyan átalakítani a kinézetét.

10.2.1.1. Telepítés

Figyelmeztetés! A Breeze a *„scaffolding”* (kezdőcsomag fájljainak automatikus generálása) során nem csak új fájlokat hoz létre, de megváltoztat már meglévő fájlokat is, amely folyamat így adatvesztéshez vezethet. Úgyhogy érdemes a felhasználói hitelesítési eszközt már a munkáink elején, projekt létrehozásakor kiválasztani, mivel ekkor még nem okozhat ez gondot. De ha például egy már régebb óta fejlesztett projektünkbe telepítjük, akkor a **routes / web.php**-ban lévő útvonalaink eltűnhetnek, és a létrehozott weboldal sablonunk is felülíródhat, eltűnhet, működésképtelenné válhat. Ezt a problémás helyzetet persze megoldhatja, ha a verziókezelő rendszert megfelelően használtuk, mert ekkor nem veszhetnek el nyomtalanul például az útvonalaink a **web.php** fájlból, vagy a sablonhoz tartozó fájljaink.

Hozzunk létre egy új projektet! Az új alkalmazás neve: **l10-auth-breeze** (az alkalmazás GitHub repo-ja itt érhető el).

Végezzük el a kezdeti adatkapcsolati beállítást (hozzunk létre új adatbázist **l10_auth_breeze_db** névvel), majd migráljuk az új projekt adattábláit!

Utána következhet a Breeze csomag letöltése, majd telepítése:

```
composer require laravel/breeze --dev
```

```
php artisan breeze:install
```

Ez utóbbi parancs fogja publikálni a hitelesítési csomag nézeteit, útvonalait, Controller-jeit és más erőforrásokat az alkalmazásunkban. Válasszuk a felajánlott lehetőségek közül a **blade**-et (ami a *„Blade with Alpine”* opciót jelenti)! Sötét módot a kinézetben is hozzáadhatunk, illetve a hitelesítést tesztelő csomagot is kiválaszthatjuk (PHPUnit vagy Pest). A korábbi ismereteinket figyelembe véve a PHPUnit használatát javaslom még ezen a ponton.

Előfordulhat egy hiba a telepítés során (ha nem kaptunk hibát, akkor ezek a sorok figyelmen kívül hagyhatók.): *„[SyntaxError] Unexpected token 'export' C:\xampp\htdocs\l\10-auth-breeze\postcss.config.js:1”.* A megoldás az, hogy a projekt gyökerében lévő **package.json** fájlhoz hozzáadjuk a fő szinten a következő kulcs-érték párost:

```
"type": "module"
```

10–15. kódrészlet: A package.json fájl bővítése

Ezután futtassuk újra a php artisan breeze:install parancsot!

A telepítés végén az alkalmazásunkat a Vite felépíti, hogy készen álljon éles környezetben optimálisan futni. Így kb. 60 új fájl jött létre vagy módosult a meglévők közül. Néhány változást áttekinthetünk az érdekesség kedvéért:

- Új útvonalak kerültek regisztrálásra a **routes** mappa fájljaiban:
  - **web.php:** bekerültek olyan útvonalak, amelyek a sikeres hitelesítés után lesznek elérhetőek, továbbá az **auth.php** útvonal fájl importálása.
    - A **dashboard** útvonalnál megjelent két Middleware is, amelyek közül az **auth** arra vonatkozik, hogy csak bejelentkezett felhasználók érhetik majd el ezt az útvonalat. Amiért ezt lehet tudni, az egy picit szövevényesebb[14], de bárki utána tud járni a háttérben, mert a Laravel forráskódja nyílt.
  - **auth.php:** látogatóknak (guest) és regisztrált felhasználóknak szóló további útvonalak kerültek itt regisztrálásra.
  - Az útvonalakat a php artisan route:list paranccsal le tudjuk kérni és meg tudjuk tekinteni az új, felhasználói hitelesítéssel kapcsolatos útvonalainkat.
- Számos új Controller jött létre, amelyek az **app / Http / Controllers / Auth** könyvtárba kerültek be.
- A **Request** és **Middleware** osztályok is a hitelesítési folyamat lefolytatását támogatják.
- A **resources / views** mappában pedig olyan szerkezeti (layout), nézet (view) és komponens (component) fájlok jöttek létre, amelyek a hitelesítési folyamatban megjelenő oldalak szerkezetét tartalmazzák. Ha a kinézetben bármit meg szeretnénk változtatni, akkor azt itt megtehetjük majd.
- Mindezeket, illetve a helyes működésüket előre definiált automatikus tesztekkel is tudjuk ellenőrizni. A tesztek a **tests / Feature** mappába és **Auth** almappájába kerültek be.

10.2.1.2. Telepítés utáni tesztelés

Ha nem szeretnénk elveszíteni az adatainkat (ismételt figyelmeztetés), akkor hajtsuk végre a következő módosítást a tesztelés előtt: nyissuk meg a projekt mappánk gyökerében lévő **phpunit.xml** fájlt és a következő két sort vegyük ki a megjegyzésből:

```
<env name="DB_CONNECTION" value="sqlite"/>
<env name="DB_DATABASE" value=":memory:"/>
```

10–16. kódrészlet: Tesztelési adatbázis beállítása adatvesztés elkerülésére

[14] Az **app / Http / Kernel.php** fájlban van regisztrálva az **auth** Middleware a **$middlewareAliases** tömbben. Az ehhez tartozó Middleware osztály az **app / Http / Middleware / Authenticate.php** fájlban van. Ebben a fájlban viszont csak egy átirányító metódus látszódik a login útvonalra... de mikor irányítson át? Ehhez meg kell nézni az ősosztályában lévő **handle()** metódust és itt próbál meg hitelesíteni a rendszer. A **vendor / laravel / framework / src / Illuminate / Auth / Middleware / Authenticate.php**-ben megtalálható **authenticate()** metódus segítségével próbálja meg hitelesíteni a felhasználót, és ha nem sikerül neki, akkor úgy tekint rá, hogy nincs bejelentkezve, tehát át is fogja irányítani a login útvonalra, hogy hitelesítse magát.

Az új teszteseteinket is lefuttathatjuk (a két ExampleTest osztályt törölhetjük előtte) a php artisan test paranccsal és megkapjuk a helyes lefutások eredményeit.

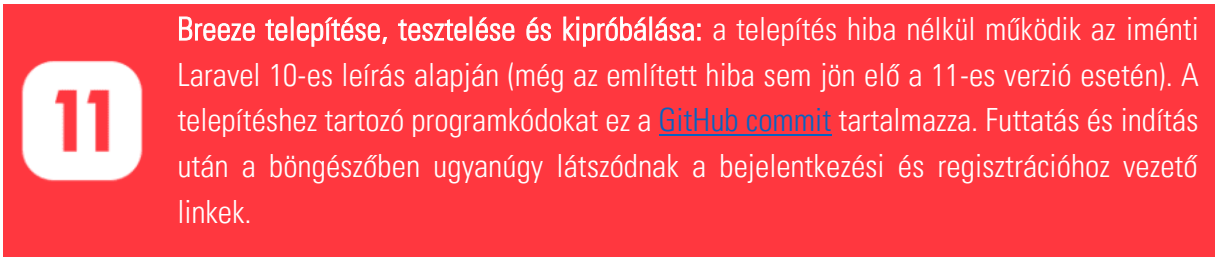
A Laravel 10-ben történő Breeze telepítés programkód változásait ez a GitHub commit tartalmazza.

```
PASS  Tests\Feature\Auth\AuthenticationTest
✓ login screen can be rendered                                          0.30s
✓ users can authenticate using the login screen                         0.13s
✓ users can not authenticate with invalid password                      0.24s
✓ users can logout                                                      0.03s

PASS  Tests\Feature\Auth\EmailVerificationTest
✓ email verification screen can be rendered                             0.03s
✓ email can be verified                                                 0.03s
✓ email is not verified with invalid hash                               0.03s

PASS  Tests\Feature\Auth\PasswordConfirmationTest
✓ confirm password screen can be rendered                               0.04s
✓ password can be confirmed                                             0.08s
✓ password is not confirmed with invalid password                       0.24s

PASS  Tests\Feature\Auth\PasswordResetTest
✓ reset password link screen can be rendered                            0.03s
✓ reset password link can be requested                                  0.05s
✓ reset password screen can be rendered                                 0.03s
✓ password can be reset with valid token                                0.04s

PASS  Tests\Feature\Auth\PasswordUpdateTest
✓ password can be updated                                               0.08s
✓ correct password must be provided to update password                  0.08s

PASS  Tests\Feature\Auth\RegistrationTest
✓ registration screen can be rendered                                   0.03s
✓ new users can register                                                0.03s

PASS  Tests\Feature\ProfileTest
✓ profile page is displayed                                             0.05s
✓ profile information can be updated                                    0.03s
✓ email verification status is unchanged when the email address is unchanged  0.03s
✓ user can delete their account                                         0.08s
✓ correct password must be provided to delete account                   0.09s

Tests:    23 passed (59 assertions)
Duration: 2.01s
```

10–9. ábra: Felhasználói hitelesítés: Breeze teszteseteinek helyes lefutása

10.2.1.3. Kipróbálás

Ha most elindítjuk az alkalmazásunk kiszolgálását (php artisan serve) akár egy másik terminal segítségével, és megnézzük az alkalmazásunkat a böngészőben, akkor azt vehetjük észre, hogy a főoldalunkon jobb felül megjelentek a *„Log in”* és a *„Register”* feliratok (linkek).



10–2. újdonság: Breeze telepítése, tesztelése és kipróbálása

Hajtsunk végre egy regisztrációt, és ha megfelelünk a kliens- és szerveroldali validációnak egyaránt, akkor létrejön a felhasználónk a **users** adattáblában. Továbbá rögtön be is jelentkeztet a rendszer minket, így egy **/dashboard** oldalra jutunk, ahol jelzi számunkra a rendszer, hogy sikeresen bejelentkeztünk. Jobb felül a regisztrációkor megadott nevünk fog megjelenni, amely egy lenyíló menü is egyben. A menü a profil (*„Profile"*) oldalra vezető és egy kijelentkezési (*„Log Out"*) linket tartalmaz.

Telepítés és használatba vétel után szükségünk lehet arra a felhasználóra, aki éppen be van jelentkezve az alkalmazásunkba. Ezt az **Auth** Facade-dal tudjuk elérni. Például köszöntsük a bejelentkezett felhasználót a dashboard nézet oldalon (a *„You're logged in!"* köszöntés után befűzhetjük ezt az új szöveget):

```
{{ Auth::user()->name }}
```

10–17. kódrészlet: Bejelentkezett felhasználó nevének kiíratása az Auth Facade segítségével

De ugyanerre az **auth()** segédmetódus is használható.

```
{{ auth()->user()->email }}
```

10–18. kódrészlet: Bejelentkezett felhasználó e-mail címének kiíratása az auth() segédmetódussal

Ha ezeket a kiíratásokat szépen egymás után fűzzük, akkor a következő szöveg látszódhat a **dashboard** (bejelentkezés utáni) oldalunkon. Természetesen a saját regisztrált nevünkkel és e-mail címünkkel.

You're logged in! Gludovátz Attila attila@gludovatz.hu

10–10. ábra: Bejelentkezett felhasználó adatainak kiíratása a nézet fájlban

Így tudunk tehát felhasználó specifikus adatmezőket használni azokban a nézetekben, amelyeknél elvárás, hogy csak bejelentkezés után legyenek elérhetőek.

Előfordulhat viszont, hogy vannak olyan nézet fájljaink, amelyeknél nincs az imént említett elvárás: tehát egyszerű látogatók és bejelentkezett felhasználók is meg tudják tekinteni a nézetet, de esetleg a bejelentkezett felhasználóknak plusz információkat is meg szeretnénk jeleníteni az oldalon. Ilyenkor az nem megoldás, amit az imént használtunk, mivel amikor az **Auth** osztályon vagy **auth()** metóduson keresztül szeretnénk lekérni a felhasználó adatait, akkor a sima látogatók egy **Exception** hibát kapnának, mivel nekik **null** értékkel térnének vissza ezek a lekérések, és a **null**-nak nem lehetne lekérni a **name** és **email** mezőit. Ennek megoldásához egy feltételvizsgálatot kell elvégeznünk a nézetben ahhoz, hogy tudjuk, be van-e jelentkezve az oldalunkra a látogató (tehát bejelentkezett felhasználónk böngészi-e a nézetet). A Blade eszközkészletének direktívái lesznek itt a segítségünkre. Nyissuk meg a **welcome** nézetünket és szerkesszük a **<head>**-ben lévő **<title>** oldalcímet, ami egyelőre csak simán *„Laravel"*.

```
<title>
  @if (Auth::check())
    Hi, {{ auth()->user()->name }}!
  @else
    Laravel
  @endif
</title>
```

10–19. kódrészlet: Bejelentkezés ellenőrzése után a logika kettéválasztása és eszerint a megjelenítés

A **<title>** tag-en belüli részt tehát kibővítettük egy if-else elágazással, ami annyit csinál, hogy ellenőrzi, hogy a látogató bejelentkezett felhasználó-e: ha igen, akkor az oldal címében (itt látható: a böngésző lapfülén vagy a tálcán lévő böngészőnél) köszöntjük a felhasználót (nem a **dashboard**, hanem a **welcome** nézetben vagyunk!), ha pedig nincs bejelentkezve a látogató, akkor csak simán Laravel az oldal címe.

Az **@if** és **@endif** páros ebben az esetben, mivel esetleg sokszor kell ellenőriznünk a nézetben, hogy bejelentkezett felhasználóval van-e dolgunk, ezért lecserélhető erre az egyszerűsített formára: **@auth** és **@endauth** (az **@else** marad ugyanúgy):

```
<title>
  @auth
    Hi, {{ auth()->user()->name }}!
  @else
    Laravel
  @endauth
</title>
```

10–20. kódrészlet: Felhasználói bejelentkezés ellenőrzése az @auth Blade direktívával

Egyszerűbb és letisztultabb megoldás ez így. De meg is tudjuk fordítani ezt a logikát, hogy ne azt ellenőrizzük, bejelentkezett-e a felhasználó, hanem azt, hogy látogató-e (vendég-e). Ezt így tudjuk megtenni az újabb Blade direktívákkal:

```
<title>
  @guest
    Laravel
  @else
    Hi, {{ auth()->user()->name }}!
  @endguest
</title>
```

10–21. kódrészlet: Látogató szerepkör ellenőrzése a @guest Blade direktívával

Ebben az esetben, ha látogató az, aki böngészi az alkalmazásunkat, akkor simán Laravel a weboldal címe, ha pedig nem-egyszerű látogató, tehát be van jelentkezve a felhasználó, akkor köszöntjük a címben.

További példát láthatunk a **welcome** nézetben a **<body>** tartalmi rész elején, amikor a bejelentkezett felhasználónak a **dashboard** linket mutatja a rendszer, míg a látogatóknak a **login** és **register** linkeket.

```
@auth
  <a href="{{ url('/dashboard') }}" class="font-semibold
@else
  <a href="{{ route('login') }}" class="font-semibold te
  @if (Route::has('register'))
    <a href="{{ route('register') }}" class="ml-4 font-se
  @endif
@endauth
```

10–11. ábra: Linkek mutatása a főoldalon a bejelentkezés állapotának függvényében (a többi kódrészletet itt most figyelmen kívül hagyhatjuk, ezért látszódik azoknak csak töredék része)

10.2.1.4. Testreszabás

Az egy optimális helyzet, ha a Laravel által nyújtott szolgáltatás pontosan azt adja nekünk a felhasználói hitelesítésnél, amire nekünk szükségünk van. Ebben az alfejezetben viszont azokat az eshetőségeket nézzük át, amikor valamit másképpen szeretnénk, mint ahogy alapértelmezetten megkapjuk a keretrendszertől.

10.2.1.4.1. Regisztráció letiltása

A letiltás igénye amiatt lehet, mert például egy olyan webalkalmazást készítünk, amikor a hozzáféréssel rendelkező felhasználók köre szűk, véges. Például egy kis cég adminisztrációs tevékenységeit támogató webes alkalmazásnál nincs szükség nyilvános regisztrációs felületre, mert az adminisztrátor vagy az arra jogosult személy létrehozza a felhasználókat és ő menedzseli őket, szükség esetén újat vesz fel, meglévőt módosít vagy töröl, de nem az egyszerű felhasználók végzik ezeket a tevékenységeket.

Tiltsuk le a regisztráció lehetőségét: nyissuk meg a **routes / auth.php** felhasználói hitelesítéssel kapcsolatos útvonalakat tartalmazó fájlt, és kommenteljük ki ezt a regisztrált útvonalat:

```
Route::get('register', [RegisteredUserController::class, 'create'])
  ->name('register');
```

10–22. kódrészlet: Regisztrációs útvonal a routes / auth.php fájlban

Ha ezután megpróbálnánk megnyitni a böngészőben a regisztrációs útvonalunkat (**/register**), akkor 404-es hibaoldalt kapnánk. Sőt, vegyük észre, hogy ha megnézzük az alkalmazásunk főoldalát, akkor arról is eltűnt *automatikusan* a jobb felső sarokból a regisztráció lehetősége.

10.2.1.4.2. E-mail-es felhasználói megerősítés letiltása

A regisztráció során a Breeze lehetővé teszi nekünk, hogy e-mail-es megerősítést küldjünk a leendő regisztrált felhasználóinknak, és így várjuk el tőlük a megerősítést a regisztrációs szándékukról. Ezt a funkcionalitást is hasonlóan tudjuk letiltani, mint ahogy már a regisztrációnál tettük az imént.

```
Route::get('verify-email', EmailVerificationPromptController::class)
  ->name('verification.notice');
Route::get('verify-email/{id}/{hash}', VerifyEmailController::class)
  ->middleware(['signed', 'throttle:6,1'])
  ->name('verification.verify');
```

10–23. kódrészlet: E-mail-es felhasználói megerősítés útvonalai a routes / auth.php fájlban

Az itteni útvonalakban van egy érdekes, eddig nem használt Middleware-ünk, ami a **throttle** névre hallgat, és utána van két szám is. Ennek a segítségével tudjuk meghatározni (jelen esetben a bejelentkezett felhasználónak), hogy **x** darab alkalommal próbálkozhat ezzel a tevékenységgel **y** percen belül. Jelen esetben a második útvonal Middleware-je szerint **6**-szor próbálkozhat az e-mail-es megerősítéssel **1** percen belül.

Megjegyzés: ha még nem állítottunk volna be e-mail küldési szolgáltatást, akkor ez a funkcionalitás amúgy sem működne még.

10.2.1.4.3. Jelszó emlékeztető küldésének letiltása

Hasonlóan, mint az előzőekben, ezt az útvonalat kell kikommentelni:

```
Route::get('forgot-password', [PasswordResetLinkController::class,
'create'])
  ->name('password.request');

Route::post('forgot-password', [PasswordResetLinkController::class,
'store'])
  ->name('password.email');
```

10–24. kódrészlet: Jelszó emlékeztető szolgáltatás letiltása az útvonalak regisztrációjának megszüntetésével

Megjegyzés: ezeknél az útvonalaknál (a regisztrációnál, a megerősítésnél és a jelszó emlékeztető tiltásánál is) elég a *„get"-es* útvonalakat letiltani, mivel a *„post"-osok* védve vannak, nem elérhetőek, ha nem rendelkezik a kérés a megfelelő hitelesítéssel (token-es védelemmel vannak ellátva), így azokat nem muszáj letiltanunk ahhoz, hogy az imént említett szolgáltatásokat letiltsuk.

10.2.1.4.4. Regisztráció utáni automatikus bejelentkeztetés letiltása

Ahhoz, hogy ezt tesztelni tudjuk, persze érdemes előtte újra engedélyezni a regisztráció lehetőségét, különben nem fogunk tudni regisztrálni. Ennek a funkciónak az eléréséhez az **app / http / Controllers / Auth / RegisteredUserController.php** fájlban lévő **store()** metódus magját kell módosítanunk. Vegyük ki belőle a következő sort (vagy kommenteljük ki, hogy a későbbiekben megmaradjon, ha vissza szeretnénk illeszteni):

```
Auth::login($user);
```

10–25. kódrészlet: Regisztráció utáni automatikus bejelentkeztetés letiltása

Ennek hatására nem fog minket bejelentkeztetni az alkalmazás automatikusan, hanem regisztráció után a bejelentkezési űrlapot fogjuk megkapni a böngészőben.

10.2.1.4.5. Regisztráció utáni átirányítás

Ha már az automatikus bejelentkeztetés tiltása után úgyis ott vagyunk a **RegisteredUserController store()** metódusánál, akkor ugyanitt a metódus végén van egy átirányítás:

```
return redirect(RouteServiceProvider::HOME);
```

10–26. kódrészlet: Regisztráció és bejelentkeztetés utáni automatikus átirányítás

Ez az **App / Providers / RouteServiceProvider** osztályban lévő HOME konstans szerint a **dashboard** útvonalra irányít át minket, amit mi szabadon megváltoztathatnánk. Ez a konstans változó az **app / Providers / RouteServiceProvider.php** fájlban található meg, így ha máshova szeretnénk a felhasználókat irányítani sikeres bejelentkezés után, akkor itt kell módosítani ezt az értéket, például ha ezt átírjuk a "/"-ra, akkor a regisztráció és a bejelentkeztetés végrehajtódna, és utána bejönne a főoldalunk.

Kiegészítés és magyarázat: amikor letiltottuk a regisztráció utáni automatikus bejelentkeztetést, attól a **RegisteredUserController** osztály **store()** metódusának végén még ugyanúgy erre a **/dashboard** útvonalra irányított minket a rendszer, de mivel az a **routes / web.php**-ban látható módon ez az útvonal **auth** és **verified** Middleware által védett, tehát csak bejelentkezett felhasználók érhetik el, emiatt jutottunk ott el a bejelentkezési útvonalra és űrlapra.

10.2.1.4.6. Bejelentkezés utáni alapértelmezett útvonal megváltoztatása

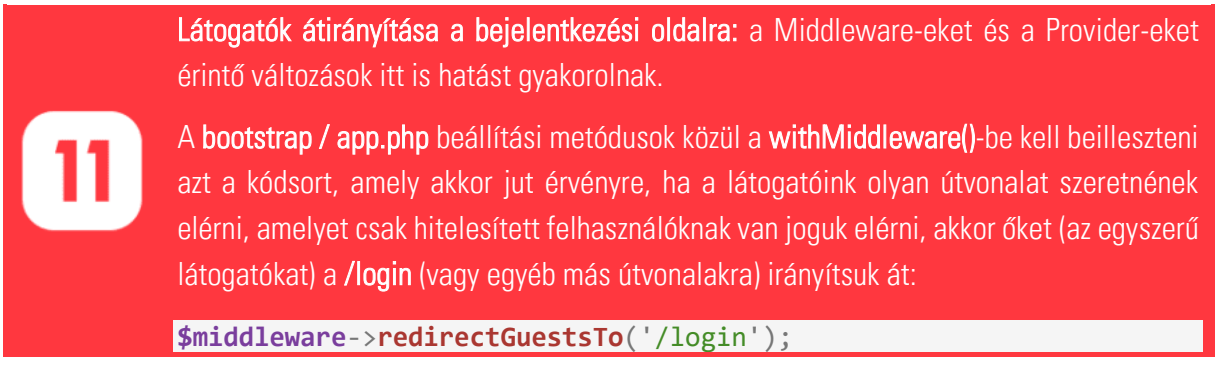
Bejelentkezés után alapértelmezetten az **app / Providers / RouteServiceProvider.php** fájlban található **HOME** konstansban megadott címre lesz átirányítva a felhasználó. Ezt az értéket felül tudjuk írni a **/home**-ról valami másra, amire szeretnénk.

Ha a bejelentkezés utáni átirányítást szeretnénk felülírni egy másik módon, akkor az **app / Http / Controllers / Auth / AuthenticatedSessionController** osztály **store()** metódusát kell módosítanunk és az ottani **return** utasítást kell módosítanunk, például így, ha a profil szerkesztési oldalára szeretnénk irányítani a felhasználót a sikeres bejelentkezés után:

```
return redirect(route('profile.edit'));
```

10–27. kódrészlet: Bejelentkezés utáni átirányítás

Megjegyzés: ha itt maradt volna a **redirect()->intended()** utasítás, akkor ugyanoda irányítottuk volna vissza a felhasználót, ahonnan érkezett a bejelentkezési felületre. Ez egy jobb felhasználói élményt (User eXperience, UX) tud biztosítani bizonyos esetekben.



10–3. újdonság: Breeze: látogatók átirányítása

10.2.1.4.7. Regisztrációs és bejelentkezési folyamat bővítése, módosítása

Módosítsuk a regisztrációs, és ezzel együtt a bejelentkezési folyamatot úgy, hogy hozzáadunk egy felhasználónevet (**username**) a rendszerhez (név és e-mail már van benne). Ehhez először kell egy username nevű mező a users táblába, hozzuk létre az ehhez szükséges migrációs fájlt:

```
php artisan make:migration add_username_to_users_table
```

Ha a **name** mező után szeretnénk beszúrni, az csak MySQL-ben működne, SQLite-ban sajnos az utolsó helyre szúrja be mindig az új mezőt.

Az **up()** metódus **Schema** utasításába írjuk ezt:

```
$table->string('username')->after('name')->unique();
```

10–28. kódrészlet: Új mező (username) a users táblában

A **down()** metódus **Schema** utasításába írjuk be ezt:

```
$table->dropColumn('username');
```

10–29. kódrészlet: Új username mező törlése a users táblából

Ha most szeretnénk futtatni a `php artisan migrate` parancsot, akkor hibát kapnánk, mivel már egy felhasználóm van a **users** adattáblában, emiatt egy kötelezően egyedi értékeket tartalmazó oszlop nem adható hozzá alapértelmezett érték nélkül. Így inkább egy friss adatbázisszerkezetet hozzunk létre, üres **users** táblával:

```
php artisan migrate:fresh
```

Így már működni fog, létrejön a táblában a **username** mező, de üres lett a **users** tábla.

Következhet a User Eloquent Model fájl bővítése. A **$fillable** mező tömbjébe adjuk hozzá a **username**-et, mint új kitölthető mezőt.

```
protected $fillable = [
  'name',
  'email',
  'password',
  'username',
];
```

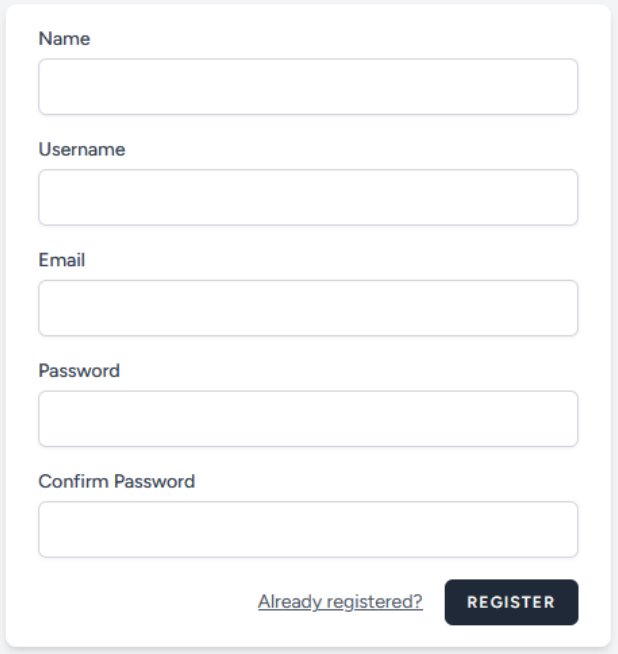
10–30. kódrészlet: User Model osztály $fillable tömbjének bővítése a username mezővel

A regisztrációs nézet fájl űrlapját is át kell alakítani. Nyissuk meg a **resources / views / auth / register.blade.php**-t. Itt a regisztráció során látható űrlap kódját láthatjuk. Talán a legegyszerűbb, ha lemásoljuk a *„Name”* szekciót, és kicsit módosítottan utána beszúrjuk ugyanúgy:

```
<!-- Username -->
<div class="mt-4">
  <x-input-label for="username" :value="__('Username')" />
  <x-text-input id="username" class="block mt-1 w-full" type="text"
name="username" :value="old('username')" required autofocus
autocomplete="username" />
  <x-input-error :messages="$errors->get('username')" class="mt-2" />
</div>
```

10–31. kódrészlet: A username label és input mező a regisztrációs űrlapon

Így az egyes elemek megfelelőek lesznek. A regisztrációs űrlapon pedig meg is jelenik az új mező:



10–12. ábra: Felhasználónévvel kibővített regisztrációs űrlap

Érvényesített felhasználó eltárolása a regisztrációkor: következhet az **app / Http / Controllers / Auth / RegisteredUserController** osztály **store()** metódusának átalakítása. A metódus elején az érkező kérés (**$request**) validálása történik meg, itt és az utána következő felhasználó létrehozásánál szintén másoljuk le a **name**-hez kötődő sorokat és írjuk át **username**-re, ahogy azt a regisztrációs űrlapnál már megtettük.

Így a felhasználónév kötelező mező lesz, szöveges tartalmú, maximum 255 karakter hosszú és plusz elvárás vele szemben (a **name**-hez képest), hogy egyedi érték legyen a táblában a felhasználónév.

```
$request->validate([
  'name' => ['required', 'string', 'max:255'],
  'username' => ['required', 'string', 'max:255', 'unique:' . User::class],
  'email' => ['required', 'string', 'lowercase', 'email', 'max:255',
'unique:' . User::class],
  'password' => ['required', 'confirmed', Rules\Password::defaults()],
]);

$user = User::create([
  'name' => $request->name,
  'username' => $request->username,
  'email' => $request->email,
  'password' => Hash::make($request->password),
]);
```

10–32. kódrészlet: Felhasználói adatok érvényesítése és a létrehozás (username mezővel bővítve)

Így most már a regisztráció folyamatát tesztelhetjük, ha a többi mezőt kitöltjük és a **Username**-et üresen hagyjuk, akkor a kliens oldali validáció jelezni fog, hogy kötelezően kitöltendő a mező. Ha egy kicsit *„trükköznénk”,* és belenyúlnánk a böngészőben az űrlap kódjába: kivennénk a **required** attribútumot a

username mezőnél és úgy küldenénk el a szervernek, akkor a szerver oldali validáció bukna el és jelezné, hogy ez egy kötelezően kitöltendő mező. Minden mező helyes kitöltése után a regisztráció megtörténik, a felhasználó létrejön a **users** adattáblában, és már be is tudunk jelentkezni a bejelentkezési űrlapon.

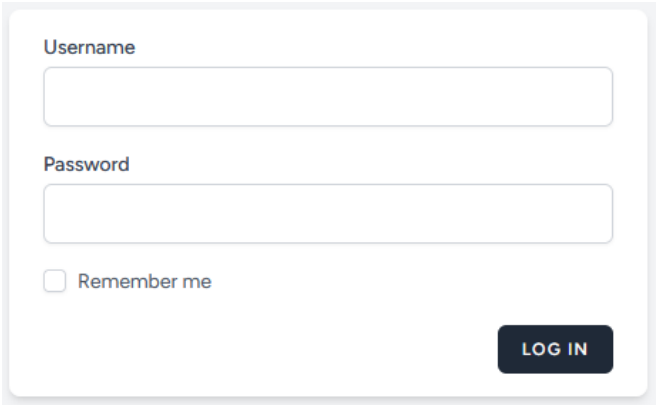
Valósítsuk meg a bejelentkezést úgy, hogy ne e-mail címet kérjen, hanem az új **username** mező alapján történjen meg az azonosítás!

A **resources / views / auth / login.blade.php**-ból vegyük ki az e-mail-es szekciót és illesszük be a **username**-es részt, amit már egyszer bemásoltunk a regisztrációs űrlapba is.

```
<!-- Username -->
<div>
  <x-input-label for="username" :value="__('Username')" />
  <x-text-input id="username" class="block mt-1 w-full" type="text"
name="username" :value="old('username')" required autofocus
autocomplete="username" />
  <x-input-error :messages="$errors->get('username')" class="mt-2" />
</div>
```

10–33. kódrészlet: Felhasználónév mező az e-mail helyett a bejelentkezési űrlapon

Itt látható az eredménye:



10–13. ábra: Új bejelentkezési űrlap (e-mail helyett felhasználónév alapú azonosítással)

A bejelentkezési logikát is módosítanunk kell még a helyes működéshez. Adódik a felvetés, hogy ennek a megoldását az **AuthenticatedSessionController**-ben kell keresnünk és nem is tévedünk ezzel akkorát. Az itteni **store()** metódus magjában az első lényegi sor, ami ez:

$request->authenticate();

Ez végzi el a kérés hitelesítését. Viszont jó lenne, ha egy picit többet látnánk az **authenticate()** metódus magjából, úgyhogy keressük meg a forrását: ez az **App / Http / Requests / Auth / LoginRequest.php** fájlban található meg.

Ennek a fájlnak a logikáját több helyen is módosítanunk kell! Kezdjük a **rules()** metódussal: töröljük ki belőle az **email**-re vonatkozó sort, és vegyük fel a **username**-re vonatkozó szabályokat:

```
public function rules(): array
{
```

```
  return [
    'username' => ['required', 'string', 'max:255'],
    'password' => ['required', 'string'],
  ];
}
```

10–34. kódrészlet: Bejelentkezési űrlapon vizsgált szabály módosítása e-mail-ről a felhasználónévre

Következik az **authenticate()** metódus, amelyben az *„email”* szövegeket *„username”*-re cseréljük, ennél többet, most nem kell tudnunk erről:

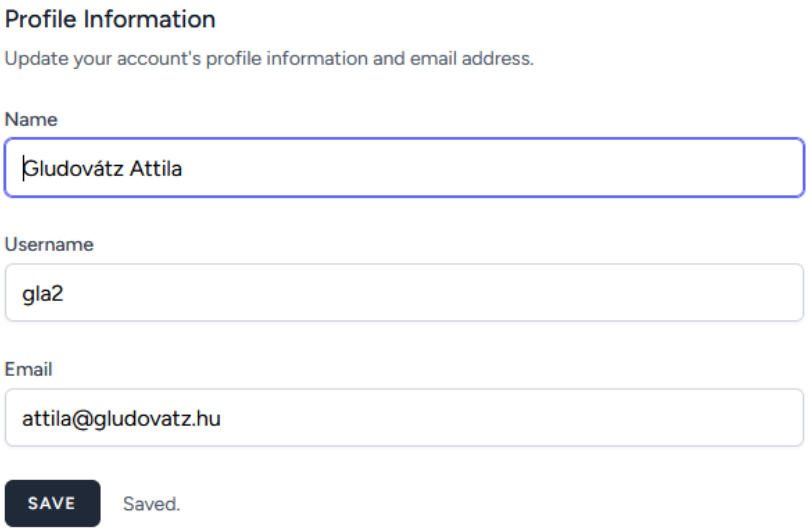
```
public function authenticate(): void
{
  $this->ensureIsNotRateLimited();

  if (!Auth::attempt($this->only('username', 'password'), $this-
>boolean('remember'))) {
    RateLimiter::hit($this->throttleKey());

    throw ValidationException::withMessages([
      'username' => trans('auth.failed'),
    ]);
  }

  RateLimiter::clear($this->throttleKey());
}
```

10–35. kódrészlet: A hitelesítési metódus módosítása e-mail-ről felhasználónévre cseréljük az azonosítást

A további két metódusunkban (**ensureIsNotRateLimited()** és **throttleKey()**), amelyek a bejelentkezési próbálkozások limitálásáért felelősek, is csak cseréljük ki az *„email”*-t *„username”*-re, és készen is leszünk.

Ha most teszteljük, akkor már működik is a bejelentkezés az imént regisztrált felhasználónk felhasználónév és jelszó párosával.

Egy kicsit zavaró lehet, de jelenleg a bejelentkezett felhasználó profil oldalán a felhasználónév módosítására még nincs lehetőség, de gyakorlásként érdemes lehet megcsinálni, hogy az is működjön az alkalmazásunkban. Ennek megvalósítása itt nem kerül részletezésre (mindössze két fájlt kell módosítani hozzá: a nézet fájl: **resources / views / profile / partials / update-profile-information-form.blade.php** és a frissítés elemeit ellenőrző request osztály **rules()** metódus szabályát: **app / Http / Requests / ProfileUpdateRequest.php**), de az alfejezet programkód módosításait tartalmazó GitHub commit-be bekerülnek a változások. A cél az lesz, hogy a felhasználónév felülírása, frissítése működjön, például így:



10–14. ábra: Felhasználónév frissítésének működése: "Saved." felirat megjelenik a Save gomb megnyomása után

Az alfejezetben végrehajtott programkód módosítások ebben a GitHub commit-ben találhatók meg.

10.2.1.5. Testreszabás utáni tesztelés

Az alfejezet második részében az automatikus teszteseteket futtattuk és azt tapasztaltuk, hogy helyesen működnek, pozitív eredménnyel zárulnak. Futtassuk most őket újra!

```
php artisan test
```

Azt tapasztalhatjuk, hogy a legtöbb tesztesetünk *„eltört”,* elromlott. Mindössze a regisztrációs és bejelentkezési nézetek megjelenítése zárult pozitív eredménnyel. Állítsuk helyre az elromlott teszteseteinket és közben vizsgáljuk meg azokat a különlegességeket a tesztesetekben, amelyeket eddig nem alkalmaztunk! Teszteseteink a **Feature** mappán belül helyezkednek el és főleg azon belül is az **Auth** mappában.

10.2.1.5.1. Regisztráció tesztelése

Kezdjük az **Auth / RegistrationTest.php** fájllal. Az új felhasználó regisztrálására azt a hibát kaptuk a terminal-ban, hogy *„The username field is required.”,* ami így az elvárások szerint működik is, hiszen hozzáadtuk a regisztrációs űrlaphoz és a kérés validálásához is a **username** mezőt, amelyet viszont a tesztesetben eddig nem követtünk le. Adjuk hozzá a **post()** metódushívás második paraméterének tömbjéhez a következőt:

```
'username' => 'testuser',
```

10–36. kódrészlet: username mező hozzáadása a regisztrációt tesztelő metódus POST kéréséhez

Futtassuk a konkrét fájl teszteseteit:

```
php artisan test tests/Feature/Auth/RegistrationTest.php
```

Megváltozott hibaüzenetet kaptunk, mégpedig az hitelesítéssel kapcsolatban. Az alapműködés az volt, hogy a regisztráció után rögtön be is jelentkeztette a rendszer a felhasználót, és utána átirányította a **dashboard** oldalra. A teszteset is eszerint működik. Így most két lehetőségünk van, vagy megváltoztatjuk

a tesztesetet és kivesszük belőle ezeket a részeket, vagy újra engedélyezzük az **app / Http / Controllers / Auth / RegisteredUserController.php** fájlban az **Auth::login($user);** kódsort (*megjegyzés:* én ezt választom), amivel automatikusan bejelentkeztetjük a felhasználót a regisztrációja után. Bármelyiket választjuk a tesztfájl ismételt lefuttatása ezután már helyes eredménnyel kell, hogy záruljon.

10.2.1.5.2. Bejelentkezés tesztelése

Az **Auth / AuthenticationTest.php** fájl hibás tesztjeinél szintén a **username** mező hiánya okozza a problémát, de egy kicsit másképp, mint a regisztrációnál. Itt ugyanis a **UserFactory**-t használják a tesztesetek ahhoz, hogy új felhasználókat hozzanak létre, de ehhez kötelező lenne a **username** megadása, amelyet mi még a felhasználó gyárban nem definiáltunk. Nyissuk meg a **database / factories / UserFactory.php** fájlt és bővítsük ki a **definition()** metódus visszatérési tömbjét ezzel:

```
'username' => fake()->userName(),
```

10–37. kódrészlet: Felhasználó gyár bővítése a felhasználónévvel

Az **AuthenticationTest** osztály többi tesztelő metódusánál POST login útvonal elérésében cseréljük ki az e-mail-es bejelentkezéseket **username**-es bejelentkezésekre:

```
'username' => $user->username,
```

10–38. kódrészlet: Bejelentkezés szimulálásánál cseréljük le az e-mail-t felhasználónévre

Ezzel már egy kivételével minden tesztesetet helyreállítottunk. Az utolsó a bejelentkezés utáni átirányítás miatt nem jó, úgyhogy az **app / Http / Controllers / Auth / AuthenticatedSessionController store()** metódusában állítsuk vissza az átirányítás útvonalát **dashboard**-ra. Így már az összes teszteset lefutása helyes lesz ebben a fájlban.

Egy nagyon fontos segédmetódust láthatunk a kijelentkezés (**logout**) tesztelésénél, amelyet a saját teszteléseink során is bármikor használhatunk: **actingAs($user)**. Ezzel gyakorlatilag egy felhasználó szerepébe helyezkedve, mint bejelentkezett felhasználó tudjuk végrehajtani a további tesztelési eljárásokat, kéréseket.

10.2.1.5.3. További Auth mappán belüli tesztesetek javítása

Az **Auth** mappán belül hasonlóan működik a többi (e-mail megerősítés, jelszó megerősítés, helyreállítás, frissítés) tesztelésének javítása is, ezeknél már csak arra van szükség, hogy a **routes / auth.php** útvonal regisztrációs fájlban engedélyezzük az e-mail-es megerősítési, illetve a jelszó helyreállításához kapcsolódó korábban kikommentelt útvonalainkat.

10.2.1.5.4. Profil oldal és funkcionalitásainak tesztelése

Végül az **Auth** mappán kívül lévő **ProfileTest.php** teszteseteivel kell foglalkoznunk. A korábbiak kijavítása után itt már egyszerű dolgunk lesz. A PATCH **/profile** útvonal kérésekhez kell csak hozzáadnunk például ugyanazt a 10–36. kódrészletet, amit már egyszer megtettünk máshol. Így hibamentes tesztelésekhez jutunk, ha ismét az összes fájlt tesztelünk (`php artisan test`).

Az alfejezetben végrehajtott programkód módosítások ebben a GitHub commit-ben találhatók meg.

10.2.2. Jetstream hitelesítési csomag

Ez a Breeze-hez képest egy robosztusabb kezdőkészlet, amely a Livewire vagy az Inertia + Vue kombinációval segíti a felhasználói hitelesítés folyamatát. Az alapvető funkcionalitásokon túl biztosít még a számunkra két faktoros hitelesítést, csapatokat, webböngésző menedzsmentet, felhasználó törlést, API támogatást is (a Laravel Sanctum csommagal).

A Laravel Jetstream egy szép grafikus felületen keresztül biztosítja a számunkra a felhasználói hitelesítés lehetőségét. Ha tudjuk, hogy a következő alkalmazásunkban szükség lesz felhasználói hitelesítési folyamatokra, funkciókra, komplexebb megoldásokra, akkor mindenképpen érdemes már a projekt fejlesztésének indításakor telepíteni és beüzemelni a Laravel Jetstream-et. Ez biztosítani fogja a számunkra a felhasználói regisztrációt, be- és kijelentkezést, e-mail-es megerősítést, két-faktoros azonosítást, felhasználói munkamenet (session) kezelését, opcionálisan csapat menedzselését és még több minden egyéb funkcionalitást is.

A Tailwind CSS keretrendszer biztosítja a számunkra a szép felhasználói felületet, úgyhogy aki mélyebben szeretne megismerkedni a Jetstream-mel, az mindenképpen térképezze fel a részletes működését. Korábban ezt önmagában telepítettük a 4.4.1.1. fejezet alapján, de itt a Jetstream telepítése magába foglalja a Tailwind keretrendszer telepítését is.

10.2.2.1. Telepítés

A felhasználói hitelesítési kezdőkészletek nem igazán férnek meg egymás mellett, úgyhogy, ha döntünk valamelyik mellett, akkor használjuk azt a Laravel alkalmazásainkban. A döntés kihatással van tehát a projektek későbbi életciklusára is. Hozzunk létre egy új projektet, amely tartalmazza majd a Jetstream kezdőkészlet csomagot a felhasználói hitelesítéshez: **l10-auth-jetstream** néven (itt lehet nyomon követni a változásait: https://github.com/gludovatza/l10-auth-jetstream). Hozzuk létre a MySQL-ben az adatbázist is neki Laravel 10 esetén **l10_auth_jetstream_db** néven! *Megjegyzés:* ezzel párhuzamosan a Laravel 11-es projekt is létrehozásra került: **l11-auth-jetstream** néven (itt lehet nyomon követni a változásait: https://github.com/gla-elte/l11-auth-jetstream).

Ezekután két irányba indulhatunk el, amikor a kliens oldali keretrendszerek között dönthetünk (az irányok főleg a JavaScript keretrendszer használatában különböznek):

1. Livewire + Blade
2. Inertia.js + Vue.js

Megjegyzés: mindkét irányvonal megfelelő és csak tőlünk függ, hogy melyiket választjuk, mert mindkettővel szép és hatékonyan működő megoldásokat tudunk megvalósítani. Viszont mindkét tématerület egy-egy jó nagy háttéranyagot és tudást igényel. Segítségül mindkettőhöz megosztok egy-egy videós tanfolyamot, amelyekből lehet tanulni:

1. Livewire sorozat
2. Inertia sorozat
    - Kiegészítésként: Vue.js sorozat

A jó hír az, hogy ha nem is ismerjük ezeket a fenti technológiákat, csak szeretnénk egy jól működő, szép, komplett hitelesítési rendszert használni a Laravel webalkalmazásunkban, akkor a Jetstream tökéletesen alkalmas erre, hiszen már alapból rengeteg mindent ad nekünk, amit ebben a fejezetben át is nézünk.

Kiinduló weboldalunk, ahol bővebben is ismerkedhetünk a Jetstream-mel: https://jetstream.laravel.com/

Telepíthetjük az új projektünkbe a Jetream-et:

```
composer require laravel/jetstream
```

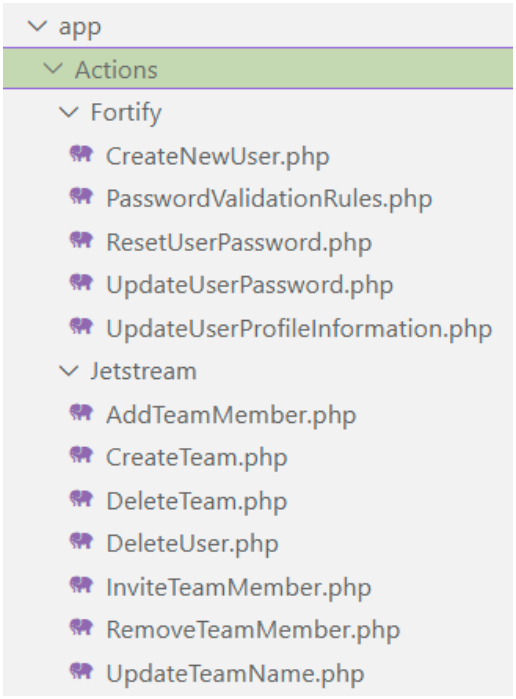
Laravel 10-ben a 4.3. verziószámú Jetstream települ, Laravel 11-ben az 5.0 verziójú Jetstream. A csomag projektben való elhelyezése után következhet a konkrét telepítés, amely *„livewire"* vagy *„inertia".* Opcionálisan a parancshoz hozzáfűzhetjük a `--teams` kapcsolót, így csapatokat is tudunk menedzselni a Jetstream segítségével. Mivel itt most az a cél, hogy a Laravel Breeze-hez képest a plusz dolgokra, új funkcionalitásokra koncentráljunk, ezért alkalmazzuk a csapatok engedélyezését (*megjegyzés:* Laravel 10 esetén adjuk hozzá a **package.json** fájlhoz a **"type": "module"** kódsort a kezdő kapcsos zárójel után rögtön, hogy ne kapjunk hibát a telepítés során, Laravel 11-nél nincs probléma):

```
php artisan jetstream:install
```

Telepítésnél adjuk meg a *livewire*-t, *teams*-t és a PHPUnit-hoz tartozó azonosító *számot*.

A telepítés végére a rendszer a Vite segítségével felépíti (elkészíti) azokat a fájlokat, amelyeket aztán rögtön optimalizál is, így tudjuk őket használni a továbbiakban. Így nem is kell már lefuttatnunk az `npm run build` parancsot. Szükségünk ellenben az adatbázis migrálásra: `php artisan migrate`

A Jetstream telepítése részben magába foglalta a Fortify hitelesítési kezdőkészlet bizonyos funkcionalitásait is, de a Fortify csomaggal a Jetstream után fogunk foglalkozni bővebben. Annyit azonban már itt érdemes megjegyezni, hogy a Fortify a felhasználói hitelesítéshez nyújt szerver oldali funkcionalitásokat, szolgáltatásokat és nem rendelkezik kliens oldali megoldásokkal (nézetekkel, komponensekkel stb.). Ez részben abban nyilvánul meg, hogy nincsenek legenerált (scaffolded) Controller osztályok, helyettük *„Action"* osztályokkal szabható majd testre a működés. Ezek az **app / Actions** mappában találhatók meg, szétválasztva Fortify-os és Jetstream-es akciókra.



10–15. ábra: Fortify és Jetstream akció osztályok

A fájlok és így a bennük található osztályok neve meglehetősen beszédes, így ezek magyarázatába most részletesen nem megyünk bele. Viszont ezek az osztályok tipikusan egy olyan funkcionalitást hajtanak végre, amelyeket a nevük is jelez: új felhasználó létrehozása, jelszó szabályok érvényesítése, új csapattag hozzáadása stb.

Ezen kívül azt is észrevehetjük, hogy új útvonalak (a **dashboard**-on kívül) regisztrálása sem történt meg a **routes / web.php** fájlban és további hitelesítési útvonalak sem jöttek létre itt a **routes** mappában. Viszont, ha lekérjük az útvonalak listáját, akkor máris azt fogjuk tapasztalni, hogy mégis létrejöttek ezek az útvonalak (10–16. ábra).

Ezek az útvonalak, ahogy a Controller-ek nevéből is látszódik és mivel új Controller-ek nem kerültek be az **app / Http / Controllers** mappába, ezért tudható, hogy a telepített csomagokban (Fortify, Jetstream) találhatók meg. A Fortify-os útvonalak itt találhatók meg: **vendor / laravel / fortify / routes / routes.php** a Jetstream-es útvonalak pedig itt: **vendor / laravel / jetstream / routes / livewire.php**

```
PUT       current-team ..................................... current-team.update › Laravel\Jetstream › CurrentTeamController@update
GET|HEAD  dashboard ..................................................................................................... dashboard
GET|HEAD  forgot-password ................................ password.request › Laravel\Fortify › PasswordResetLinkController@create
POST      forgot-password ................................... password.email › Laravel\Fortify › PasswordResetLinkController@store
GET|HEAD  livewire/livewire.js ....................................... Livewire\Mechanisms › FrontendAssets@returnJavaScriptAsFile
GET|HEAD  livewire/livewire.min.js.map ................................... Livewire\Mechanisms › FrontendAssets@maps
GET|HEAD  livewire/preview-file/{filename} ............... livewire.preview-file › Livewire\Features › FilePreviewController@handle
POST      livewire/update ..................................... livewire.update › Livewire\Mechanisms › HandleRequests@handleUpdate
POST      livewire/upload-file ............................ livewire.upload-file › Livewire\Features › FileUploadController@handle
GET|HEAD  login ..................................................... login › Laravel\Fortify › AuthenticatedSessionController@create
POST      login ........................................................... Laravel\Fortify › AuthenticatedSessionController@store
POST      logout ............................................ logout › Laravel\Fortify › AuthenticatedSessionController@destroy
GET|HEAD  register ................................................ register › Laravel\Fortify › RegisteredUserController@create
POST      register ................................................... Laravel\Fortify › RegisteredUserController@store
POST      reset-password ..................................... password.update › Laravel\Fortify › NewPasswordController@store
GET|HEAD  reset-password/{token} ................................ password.reset › Laravel\Fortify › NewPasswordController@create
GET|HEAD  sanctum/csrf-cookie ................................ sanctum.csrf-cookie › Laravel\Sanctum › CsrfCookieController@show
GET|HEAD  team-invitations/{invitation} ............. team-invitations.accept › Laravel\Jetstream › TeamInvitationController@accept
GET|HEAD  teams/create ................................................ teams.create › Laravel\Jetstream › TeamController@create
GET|HEAD  teams/{team} ..................................................... teams.show › Laravel\Jetstream › TeamController@show
GET|HEAD  two-factor-challenge ................ two-factor.login › Laravel\Fortify › TwoFactorAuthenticatedSessionController@create
POST      two-factor-challenge ................................. Laravel\Fortify › TwoFactorAuthenticatedSessionController@store
GET|HEAD  user/confirm-password ........................................... Laravel\Fortify › ConfirmablePasswordController@show
POST      user/confirm-password ......................... password.confirm › Laravel\Fortify › ConfirmablePasswordController@store
GET|HEAD  user/confirmed-password-status ......... password.confirmation › Laravel\Fortify › ConfirmedPasswordStatusController@show
POST      user/confirmed-two-factor-authentication two-factor.confirm › Laravel\Fortify › ConfirmedTwoFactorAuthenticationControll…
PUT       user/password ..................................... user-password.update › Laravel\Fortify › PasswordController@update
GET|HEAD  user/profile ............................................ profile.show › Laravel\Jetstream › UserProfileController@show
PUT       user/profile-information ........ user-profile-information.update › Laravel\Fortify › ProfileInformationController@update
POST      user/two-factor-authentication ............ two-factor.enable › Laravel\Fortify › TwoFactorAuthenticationController@store
DELETE    user/two-factor-authentication ......... two-factor.disable › Laravel\Fortify › TwoFactorAuthenticationController@destroy
GET|HEAD  user/two-factor-qr-code ........................... two-factor.qr-code › Laravel\Fortify › TwoFactorQrCodeController@show
GET|HEAD  user/two-factor-recovery-codes ............... two-factor.recovery-codes › Laravel\Fortify › RecoveryCodeController@index
POST      user/two-factor-recovery-codes ....................................... Laravel\Fortify › RecoveryCodeController@store
GET|HEAD  user/two-factor-secret-key .................. two-factor.secret-key › Laravel\Fortify › TwoFactorSecretKeyController@show
```

10–16. ábra: Fortify és Jetstream útvonalak

Ebből is adódik, hogy a testre szabás – a Breeze-zel ellentétben –, nem útvonalak engedélyezésével vagy letiltásával módosítható, hanem beállítási fájlok segítségével fogunk tudni funkcionalitásokat engedélyezni vagy letiltani a webalkalmazásban. Ezek a beállítási fájlok a **config** mappába kerültek be: **fortify.php** és **jetstream.php** fájlok, amelyekkel a kipróbálási és testre szabási alfejezetekben részletesen is foglalkozunk.

A felhasználói munkameneteket a **config / sessions.php**-ban tudjuk beállítani, illetve a konkrét aktív munkamenetek eltárolására az adatbázisban a **sessions** adattáblát használja a rendszer.

A generált nézetek szerkezetbe (layout) rendeződnek és főleg komponens alapúak, illetve azt is észrevehetjük, hogy olyan funkcionalitásoknak is létrejöttek a nézetei, amelyek alapértelmezetten még nincsenek engedélyezve, például az API használatához tartozó nézetek.

Telepítettük a Laravel Jetstream-et, következhet a tesztelése, kipróbálása, testre szabása hangsúlyozottan azoknak a részeknek, amelyek a Breeze-nél nem voltak elérhetők.

A telepítéshez tartozó programkód változások a Laravel 10 esetén ebben a GitHub commit-ben, Laravel 11 esetén pedig ebben a GitHub commit-ben találhatók meg.

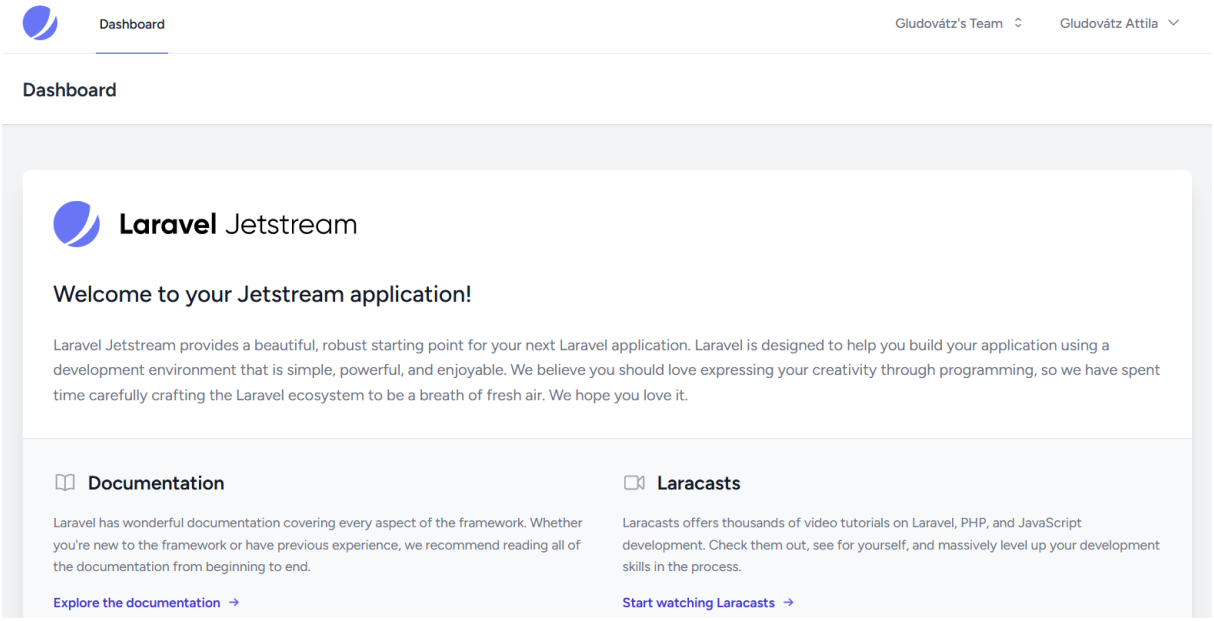
10.2.2.2. Telepítés utáni tesztelés

A tesztesetek lefuttatásához ugyanazt tegyük, mint a 10–16. kódrészletben. Utána, ha futtatjuk a teszteket, akkor a többségük zölden (PASS) lefut, néhány pedig narancssárgán (WARN) jelez. A figyelmeztetések többsége abból adódik (a problémás teszteset lefutása mellett olvasható magyarázatokból kiindulva), hogy bizonyos Jetstream szolgáltatások nincsenek alapértelmezetten engedélyezve, aktiválva, illetve, hogy az e-mail-es megerősítések nincsenek engedélyezve.

10.2.2.3. Kipróbálás

A kiszolgálás megkezdése után azt tapasztalhatjuk, hogy ugyanúgy van *„Log in"* és *„Register"* linkek a kezdőoldalon jobb felül.

Regisztráljunk is egy felhasználót! A validáció, vagyis a mezők helyességének ellenőrzése itt alapértelmezetten működik, tehát például, ha a jelszónak 8 karakternél rövidebbet szeretnénk beállítani, akkor jelezni fog a *„Register"* gomb megnyomása után, hogy túl rövid a jelszavunk. Ha mindent helyesen adtunk meg, akkor megtörtént a regisztráció és automatikusan be is jelentkeztet minket a rendszer. A regisztráció után a Jetstream vezérlőpultja (**dashboard**) oldala jön be.



10–17. ábra: Jetstream vezérlőpultja bejelentkezés után

A felhasználói menü (amikor a regisztrált felhasználónk nevére kattintunk) ugyanaz szintén, mint a Breeze esetén. Azonban megjelent egy csapathoz tartozó lenyíló menü is. Ebben az alapértelmezetten létrejövő saját csapat beállításait tudjuk megtekinteni, vagy akár új csapatot is létrehozhatunk.

A kipróbálás folyamatát a profilhoz és a csapathoz tartozó funkcionalitások bemutatásával és testre szabásával folytathatjuk.

10.2.2.4. Profilhoz tartozó funkcionalitások lehetőségei

A nevünkön kattintva lenyílik a helyi menü és kiválaszthatjuk a profil beállításait (Profile). Lehetőségünk van itt a következőkre:

1. alapadatok (név, e-mail cím) megváltoztatására,
2. jelszó megváltoztatására (de kell hozzá mindenképpen a régi jelszó is, így biztonságosabb),
3. kétfaktoros azonosításra, vagyis hitelesítésre (TFA[15] - bővebben erről, illetve a beállításairól itt is olvashatunk),

[15] Two-Factor Authentication

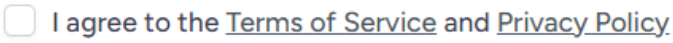
4. kezelhetjük a munkameneteinket, ha esetleg több helyen (böngészőben, eszközön) vagyunk bejelentkezve, akkor itt ki tudjuk jelentkeztetni a másik munkameneteinket,
5. törölhetjük is a felhasználónkat.

Próbáljunk ki néhány egyszerűbb, személyes profilhoz tartozó funkcionalitást!

Engedélyezzük, vagyis vegyük ki a kommentelést a **config / jetstream.php 'features'** tömbjének kikommentelt elemei elől. Így a *„Felhasználási és adatvédelmi feltételek elfogadása”*, *„Profilkép hozzáadása és kezelése”* és az *„API szolgáltatások”* funkcionalitások is elérhetővé válnak.

10.2.2.4.1. Felhasználási és adatvédelmi feltételek funkcionalitás

A felhasználási és adatvédelmi feltételek funkció engedélyezése a regisztrációs űrlapon jelenik meg egy új elemként.

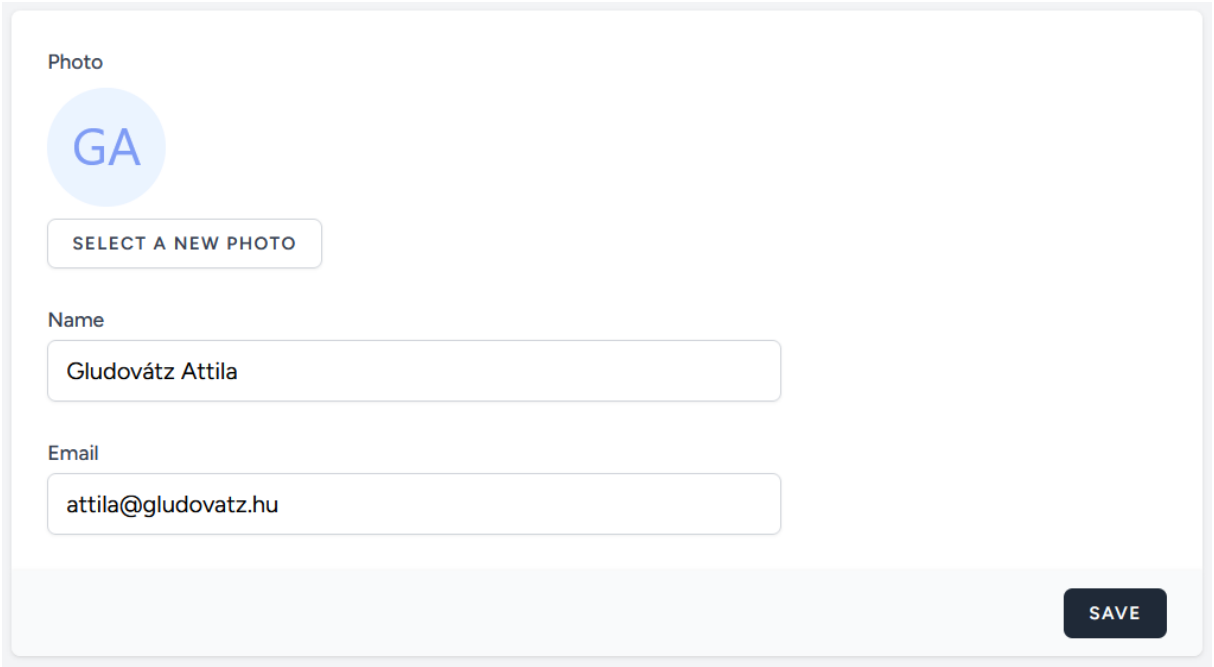


10–18. ábra: Felhasználási és adatvédelmi feltételek elfogadása checkbox és label a regisztrációs űrlapon

A szolgáltatási feltételek és az adatvédelmi irányelvek linkek egy-egy nézethez vezetnek, amelyeket a **resources / views** mappában találunk és szabhatunk testre igény szerint: **terms.blade.php** és **policy.blade.php** nézet fájlokban.

10.2.2.4.2. Profilkép hozzáadása, szerkesztése, törlése funkcionalitások

A profilkép beállítását a *„Profile”* felhasználói menüben, a *„Profile Information”* szekcióban tudjuk megtenni. Alapértelmezetten generált egy képet nekünk a rendszer: a regisztrációkor megadott névből létrehozott egy monogramot.



10–19. ábra: Profil információ szekció bővült a profilkép résszel

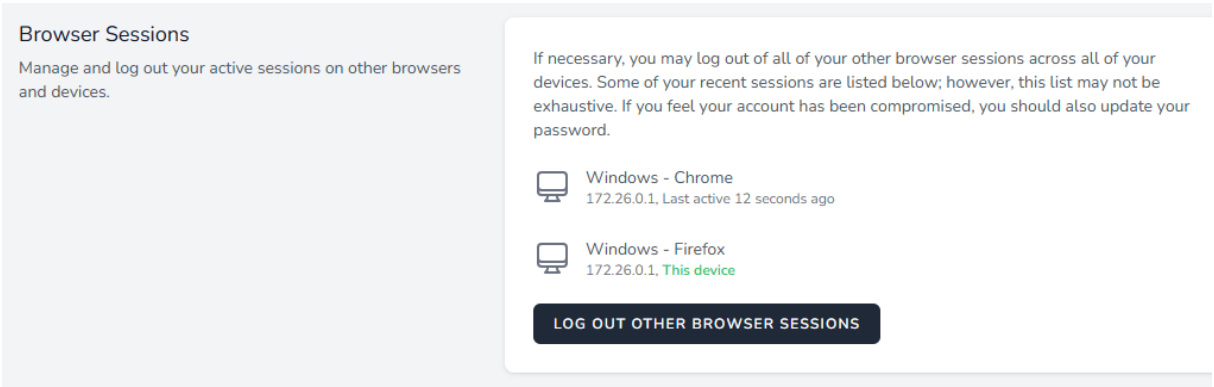
Ezt a képet felülírhatjuk és feltölthetünk egy saját profilképet a *„Select a new photo”* gomb megnyomása utáni tallózással. Alapértelmezetten az új kép a projektünk **storage / app / public / profile-photos** mappájába kerül be egy véletlenszerűen generált karaktersorozatú névvel (azért, hogy ne fordulhasson elő ütközés a fájlneveknél). Ha a feltöltés után nem jelenik meg az új képünk, csak az **<img>** tag **alt** attribútumába elhelyezett szöveg (a regisztrált felhasználónk neve). Ez a probléma leggyakrabban akkor szokott felmerülni, ha az **.env** fájlban lévő **APP_URL** beállítás értéke helyileg http://localhost míg a futtatást a http://127.0.0.1:8000 címen nézzük a böngészőben. Változtassuk meg az **APP_URL**-t a megfelelő címre, majd frissítsük az oldalt a böngészőben, és már jó is lesz a profilképünk elérhetősége. Ez a képfeltöltési és -megjelenítési probléma a webes alkalmazásunk kihelyezésekor (deployment), élesre állításakor is elő szokott fordulni, ekkor ugyanezt a két értéket kell megvizsgálni, hogy az **APP_URL**-ben megadott értéken található-e az alkalmazásunk publikus mappája.

Ha mégsem tetszik a beállított profilkép, akkor el is távolíthatjuk azt.

10.2.2.4.3. Munkamenetek kezelése

Előfordulhat, hogy valahol bejelentkeztünk a webalkalmazásunkba, és ottfelejtettük megnyitva a böngészőt. Ez nyilván problémákat okozhat, de segít nekünk a profil oldal *„Browser Sessions”* szekciója, amely megmutatja, hogy milyen rendszeren, milyen böngészővel és milyen IP címről vagyunk bejelentkezve még az alkalmazásba az aktuálison kívül. Próbáljuk is ezt ki!

Ha alapértelmezetten Firefox böngészőt használunk, majd utána a Chrome-ban is bejelentkezünk ugyanazzal a felhasználóval, akkor a Firefox-ban, ha frissítjük a profil oldalunkat, akkor láthatjuk a két különböző munkamenetet.



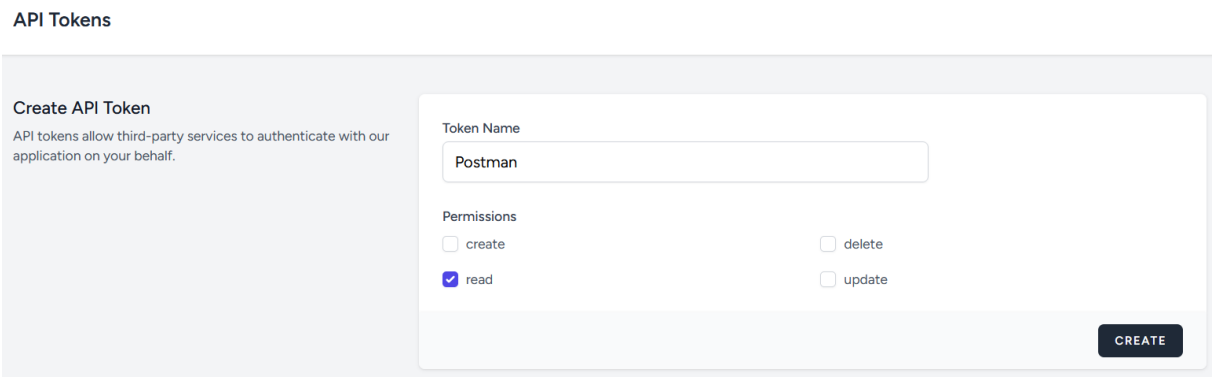
10–20. ábra: Felhasználói munkamenetek listázása és megszüntetésének lehetősége a profil oldalon

Ha azt szeretnénk, hogy ne lehessen a másik böngészőben (vagy helyen) problémás helyzeteket előidézni, akkor kattintsunk a *„Log out other browser sessions”* gombra, ami után egy jelszavas megerősítést kér tőlünk a rendszer, és utána már ki is jelentkeztette a *„Chrome-os felhasználót”*: ha ráfrissítünk a Chrome-ban a **dashboard** oldalra, akkor már újra a bejelentkező űrlap fog látszódni ott. Ezzel a funkcióval tehát a feledékenységünket tudjuk kijavítani, ami sokszor hasznos tud lenni.

10.2.2.4.4. Jetstream API beállítása és működése (a Sanctum csomag segítségével)

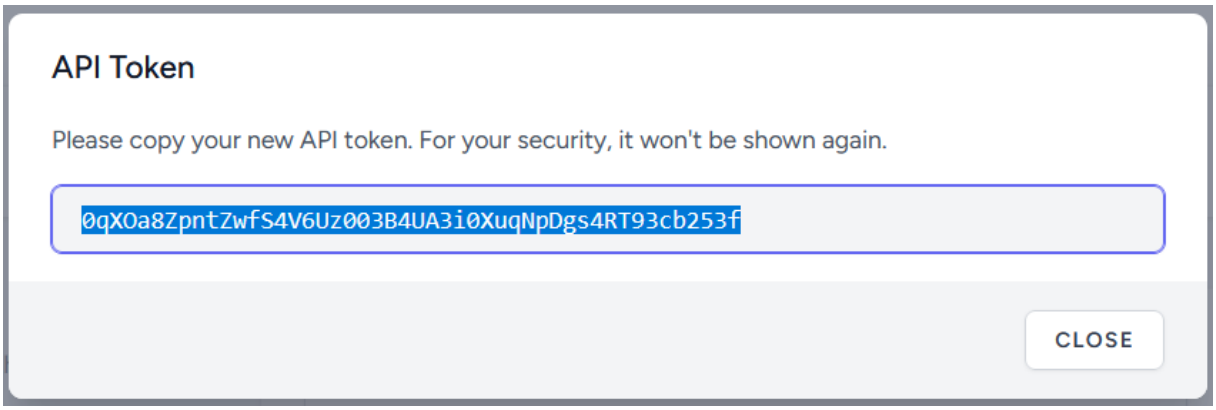
A Jetstream nem csak a felhasználókat tudja hitelesíteni, hanem azokat a programokat is, amelyek szolgáltatásokat vennének igénybe a webalkalmazásunktól. A **config / jetstream.php**-ben engedélyezett

Features::api() utasításnak köszönhetően a felhasználói menü egy új menüponttal bővült: *„API Tokens”*. Itt tudunk nevet adni annak a külső alkalmazásnak, aminek valamilyen engedélyt akarunk adni ahhoz, hogy mit és mire (létrehoz, kiolvas, szerkeszt és frissít esetleg töröl) használhat az alkalmazásunkban vagy például ezáltal az adatbázisban. Természetesen ez a nézet is testre szabható a **resources / views / api / api-token-manager.blade.php** fájl szerkesztésével, de most koncentráljunk inkább az elérhető funkcionalitásra.



10–21. ábra: Postman alkalmazás API engedélyének létrehozása a kiolvasáshoz

Például a Postman alkalmazásnak adjunk olvasási engedélyt, majd megkapjuk a hozzá tartozó token-t.



10–22. ábra: A Postman alkalmazáshoz kapott API token

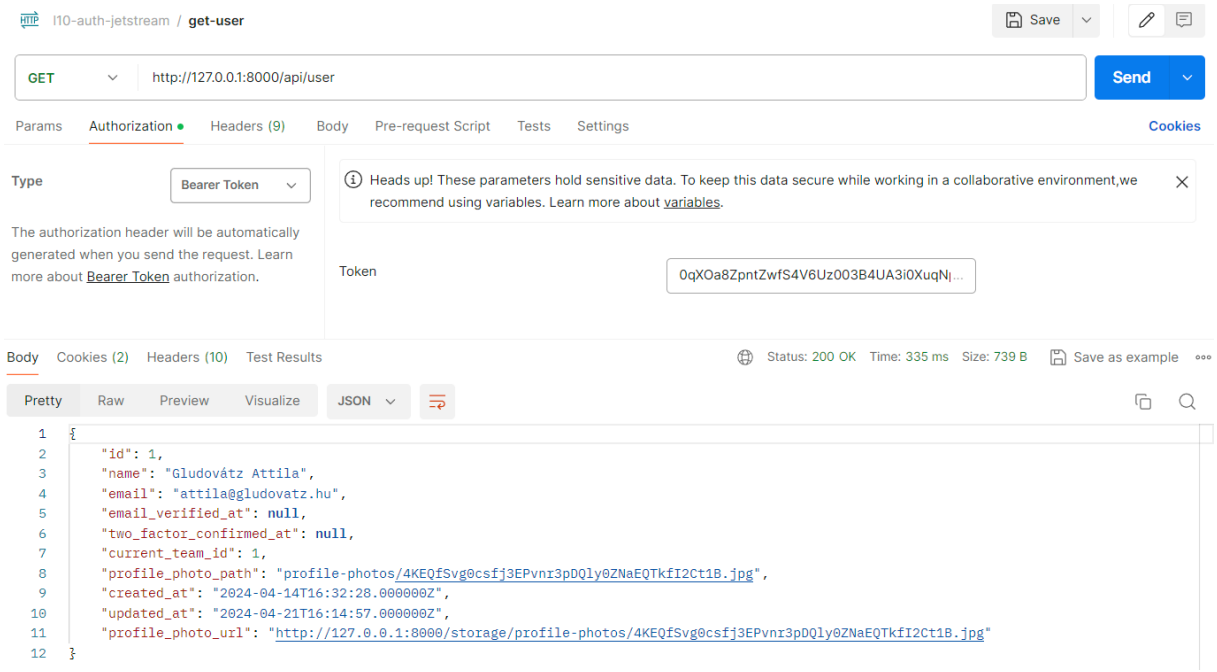
Létrehozás után is menedzselhető a létrehozott hozzáférés és az engedélyei: a *„Permissions”* linkre kattintva egy modal ablakban ugrik fel a jogok szerkesztése. Ez a token be is kerül a **personal_access_tokens** adattáblába az adatbázisban.

Megjegyzés: bár a felhasználói hitelesítés (authentication) és az engedélyeinek kezelése (authorization) szorosan összefüggő témakörök, itt most még csak a hitelesítés szempontjából fontos, hogy létrehoztuk az API hozzáférési token-t és a további engedélyekkel (létrehozás, frissítés, törlés) nem foglalkozunk itt, csak a következő, 10. fejezetben.

Térjünk át a Postman alkalmazásra és hozzunk létre egy új gyűjteményt, amelybe egy új kérést tudunk definiálni. A kérés neve lehet ez (de a neve nem releváns): **get-user**, a kérés legyen GET-es és ezt az alapértelmezetten létező API-os útvonalat kérjük le: http://127.0.0.1:8000/api/user, de még előtte a *„Headers”* lapfül elemeiben vegyük ki az *„Accept”* beállítás elől a pipát, és hozzuk újra létre az *„Accept”*-et **application/json** értékkel! Ha ezt nem tennénk meg, akkor 200-as OK választ kapnánk eredményül, de

az nem a tényleges felhasználót adná vissza, hanem a bejelentkezési űrlap HTML kódját. Ha így most lekérjük az útvonalat, akkor a válasz üzenet: 401-es állapotkódú *„Unauthenticated."* lesz, ami annyit tesz, hogy nem vagyunk hitelesítve.

A megoldás az, ha a *„Headers"* mellett kiválasztjuk az *„Authorization"* lapfület és a *„Type"* lenyíló listában a *„Bearer Token"*[16] értéket. Ekkor megjelenik jobbra mellette a *„Token"* szövegdoboz, ahova be kell másolnunk a fenti véletlen karaktersorozatot, token-t, majd utána újra le kell kérnünk az URL-t. Eredményül meg kell kapnunk a felhasználónkat:

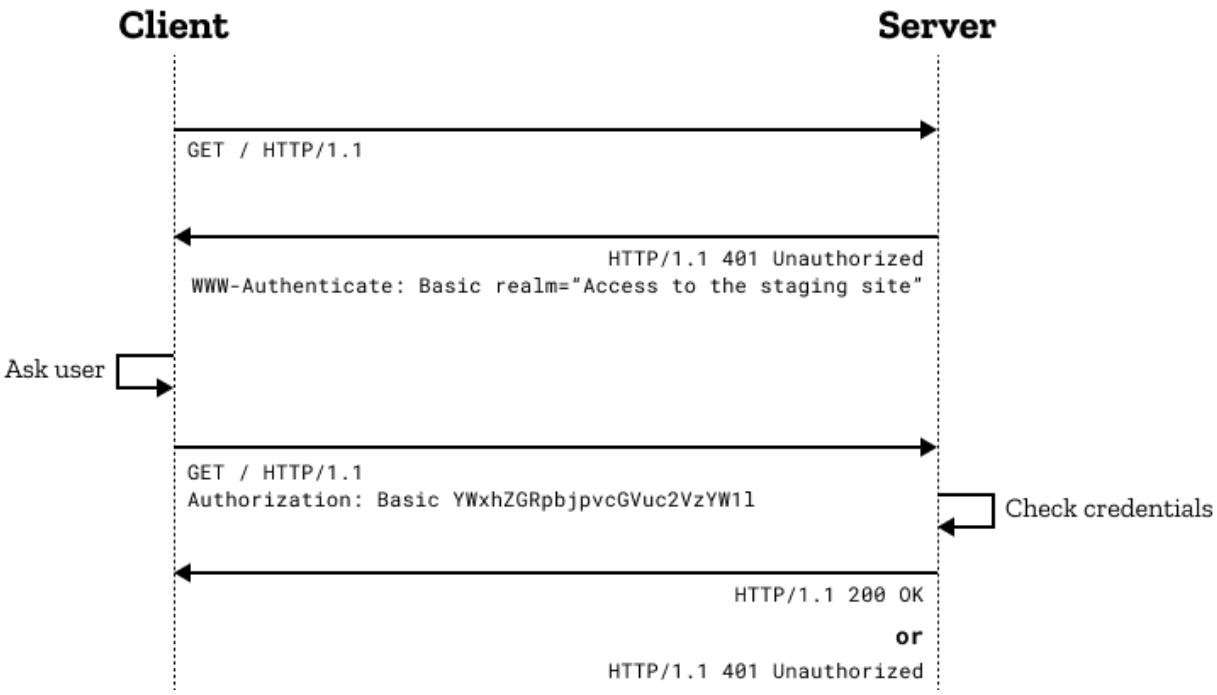


10–23. ábra: Postman alkalmazásban hitelesített API kérés futtatása megfelelő válasszal

Megjegyzés: ekkor automatikusan bekerül a kérés *„Headers"* részébe egy új, *„Authorization"* kulcs elem a *„Bearer"* szóval és a hozzáfűzött token értékkel.

A *„Type"* részben több más opció (séma) is választható, amelyek mind a HTTP protokoll hitelesítési eljárását teszik lehetővé. Erről további információ itt olvasható: https://developer.mozilla.org/en-US/docs/Web/HTTP/Authentication de talán egy ábra ilyenkor sokatmondóbb, mint a teljes leírás:

[16] Ezek a token-ek lehetővé teszik a felhasználói kérések hitelesítését egy hozzáférési kulcs használatával. Ezt a token-t API kulcsnak is hívják. Egy ilyen kulcs lehet például egy JSON webes token (JWT). A token, ahogy azt láthattuk is, egy szöveges karakterlánc, amely a kérés fejlécében szerepel.



10–24. ábra: Kliens-szerver közötti kommunikációhoz tartozó HTTP hitelesítés eshetőségei (Forrás: [7])

Az ábra magyarázata: üzenetváltások történnek a kliens és a szerver között. Az üzenetváltások fentről lefelé tekintve időben előrefelé történnek. Kezdetben a kliens indít egy GET-es HTTP kérést, de nem mellékeli hozzá a hitelesítési információkat (esetünkben a token-t). Emiatt a szervertől egy 401 státuszkódú választ kap, amely nem ad hozzáférést a kért erőforráshoz. A kliens a felhasználó megkérdezése után már úgy indítja el a GET-es HTTP kérést, hogy mellékeli hozzá az *„Authorization"* információt, vagyis a token-t. A szerver ellenőrzi a token megfelelőségét és érvényességét. Ha mindent rendben talál, akkor a sikeres hitelesítés után visszaküldi a kliensnek a kért erőforrást (200-as OK kóddal), ha valami gond volt a hitelesítéssel, akkor ismét 401-es hibakódot küld vissza a kliensnek.

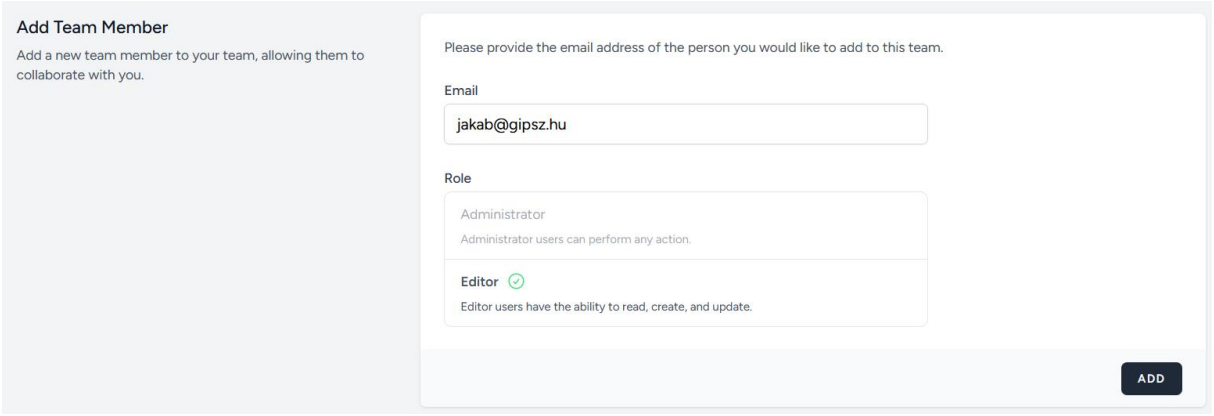
Ezzel gyakorlatilag megvalósítottuk a (Jetstream csomaggal együtt feltelepített) Sanctum által az API token alapú hozzáférést az alkalmazásunkhoz. Ennek a használata egyszerűbb, mint a cookie alapú hitelesítés, viszont veszélyesebb is, mivel a token maga ellopható, illetve kevésbé véd a CSRF és XSS támadások ellen. A cookie alapú hitelesítést a 10.2.3.2.3. alfejezetben tekintjük át.

10.2.2.5. Csapathoz tartozó funkcionalitások lehetőségei

A csapat beállításainál képesek vagyunk átírni a csapatunk nevét és tudunk új felhasználókat (e-mail cím alapján) meghívni a csapatba. Az új tag meghívásánál be tudjuk még állítani, hogy milyen szerepkörrel rendelkezzen az új csapattag: alapértelmezetten adminisztrátor (Administrator) és szerkesztő (Editor) szerepkörök vannak. Az adminisztrátorok mindent képesek csinálni, amit most eddig mi is csináltunk, a szerkesztőnek egy kicsit kevesebb joga van, ők főleg a weboldal tartalmait tudják majd szerkeszteni. Ez a két szerepkör kiindulásnak elég, de a jövőben majd többet is hozzá tudunk adni, ha szeretnénk.

Ha most szeretnénk meghívni valakit a csapatba, akkor a rendszer kéri az e-mail címet és az új tag szerepkörét, és miután helyesen megadtuk ezeket, küldene egy e-mail-t a leendő tagnak, hogy

regisztráljon be. Viszont a mi alkalmazásunkban még nincsen beállítva az e-mail küldő SMTP[17] szolgáltatás. Ezért a meghívás (invite) még nem fog tudni működni. Ezért egy kicsit trükközünk, és az **.env** fájlban a **MAIL_MAILER** attribútum értékét állítsuk át **smtp**-ről **log**-ra. Ekkor a kiküldendő e-mail-eket a **storage / logs / laravel.log** fájlban tudjuk böngészni.



10–25. ábra: Új csapattag hozzáadása meghívással

Az *„Add"* gomb megnyomása után bekerült az említett naplózási fájlba a HTML alapú levél teljes tartalma, kiegészítve azokkal az információkkal, amelyek az új felhasználó regisztrációjához, illetve így közvetve a csapathoz való csatlakozáshoz kellenek. Alább látható egy részlet a **laravel.log** fájlból:

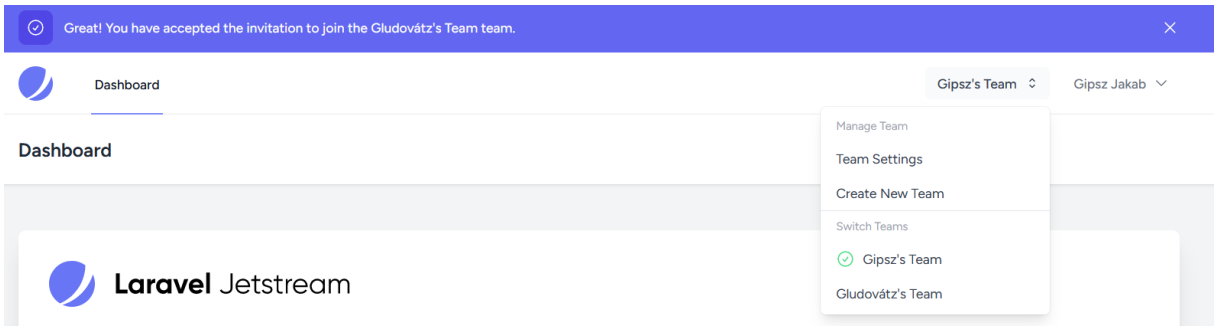
storage > logs > laravel.log

```
1  [2024-04-14 20:56:22] local.DEBUG: From: Laravel <hello@example.com>
2  To: jakab@gipsz.hu
3  Subject: Team Invitation
4  MIME-Version: 1.0
5  Date: Sun, 14 Apr 2024 20:56:22 +0000
6  Message-ID: <741d29feb0ac07012cb740676e03dd3d@example.com>
7  Content-Type: multipart/alternative; boundary=IbcLPcNP
8
9  --IbcLPcNP
10 Content-Type: text/plain; charset=utf-8
11 Content-Transfer-Encoding: quoted-printable
12
13 Laravel: http://localhost
14
15 You have been invited to join the Gludovátz's Team team!
16
17 If you do not have an account, you may create one by clicking the button below. After creating an account, you may click the invitation
18
19 Create Account: http://127.0.0.1:8000/register
20
21 If you already have an account, you may accept this invitation by clicking the button below:
22
23 Accept Invitation: http://127.0.0.1:8000/team-invitations/1?signature=0427cd96e49645c7e5a2ae02a1be8c02fbb26ea4889bf30690ae816066cc611d
24
25 If you did not expect to receive an invitation to this team, you may discard this email.
26
27 © 2024 Laravel. All rights reserved.
```

10–26. ábra: Invitációs levél kódja, tartalma a napló fájlban

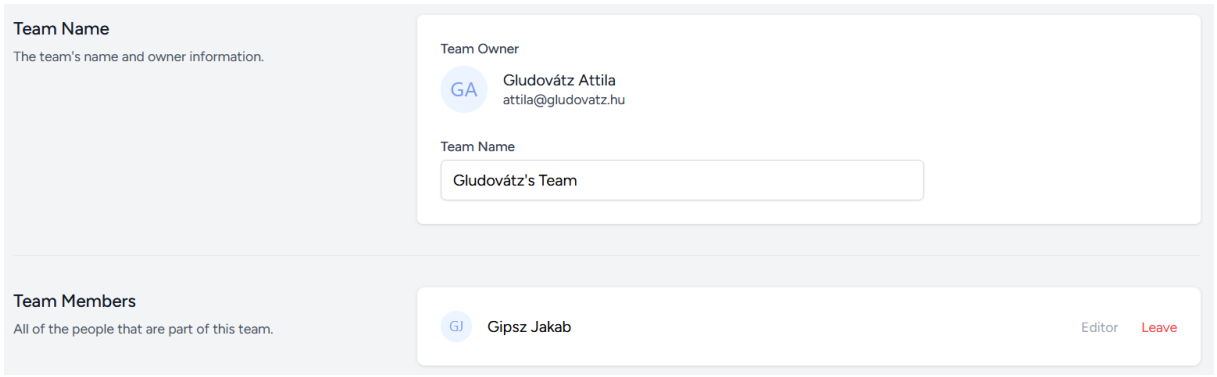
Nyissunk meg egy másik típusú böngészőt, vagy annak a böngészőnek egy inkognitó ablakát, amelyet már használtunk a Jetstream-be való bejelentkezésre. Hajtsuk végre a regisztrációt jakab@gipsz.hu e-mail címmel, majd amelyik böngésző ablakban regisztráltunk, oda másoljuk be az iménti ábrán látható (23. sor) *„Accept Invitation"* utáni linket és látogassuk meg az oldalt. Így azon túl, hogy lett Gipsz Jakabnak egy saját csapata, már elfogadta a meghívást Gludovátz Attila csapatába és azt a csapatos menüben látja is.

[17] Simple Mail Transfer Protocol



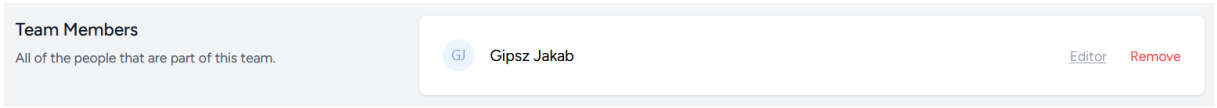
10–27. ábra: Új felhasználó hozzáadásra került egy másik csapathoz

A csapat átváltása után, a *„Team Settings"*-re kattintva láthatja is a csapat beállításait: módosítani tudja a csapat nevét, látja a saját szerepkörét, illetve el tudja hagyni a csapatot (*„Leave"*), ha szeretné.



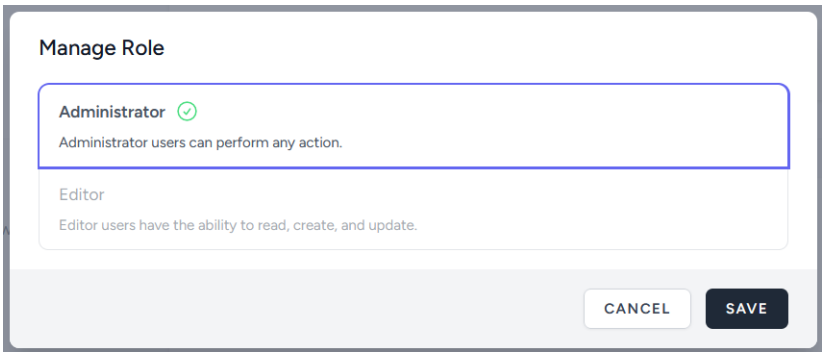
10–28. ábra: Csapat beállításai szerkesztőként, nem adminisztrátorként

Ugyanez a csapat beállítási felület az adminisztrátor szemszögéből így néz ki:



10–29. ábra: Elfogadott invitáció (új csapattag) a csapat beállításainál

Adminisztrátorként meg tudjuk változtatni a hozzáadott csapattagok szerepkörét (kattintással), illetve ki is tehetjük a csapatból (*„Remove"*). A szerepkör megváltoztatása egy felugró (modal) ablakban lehetséges, az elérhető szerepkörök közül választhatunk itt:



10–30. ábra: Csapattag szerepkörének módosítása (felugró ablakban)

A szerepkörökről és az engedélyeztetésről a 1. fejezetben lesz bővebben szó.

10.2.2.6. További testre szabási lehetőségek

A felhasználói hitelesítési folyamat számos olyan funkciót, finomhangolási lehetőséget rejt magában, amelyeket programozóként testre szabhatunk, akár egyszerű beállítások engedélyezésével, letiltásával, vagy a meglévő funkciók felüldefiniálásával.

10.2.2.6.1. Funkcionalitások (Fortify és Jetstream) menedzselése

A felhasználói profilos és csapatos felsorolásban vannak olyan funkcionalitások, amelyek mindenképpen elérhetőek a felhasználók számára, de vannak köztük opcionálisan engedélyezhetők vagy letilthatók is. A funkcionalitásokat két csoportba tudjuk sorolni, mivel két felhasználói hitelesítései kezdőkészlet szolgáltatja őket a számunkra:

1. Fortify szolgáltatásai (a config / fortify.php fájlban engedélyezhetők / letilthatók kommenteléssel):
    a. Regisztráció
    b. Jelszó helyreállítás
    c. E-mail-es megerősítés regisztrációkor
    d. Profil adatok frissítés
    e. Jelszó frissítés
    f. Két faktoros hitelesítés
2. Jetstream szolgáltatásai (a config / jetstream.php fájlban engedélyezhetők / letilthatók kommenteléssel):
    a. Felhasználási és adatvédelmi feltételek elfogadása
    b. Profilkép hozzáadása és kezelése
    c. API szolgáltatások
    d. Csapat(ok) menedzselése
    e. Felhasználói profil törlése

Vannak alapértelmezetten letiltott (kikommentelt) funkcionalitások, de innentől mi, fejlesztők döntjük el, hogy melyik szolgáltatásokat tesszük elérhetővé az alkalmazásunkban a felhasználóink számára. Ha például a twoFactorAuthentication sorait (több van!) kommenteljük ki, akkor eltűnik ez a beállítási lehetőség a felhasználói profil beállításai közül. A profil oldal szerkesztése során a Livewire komponensek lesznek segítségünkre, amelyek aszinkron JavaScript hívásokkal javítják a felhasználói élményt, így egy-egy szekció frissítésekor nem töltődik újra a teljes oldal, hanem csak az adott szekció frissül.

A beállítások további finomhangolása az app / Providers / FortifyServiceProvider.php és a JetstreamServiceProvider.php fájlokban lévő boot() metódusok magjában lévő kódsorok segítségével lehetséges. Itt tudjuk beállítani azokat, hogy melyik Fortify vagy Jetstream funkcionalitást melyik Action osztály fog végrehajtani. A FortifyServiceProvider osztályban a hitelesítési folyamatot tudjuk még részletesebben hangolni. A JetstreamServiceProvider osztályban főleg a csapatos és szerepkörös beállításokat tudjuk pontosítani.

10.2.2.6.2. Szerepkörök hozzáadása, szerkesztése, törlése

Az app / Proivders / JetstreamServiceProvider.php fájl configurePermissions() metódusában tudjuk megtenni a szerepkörök menedzselését. Az ott lévő példák jól mutatják, hogy az admin szerepkörű

felhasználó minden CRUD műveletre képes, míg az editor szerepkörűek törlésre nem képesek, de a többire igen.

Az alfejezetben végrehajtott programkód módosítások, beállítási és útvonal fájlok publikálása ebben a GitHub commit-ben érhetők el.

10.2.2.6.3. Útvonalak felüldefiniálása

Útvonalak publikálása a saját projektünk magjába a következő utasítással lehetséges:

```
php artisan vendor:publish
```

Utána a rendszer megkérdezi, hogy mit publikáljon a számunkra: válasszuk ki a **jetstream-routes** értékhez tartozó azonosító számot. Miután ezt megtettük, a **routes** mappába bekerül a **jetstream.php** útvonal regisztrációkat tartalmazó fájl, amely tartalmát így felülírhatjuk, vagy törölhetünk is akár belőle útvonala(ka)t.

A testre szabási műveletek végrehajtása után érdemes lefuttatni a `php artisan test` parancsot, hogy még mindig hibamentesek vagyunk-e, esetleg vannak-e figyelmeztetések, amelyektől eltekinthetünk vagy javíthatjuk őket.

10.2.3. Fortify és Sanctum hitelesítési csomagok

A Fortify szolgáltatásaival már találkoztunk a Jetstream hitelesítési kezdőkészleten belül. A Fortify abban különbözik a Breeze-től és a Jetstream-től, hogy nincsen neki *„frontend"* része, tehát nem generálódnak nézet, szerkezet, komponens fájlok hozzá a projektben. Ellenben tartalmaz minden olyan útvonalat, Controller-t és szolgáltatást, amelyek a felhasználói hitelesítés folyamatának megvalósításához szükségeltetnek (regisztráció, bejelentkezés, jelszó visszaállítás, e-mail-es megerősítés stb.).

Mivel a Fortify nem biztosít saját felhasználói felületet, ezért ez a szolgáltatás csomag leginkább erre a két dologra alkalmas:

1. Mi magunk alakíthatjuk ki azokat a nézeteket, amelyek a hitelesítési szolgáltatások eléréséhez szükségesek (például ad nekünk regisztrációs, bejelentkezési és jelszó visszaállító űrlapot vagy éppen a két faktoros hitelesítés nézeteit stb.).
2. API útvonalakon keresztül az alkalmazás és annak szolgáltatásai megszólíthatók legyenek más programok számára a felhasználói hitelesítés után. A felhasználói hitelesítéshez itt egy másik csomagra is szükség van, ez a Sanctum lesz.

A felsorolás 1. pontjában leírt saját felhasználói grafikus felület elemeit mi magunk fogjuk implementálni egy-egy példán keresztül a fejezet további részében. Gyakorlati szempontból a felsorolás 2. pontját a Postman alkalmazással fogjuk kipróbálni. De a valós ipari környezetekben még inkább valamilyen kliens oldali keretrendszer teszi ezt meg a Postman helyett. Ezek például a következő osztálykönyvtárak: a Vue.js vagy a React, amelyeknek a Fortify biztosít hitelesítési útvonalakat és szolgáltatásokat.

A Fortify egy 3. féltől származó csomag, amely szabadon telepíthető, használható, de nem lenne kötelező ezt alkalmaznunk, csak ha *nem mi magunk vagy nem önállóan* akarjuk megírni a teljes felhasználói hitelesítési folyamat részeit. Ahogy kiemelésre került az iménti felsorolásnál, az API felületnél

végrehajtott felhasználói hitelesítés kapcsán egy másik csomag, a Sanctum is a segítségünkre lesz. A *Sanctum* azonban nem lenne elegendő a teljes hitelesítési folyamat lefedésére, mivel az csak az API token-ek kezelésével, illetve azok segítségével a felhasználók hitelesítésekor létrehozott munkamenet sütikkel foglalkozik, de nem adna hozzá önmagától a rendszerünkhöz útvonalakat és funkcionalitásokat (regisztráció, bejelentkezés, jelszó visszaállítás stb.). A Fortify és a Sanctum csomagok nem ugyanarra a célra vannak, nem tudják helyettesíteni egymást, de mivel ugyanazon a területen működnek, ezért egymást kiegészítik. Ha egy *Single Page Application*-t (SPA-t) készítünk, akkor abszolút előfordulhat, hogy a Laravel-ben, mint backend oldali kiszolgáló környezetben a Fortify csomag végzi a felhasználói regisztrációs, bejelentkezős, jelszó visszaállítós funkcionalitásokat, míg a Sanctum az API token-ek kezelését és a munkamenet hitelesítést.

Ha valamelyik korábbi hitelesítési kezdő készletet már használjuk (Breeze vagy Jetstream) a webalkalmazásunkban, akkor nincsen szükség a Fortify telepítésére, mivel azok a kezdő készletek teljeskörű szolgáltatásokat nyújtanak a felhasználói hitelesítéshez.

10.2.3.1. Telepítés

Hozzunk létre egy új projektet, amely tartalmazza majd a Fortify csomagot a felhasználói hitelesítés funkcionalitásaihoz **l10-auth-fortify** néven (itt lehet nyomon követni a változásait: https://github.com/gla-elte/l10-auth-fortify). Ugyanígy a Laravel 11-es projektet is hozzuk létre **l11-auth-fortify** néven (a változásai itt lesznek nyomon követhetőek: https://github.com/gla-elte/l11-auth-fortify). Hozzuk létre a MySQL-ben az adatbázist is neki Laravel 10 esetén **l10_auth_fortify_db** néven! Laravel 11-ben megfelelő az SQLite adatbázis használata is. Mindkét projektben telepítsük a Fortify-t:

```
composer require laravel/fortify
```

Telepítés után futtassuk le az útvonalakat lekérő parancsot, és vizsgáljuk meg, hogy milyen hitelesítéssel kapcsolatos útvonalak kerültek be a webalkalmazásunkba:

```
php artisan route:list
```

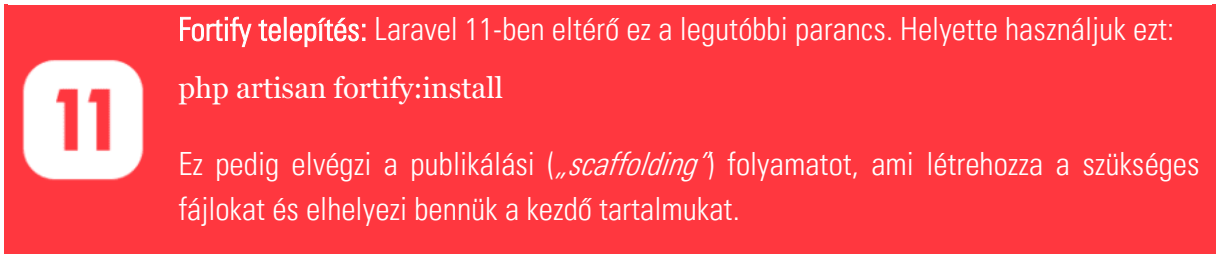
Látható, hogy jónéhány felhasználói hitelesítési útvonal regisztrálásra került:

```
POST      email/verification-notification ..... verification.send › Laravel\Fortify › EmailVerificationNotificationController@store
GET|HEAD  email/verify ..... verification.notice › Laravel\Fortify › EmailVerificationPromptController@__invoke
GET|HEAD  email/verify/{id}/{hash} ..... verification.verify › Laravel\Fortify › VerifyEmailController@__invoke
GET|HEAD  forgot-password ..... password.request › Laravel\Fortify › PasswordResetLinkController@create
POST      forgot-password ..... password.email › Laravel\Fortify › PasswordResetLinkController@store
GET|HEAD  login ..... login › Laravel\Fortify › AuthenticatedSessionController@create
POST      login ..... Laravel\Fortify › AuthenticatedSessionController@store
POST      logout ..... logout › Laravel\Fortify › AuthenticatedSessionController@destroy
GET|HEAD  register ..... register › Laravel\Fortify › RegisteredUserController@create
POST      register ..... Laravel\Fortify › RegisteredUserController@store
POST      reset-password ..... password.update › Laravel\Fortify › NewPasswordController@store
GET|HEAD  reset-password/{token} ..... password.reset › Laravel\Fortify › NewPasswordController@create
GET|HEAD  sanctum/csrf-cookie ..... sanctum.csrf-cookie › Laravel\Sanctum › CsrfCookieController@show
GET|HEAD  two-factor-challenge ..... two-factor.login › Laravel\Fortify › TwoFactorAuthenticatedSessionController@create
POST      two-factor-challenge ..... Laravel\Fortify › TwoFactorAuthenticatedSessionController@store
GET|HEAD  user/confirm-password ..... Laravel\Fortify › ConfirmablePasswordController@show
POST      user/confirm-password ..... password.confirm › Laravel\Fortify › ConfirmablePasswordController@store
GET|HEAD  user/confirmed-password-status ..... password.confirmation › Laravel\Fortify › ConfirmedPasswordStatusController@show
POST      user/confirmed-two-factor-authentication .. two-factor.confirm › Laravel\Fortify › ConfirmedTwoFactorAuthenticationController@store
PUT       user/password ..... user-password.update › Laravel\Fortify › PasswordController@update
PUT       user/profile-information ..... user-profile-information.update › Laravel\Fortify › ProfileInformationController@update
POST      user/two-factor-authentication ..... two-factor.enable › Laravel\Fortify › TwoFactorAuthenticationController@store
DELETE    user/two-factor-authentication ..... two-factor.disable › Laravel\Fortify › TwoFactorAuthenticationController@destroy
GET|HEAD  user/two-factor-qr-code ..... two-factor.qr-code › Laravel\Fortify › TwoFactorQrCodeController@show
GET|HEAD  user/two-factor-recovery-codes ..... two-factor.recovery-codes › Laravel\Fortify › RecoveryCodeController@index
POST      user/two-factor-recovery-codes ..... Laravel\Fortify › RecoveryCodeController@store
GET|HEAD  user/two-factor-secret-key ..... two-factor.secret-key › Laravel\Fortify › TwoFactorSecretKeyController@show
```

10–31. ábra: Fortify csomag felhasználói hitelesítéshez kapcsolódó útvonalai (részlet)

Ezek után publikálhatjuk a Fortify-hoz tartozó beállítások finomhangolásáért felelős fájlokat:

```
php artisan vendor:publish --provider="Laravel\Fortify\FortifyServiceProvider"
```



10–4. újdonság: Fortify telepítése

Az utasításra adott válasz a terminal-ban pontosan leírja, hogy milyen fájlok kerültek publikálásra, de szerencsére, mivel verziókövetjük a projektünket, ezért a VSCode *„Source Control”* menüjében is láthatóak ezek az új fájlok. Ezen fájlok közül többel már találkoztunk a Jetstream kezdő készlet működése során, mivel az közvetlenül tartalmazta a Fortify útvonalait, akció fájljait, beállításait is. Ebben a fejezetben arra a szolgáltatásra koncentrálunk majd, amelyet a Jetstream-nél még nem próbáltunk ki. Ez majd a kicsit bonyolultabb két faktoros hitelesítés lesz, de előtte még a bejelentkezés folyamatát is áttekintjük. Ehhez hozzáadásra került egy migrációs fájl is, amelyet migráljunk az adatbázisba:

```
php artisan migrate
```

A telepítés nem hozott létre automatikus teszteket, így ebben az állapotban még nem tudjuk tesztelni az alkalmazásunkat, csak manuálisan, például weben keresztül saját nézetekkel vagy a Postman alkalmazás segítségével.

A telepítéshez tartozó programkód módosításokat ez a GitHub commit tartalmazza.

10.2.3.2. Kipróbálás

Futtassuk az alkalmazásunkat, és nyissuk meg a böngészőben! A kezdőlapon jobb felül látható lesz a bejelentkezési és regisztrációs link, mivel a GET-es **/login** és **/register** útvonalak regisztrálásra kerültek a

rendszerbe, viszont a nézet nem elérhető hozzájuk, ami rendben is volna a Fortify alapkoncepciója alapján. A hibaüzenet azonban, amit kapunk, ha meglátogatjuk például a bejelentkezési útvonalat, az nem azt mutatja, hogy a bejelentkezési űrlaphoz tartozó nézet hiányzik: *„Target [Laravel\Fortify\Contracts\LoginViewResponse] is not instantiable."*. A probléma egyik megoldási lépése az, hogy regisztrálnunk kell a publikált **FortifyServiceProvider** osztályt a rendszerbe. Nyissuk meg a **config / app.php** fájlt, és keressük meg benne a **'providers'** tömböt (a fájl vége felé lesz), majd adjuk hozzá az *„Package Service Providers"* megjegyzés után ezt (bár jelentősége nincs, hogy hol adjuk hozzá, de itt logikus):

```
App\Providers\FortifyServiceProvider::class,
```

10–39. kódrészlet: FortifyServiceProvider osztály regisztrálása a rendszerbe

10–5. újdonság: Fortify telepítése utáni szolgáltatás regisztrálása

Ez még nem fogja megoldani a problémát, de innentől haladjunk tovább a funkcionalitások megvalósítása felé!

10.2.3.2.1. Bejelentkezés funkció megvalósítása saját nézettel

Nyissuk meg az **app / Providers / FortifyServiceProvider.php** fájlt, és a **boot()** metódusához adjuk hozzá ezt a kódrészletet:

```
Fortify::loginView(function () {
  return view('auth.login');
});
```

10–40. kódrészlet: Login nézet helyének meghatározása

Utána a böngészőben a bejelentkezési (**login**) linkre való kattintás után azt fogjuk kapni, hogy hiányzik a login útvonalhoz tartozó nézet. Hozzunk létre egy új nézet fájlt a **resources / views / auth** új mappában és **login.blade.php** névvel. Most nem a design-on és a szépségen lesz a fő hangsúly, úgyhogy a fájlban egyszerűen csak hozzunk létre egy bejelentkezési űrlapot a következők szerint:

```
<form action="{{ route('login') }}" method="post">
  @csrf
  <div>
    <label for="email">E-mail:</label>
    <input type="email" name="email" id="email">
  </div>
  <div>
    <label for="password">Jelszó:</label>
    <input type="password" name="password" id="password">
  </div>
  <div>
    <input type="submit" value="Bejelentkezés">
```

```
  </div>
</form>
```

10–41. kódrészlet: Bejelentkezési űrlap a Fortify POST login útvonal kipróbálásához

Azonban az adatbázisban és a **users** adattáblában nincsen még egyetlen felhasználónk sem. Nyissuk meg a **database / seeders / DatabaseSeeder.php** fájlt! A **run()** metódus magjából vegyük ki a kommentelést a *„Test User"* adatgyár működtetésének soraiból:

```
\App\Models\User::factory()->create([
  'name' => 'Test User',
  'email' => 'test@example.com',
]);
```

10–42. kódrészlet: Felhasználó gyártása a DatabaseSeeder osztályban

Mentés után futtassuk a következő utasítást:

```
php artisan db:seed
```

Így létrejön a **users** táblában egy *„Test User"* nevű felhasználónk. Ahhoz, hogy a bejelentkezés után elérjünk egy létező, regisztrált **/home** útvonalat, hozzuk létre a **routes / web.php** fájlban a következő útvonalat:

```
Route::get('/home', function () {
  return "Üdv, " . auth()->user()->name;
})->middleware('auth');
```

10–43. kódrészlet: Sikeres bejelentkezés utáni köszöntés megjelenítése a /home útvonalon

Ha ezekután felkeressük a böngészőben a http://127.0.0.1:8000/login útvonalat, és a kapott űrlapot kitöltjük (e-mail: test@example.com, jelszó: password – a **UserFactory** osztály miatt ez az alapértelmezett jelszó), akkor utána a rendszer be is jelentkeztet, és köszönt minket (mivel az **auth** middleware-en sikeresen átmegy a hitelesített kérésünk, és így elérhetővé válik számára a **/home** útvonal). Ha egy másik útvonalra szeretnénk irányítani a felhasználóinkat a bejelentkezés után, akkor a **config / fortify.php** fájlban a **'home'** kulcsot kell átállítani egy másik útvonalra.

Így tehát a Fortify biztosította a felhasználói hitelesítési útvonalakat és funkciót, de magunknak manuálisan kellett a nézetet megvalósítani. Az alfejezet eddigi részében található programkód módosítások ebben a GitHub commit-ben találhatók meg.

10.2.3.2.2. Két faktoros hitelesítés (2FA) megvalósítása saját nézetekkel

Amikor ez a **Features::twoFactorAuthentication()** szolgáltatás aktiválva van (nincs kikommentelve) a **config / fortify.php** beállítási fájl **'features'** tömbjében, akkor a felhasználónak egy 6 számból álló token-t kell megadnia a hitelesítési folyamat során. Ez a token egy időalapú egyszer használatos jelszó (time-based one-time password – TOTP), amit egy arra alkalmas mobil hitelesítési alkalmazásból, például a Google Authenticator-tól kaphatunk meg. Ezt a folyamatot most – a Jetstream-es kész megoldással szemben –, saját magunk fogjuk megvalósítani saját nézetekkel. Kezdjük azzal, hogy a **config / fortify.php** kétfaktoros hitelesítés részében a **confirm** és **confirmPassword** értékeket **false**-ra állítjuk, így nem kell mindig

megerősítenünk a profil beállítások megváltoztatását a jelszavunk megadásával, viszont a bejelentkezés után szükség lesz a hitelesítő (vagy valamelyik érvényes helyreállító) kód megadására. Ha ezt nem tennénk meg, akkor a **confirm-password** nézetet is meg kellene most valósítanunk a 2FA működéséhez.

A feladat megvalósítását a **User** Model osztállyal kezdjük, és használjuk az osztályon belül a **TwoFactorAuthenticatable** trait-et.

```
use Laravel\Fortify\TwoFactorAuthenticatable;
use Laravel\Sanctum\HasApiTokens;

class User extends Authenticatable
{
  use HasApiTokens, HasFactory, Notifiable, TwoFactorAuthenticatable;
```

10–44. kódrészlet: 2FA szempontból releváns kódrészlet a User Model osztályban

Ezután létre kell hoznunk egy nézetet, ahol a felhasználók engedélyezhetik, vagy letilthatják saját maguk számára a 2FA szolgáltatást, és újra generálhatják azokat a helyreállítási kódokat, amelyeket felhasználva bármikor – a 2FA hitelesítő alkalmazás futtatása nélkül is – bejelentkezhetnek.

A 2FA engedélyezéséhez szükség lesz egy saját nézetre, ahol el lehet végezni az engedélyezési, letiltási funkcionalitásokat, továbbá megtalálhatók lesznek rajta a regisztrációs QR kód és a helyreállítási kódok. De előbb az ehhez nézethez vezető útvonalat regisztráljuk a **routes / web.php** fájlban:

```
Route::get('/user/two-factor-auth', function () {
  return view('auth.two-factor-auth');
})->middleware('auth');
```

10–45. kódrészlet: 2FA felhasználói profil beállításokat tartalmazó nézethez vezető védett útvonal

Továbbá a 2FA engedélyezéséhez a Fortify nyújt egy regisztrált útvonalat, amely POST-os elérést igényel a **/user/two-factor-authentication** végponton. Ha a kérés sikeres volt, akkor a felhasználó visszairányításra kerül arra az URL-re, ahol előtte volt. A felhasználói munkamenetet a **session**-ben tároljuk, azon belül is a 2FA engedélyezését a **status** nevű attribútumában. A session, vagyis a munkamenet adatainak elérése a nézetekben a **@session** Blade direktívával lehetséges. Hozzunk létre ehhez egy nézetet a **resources / views / auth** mappában **two-factor-auth.blade.php** néven. Bontsuk három részre logikailag a nézet fájl felépítését:

1. Bevezető szöveg és információ arról, hogy éppen engedélyezve van-e a 2FA vagy sem.
2. Ha nincs engedélyezve a 2FA, akkor egy űrlap és a rajta lévő gomb segítségével engedélyezhetjük.
3. Ha engedélyezve van a 2FA, akkor megjelenik a QR kód, amelyet az Authenticator alkalmazással be tudunk majd olvasni. Továbbá láthatóak lesznek a helyreállítási kódok, amelyeket akár újra is generálhat a felhasználó. Végül pedig a 2FA letiltási űrlap és gombja.

A nézet kódrészletei az iménti felsorolás sorrendjében következnek alább:

```
{{-- 1. rész --}}
<h1>Két faktoros hitelesítés részetei</h1>
<h2>Itt frissítheti a bejelentkezési beállításait.</h2>
@if ( session('status') == 'two-factor-authentication-enabled' )
```

```
  <div>
    A két faktoros hitelesítés (2FA) engedélyezve van.
  </div>
@elseif ( session('status') == 'two-factor-authentication-disabled' )
  <div>
    A két faktoros hitelesítés (2FA) nincs engedélyezve.
  </div>
@endif
```

10–46. kódrészlet: Aktuális kérés szerint engedélyezve van-e a 2FA a felhasználónál

Az engedélyezési folyamat egy feltételvizsgálattal kezdődik, hiszen csak akkor tudjuk engedélyezni, ha az még a felhasználónál nem volt engedélyezve. Ez a **users** adattábla **two_factor_secret** mező értéke szerint tud megtörténni az adott felhasználó adatsorában:

```
{{-- 2. rész --}}
@if ( ! auth()->user()->two_factor_secret )
  <form method="post" action="{{ route('two-factor.enable') }}">
    @csrf
    <div>
      <button>2FA: Engedélyez</button>
    </div>
  </form>
@endif
```

10–47. kódrészlet: 2FA engedélyező űrlap és gomb

Ha pedig engedélyezve van a 2FA, akkor a Blade nézetben egy script alapú megoldással megjeleníthetjük a QR kódot, amely szükséges a Google vagy Microsoft Authenticator alkalmazásban való regisztráláshoz. Ez a háttérben a GET-es **/user/two-factor-qr-code** útvonal végpont alapján kérhető le, de ezt elvégzi helyettünk a rendszer (a **twoFactorQrCodeSvg()** metódushívás által), majd meg is jelenik a QR kód (ami egy JSON objektum, amely svg kulcsot tartalmaz).

A helyreállítási kódok lekérése az adatbázisból, pontosabban a **users** adattábla aktuális felhasználóhoz tartozó sorának **two_factor_recovery_codes** mezőjéből érkeznek. Ezeket a háttérben a GET-es **/user/two-factor-recovery-codes** útvonal végpont megszólításával kéri le az alkalmazás. Ezeket a kódokat a felhasználónak érdemes elmenteni, hiszen a későbbiekben, ha nem működne a 2FA hitelesítő alkalmazás a mobiltelefonján, akkor ezek segítségével bármikor be tudnak lépni. Ezek a kódok azonban csak egyszer használhatóak, ha valamelyiket elhasználta a felhasználó, akkor az utána letiltásra kerül, vagyis megszűnik. A felhasználó rendelkezésére áll ugyanakkor egy gomb, amellyel újra tudja generálni ezeket a helyreállítási kódokat. Ezt a POST-os **/user/two-factor-recovery-codes** útvonal végpont elérésével küldheti be, és egyúttal kap is új kódokat. Végül ebben a szekcióban biztosítunk lehetőséget arra, hogy a felhasználó le tudja tiltani a 2FA funkcionalitást a bejelentkezésénél:

```
{{-- 3. rész --}}
@if ( auth()->user()->two_factor_secret )
  {{-- QR kód a hitelesítési mobil alkalmazásnak --}}
  <div>
    {!! auth()->user()->twoFactorQrCodeSvg() !!}
```

```
  </div>

  {{-- Helyreállítási kódok és újragenerálási lehetőségük --}}
  <ul>
    @foreach ( auth()->user()->recoveryCodes() as $code)
      <li>{{ $code }}</li>
    @endforeach
  </ul>
  <form method="post" action="/user/two-factor-recovery-codes">
    @csrf
    <div>
      <button>2FA: Helyreállítási kódokat újragenerál</button>
    </div>
  </form>

  {{-- Letiltás --}}
  <form method="post" action="{{ route('two-factor.disable') }}">
    @csrf
    @method('delete')
    <div>
      <button>2FA: Letilt</button>
    </div>
  </form>
@endif
```

10–48. kódrészlet: A 2FA funkcionalitás engedélyezés utáni lehetőségei

A folyamat így teljes a felhasználói profil beállításának 2FA szempontjából, de ezeket a lépéseket nézzük végig a böngészőben, és próbáljuk ki őket!

10–32. ábra: Bejelentkezés után a /user/two-factor-auth útvonal meglátogatása

Engedélyezés után megjelennek a szükséges részek:

10–33. ábra: Engedélyezett 2FA beállításai és lehetőségei

Megnyithatjuk például a Google Authenticator alkalmazást, és bescannelhetjük a QR kódot. A scannelés után megjelenik a listában az alkalmazásunk neve (Laravel), majd utána kettőspont és a felhasználónk e-mail címe, akivel be vagyunk jelentkezve a webes alkalmazásba (test@example.com). A helyreállítási kódokat újra generálhatjuk gombnyomásokkal, a letiltást egy kicsit később próbáljuk ki.

Mindenekelőtt készítsük el a kijelentkezés logikáját, a POST **/logout** útvonal adott a Fortify által. De még szükség van arra, hogy például a **welcome** nézetben a **Home** link megjelenítése után helyezzünk el egy űrlapot és egy gombot, amivel ki tud jelentkezni a felhasználó (a gomb stílusát a *„Home"* link elemből kimásolhatjuk).

```
<form action="{{ route('logout') }}" method="post">
  @csrf
  <button class="font-semibold text-gray-600 hover:text-gray-900 dark:text-
gray-400 dark:hover:text-white focus:outline focus:outline-2 focus:rounded-
sm focus:outline-red-500">Kijelentkezés</button>
</form>
```

10–49. kódrészlet: Kijelentkezési űrlap kódja a főoldal @auth szekciójában (a home link után)

Továbbá szükség van még egy nézetre annak a teszteléséhez, hogy a kijelentkezésünk után a bejelentkezési adatok (e-mail cím és jelszó) megadása után, a 6 számból álló hitelesítési kódot meg tudjuk

adni (ez a kód a mobil alkalmazásban időről-időre frissül). Ehhez nyissuk meg az **app / Providers / FortifyServiceProvider.php** fájlt, és a **boot()** metódusához adjuk hozzá az alábbi kódot:

```
Fortify::twoFactorChallengeView(function () {
  return view('auth.two-factor-challenge');
});
```

10–50. kódrészlet: A 2FA kihívás nézetéhez vezető útvonal regisztrálása

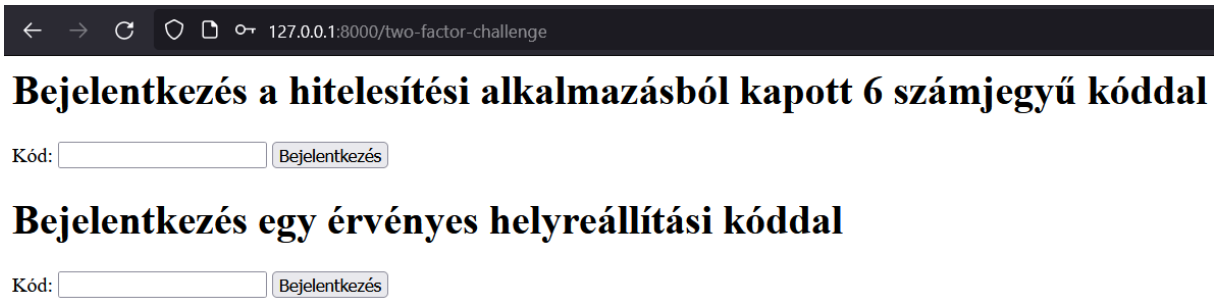
Hozzuk létre az új nézetet a **resources / views / auth** mappában **two-factor-challenge.blade.php** névvel és az alábbi tartalommal:

```
<h1>Bejelentkezés a hitelesítési alkalmazásból kapott 6 számjegyű
kóddal</h1>
<form method="POST" action="/two-factor-challenge">
    @csrf
    <label for="code">Kód:</label>
    <input id="code" type="text" name="code" />
    @error('code')
      <div>{{ $message }}</div>
    @enderror
    <button>Bejelentkezés</button>
</form>

<h1>Bejelentkezés egy érvényes helyreállítási kóddal</h1>
<form method="POST" action="/two-factor-challenge">
  @csrf
  <label for="recovery_code">Kód:</label>
  <input id="recovery_code" type="text" name="recovery_code" />
  @error('recovery_code')
    <div>{{ $message }}</div>
  @enderror
  <button>Bejelentkezés</button>
</form>
```

10–51. kódrészlet: 2FA kihívás teljesítésének nézete a két űrlappal

A kód lényegi része két űrlapból áll, az első a hitelesítési kódot várja a mobilos hitelesítő alkalmazásból, ha pedig ez nem működne, akkor megadhatjuk a még érvényes helyreállítási kódok valamelyikét. Az érvényesítések működnek és ha szerver oldali hiba érkezik, akkor az meg fog jelenni az adott bemeneti mező mellett.

Most folytathatjuk a manuális tesztelést, ehhez jelentkezzünk ki a főoldalon, majd jelentkezzünk újra be! Utána megkapjuk ezt a 2FA kihívást tartalmazó nézetet:



10–34. ábra: 2FA kihívás nézete a két űrlappal

Ha bármelyiket elrontjuk, akkor hibaüzentet kapunk a bemeneti mezők után, ellenben, ha helyes és érvényes kódot adunk meg, amely akár a mobilalkalmazásból, akár a helyreállítási kódlistából érkezik, akkor bejelentkeztet minket a rendszer, és a **/home** útvonalon köszönt minket, ahogy azt korábban már implementáltuk.

Ha most újra meglátogatjuk a **/user/two-factor-auth** útvonalat és letiltjuk a 2FA-t, akkor utána egy kijelentkezést követő bejelentkezésnél már nem kéri tőlünk a 2FA kódokat (a **users** adattábla adott felhasználóhoz tartozó sorából is törlődnek ilyenkor a **two_factor_secret** és **two_factor_recovery_codes** mező értékei).

10–35. ábra: 2FA letiltásakor látható munkamenet információ és az újra megjelenő engedélyező gomb

Ennek a két Fortify által nyújtott szolgáltatásnak, folyamatnak (bejelentkezés, 2FA) a bemutatás remélhetőleg elegendő tapasztalatot nyújtott ahhoz, hogy akár a többi folyamatot (regisztráció, jelszó megerősítés, e-mail-es megerősítés stb.) is képesek legyünk akár önállóan implementálni. Ehhez persze nagy segítségünkre van a Fortify dokumentációja, amely főleg a felhasználói kérések (útvonal végpontok) kötelező bemeneti paramétereit és a kérésekre adott lehetséges válaszait is tartalmazza.

Az alfejezetben megtalálható programkód módosítások elérhetők ebben a GitHub commit-ben.

10.2.3.2.3. Bejelentkezés funkció megvalósítása API hívás számára cookie alapon

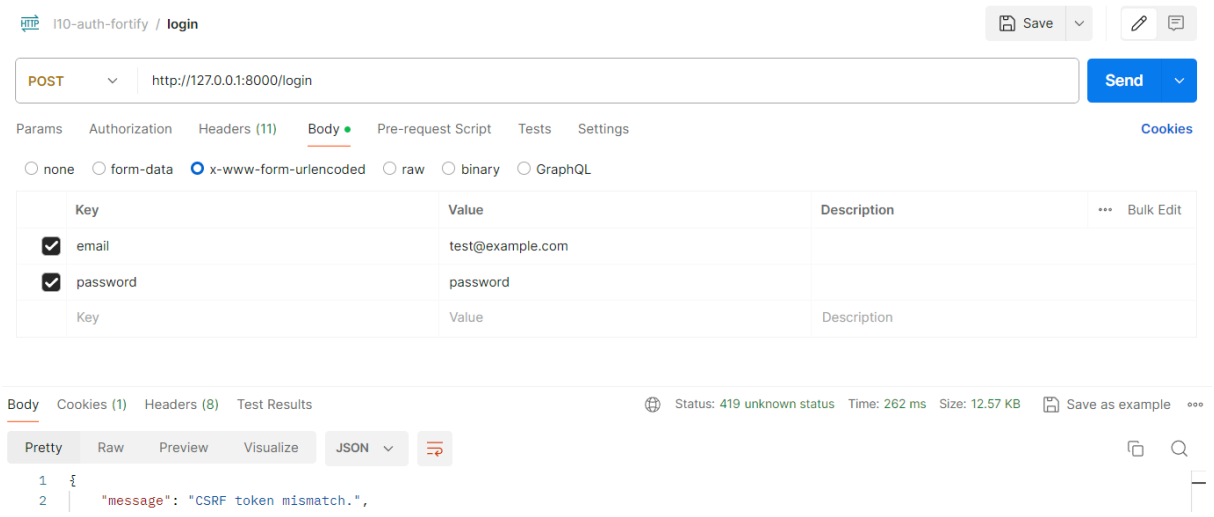
Ahogy arra korábban tettünk utalást, a Sanctum csomag segít minket a felhasználói hitelesítésben API kérések esetén. Lehetőségünk van API token alapú azonosításra (ezt a 10.2.2.4.4. alfejezetben tekintettük át), illetve cookie alapú azonosításra is, amelynek a beállítása bonyolultabb, ellenben a megfelelő beállítások után jobban véd a CSRF és XSS támadástípusok ellen (ezekről részletesebben a 15.4. alfejezetben olvashatunk).

Láthattuk, hogy alapértelmezetten a Fortify regisztrál útvonalakat, amelyek nézetekkel térnének vissza. Ha azonban egy SPA-t készítünk, akkor ott a kliens oldali keretrendszerben kell a megjelenítést elkészíteni, így nincs szükség arra, hogy a Laravel-ben a Fortify regisztráljon olyan GET-es útvonalakat, amelyek egy-egy nézettel térnek vissza, például a **login**, **register**, **two-factor-challenge** stb. Ekkor megtehetjük azt, hogy a **config / fortify.php** fájlban a **'views'** kulcsot megtalálva átállítjuk az értékét **true**-ról **false**-ra. Ekkor ezek a nézettel visszatérő GET-es útvonalak kikerülnek a regisztrált (és így elérhető) útvonalak listájából.

A Fortify tehát adja nekünk az útvonalakat és a funkcionalitásokat, amelyek a felhasználói hitelesítés során elengedhetetlenek. Alkalmazzuk egy API hívásnál a POST **/login** útvonalat a Postman alkalmazásban. Az itt végrehajtott kéréseknek hozzunk létre egy új gyűjteményt (collection) a Postman-ben **l10-auth-fortify** néven. Az új gyűjteményhez adjunk hozzá egy új kérést! A kérés beállítási paraméterei:

- *Kérés neve:* login
- *HTTP metódus:* **POST**
- *Útvonal:* **http://127.0.0.1:8000/login**
- *Headers*-nél egy új **Accept** fejléc elem, ahogy már korábban is láthattuk ezzel az értékkel: **application/json**
- *Body: x-www-form-urlencoded* lapon a következő kulcs-érték párok:
  - **email → test@example.com**
  - **password → password**

Majd futtassuk a kérést! Nem fog lefutni, mivel egy 419-es HTTP státuszkódú választ kaptunk. *„CSRF token mismatch."* üzenettel (lásd alább).



10–36. ábra: Bejelentkezés útvonal elérése a Postman-ben, sikertelen válasszal

Ezt a hibakódot már többször megkaptuk korábban is, amikor elküldtük valamelyik űrlapunkat (a **create** vagy az **edit** nézeteknél) úgy, hogy nem szerepelt benne a **@csrf** Blade direktíva. Ez tette hozzá azt a token-t az űrlap elemeihez, ami védte az alkalmazásunkat a Cross-Site Request Forgery támadás ellen. API hívásnál azonban egy kicsit más módon kell hozzáadni ezt a CSRF token-t a kérésekhez.

Ennek a problémának a megoldásánál segít minket a Sanctum csomag (hivatalos dokumentációja itt érhető el: https://laravel.com/docs/10.x/sanctum). Ez egy olyan felhasználói hitelesítést segítő rendszer,

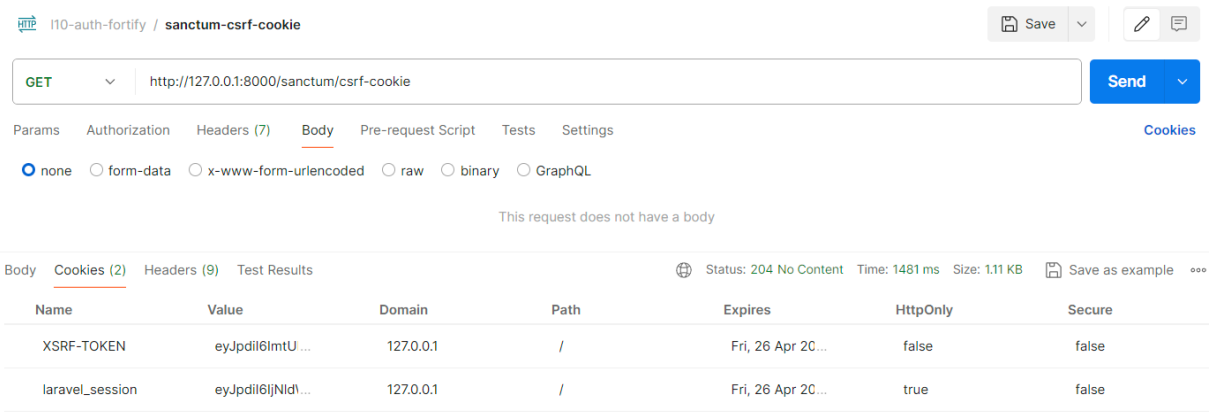
amely tipikusan SPA alkalmazásoknál vagy akár mobil alkalmazásoknál, esetleg token alapú API hívások számára ad támogatást a Laravel keretrendszer magjának eléréséhez. A Sanctum lehetővé teszi, hogy az alkalmazás minden felhasználója API token-t generáljon a fiókjához vagy akár csak egy-egy (például űrlap elküldési) kéréséhez is. A későbbiekben láthatjuk majd, hogy ezeken a token-eken keresztül képességekkel vagy hatáskörökkel ruházhatjuk fel a felhasználót bizonyos webalkalmazásban végrehajtható funkcionalitások eléréséhez vagy éppen letiltásához (*megjegyzés:* a 10.2.2.4.4. alfejezetben a Jetstream API szolgáltatásainál is láttunk már hasonlót).

A Sanctum-ot nem kell most külön telepítenünk, mivel a Fortify telepítése magába foglalta a Sanctum telepítését is.

A Sanctum a CSRF token hozzáféréséhez biztosít egy nyilvánosan elérhető útvonalat, amelyet a Postman segítségével érhetünk el:

- *Kérés neve:* sanctum-csrf-cookie
- *HTTP metódus:* GET
- *Útvonal:* http://127.0.0.1:8000/sanctum/csrf-cookie

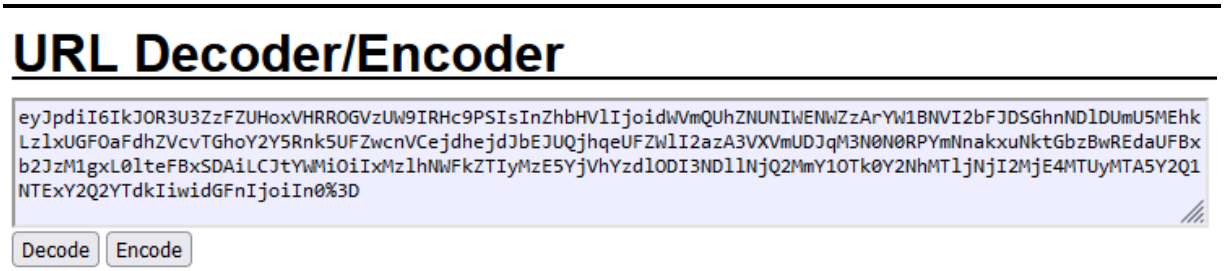
Kapunk egy sikeres választ, amelynek a *„Body"* része üres, de ha átváltunk a *„Cookies"* lapfülre, akkor ott látható lesz a számunkra fontos XSRF-TOKEN és értéke.



10–37. ábra: CSRF token sikeres lekérése a Postman-ben

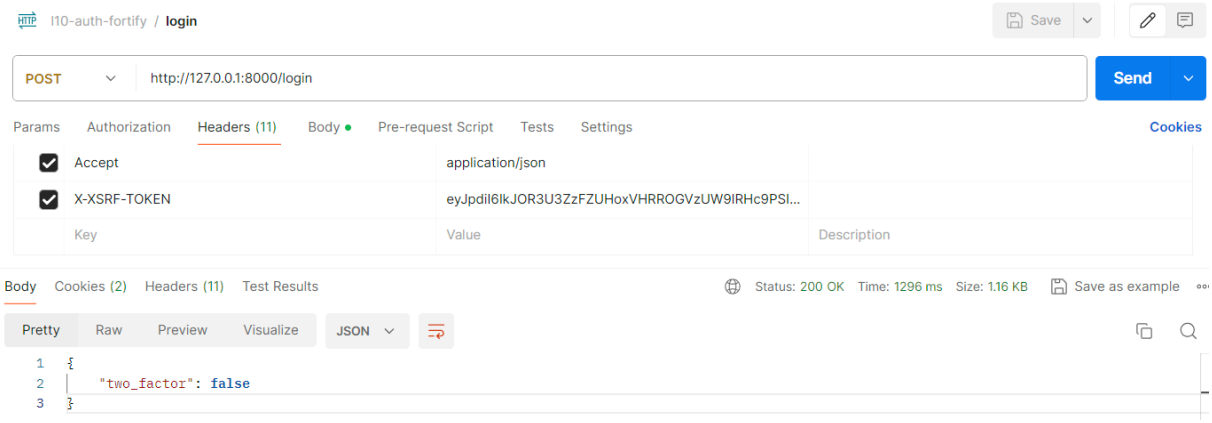
Most térjünk vissza a korábbi bejelentkezési kérésünkhöz, de a kérés *„Headers"* részéhez adjunk hozzá egy új fejlécet X-XSRF-TOKEN (figyelem, az új fejléc mező kapott egy X- előtagot is) névvel és az iménti sanctum/csrf-cookie útvonal lekérésekor megkapott XSRF-TOKEN Cookie értékével (másolás-beillesztés).

Azonban, ha most újraindítjuk a kérést, akkor még mindig a 419-es HTTP hibakódot kapjuk. Ez amiatt van, mert a Laravel kódolja a token-t. A visszakódoláshoz látogassuk meg például ezt az oldalt: https://meyerweb.com/eric/tools/dencoder/ Illesszük be a vágólapunkon lévő XSRF-TOKEN értéket a szöveges mezőbe és nyomjuk meg a *„Decode"* gombot.



10–38. ábra: URL dekódolás

A dekódolási tesztek során leggyakrabban azt tapasztalhatjuk, hogy a karaktersorozat végén látható *„%3D"* érték átváltozik *„="*(egyenlőségjelre). Ha ezt az új értéket adjuk meg a Postman kérés új **X-XSRF-TOKEN** értékének, és újra elküldjük a kérést, akkor most már megfelelő eredményt fogunk kapni.



10–39. ábra: Bejelentkezés útvonal elérése a Postman-ben, sikeres válasszal

Mivel a Fortify telepítésre került korábban, azt a választ fogjuk kapni a 200-as HTTP státuszkódon és OK üzeneten kívül, hogy ennek a felhasználónak jelenleg éppen nincsen engedélyezve a 2FA (attól függően, hogy a korábbi teszteléseknél éppen milyen állapotban hagytuk ezt az értéket kell most **true** vagy **false** értéket megkapnunk).

10.2.3.2.4. Süti automatikus hozzáadása a kérésekhez

A CSRF token-ek csak aránylag rövid ideig érvényesek, ellentétben az API hívásoknál használható felhasználói token-ekkel. Először muszáj manuálisan elérni/lekérni a Postman-ben a **sanctum/csrf-cookie** útvonalat. Aztán az itt végrehajtott manuális folyamat további részeit viszont érdemes lenne programozhatóan automatikussá tenni, hogy a szükséges hívásoknál hajtsa végre ezeket:

- kérje le a **sanctum-csrf-cookie** útvonalat,
- nyerje ki a kapott válaszból a Cookie értékét,
- dekódolja az URL-t,
- és végül adja hozzá a tényleges API kérésünkhöz.

Ezt az automatizálást a POST **/login** útvonalunk kérésénél a *„Pre-request Script”*[18] lapfülön tudjuk megtenni. Ide JavaScript kódot tudunk írni, amely a kérés elindítása előtt mindig lefut. *Megjegyzés:* a Postman-ben a script írásakor meglehetősen jól működik a kódolást segítő Intellisense[19] szolgáltatás.

```
const jar = pm.cookies.jar();

jar.get("http://127.0.0.1:8000", "XSRF-TOKEN", (error, cookie) => {
    console.log(error, cookie);
});
```

10–52. kódrészlet: POST /login kérés "Pre-request Script"-je a Postman-ben

Elindíthatjuk a kérést, majd bal alul nyissuk fel a *„Console”*-t a Postman-ben! A következő hibát kapjuk meg: *„CookieStore: programmatic access to "127.0.0.1" is denied”*. A probléma megoldásához a kérés elküldésének nagy *„Send”* gombja alatt találjuk a *„Cookies”* linket, kattintsunk rá! A felugró kisablakban bal alul találjuk a *„Domains Allowlist”* gombot, nyomjuk meg! A szövegmezőbe adjuk meg ezt: **127.0.0.1** majd nyomjuk meg mellette az *„Add”* gombot, és bezárhatjuk ezt a kis ablakot. Ha most újra elküldjük a kérést, akkor alul a *„Console”*-ban látnunk kell a kiírt dekódolt CSRF token értéket.

Folytathatjuk a *„Pre-request Script”* kódunkat, a kiíratást vegyük ki belőle, és bővítsük a callback függvényt a fejléc automatikus, programozott hozzáadásával (a kódban a **pm** a Postman-t, saját magát jelöli, az ő kéréseit, környezetét, Cookie tárját stb. tudjuk vele elérni).

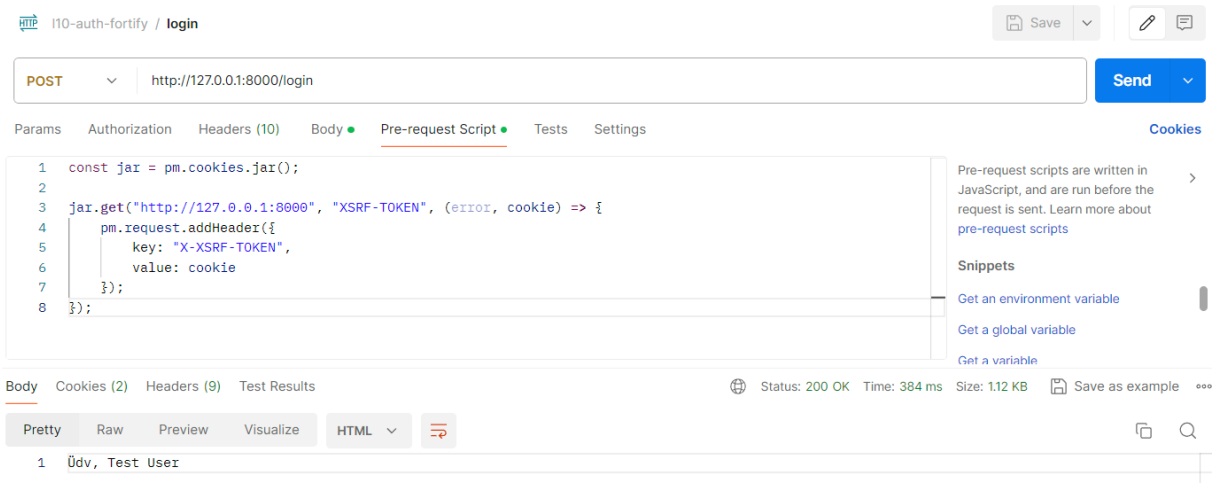
```
pm.request.addHeader({
    key: "X-XSRF-TOKEN",
    value: cookie
});
```

10–53. kódrészlet: A fejléc automatikus, programozott hozzáadása a kéréshez

Ha azonban most újra elküldjük a kérést, még mindig a 419-es *„CSRF token mismatch.”* válasz üzenetet kapjuk. Ez azért van, mert a *„manuális”* token hozzáadást ki kell törölnünk a kérés *„Headers”* részéből úgy, hogy a sorának a jobb szélén a kuka gombra kattintunk. Így most már sikeres választ kapunk a kérésünkre.

[18] Az elnevezés a Postman legújabb verziójában (v11.1.14) már külön lett szedve és csak *„Scripts”* szekció van, amin belül tovább lett bontva: *„Pre-request”* és *„Post-request”* részekre. Az itt olvasható kódokat mindig a *„Pre-request”* lapfülön írjuk be a Postman-be.

[19] Az IntelliSense egy általános kifejezés a különböző kódszerkesztési funkciókra, beleértve a következőket: kódkiegészítés, paraméterinformáció, gyorsinformáció, és tagfüggvény vagy tagváltozó listák.



10–40. ábra: Sikeres bejelentkezés után a válaszban vissza is kapjuk az üdvözlő /home útvonalunk eredményét

Feltételezhetjük, hogy a gyűjteményben több ilyen POST kérésünk is lesz, amelyekbe mindenhova be kellene másolni a *„Pre-request Script"* kódunkat, de megtehetjük ezt úgy is, hogy a gyűjteményt magát (**l10-auth-fortify** névre hallgat) nyitjuk meg, és az ott is a *„Pre-request Script"* lapfülön lévő mezőbe másoljuk be a kódunkat, a **login** kérésünknél pedig így már ki is vehetjük az ottani kódot.

Végül teszteljük le a **routes / api.php**-ban található útvonalunkat, amely a bejelentkezett felhasználót magát adja vissza akkor, ha megfelel az **auth:sanctum** Middleware-nek a kérés:

- *Kérés neve:* user
- *HTTP metódus:* **GET**
- *Útvonal:* **http://127.0.0.1:8000/api/user**
- *Headers*-nél egy új **Accept** fejléc elem, ahogy már korábban is láthattuk ezzel az értékkel: **application/json**

Eredményül egy 401-es HTTP hibakódot kapunk, ami *„Unauthorized"*-ot jelent, de emellett a válasz az *„Unauthenticated."* üzenetet tartalmazza. Ez hasonló a 403 *„Forbidden"*-hez, de kifejezetten akkor használható, ha a hitelesítés lehetséges, viszont nem sikerült vagy még nem történt meg. A válasznak tartalmaznia kell egy *„WWW-Authenticate"* fejlécmezőt, amely a kért erőforrásra (útvonalon keresztül az adott felhasználóra) vonatkozó kihívást tartalmazza.

Természetükből adódóan az API hívások nem használják a CSRF ellenőrzést, mivel ezek az API hívások (útvonal elérések) *állapotmentesek*, ami azt jelenti, hogy nem tárolnak munkamenet adatokat, és nem tartják meg a felhasználó állapotát a kérések között. Ezt a Laravel 10-ben az **app / Http / Kernel.php**-ban lévő **$middlewareGroups** tömb **'api'** elemeinél is láthatjuk (csak a **ThrottleRequests** és a **SubstituteBindings** Middleware-ek vannak regisztrálva az API útvonal kérésekhez alapértelmezetten). Azonban az a legritkább esetben fordul elő, hogy a webalkalmazásunk nyilvános API felülete semmilyen felhasználói azonosítást nem várna el azoktól, akik el szeretnék érni… Mi viszont szeretnénk állapotfüggővé tenni az API kéréseinket, hogy a felhasználói azonosítást elvégezzük, és utána a saját munkamenetét kezeljük.

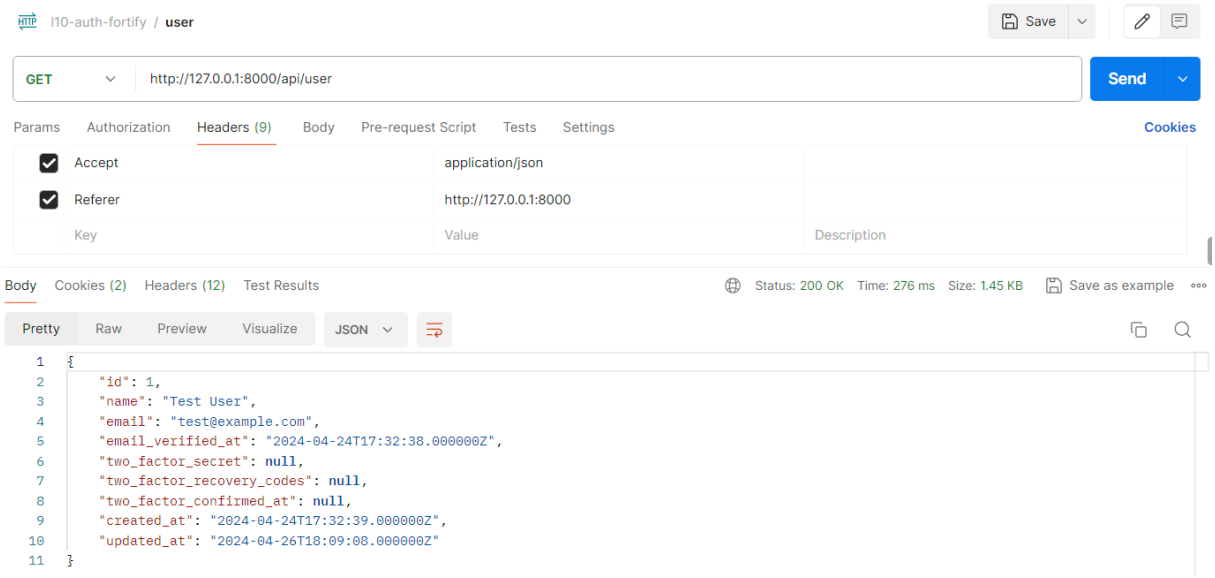
A **Kernel.php**-ban az **'api' $middlewareGroups** tömbhöz adjuk hozzá elsőként a következőt, illetve, ha ott van, akkor vegyük ki előle a kommentelést, hogy érvényre jusson:

```
\Laravel\Sanctum\Http\Middleware\EnsureFrontendRequestsAreStateful::class,
```

10–54. kódrészlet: Kérést indító kliens azonosítását elvégző köztes réteg

A **EnsureFrontendRequestsAreStateful** Middleware **handle()** metódusát, ha megvizsgáljuk, akkor van benne egy **fromFrontend()** metódushívás a kérés ellenőrzéséhez. Ez a **fromFrontend()** metódus a kérés fejléc értékei között keresi a *referer* vagy *origin* fejléc mező értékét. A metódus az állapottal rendelkező domain-ek listáját a **config / sanctum.php** fájlban lévő **'stateful'** értékei között keresi. Itt van például a localhost, a localhost:3000, 127.0.0.1, 127.0.0.1:8000 és még más értékek is. Ha ezekhez szeretnénk hozzáadni újat, akkor az **.env** fájlt érdemes bővíteni egy **SANCTUM_STATEFUL_DOMAINS** kulccsal és további értékeket hozzáadni az alapértelmezetteken definiált értékeken túl. Például a **localhost:5173** címet (http:// protokoll előtag nélkül), ha az **.env** fájlban van **FRONTEND_URL** beállítva ugyanerre a címre (de itt már http:// protokoll előtaggal).

A korábban említett, Postman-ben látható 401-es hiba abból adódik, hogy van még egy hiányzó fejléc mezőnk a kérésnél, amely vagy a *„Referer”* vagy az *„Origin”*. Válasszuk a *„Referer”*-t, amivel a klienst azonosítjuk, aki a kérést küldte. Állítsuk be neki a **http://127.0.0.1:8000** címet. Ha most lefuttatjuk a Postman-ben először a **login** kérést (tehát bejelentkezünk, ha nem lettünk volna bejelentkezve), utána pedig a most felparaméterezett **user** kérésünket, akkor meg kell kapnunk egy 200-as OK válasz mellett a felhasználónk adatait JSON formátumban.



10–41. ábra: Bejelentkezett felhasználónk adatainak sikeres lekérése

Ez a *„Referer”* fejléc érték persze az összes gyűjteménybe foglalt kéréshez szükségeltetik, úgyhogy helyezzük át ebből a kérésből a gyűjtemény *„Pre-request Script”* kódrészébe, méghozzá a másik fejléc mező (**X-XSRF-TOKEN**) elé vagy után, nincs jelentősége a helyének ilyen szempontból.

```
pm.request.addHeader({
    key: "Referer",
    value: "http://127.0.0.1:8000"
```

```
});
```

10–55. kódrészlet: Referer fejléc mező értékének hozzáadása a gyűjteményhez tartozó összes kéréshez

Ha az átalakítás után is működik a **user** kérésünk, akkor ez így megfelelő lesz és jó munkát végeztünk!

10.2.3.2.5. Kijelentkezés megvalósítása API hívással

Ezekután a kijelentkezés kipróbálása rendkívül egyszerű, mindössze egy POST **/logout** kérést kell létrehozni a Postman-ben. Ha be vagyunk jelentkezve, akkor pedig végre tudjuk hajtani a kérést, és üres választ, de 204 No Content HTTP státuszkódot ad vissza a rendszer. Utána pedig nem lesz már elérhető például a GET **api/user** útvonal, 401 Unauthenticated választ ad rá a rendszerünk.

10.2.3.2.6. Regisztráció megvalósítása API hívással

A két faktoros hitelesítés előtt, hozzunk létre, vagyis regisztráljunk API hívással egy új felhasználót!

Ehhez egy új kérés létrehozására lesz szükségünk a Postman-ben: POST **/register**

A http://127.0.0.1:8000/register útvonal végpont bemenetként kötelezően elvárja tőlünk a következőket:

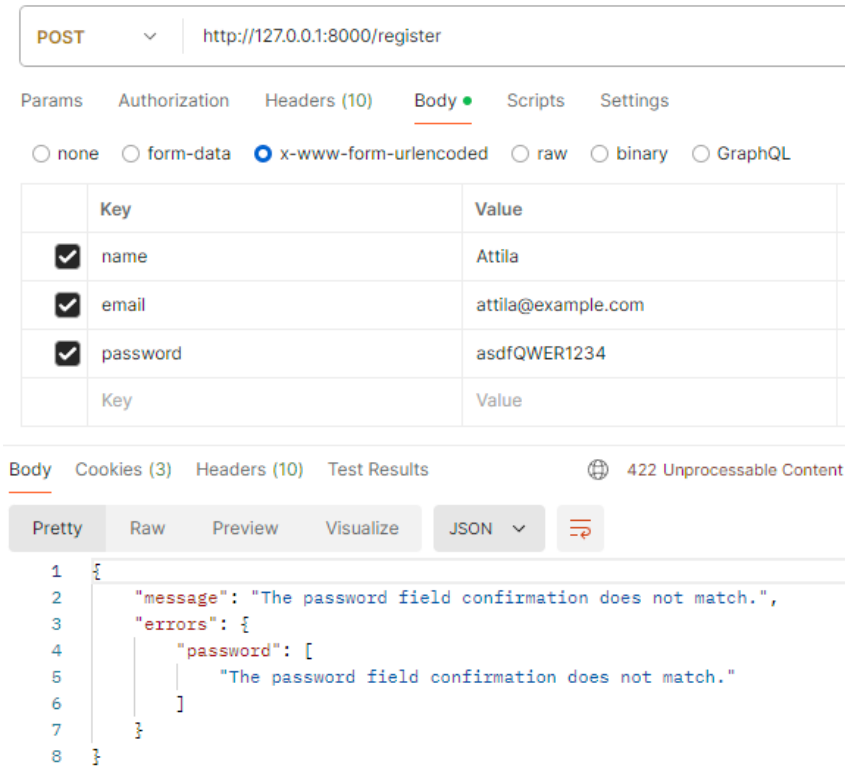
1. *name* (string vagyis szöveges elem)
2. *email* (ami a **config / fortify.php** alapértelmezett beállítása szerint az e-mail cím lesz, de attól függően, hogy a username-et vagy az email-t állítottuk alapértelmezettnek, ez lehet a **User** Model osztály username mezője is, ha van ilyenünk)
3. *password* (jelszó, a házirend betartásával: a szabályokért nézzük meg az **app / Actions / Fortify / CreateNewUser.php** fájl validációs részét)
4. *password_confirmation* (elvárt a megfelelő jelszó mező megismétlése)

Body nélküli kérés elküldésénél működik a validáció, és jelzi nekünk a válaszban a rendszer, hogy milyen mezők hiányoznak (422-es HTTP hibakódot is visszaad):


```
{
    "message": "The name field is required. (and 2 more errors)",
    "errors": {
        "name": [
            "The name field is required."
        ],
        "email": [
            "The email field is required."
        ],
        "password": [
            "The password field is required."
        ]
    }
}
```


10–42. ábra: Validáció működése API-n keresztüli regisztráció során

Ha a *„Body”* (x-www-form-urlencoded) lapfülön hozzáadjuk ezeket a mezőket, akkor még mindig hibát fogunk kapni, mert hiányzik a jelszó megerősítése:



10–43. ábra: Jelszó megerősítés mező is kötelező és egyeznie kell a jelszóval a regisztrációnál

Ha megadjuk a *password_confirmation* mezőt is, akkor végre fog hajtódni a regisztráció, kapunk egy üres választ, de 201 Created HTTP állapotkódot is, ami a sikeres létrehozásra utal.

10.2.3.2.7. Két faktoros hitelesítés megvalósítása API hívások számára

A két faktoros hitelesítést meg lehet oldani API hívásokkal is, tehát nem szükségesek hozzá nézetek. A **User** Model osztályban már elvégeztük korábban a módosításokat, így az fel van készítve a 2FA működtetésére. Az iménti regisztrált felhasználót pedig igénybe vehetjük az itteni 2FA működésének tesztelése során.

Tekintsük át, hogy milyen útvonal végpontok tartoznak a 2FA témaköréhez! Ezeket aztán a Postman-ben a már meglévő gyűjteményünkben egy külön könyvtárba is szervezhetjük (adjuk hozzá a 2FA könyvtárat a gyűjteményhez). A folyamat onnantól indul, hogy bejelentkezünk egy olyan regisztrált felhasználóval, akinek nincs engedélyezve a 2FA, ő a következő funkcionalitásokat éri el az útvonal végpontokon keresztül:

- 2FA engedélyezése: POST **/user/two-factor-authentication**
    - Siker esetén 200 OK HTTP állapotkóddal tér vissza a rendszer és egy üres válasszal.

Engedélyezés után már több funkcionalitás, útvonal végpont elérése ad érdemi válaszokat, amikor be vagyunk jelentkezve:

- 2FA letiltása: DELETE **/user/two-factor-authentication**
    - Siker esetén 200 OK HTTP állapotkóddal tér vissza a rendszer és egy üres válasszal.
- Helyreállítási kódok lekérése: GET **/user/two-factor-recovery-codes**
    - Siker esetén 200 OK HTTP állapotkóddal tér vissza a rendszer és a válasz törzsében a nyolc darab helyreállítási kódot tartalmazó gyűjteménnyel.

- Helyreállítási kódok újra generálása: POST /user/two-factor-recovery-codes
  - Siker esetén 200 OK HTTP állapotkóddal tér vissza a rendszer és egy üres válasszal.
- QR kód lekérése: GET /user/two-factor-qr-code
  - A kérés *„Accept" „Header"* elemét szokás szerint application/json-re kell állítani.
  - Siker esetén 200 OK HTTP állapotkóddal tér vissza a rendszer és a válasz törzsében az svg QR kód képfájl digitális kódjával. Megjegyzés: akkor is meghívható ez az útvonal, ha nincs engedélyezve a 2FA, viszont ebben az esetben csak egy üres tömbbel tér vissza a QR kód kódja helyett.
  - A QR kód megjelenítése (vizualizálása) sajnos nem működik a Postman-ben (talán azért is, mert kódolva van, vagyis rengeteg backslash karaktert tartalmaz a különleges karakterek – például az idézőjelek – levédése miatt). Viszont nekünk szükségünk van arra, hogy az új felhasználónkhoz tartozó QR kódot elmentsük az okostelefonunk hitelesítő alkalmazásába. Emiatt az a célunk, hogy megjelenítsük valamilyen módon a visszakapott svg kódból magát a QR kód képét. Ezt két lépésben tudjuk megtenni:
    - Másoljuk ki a válaszból az <svg> nyitó- és zárótag-je közötti értéket a nyitó- és zárótag-gel együtt.
      - A dekódoláshoz a PHP nyelv stripslashes() metódusát kell használnunk, amit online is megtehetünk például itt: https://onlinephp.io/stripslashes
      - Beillesztjük a vágólapunkra helyezett <svg> tag-et és tartalmát. Futtatjuk a kódot és megkapjuk az eredményt már backslash karakterek nélkül.
      - A backslash-ektől *„megtisztított"* <svg> kódot egy online svg megjelenítőbe illesszük be, például itt: https://www.svgviewer.dev/
      - A generált QR kódot már scan-nelhetjük az okostelefonunk valamely hitelesítő (Google, Microsoft stb.) alkalmazásával.
      - A beolvasás után megjelenik a hitelesítő alkalmazásban a webes projektünk neve (ha nem változtattuk meg az .env fájlban az APP_NAME attribútumot, akkor Laravel) és az e-mail cím, amivel bejelentkeztünk korábban. A 6 számot később a bejelentkezés megerősítésénél tudjuk alkalmazni, ha az adott felhasználónál engedélyezésre került a 2FA.
- Jelszó megerősítése: POST /user/confirm-password
  - Ha a config / fortify.php beállítási fájlban, 'features' tömbben a twoFactorAuthentication() paraméterében a 'confirmPassword' beállítás true-ra van állítva, akkor a fontosabb funkcionalitások (2FA engedélyezés/letiltás, új helyreállítási kódok generálása stb.) végrehajtásához jelszavas megerősítést kér a rendszer. Ekkor a folyamat részévé kell tenni ennek a kérésnek a végrehajtását, például a 2FA engedélyezése/letiltása után ezt a kérést kell elindítani, hogy ténylegesen érvényre jusson a 2FA engedélyezése/letiltása.
  - A kérés *„Accept" „Header"* elemét szokás szerint application/json-re kell állítani.
  - A kérés *„Body"* részében pedig szükség van a *password* mező definiálására, aminek értékül a felhasználó aktuális jelszavát kell megadni.

Engedélyezés után, de kijelentkezett állapotban szükség van a bejelentkezésre, akkor nem elég csak a POST **/login** útvonal elérése, hanem meg kell erősíteni bejelentkezést:

- 2FA kihívás: POST **/two-factor-challenge**
    - A kérés *„Accept" „Header"* elemét szokás szerint **application/json**-re kell állítani.
    - A kérés *„Body"* részében pedig két módon is elvégezhetjük a bejelentkezést:
        - *code* értéke az okostelefonos hitelesítő alkalmazásból kapott (időben lejáró és frissülő) 6 számból álló kód.
        - *recovery_code* értéke a helyreállítási kódok közül az egyik, amely még érvényes. Használat után az adott kód érvénytelenné válik, de belépés után bármikor újra tudjuk generálni a helyreállítási kódok gyűjteményét.
        - Amelyiket használjuk a fenti kettő közül az legyen kipipálva a Postman-ben, a másik ne legyen kipipálva.

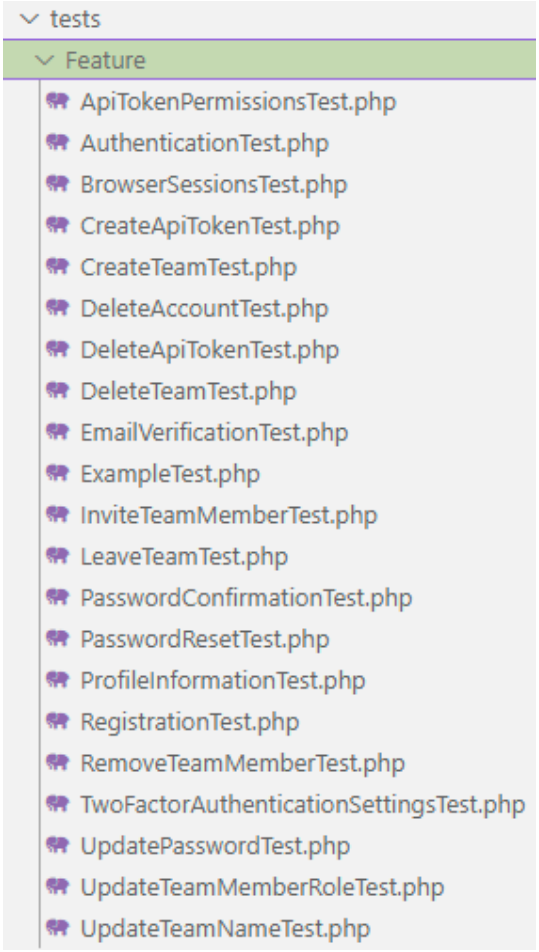
A fentiek ismertetése után érdemes kipróbálni ezeket a funkcionalitásokat, hogy megtapasztaljuk a Fortify kezdő készlet API funkcionalitásainak nézetek nélküli működését a Laravel projektünkben.

A további Fortify funkcionalitásokat (például az e-mail megerősítést vagy a jelszó helyreállítást) már a fentiek alapján önállóan is meg kell tudnunk valósítani a hivatalos dokumentáció segítségével.

10.2.3.3. Automatikus tesztek implementálása és futtatása

Ahogy azt már korábban megismertük (lásd például a 10.2.2.1. alfejezetet), a Jetstream magába foglalja a Fortify funkcionalitásait is, de láthatjuk azt, hogy a Fortify alapértelmezetten nem tartalmaz teszteseteket önmaga funkcionalitásaira vonatkozóan. Emiatt elővehetjük és megnyithatjuk az **l10-auth-jetstream** projektünket és a **tests / Feature** mappából kiválogathatunk olyan funkciókat, teszteset készleteket (10–44. ábra), amelyek a Fortify kapcsán hasznosak lehetnek és nem csak Jetstream specifikusak (mint például a csapatokra vonatkozó tesztesetek).

A tesztelés jelenleg csak a **Feature** és **Unit** mappában lévő egy-egy **ExampleTest** osztály példa tesztelő metódusait tartalmazza. A tesztelés megkezdéséhez a projekt gyökerében lévő phpunit.xml fájl két környezeti változóját (**DB_CONNECTION, DB_DATABASE**) állítsuk be a memóriabeli SQLite-os verzióra, ehhez mindössze a két erre vonatkozó kódsort kell kivennünk a megjegyzés részből, ahogy azt már csináltuk ebben a fejezetben korábban a 10–16. kódrészletben, akkor még a Breeze-es projektben.



10–44. ábra: Jetstream Feature teszteset készletei

10.2.3.3.1. Regisztráció tesztelése

Kezdjük a funkcionalitások tesztelését a regisztrációval, és vizsgáljuk meg a **RegistrationTest.php** fájl tartalmát, miközben átmásoljuk a fájlt a Jetstream-es projektből a Fortify-os projektünkbe. A **RefreshDatabase** trait alkalmazása a teszteset futtatásakor mindig újraépíti a migrációs fájlok alapján a tesztelési adatbázis szerkezetét, így mindenképpen külön adatbázist érdemes használni a teszteléshez, nehogy véletlenül az éles adataink eltűnjenek. A metódusok vizsgálatához először változtatást ne hajtsunk még végre, csak futtassuk le a tesztelést erre a fájlra vonatkozóan:

```
php artisan test tests/Feature/RegistrationTest.php
```

A regisztrációhoz tartozó nézetet pozitív (vagy az angol tesztelés szlengben: *„happy"*) és negatív (vagy *„sad"*) oldalról is teszteli a fájl, attól függően, hogy a regisztrációs funkcionalitás engedélyezve van-e a Fortify beállításai (**config / fortify.php**) között. *Megjegyzés:* a programozók, tesztelők nagyon sokszor csak a pozitív ágat vizsgálják, hogy adott esetben minek kell történnie és úgy történik-e. De a másik ág, a negatív kimenetel vizsgálata legalább ilyen fontos, hogy mi van akkor, ha valamilyen feltételnek nem felelünk meg, akkor minek kell történnie és tényleg az következik-e be.

Attól függően, hogy engedélyezve/tiltva van-e a regisztráció az egyik tesztesetet a kettő közül mindenképpen kihagyja, átugorja a feltételvizsgálat során. Nekünk viszont nincsen regisztrációhoz tartozó

nézetünk, semmilyen hitelesítéshez kapcsolódó nézetünk nincs ebben a projektben, ezt a Fortify beállításai között a **'views' => false** beállítással egyértelműsítettük is a rendszer számára korábban. Ennélfogva a GET **/register** útvonal nem is létezik a rendszerben, amiről meggyőződhetünk, ha lekérjük a regisztrált útvonalak listáját a `php artisan route:list` paranccsal a terminal-ban. A két említett teszteset így nem is tud visszatérni se 200-as, se 404-es állapotkóddal. Ez a két tesztelő metódus így törölhető is a fájlból, mivel mindkettő a regisztrációs nézet meglétét vizsgálja. Így főleg az alkalmazásunk HTTP POST metódussal elérhető útvonal végpontjait tudjuk most automatikus tesztelésekkel ellátni.

Maradt egy tesztelő függvényünk (**test_new_users_can_register()**), ami egy új felhasználó létrehozásának sikerességét teszteli. Ebben a metódusban a négy regisztrációnál elvárt paraméter mellett látható egy **'terms'** attribútum is, de ennek az értéke Jetstream-specifikus, úgyhogy ez a sor törölhető.

Utána a teszteset futtatása már pozitív eredménnyel zárul, sikeresen megtörténhet így a regisztráció, ahogy azt korábban manuális tesztelésnél a Postman-ben is tapasztalhattuk.

Létrehozhatunk egy új tesztelő metódust is, amivel a jelszó megerősítéséhez kapcsolódó validációs hibát ellenőrizhetjük a regisztráció során (ahogy azt jelezte nekünk a Postman is a manuális regisztráció során: 10–43. ábra).

```
public function test_registration_requires_a_password_confirmation(): void
{
  if (!Features::enabled(Features::registration())) {
    $this->markTestSkipped('Registration support is not enabled.');
  }

  $response = $this->post('/register', [
    'name' => 'Test User 2',
    'email' => 'test2@example.com',
    'password' => 'password',
  ]);

  $response->assertSessionHasErrors(['password']);
}
```

10–56. kódrészlet: Automatikus tesztelés a jelszó megerősítés hiányára

Ez alapján gyakorlásként azt is tesztelhetjük másik metódusokkal, hogy üres tartalmú *(„Body")* regisztrációs kéréssel milyen választ kapunk (10–57. kódrészlet: 1. metódus), esetleg érvénytelen e-mail címmel tudunk-e regisztrálni (10–57. kódrészlet: 2. metódus), vagy ugyanazzal az e-mail címmel, ami már létezik az adattáblában lehet-e, szabad-e újra regisztrálni (10–57. kódrészlet: 3. metódus). Elvileg nem szabad, de ezt teszteljük is le!

```
public function test_registration_requires_name_email_and_password()
{
  $response = $this->post('/register', [
    // Üres kérés a regisztrációhoz
  ]);
```

```
  $response->assertSessionHasErrors(['name', 'email', 'password']);
}

public function test_registration_requires_a_valid_email()
{
  $response = $this->post('/register', [
    'name' => 'John Doe',
    'email' => 'invalid-email',
    'password' => 'password',
    'password_confirmation' => 'password',
  ]);

  $response->assertSessionHasErrors(['email']);
}

public function test_registration_fails_if_email_already_exists()
{
  // Először létrehozunk egy felhasználót az adott e-mail címmel
  \App\Models\User::create([
    'name' => 'Existing User',
    'email' => 'existing@example.com',
    'password' => Hash::make('password'),
  ]);
  // Megpróbáljuk újra regisztrálni ugyanazzal az e-mail címmel
  $response = $this->post('/register', [
    'name' => 'New User',
    'email' => 'existing@example.com',
    'password' => 'password',
    'password_confirmation' => 'password',
  ]);

  $response->assertSessionHasErrors(['email']);
}
```

10–57. kódrészlet: További tesztek a regisztrációs esetek automatikus ellenőrzéséhez

A Hash Facade-ot (Illuminate\Support\Facades\Hash) importálni kell a helyes működéshez a fájl elején. A User osztály használatánál az abszolút elérésére hivatkoztunk, így ezt az osztályt nem szükséges importálni!

Megjegyzés: ezt a három új metódust is ki lehet egészíteni a korábbi feltételvizsgálattal, ami ellenőrizte a regisztráció engedélyezett státuszát, de itt most a lényegi részekre koncentráltunk a példakódoknál.

A tesztelés elindítása után azt tapasztaljuk, hogy a regisztrációs eljárásunk mind az öt tesztesete hibátlanul lefut:

```
PASS  Tests\Feature\RegistrationTest
✓ new users can register                                    0.33s
✓ registration requires a password confirmation             0.03s
✓ registration requires name email and password             0.04s
✓ registration requires a valid email                       0.03s
✓ registration fails if email already exists                0.03s

Tests:    5 passed (13 assertions)
Duration: 0.64s
```

10–45. ábra: Sikeresen lefutnak a regisztrációs eljárásnál implementált automatikus teszteseteink

10.2.3.3.2. Bejelentkezés tesztelése

Most megint felhasználhatjuk a Jetstream-es projekt **AuthenticationTest** osztályát, amelyet átmásolhatunk a Fortify-os projektünkbe. Az első metódus (**test_login_screen_can_be_rendered()**) itt ismét a bejelentkezési nézet meglétét teszteli, de ezt törölhetjük, mivel nincsenek hitelesítéshez köthető nézeteink ebben a projektben. A további két metódus a bejelentkezés sikerességét és sikertelenségét teszteli jó, illetve rossz jelszavakkal. Futtassuk ezeket a teszteseteket egy rövidített parancs kiadásával:

```
php artisan test --filter AuthenticationTest
```

A helyes eredmények, pozitív lefutás után az új első metódus nevéből érdemes kivenni a *„using_the_login_screen”* szöveget, mert félrevezető, hiszen nem a bejelentkezési nézeten keresztül lép be ott a felhasználó, hanem csak a POST **/login** útvonal felparaméterezett elérésével.

Adjunk hozzá még két tesztelő metódust, amelyek egy védett útvonal elérését tesztelik különböző végkimenetelek szempontjából. Egy ilyen védett útvonalunk van a projektben: GET **/api/user**. Ennek az elérése látogatóként (nem hitelesített felhasználóként) nem lehetséges és a rendszer átirányít a bejelentkezési oldalra (ha volna ilyen, de ezzel már nem foglalkozunk itt). Hitelesített felhasználót pedig az **actingAs()** segédmetódussal tudunk szimulálni, amellyel már elérhető a védett útvonal.

```php
public function guest_cannot_access_protected_route()
{
  $response = $this->get('/api/user');

  $response->assertRedirect('/login');
}

public function test_authenticated_user_can_access_protected_route()
{
  // Létrehozunk egy teszt felhasználót
  $user = User::factory()->create();

  // Szimuláljuk a bejelentkezést
  $this->actingAs($user);

  $response = $this->get('/api/user');

  $response->assertStatus(200);
}
```

10–58. kódrészlet: Védett útvonal elérésének különböző eseteit teszteljük

Az új teszteseteink is sikeresen lefutnak a tesztelés megindítása után!

Két faktoros hitelesítés teszteléséhez nézzük meg a Jetstream-es projektünk **TwoFactorAuthenticationSettingsTest.php** fájlját és teszteseteit!

10.2.3.3.3. Kijelentkezés tesztelése

A kijelentkezés automatikus tesztelése itt is hasonlóan egyszerű, mint a Postman-ben volt, pusztán el kell érni a POST **/logout** útvonalat és már kijelentkezésre is került a felhasználó. Először a tesztelő függvényben persze a bejelentkezést kell szimulálnunk.

```
public function test_authenticated_user_can_logout()
{
  // Létrehozunk egy teszt felhasználót
  $user = User::factory()->create();

  // Szimuláljuk a bejelentkezést
  $this->actingAs($user);

  // Szimuláljuk a kijelentkezést
  $response = $this->post('/logout');

  // Ellenőrizzük, hogy a felhasználó átirányításra került a főoldalra (vagy
esetleg a bejelentkezési oldalra)
  $response->assertRedirect('/');

  // Ellenőrizzük, hogy a felhasználó kijelentkezett
  $this->assertGuest();
}
```

10–59. kódrészlet: Bejelentkezés utáni kijelentkezés sikerességének ellenőrzése

Az eddig megismerteken kívül még számos teszteset létrehozható és ellenőrizhetünk velük mindenféle eshetőséget, amihez talán ezek a gyakorlati példák jó kiindulási alapnak minősülnek.

Az alfejezetben végrehajtott programkód módosítások ebben a GitHub commit-ben elérhetőek.

Tipp: ha komplex funkcionalitások fejlesztése során nem is feltétlenül, de a tesztesetek definiálásánál nagy segítségünkre tud lenni a ChatGPT. Úgyhogy ne féljünk használni akkor, amikor újabb és újabb automatikus teszt létrehozásával szeretnénk biztosabbá tenni az alkalmazásunk helyes működésébe vetett bizalmunkat. Ezen tesztesetek létrehozása egy idő után rendkívül *„unalmas"* tud lenni, az alkalmazás helyes működése miatt viszont elengedhetetlen. Az ilyen unalmas feladatok létrehozásánál sokat tud segíteni a ChatGPT.

10.2.4. Hitelesítési csomag integrálása meglévő projektbe

Tesztelés szempontjából hajtsuk végre azt, hogy a Fortify csomagot egy már létező és számos funkcionalitást tartalmazó webes projektünkbe telepítjük. Először megvizsgáljuk a projekt esetleges

változásait, aztán különböző teszteseteket definiálunk a webes és API kérések végrehajtási eredményeinek ellenőrzésére.

Az **l10-components** nevű webes projektünket vegyük elő és telepítsük hozzá a Fortify csomagot. Mivel a projektet verziókezeléssel követjük, ezért látni fogjuk, hogy milyen változások hajtódnak végre a mappákban, fájlokban és szerkezeteikben.

Megjegyzés: először a Jetstream csomagot szerettem volna telepíteni, de az annyira elrontotta a projekt sablonját, hogy az rengeteg kiegészítő munkával járt volna, így inkább visszavontam a telepítés okozta változtatásokat és a Fortify csomag használata mellett döntöttem, mivel az nem rontja el az alkalmazásunk sablonját.

Az automatikus tesztelés jelenleg csak a kategóriák tesztelését tartalmazza a projektben, amely több tesztesetnél is elbukik, mivel a **CategoryController** osztály konstruktorához hozzáadtuk az **auth** Middleware ellenőrzését, emiatt az adott útvonal végpontok (**create, store, edit, update, destroy**) elérésénél a rendszer tovább irányítana minket a **login** útvonalra, de ilyen még nincsen a rendszerbe regisztrálva. Továbbá azt is láthatjuk a **routes / web.php** fájlban, hogy a kommentek megtekintése és kezelése is **auth** Middleware által védett útvonalak.

Telepítsük a projektbe a Fortify-t a 10.2.3.1. alfejezet alapján, ez mindössze a **composer** fájlok módosulásával jár, illetve a többi beállítási fájl újként jön létre, tehát nem érint más, már meglévő fájljainkat.

10.2.4.1. Nézetek módosítása a hitelesítés miatt

Kövessük a 10.2.3.2. és a 10.2.3.2.1. alfejezetek leírásait, de itt most a 10–41. kódrészlethez képest egy új, saját bejelentkezési nézetet hozzunk létre, ami illeszkedik az oldalunk megjelenésébe.

```
@extends('app')
@section('main')
<article class="auth">
  <header>
    <div class="title">
      <h1>Login</h1>
    </div>
  </header>
  <x-form
    method="post"
    action="{{ route('login') }}"
  >
    <div>
      <label for="email">Email:</label>
      <input type="email" name="email" id="email" value="{{ old('email')
}}">
      <x-input-error for="email" />
    </div>
    <div>
      <label for="password">Password:</label>
```

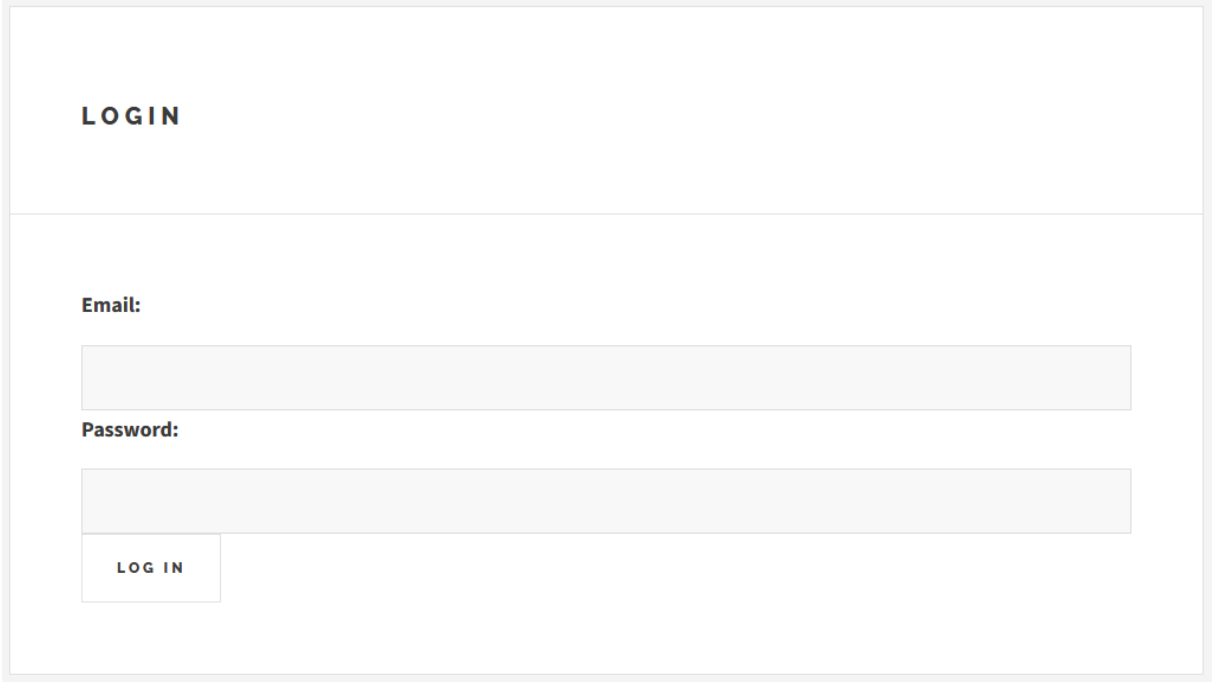
```
      <input type="password" name="password" id="password" value="{{
old('password') }}">
      <x-input-error for="password" />
    </div>
    <div>
      <input type="submit" value="Log in">
    </div>
  </x-form>
</article>
@endsection
```

10–60. kódrészlet: A resources / views / auth / login.blade.php tartalma

A tökéletesen pontos megjelenítéshez még egy apró módosításra van szükség: nyissuk meg a **resources / sass / components / _resource.scss** fájlt és bővítsük a legelső felsorolást egy **.auth** osztállyal:

.post, .category, .tag, .comment, .project, **.auth** {

Ezután már jó lesz a bejelentkezési űrlapunk megjelenítése, ha megpróbáljuk a főmenüben elérni a *„Comments”* menüpontot:



10–46. ábra: Bejelentkezési űrlap az l10-components projektünkben

Tesztelhetjük is már a bejelentkezési űrlap elérését és a validációját, például elküldhetjük üresen az űrlapot, ekkor meg kell kapnunk a szerver oldali validációs hibaüzeneteket.

Fortify-ban bejelentkezés után a hitelesített felhasználót alapértelmezetten a **/home** útvonalra irányítja át a rendszer. Ezt az értéket a **config / fortify.php** beállítási fájlban tudjuk felülírni a **'home'** paraméterben adjuk meg például csak a főoldalat: **/**

Emellett módosítsuk a szerkezet fájlunkat is, hiszen meg kell jeleníteni a bejelentkezés linket, illetve feltételesen szabad csak megjeleníteni a kommentek linket, valamint a kategóriák között a létrehozási és szerkesztési nézetet. De haladjunk csak szép sorban!

A szerkezet (**app.blade.php**) fájl magába foglalja a fejlécben lévő vízszintes menüt (**includes / _header.blade.php**) és a kattintás hatására jobb oldalról beúszó oldalsó menüt (**includes / _menu.blade.php**). Mindkét nézet fájlban használhatjuk a **@auth** vagy a **@guest** Blade direktívákat.

```
{{-- includes / _header.blade.php --}}
@auth
  <li><a href="{{ route('comments.index')}}">Comments</a></li>
@endauth

{{-- includes / _menu.blade.php --}}
@auth
<x-nav-link
  url="{{ route('comments.index') }}"
  title="Comments"
  description="List of comments"
/>
@endauth
```

10–61. kódrészlet: Kommentek menüpontjának eltüntetése csak látogatók számára

A **_header.blade.php** fájlban jelenítsük meg, ha az adott felhasználó be van jelentkezve: a **nav.links > ul** szekció végén hozzunk létre az alábbi sorokat:

```
@auth
  <li>Hi, {{ auth()->user()->name }}!</li>
@endauth
```

10–62. kódrészlet: Bejelentkezés jelzése az oldalon

Maga a köszöntés a fejlécben lévő menüsor végén fog látszódni.

A **_menu.blade.php** nézetben már van egy *„Log in”* feliratú gomb, amelynél használjuk a **@guest** direktívát, mert csak a látogatóinknak szeretnénk megjeleníteni azt, de egyúttal tegyünk be egy kijelentkezési gombot is erre a helyre (mivel ez egy POST-os útvonal végpont, ezért ehhez kell egy űrlap és úgy benne a gomb), ami pedig csak a bejelentkezett felhasználóinknak látható.

```
<ul class="actions stacked">
  <li>
  @guest
    <a href="{{ route('login') }}" class="button large fit">Log In</a>
  @else
    <x-form
      method="post"
      action="{{ route('logout') }}"
    >
      <button class="button large fit">Log Out</button>
    </x-form>
```

```
  @endguest
  </li>
</ul>
```

10–63. kódrészlet: Bejelentkezés és kijelentkezés gombok helyes megjelenítése a nézetben

Viszont, még tesztelni nem tudjuk az alkalmazást, mivel nincs még felhasználónk, ezért a Tinker-be lépjünk be, és hozzunk létre a felhasználógyárunk segítségével egy új felhasználót:

```
\App\Models\User::factory(['email' => 'test@example.com'])->create();
```

10–1. utasítások: Új felhasználó hozzáadása fix e-mail címmel

A létrehozott felhasználóval bejelentkezhetünk: az imént megadott e-mail címmel és a password jelszóval.

A **categories / index.blade.php** fájlban a **create** és **edit** linkeket ugyanígy vegyük körbe az **@auth** direktívával, a **store**, **update** és **destroy** útvonalak úgyis csak ezeken a nézeteken keresztül érhetők el, látható nyomuk azoknak nincsen. Ha pedig valaki manuálisan szeretné elérni ezeket a **create** / **edit** útvonalakat, akkor a rendszer automatikusan a bejelentkezési oldalra fogja őt navigálni.

Az alfejezethez tartozó programkód módosítások ebben a GitHub commit-ben találhatók meg.

10.2.4.2. Tesztek módosítása a hitelesítés miatt

Először vizsgáljuk meg, hogy hány tesztesetünk fut le hibátlanul, és hány nem fut le megfelelően.

```
php artisan test --filter=CategoryTest
```

6 hibás és **3** hibátlan tesztesetet tartalmaz a fájl.

Szerkesszük a **tests / Feature / CategoryTest.php** fájlt úgy, hogy a felhasználói hitelesítést igénylő teszteseteket módosítsuk! A tesztesetek *„kijavítása”* nagyon egyszerűen tud megtörténni. Mindegyik tesztelő metódus *„Given”* részéhez, vagyis ahhoz, hogy mi adott a metódus elején, adjuk hozzá a következő kódsorokat:

```
// Létrehozunk egy teszt felhasználót
$user = User::factory()->create();

// Szimuláljuk a bejelentkezést
$this->actingAs($user);
```

10–64. kódrészlet: Példa felhasználó szimulálása a tesztelő metódusokban

Hiszen arra van szükség, hogy az **auth** Middleware-nek megfeleljenek a kérések és így egy **users** táblában lévő felhasználót szimulálunk, aki a kéréseket végre fogja hajtani.

Ha már hat függvényben is működésre bírtuk az adatgyárat, akkor esetleg érdemes lehet ezt kiszervezni egy privát függvénybe az osztályon belül (amit aztán meghívogatunk), és annak a visszatérési értékébe beletenni az adatgyár működtetését. További kód újreszervezési javaslat lehet, hogy használjuk a tesztelő osztály *„konstruktorát”* (**setUp()** metódus) azért, hogy elég legyen egy helyen létrehozni magát a **$user**-t és utána azt használhatjuk minden egyes metódusban, ahol az **actingAs()** metódus felhasználásra került. Adjuk hozzá ezt a mezőt és metódusokat:

```
private User $user;

protected function setUp() : void
{
  parent::setUp();

  $this->user = $this->createUser();
}

private function createUser() : User
{
  return User::factory()->create();
}
```

10–65. kódrészlet: CategoryTest osztály újraszervezése

Magyarázat: a **parent::setUp()** kódsorral a **TestCase** ősosztály *„konstruktorát"* hívjuk meg először, aztán hajtunk végre saját beállításokat.

Utána a többi függvényben lévő **actingAs($user)** hívásokat módosítsuk ilyenre: **actingAs($this->user)** mivel így már az osztály változóját, a *„konstruktorban"* beállított **$user**-t használhatják a függvények, az azokban lévő felhasználót létrehozó adatgyár működések pedig törlésre vagy kikommentelésre kerülhetnek.

Ezáltal most már mind a 9 teszteset sikeres lefutást fog eredményezni. Ezek viszont mind a pozitív *(„happy")* teszteseteink voltak. Gyakorlásként érdemes megcsinálni a negatív párjukat, vagyis például azt, hogy egy nem hitelesített látogató milyen elvárt HTTP hibakódot kell, hogy visszakapjon.

```
function test_a_visitor_cant_see_the_create_category_page()
{
  $response = $this->get('/categories/create');
  $response->assertStatus(302);
  $response->assertRedirect('login');
}
```

10–66. kódrészlet: A nem hitelesített látogató nem tudja elérni a kategóriát létrehozó oldalt

Továbbá hasonló automatikus teszteseteket is létrehozhatunk a kommentek működésének helyességéhez is!

php artisan make:test CommentTest

Az egyszerű látogatók nem képesek a kommentekhez hozzáférni, megtekinteni őket, létrehozni őket, szerkeszteni és törölni sem tudják őket.

```
public function test_visitors_cannot_access_comments(): void
{
  $response = $this->get('/comments');
  $response->assertStatus(302);
  $response->assertRedirect('login');
}
```

10–67. kódrészlet: Kommentek nem elérhetők a látogatók számára

Utoljára pedig teszteljük azt, hogy bejelentkezés után a főoldalra jut-e a felhasználó, ahogy azt a Fortify beállításai között meghatároztuk.

php artisan make:test AuthTest

```
use RefreshDatabase;

public function test_login_redirects_to_home_page(): void
{
  $user = User::factory([
    'name' => 'Attila',
    'email' => 'attila@example.com'
  ])->create();

  $response = $this->post('/login', [
    'email' => $user->email,
    'password' => 'password'
  ]);

  $response->assertStatus(302);
  $response->assertRedirect('/');
}
```

10–68. kódrészlet: Teszt: bejelentkezés után a főoldalra kerül a felhasználó

Hibát kapnánk, azt jelezné a rendszer, hogy nem létezik a **users** adattábla az SQLite adatbázisunkban, ha nem használnánk az osztályon belül a **RefreshDatabase** trait-et. Futtassuk a tesztet:

php artisan test --filter=AuthTest

Így pozitív eredményt fog adni ez a tesztesetünk és a korábban létrehozottak is (minden zöld).

Az alfejezetben végrehajtott programkód módosítások ebben a GitHub commit-ben találhatók meg.

Tipp: Teszteld a tudásod!

Léteznek nyilvános repo-k, amelyekkel a felhasználói hitelesítés során megszerzett tesztelési ismereteinket is próbára tudjuk tenni, például:

https://github.com/LaravelDaily/Test-Laravel-Auth-Basics

Ha klónozzuk a projektet és elindítjuk a tesztelést, akkor kezdetben minden tesztünk elbukik. Ezt a kódbázist kell úgy átalakítani, hogy hiba nélkül lefussanak, átmenjenek a teszteseteink. További útmutatások, javaslatok a GitHub projekt leírásában érhetők el a könyvtár- és fájlstruktúra alatt.

10.3. Összegzés

A fejezet során először megismerkedtünk a köztes rétegek (Middleware) használatával. Ez a terület, illetve a felhasználásuk mikéntje sokat változott a Laravel 10-hez képest a 11-ben, úgyhogy a verzióváltás miatt eltérően kell kezelni őket. Ezen folyamatoknak részleteit áttekintettük a fejezet első részében.

Utána következett a felhasználói hitelesítés (Authentication) meglehetősen nagy témaköre. Egy webes alkalmazásban legtöbbször alapvetés, hogy valamilyen módon hitelesítsük a felhasználóinkat, akiknek különböző védett funkcionalitásokhoz lesz így hozzáférésük. A felhasználói hitelesítés kapcsán főleg a Laravel által nyújtott vagy 3. féltől származó kezdő készletek (Breeze, Jetstream, Fortify, Sanctum) telepítésével, használatával, testre szabásával és a tesztelésükkel foglalkoztunk és tekintettünk át számos gyakorlati példát. A definiált feladatok között volt webes kérések feldolgozásához tartozó funkciók csoportja, illetve API kérésekhez tartozó funkciók összessége is. Mindkét területen a felhasználói hitelesítés működését mélységében ismertük meg így. Majd számos további automatikus tesztesetet definiáltunk és tekintettünk át a lehetőségekkel együtt.

Legvégül pedig a Fortify kezdő készletet integráltuk a korábban leginkább használt és fejlesztett projektünkbe (l10-components), így kiegészítve azt egy felhasználói hitelesítés funkcionalitással, és annak tesztelésével is.

11. Felhasználói engedélyezések (Authorization)

A felhasználói engedélyezések témakörével mindenképpen a felhasználói hitelesítés után kell foglalkoznunk. Akkor válik *„élessé”* ez a téma, amikor már vannak regisztrált felhasználóink, és nekik szeretnénk meghatározni: engedélyezni, korlátozni vagy éppen tiltani, hogy milyen funkcionalitásokhoz férhetnek hozzá a webes alkalmazásunkon belül.

11.1. Bevezetés egy egyszerű gyakorlati példán keresztül

Azért, hogy ráérezzünk az engedélyezési folyamat működésére, rögtön meg is nézhetünk rá egy nagyon egyszerű gyakorlati példát az **l10-components** projektünkben:

1. Hozzunk létre véletlenszerűen adatgyárak segítségével 10 felhasználót!
2. A blogbejegyzések (**posts**) adattáblához adjunk hozzá egy **author_id** mezőt, amely a **users** tábla **id** mezőjére hivatkozik külső kulcsként!
3. A **Post** és **User** Model osztályokban is definiáljuk a kapcsolatokat: egy szerzőhöz (felhasználóhoz) több blogbejegyzés is tartozhat. A **Post** osztályban adjuk hozzá a **$fillable** mező értékeihez az **author_id**-t is!
4. Töltsük fel a **posts** adattáblát véletlenszerűen a létrehozott felhasználók véletlen azonosítóival.
5. *Opcionálisan:* adjuk hozzá a blogbejegyzések kezeléséhez (**PostController** metódusai és a **resources / views / posts** mappában lévő nézetekhez) a szerzői hozzárendelést és megjelenítést.
    a. **create** metódus és nézet: nem kell hozzáadni a felhasználókat szerzőként, mert mindig az aktuálisan bejelentkezett felhasználó legyen a blogbejegyzés szerzője.
    b. **store** metódusban tároljuk el az **author_id** értékét így: **auth()->id()**
    c. **edit** metódus és nézet: ugyanaz vonatkozik rá, mint a **create**-re.
    d. **update** metódus: ugyanaz vonatkozik rá, mint a **store**-ra.
6. Végül csak annak a felhasználónak engedjük megtekinteni (**show**) az adott blogbejegyzés részleteit, aki a szerzője (author).

A folyamat rövid bemutatása az érdekesebb elemeket részletesebben kiemelve alább olvasható:

Tinker segítségével létrehozhatjuk a felhasználókat:

```
App\Models\User::factory(10)->create();
```

11–1. utasítások: 10 felhasználó létrehozása

Új migrációs fájl létrehozása a **posts** táblához:

```
php artisan make:migration add_author_id_to_posts_table
```

Idegen kulcs létrehozása/eltávolítása a migrációs fájl **up()** és **down()** metódusaiban:

```
public function up(): void
{
  Schema::table('posts', function (Blueprint $table) {
```

```
    $table->foreignId('author_id')->nullable()->constrained('users')-
>onUpdate('cascade')->onDelete('cascade');
  });
}

public function down(): void
{
  Schema::table('posts', function (Blueprint $table) {
    if (DB::getDriverName() !== 'sqlite') {
      $table->dropForeign('posts_author_id_foreign');
    }
    $table->dropColumn('author_id');
  });
}
```

11–1. kódrészlet: Blogbejegyzés szerzőire vonatkozó idegen kulcs létrehozása/törlése a migrációs fájlban

Az **author_id** mezőt nullázhatóvá tettük, mivel már a **posts** táblánkban vannak adatok, és ezen módosító nélkül nem működne az idegen kulcs kényszer hozzáadása. Migráljunk:

```
php artisan migrate
```

Post Model osztály kapcsolata itt látható (de a **$fillable**-t se felejtsük el bővíteni ezzel a mezővel!):

```
public function author() {
  return $this->belongsTo(User::class, 'author_id');
}
```

11–2. kódrészlet: Post Model osztályban lévő kapcsolat a szerzők (felhasználók) felé

User Model osztályba másolhatjuk ugyanazt a kapcsolati metódust, ami a **Category** osztályban van: **posts()**

Frissítsük fel a létező, nem törölt blogbejegyzéseket véletlenszerűen kiválasztott szerzői azonosítókkal a Tinker segítségével:

```
$user_ids = App\Models\User::get()->pluck('id');

$posts = App\Models\Post::get();

$posts->each(function ($post) use ($user_ids) { $post->update(['author_id' => $user_ids->random()]); });
```

11–2. utasítások: Létező blogbejegyzésekhez szerzői azonosítók véletlenszerű beállítása

Az 5. opcionális pont megvalósítását az Olvasóra bízom, itt most folytassuk a munkát a 6. ponttal: a **PostController** osztály **show()** metódusát kell módosítani, amelyben ténylegesen megvalósítjuk az engedélyezési eljárást, de ha más helyen is szükségünk lenne rá, akkor már most érdemes ezt az üzleti logikai rétegben, vagyis a **User** Model osztályban implementálni egy tagfüggvényben:

```
public function owns($post) : bool {
  return $post->author_id == auth()->id();
}
```

11–3. kódrészlet: Ellenőrzi, hogy a felhasználó birtokolja-e az adott blogbejegyzést

Az alábbi kódsorokat szúrjuk be a **PostController show()** metódusába a **return** utasítás elé! Ezek pedig mind egyenértékűek (ekvivalensek) egymással, viszont az a leginkább olvasható (álljon meg a végrehajtás akkor, ha a bejelentkezett felhasználó nem birtokolja szerzőként az adott blogbejegyzést), amit utoljára megjegyzés nélkül hagyunk:

```
// abort_if($post->owner_id !== auth()->id(), 403);
// abort_if(!auth()->user()->owns($post), 403);
abort_unless(auth()->user()->owns($post), 403);
```

11–4. kódrészlet: Hozzáférés engedélyének ellenőrzése többféle módon a PostController show() metódusában

Így a bejelentkezés után az adott felhasználó csak azt a blogbejegyzést tudja megtekinteni, amelynek ő a szerzője (**posts** adattábla **author_id** mezője az adott sorban a bejelentkezett felhasználó azonosítója).

Ez azonban még csak egy nagyon leegyszerűsített megoldása az engedélyezési eljárásnak, viszont jól látható belőle, hogy hogyan kell működnie.

Az alfejezethez tartozó programkód módosítások ebben a GitHub commit-ben találhatók meg.

11.2. Felhasználói engedélyezési technikák a Laravel-ben

A beépített hitelesítési szolgáltatások (kezdő készletek) mellett a Laravel több módot is biztosít a felhasználói műveletek engedélyezésére egy adott erőforrás menedzselése során. Előfordulhat például, hogy hiába került hitelesítésre az adott felhasználó, de nem jogosult mégsem bizonyos erőforrásokkal kapcsolatos funkcionalitások végrehajtására, úgy, mint például a frissítés vagy a törlés. A Laravel erre a folyamatra (az engedélyezések ellenőrzésére) is több egyszerű, de szervezett módot biztosít a számunkra.

A Laravel két módot biztosít a műveletek engedélyezésére, ezek a kapuk, átjárók (Gate) és a házirendek (Policy) – az angol kifejezések meglehetősen elterjedtek, jól érthetőek, így a továbbiakban ezeket használjuk a magyarra fordított változatuk helyett. Nem kell ezek közül választanunk, akár kevert módon is használhatjuk őket, sőt, ez az együttes használat a leginkább elterjedt mód. Ha különbséget kellene tenni köztük, akkor a Gate-ek leginkább olyan műveletekre alkalmazhatók, amelyek nem kapcsolódnak semmilyen Model osztályhoz vagy erőforráshoz, ilyen művelet lehet például az adminisztrátori vezérlőpult megtekintése. Ezzel szemben a Policy-ket akkor kell használni, ha valamilyen műveletet engedélyezni (korlátozni vagy tiltani) szeretnénk szorosan a modellek vagy erőforrások kapcsán. A Gate-ek remek lehetőséget nyújtanak a Laravel engedélyezési funkcionalitás alapjainak megismeréséhez és megértéséhez, de egy komplexebb, robosztusabb alkalmazásnál sokkal inkább elterjedtebb a Policy-k használata az engedélyezési eljárásoknál, vagy még inkább egy kevert használat.

Mindkét eszközt az **app / Providers / AuthServiceProvider.php** fájlban kell beregisztrálni a rendszerbe. A fájl mutat is nekünk példát, és magyarázatokat helyeztek el a kommentekben a jobb megértés érdekében.

```php
class AuthServiceProvider extends ServiceProvider
{
    /**
     * The model to policy mappings for the application.
     *
     * @var array<class-string, class-string>
     */
    0 references
    protected $policies = [
        // 'App\Models\Model' => 'App\Policies\ModelPolicy',
    ];

    /**
     * Register any authentication / authorization services.
     */
    0 references | 0 overrides
    public function boot(): void
    {
        //
    }
}
```

11–1. ábra: AuthServiceProvider az engedélyezési folyamat kiinduló pontja

Az iménti ábrán látható, hogy a Policy osztályok regisztrálásának helye a **$policies** tömbben van, míg a **boot()** metódus szolgál arra, hogy a Gate-eket definiáljuk.

11.2.1. Engedélyezési technikák alkalmazása Gate eszközzel

A Gate-ek használatát Facade-okkal valósítjuk meg. Ilyet korábban használtuk már, például a DB is ilyen volt, amely adatbázisműveletek végrehajtásában nyújtott nekünk egy eszközkészletet, metódusokat, tehát segített minket, hatékonyabbá és egyszerűbbé tette a munkánkat. A Gate Facade is hasznos, mert metódusokat biztosít a Gate-ek definiálásához és ellenőrzéséhez. Így például létrehozhatunk egy olyan átjárót (kaput) az ellenőrzéshez, ami például csak adminisztrátor szerepkörű felhasználót enged hozzáférni bizonyos oldalakhoz vagy funkciókhoz.

Nézzük is meg a gyakorlatban, hogy hogyan lehet Gate-eket definiálni, ellenőrizni és paraméterekkel ellátni őket stb.

A Gate-eket a következő esetekben érdemes használni:

- *Erőforrás-birtoklás esetén:* ekkor használható az erőforrásokhoz való hozzáférés korlátozására a tulajdonjog alapján. Például korlátozhatja a hozzáférést egy felhasználó saját profiloldalához, vagy korlátozhatja egy adott bejegyzés szerkesztését csak arra a felhasználóra, aki létrehozta azt. Ilyenre láthattunk példát az előző 11.1. alfejezetben, de akkor még a Gate-ek használata nélkül.
- *Speciális üzleti logikai réteg kialakításakor:* a Model osztályokban lehet ellenőrizni, egy funkció kapcsán, hogy például egy könyvtári tag hány könyvet kölcsönözhet ki egyidejűleg, vagy rendelkezik-e az adott tagsági, előfizetői szinttel egy oktatási weboldalon.
- *Szerepkör alapú hozzáférés megvalósításakor:* az eddigiekből is kitűnhetett, hogy a Gate-ek segítségével a webalkalmazás bizonyos funkcióihoz, részeihez való hozzáférés korlátozható.

Például egy webshop adminisztrációs felületéhez csak az eladónak vagy a fő adminisztrátor szerepkörű (csoport) tagoknak van lehetőségük hozzáférni, a többieknek (egyszerű vásárlóknak) nincs. Ilyen megoldásokra látunk majd példát a 11.3. alfejezetben.

Gyakorlat szempontjából maradjunk az **l10-components** projektünknél, és itt hajtsuk majd végre a szükséges módosításokat!

11.2.1.1. Gate működési hátterének előkészítése

Módosítsuk a **users** adattáblánkat, adjunk hozzá egy **isAdmin** mezőt!

```
php artisan make:migration add_isAdmin_to_users_table
```

Az **up()** metódusban adjuk hozzá ezt az **isAdmin** mezőt **boolean** értékként és alapértelmezetten (**default()**) állítsuk be **false** értékre. A **down()** metódusban csak töröljük (**dropColumn()**) ki ezt az új mezőt. Mentés után migráljunk!

Ezzel párhuzamosan a **User** Model osztályunkat is készítsük fel ennek a mezőnek a kezelésére: a **$fillable** tömbhöz adjuk hozzá ezt az új **isAdmin** mezőt.

Hozzunk létre egy új felhasználót, és tegyük egyből adminisztrátorrá például a Tinker segítségével:

```
App\Models\User::factory(['email' => 'admin@example.com', 'isAdmin' => 1])->create();
```

11–3. utasítások: Új adminisztrátor felhasználó létrehozása

Az utasításban az e-mail cím nem is érdekes, csak válasszunk egy könnyen megjegyezhetőt. A jelszó alapértelmezetten password lesz. Az **isAdmin** mező értéke pedig a MySQL-ben a **true**-t az 1-gyel kell jelölni, míg a **false**-ot a 0-val.

11.2.1.2. Gate definiálása

A példa legyen szerepkör alapú: bizonyos oldalakhoz csak az adminisztrátor (admin) férhet hozzá.

Ezután definiálhatjuk a Gate-et az **app / Providers / AuthServiceProvider** osztály **boot()** metódusában:

```php
Gate::define('admin', function (User $user) {
  return $user->isAdmin;
});
```

11–5. kódrészlet: Gate definiálása az adminisztrációs funkcionalitások eléréséhez

A **Gate** Facade-ot és a **User** Model osztályt importálni kell a fájl elején! A **define()** metódus két paramétert vár el, az első a Gate neve, amire majd lehet hivatkozni, a második paramétere egy closure, egy helyben definiált névtelen metódus, amely a leggyakrabban a **User** Model osztály egy példányát kapja meg első paraméteréül. Ez képviseli azt a felhasználót, aki hozzá akar majd férni bizonyos funkcionalitásokhoz az oldalon. Ha ennek a névtelen metódusnak a visszatérési értéke igaz lesz, akkor a hozzáférés engedélyezve lesz, ha pedig hamis, akkor tiltásra kerül a hozzáférés az adott felhasználó részéről.

A definiált Gate-et innentől kezdve több helyen is felhasználhatjuk.

11.2.1.3. Gate felhasználása útvonalaknál

Kezdetben rendeljük hozzá egy nagyon egyszerű útvonalhoz a **routes / web.php** fájlban:

```
Route::get('/admin-panel', function () {
  if (Gate::allows('admin')) {
    return "Admin Panel";
  }
  abort(403);
});
```

11–6. kódrészlet: Admin Gate engedély ellenőrzése útvonalnál

Az **allows()** metódust adja nekünk a Gate Facade (ezt importáljuk is a fájl tetején), amellyel könnyedén ellenőrizhetjük, hogy a hitelesített (bejelentkezett) felhasználó rendelkezik-e ezzel a jogkörrel az **isAdmin** mezőjén keresztül.

Az **allows()** tagadása megoldható lenne előtte a **!** használatával is, de olvashatóbb, ha a Gate Facade által nyújtott **denies()** segédmetódust alkalmazzuk helyette.

11.2.1.4. Gate felhasználása Controller-ben

Bár említésre került, hogy a Gate definiálásánál az adott felhasználót adjuk át neki, de előfordulhat, hogy egy erőforrást is megadunk neki, és így például meg tudjuk valósítani, hogy ne csak az adminisztrátorok tudják szerkeszteni az adott blogbejegyzést, hanem az is, aki nem biztos, hogy adminisztrátor, de a szerzője volt.

```
Gate::define('update-post', function (User $user, Post $post) {
  return $user->isAdmin || $user->id === $post->author_id;
});
```

11–7. kódrészlet: Gate definiálása erőforrással való művelethez (adminisztrátor vagy szerző szerkesztheti a blogbejegyzést)

Ennek a Gate-nek az ellenőrzését elhelyezhetjük a **PostController update()** metódusának elején, elég csak az adott blogbejegyzés példányt átadni az **allows()** metódusnak, a bejelentkezett felhasználót alapértelmezetten megkapja.

```
public function update(UpdatePostRequest $request, Post $post)
{
  if(Gate::allows('update-post', $post)) {
    DB::transaction(function () use ($post, $request) {
      $post->update($request->safe()->only(['title', 'slug', 'body',
'category_id', 'published_at']));

      $post->tags()->sync($request->tags);

      Rating::where('post_id', $post->id)->update(['score' => $request-
>score]);
    });

    return redirect(route('posts.index'));
  }
```

```
    abort(403);
}
```

11–8. kódrészlet: Engedélyvizsgálat az adott blogbejegyzés frissítéséhez

Ha sok erőforrásunk van, akkor a Gate-ek használata kényelmetlen lehet, és sokkal hatékonyabb, ha Policy-ket alkalmazunk ilyenkor helyette (11.2.2. alfejezet).

11.2.1.5. Gate felhasználása a nézetben

A nézetekben lehetőségünk van már a szerkesztési linket is elrejteni azért, hogy ne tudjon rákattintani az a felhasználó, akinek nincsen joga bizonyos bejegyzés frissítéséhez.

A nézetekben is ugyanúgy használhatók a Gate **allows()** és **denies()** metódusai egy **@if-@endif** Blade direktíva párosban. A Blade sablon motor viszont itt az engedélyeztetés témakörénél is szolgáltat a számunkra egy egyszerűsítést.

Nyissuk meg a blogbejegyzések **index** nézetét, amely a szerkesztés linkeket is tartalmazza. Alapból, ha ránéz az oldalra egy felhasználó, azt gondolhatja, hogy bármelyik blogbejegyzést tudja frissíteni, azonban ez most már nincs így. Tegyük ezt láthatóvá a nézetben is, vagyis tüntessük el azokat a szerkesztési linkeket, amelyekhez az adott felhasználónak nincsen frissítési joga.

```
@can('update-post', $post)
    <a href="{{ route('posts.edit', $post->id) }}">Edit</a>
@endcan
```

11–9. kódrészlet: Blogbejegyzés frissítésének engedélyét vizsgáljuk a posts.index nézetben

Az engedély ellenőrzésének hatására egy látogató (nem hitelesített felhasználó) számára eltűnik az összes szerkesztési link a blogbejegyzéseknél.

Title	No. of tags	No. of comments	Operations
Test: storing a post	3	0	
Test: updating a post	0	0	
ad	2	0	
Voluptatem qui.	0	0	

11–2. ábra: Látogatók számára nincs frissítési engedély (és így szerkesztési link sem) a blogbejegyzéseknél

Alapértelmezés szerint minden Gate (és Policy is) automatikusan **false** értékkel tér vissza, ha a bejövő kérés egy nem-hitelesített felhasználótól érkezett (tehát látogatótól jött). *Megjegyzés:* lehetőségünk van azért arra is, hogy a látogatóktól érkező kéréseket is átengedjük, ehhez érdemes ezt a példát megtekinteni.

Egy bejelentkezett felhasználónál, aki létrehozta az adott blogbejegyzést, látható lesz a szerkesztési link, és tudja is frissíteni a blogbejegyzés részleteit, a többi blogbejegyzést azonban nem tudja frissíteni, és így alapból a szerkesztési link sem látszódik nála azokhoz.

Title	No. of tags	No. of comments	Operations
Test: storing a post	3	0	
Test: updating a post	0	0	Edit
ad	2	0	Edit
Voluptatem qui.	0	0	

11–3. ábra: Bejelentkezett szerzők a saját blogbejegyzéseiket frissíthetik csak (azokhoz van szerkesztési link is)

Egy adminisztrátor felhasználó tudja szerkeszteni és frissíteni is az összes blogbejegyzést.

Title	No. of tags	No. of comments	Operations
Test: storing a post	3	0	Edit
Test: updating a post	0	0	Edit
ad	2	0	Edit
Voluptatem qui.	0	0	Edit

11–4. ábra: Adminisztrátor engedéllyel bíró felhasználó minden blogbejegyzést frissíthet (így szerkeszthet is)

Mint az **@if**-**@endif** szerkezetben, itt is lehet az elágazásnál másik ágat definiálni az **@elsecan** és **@else** direktívákkal, illetve a **@can** helyett a **@cannot**-ot is használhatjuk (ekkor persze **@elsecannot** ág és **@endcannot** záró direktív áll rendelkezésünkre), ha rögtön a negáltat akarjuk ellenőrizni a nézetben, nem pedig a pozitív ágat.

Ezeknek a direktíváknak a használata persze még nem akadályozza meg a jogosulatlan felhasználót abban, hogy a linket esetleg manuálisan összerakva (böngészőbe beírhatja ezt például: **/posts/1/edit**) el tudja érni a blogbejegyzéshez tartozó szerkesztési űrlapot, de ha ezt a kiskaput is be szeretnénk zárni előtte (érdemes!), akkor a **PostController edit()** metódusában lehet definiálni ezt, ahogy tettük az **update()** metódusban is (11–8. kódrészlet).

11.2.1.6. Gate testreszabása egyedi hibaüzenettel

Ha saját, egyedi hibaüzenetet szeretnénk hozzáfűzni ahhoz az esethez, amikor elbukik az engedélyeztetési folyamat, akkor a definiáláshoz kell visszatérnünk, vagyis az **AuthServiceProvider boot()** metódusában létrehozott Gate-hez.

```
Gate::define('update-post', function (User $user, Post $post) {
```

```
  return $user->isAdmin || $user->id === $post->author_id
    ? Response::allow()
    : Response::deny('You must be the post\'s author or an adminstrator.');
});
```

11–10. kódrészlet: Gate definiálása hibaüzenettel kiegészítve

A fájl elején importáljuk a **Response** eléréséhez az **Illuminate\Auth\Access\Response** osztályt.

A hibaüzenet azt jelzi majd vissza a látogatónak/felhasználónak, aki esetleg olyan blogbejegyzést szeretne frissíteni, amihez nincs engedélye, hogy nem ő a blogbejegyzés szerzője és nem is adminisztrátor, úgyhogy nincsen a frissítésre lehetősége.

A hibaüzenet megjelenítéséhez azonban módosítanunk kell a **PostController update()** metódusát is (ha az **edit()**-et is megcsináltuk gyakorlásként, akkor természetesen azt is módosítsuk eszerint).

```
public function update(UpdatePostRequest $request, Post $post)
{
  $response = Gate::inspect('update-post', $post);

  if ($response->allowed()) {
    DB::transaction(function () use ($post, $request) {
      $post->update($request->safe()->only(['title', 'slug', 'body',
'category_id', 'published_at']));

      $post->tags()->sync($request->tags);

      Rating::where('post_id', $post->id)->update(['score' => $request-
>score]);
    });

    return redirect(route('posts.index'));
  }
  else {
    echo $response->message();
  }
  // abort(403);
}
```

11–11. kódrészlet: Engedélyvizsgálat az adott blogbejegyzés frissítéséhez egyedi hibaüzenettel

A hibaüzenetet természetesen többnyelvűsíthetjük is, ha követjük a 15.3. alfejezet útmutatásait.

11.2.1.7. Egy engedély mindenekelőtt…

Előfordulhat, hogy bizonyos engedély meglétét ki szeretnénk emelni a többi közül, és azt leellenőrizni minden más egyéb engedélyvizsgálat előtt. Ilyen lehet például, hogy az adott felhasználó adminisztrátor-e. Emeljük ki ezt a már definiált Gate-ünkből (11–10. kódrészlet), és használjuk az ellenőrzést a **before()** segédmetódusban (egyúttal vegyük ki a 11–10. kódrészletben lévő **return** utasítás első feltételét).

```
Gate::before(function (User $user) {
  if ($user->isAdmin) {
```

```
    return true;
  }
});
```

11–12. kódrészlet: Minden más engedélyezést megelőző vizsgálat definiálása

Fontos: más visszatérési értéket itt ne használjunk, csak a **true**-t! Ha ez az igazzal tér vissza, akkor már meg is van az engedély az adott további művelet végrehajtására. Ha viszont ez nem tér vissza semmivel, akkor még mindig van lehetőség arra, hogy a további engedélyek ellenőrzésének megfeleljen az adott kérés (felhasználó).

Ekkor csak az *„elbukó"* (nem igaz értékkel visszatérő) adminisztrátori vizsgálat után fogja ellenőrizni a rendszer a többi engedély megfelelését. Ha a felhasználó adminisztrátor, akkor már meg is van az engedélye, és tevékenykedhet bármit a jelenlegi alkalmazásunk keretében.

Előfordulhat az is, hogy az engedélyek ellenőrzése után szeretnénk még valamit ellenőrizni, de ez sokkal-sokkal ritkábban történik meg a valóságban. Erre a műveletre a Gate **after()** segédmetódusa lehet a segítségünkre ugyanúgy, ahogy azt a **before()**-nál láthattuk.

Az alfejezetben végrehajtott programkód módosítások ebben a GitHub commit-ben találhatók meg.

11.2.2. Engedélyezési technikák alkalmazása Policy eszközzel

A Policy osztályok az engedélyezési logikát egy adott modell vagy erőforrás köré szervezik. Például, ha az alkalmazás egy blog, mint ugye a mi esetünkben is, akkor lehet egy **Category** Model osztály és egy vele együttműködő **CategoryPolicy** osztály a felhasználói műveletek engedélyeihez, például a blogbejegyzések kategóriáinak létrehozásához, frissítéséhez vagy törléséhez.

A Policy osztályt a `php artisan make:policy` parancs segítségével hozhatjuk létre. A generált Policy osztály az **app / Policies** könyvtárba kerül majd be. Ha ez a könyvtár nem létezik az alkalmazásban, a Laravel létrehozza, majd elhelyezi benne a fájlt. Alapértelmezetten egy kvázi üres **CategoryPolicy** osztály jönne létre, de ha szeretnénk, hogy az erőforráshoz kapcsolódóan rögtön tartalmazza a megtekintési, létrehozási, frissítési, törlés műveleteket is, akkor a `--model` kapcsoló segítségével adjuk meg neki a hozzá kapcsolódó Model osztályt is. Adjuk ki ezt az utasítást:

```
php artisan make:policy CategoryPolicy --model=Category
```

A Policy osztályokat nem feltétlenül elég létrehozni, de regisztrálni is lehet őket, hasonlóan, mint ahogy a Gate-eket is megtettük. A regisztrációt ugyanabban a fájlban az **app / Providers / AuthServiceProvider.php**-ban tehetjük meg, a **$providers** tömbhöz kell hozzáadni új elemet, ahogy az ott elhelyezett kikommentezett példa mutatja is nekünk. Az asszociatív tömb kulcs-érték párjainál a kulcs a Model osztály, az érték pedig a Policy osztály.

```
protected $policies = [
  // 'App\Models\Model' => 'App\Policies\ModelPolicy',
  Category::class => CategoryPolicy::class,
];
```

11–13. kódrészlet: CategoryPolicy osztály regisztrálása a rendszerbe

Mindkét osztály importálására szükség van, ha csak ilyen röviden akarunk itt rájuk hivatkozni.

Megjegyzés: a keretrendszerben van már automatikus Model-Policy osztály páros felderítés, így akár a manuális regisztráció nélkül is tudja a Laravel, hogy melyik Model-hez melyik Policy-t akarjuk alkalmazni, de ehhez természetesen be kell tartanunk a névkonvenciókat, és hogy melyik mappában helyezzük el a fájlokat – ha artisan parancsokat használunk a létrehozásokhoz, akkor úgyis megfelelően fogja elhelyezni őket.

Mivel a Policy osztályok leginkább a Model-ekhez kapcsolódó funkcionalitások engedélyezésére/tiltására használhatók, ezért érdemes áttekinteni, hogy hogyan, milyen Policy metódusok képesek – például a RESTful Controller-ekben – az ott lévő funkciók *„elérhetőségét"* befolyásolni. Mivel a **CategoryPolicy** osztályunkat a **Category** Model-hez hoztuk létre, ezért érdemes áttekinteni, hogy milyen metódusokat tartalmaz a hozzáférési engedélyek meghatározásásra a Controller-ek metódusaival összefüggésben.

Controller metódusok	Policy osztály metódusai
index	viewAny
show	view
create	create
store	create
edit	update
update	update
destroy	delete

11–1. táblázat: Funkciók, akciók összerendelése a Policy metódusokkal

Amikor a felhasználói kéréseket az adott Controller metódushoz irányítjuk, akkor a neki megfelelő Policy metódus automatikusan meghívásra kerül a Controller metódus végrehajtása előtt.

Jelenleg még a **CategoryPolicy** osztályban lévő összes metódus visszatérési értéke hiányzik, miközben a metódusoknál jelölésre került, hogy bool típussal fognak visszatérni. Úgyhogy vagy definiálunk valamilyen bool típusú visszatérési értéket (egy feltételvizsgálattal például, mint ahogy a Gate-ek definiálásánál tettük), vagy töröljük az adott Policy metódust. Utóbbit érdemes végig gondolni, mivel a rendszer – attól függetlenül, hogy a **Category** elemeink *„soft delete"* tulajdonsággal rendelkeznének-e – hozzáadott **restore()** és **forceDelete()** metódusokat a **CategoryPolicy** osztályhoz. A kategóriáink nem rendelkeznek *„soft delete"* funkcióval, így ez utóbbi két metódus törölhető / kikommentelhető a **CategoryPolicy** osztályban.

A többi metódus visszatérési értékének beállítását bontsuk ketté:

1. *viewAny* és *view*: amelyek tehát az **index** és **show** erőforrás útvonalakhoz és Controller metódusokhoz tartoznak, ezeknél állítsunk be egyszerűen igaz értéket, hogy mindenki láthassa a kategóriákat. Tehát a metódusok magja legyen ez:
    a. **return true;**

   b. ezekhez az útvonalakhoz/metódusokhoz eddig nem kellett felhasználói hitelesítés, így viszont már automatikusan tartozni fog hozzájuk egy hitelesítési elvárás a megtekintések engedélyezéséhez, hiába térünk vissza igaz értékkel a magjukban, a látogatók nem fogják tudni tallózni a kategóriákat. Ha mégis engedni szeretnénk nekik akkor kövessük végig a következő alfejezetben felsorolt lehetőségeket.
2. *create, update, delete:* állítsuk be ezeknél (hasonlóan, mint korábban a Gate-eknél megtettük), hogy az adott felhasználónak adminisztrátornak kell lennie, és csak akkor érheti el a kategóriák létrehozó, módosító és törlő funkcionalitásait. Tehát a metódusok magja legyen ez:
   a. **return $user->isAdmin;**

11.2.2.1. Engedélyezés a Controller-ben

Az engedélyezést több módon is tudjuk menedzselni a Controller-ekben. Az első ilyen módszer az, amikor a **User** Model osztályon keresztül használjuk az alapértelmezetten biztosított **can()** és **cannot()** metódusokat. Ezek nagyon hasonlóan működnek, mint korábban a Gate-ről szóló fejezet nézeteket és engedélyezési Blade direktívákat (11.2.1.5. alfejezet) tartalmazó részében láthattuk. A Policy metódusoknak megfelelő engedélyeket manuálisan is felhasználhatjuk az adott Controller metódus kezdetén, például így **update()** metódus magjának legelején:

```
if ($request->user()->cannot('update', $category)) {
  abort(403);
}
```

11–14. kódrészlet: Policy metódus által nyújtott ellenőrzés meghívása a Controller vezérlő metódusában

Ezután már mehet is a kategória frissítése az **update()** metóduson belül. *Megjegyzés:* a **can()** és **cannot()** második paramétere az **update()** esetén a Model osztály egy példánya volt (**$category**), de ha például a **create** Policy metódusra hivatkozunk, akkor ott (például a **CategoryController store()** metódus magjának legelején) még nincsen ilyen példány, ezért ott a **Category::class** használható.

Ha van az adott Model osztályhoz Policy osztály regisztrálva, akkor a **can()** vagy a **cannot()**, metódus automatikusan meghívja a megfelelő engedélyezési szabályzatot (Policy metódust).

A másik módszer, a **User** Model osztály alkalmazásán túl az lehet, ha a Controller **authorize()** segédmetódusát használjuk. Ez használatra ugyanúgy működik, mint a **User**-nél a **can()** vagy a **cannot()**, tehát első paraméterében a Policy osztály metódusának nevét kell írni, a másodikba pedig vagy a Model osztály nevét (**create** Policy esetén), vagy pedig a Model osztály egy példányát (**update** és **delete** Policy esetén). Így le lehet rövidíteni a 11–14. kódrészletben látott megoldást erre a **CategoryController update()** metódusán belül:

```
$this->authorize('update', $category);
```

11–15. kódrészlet: Controller segédmetódusa az engedély meglétének ellenőrzésére

Itt is igaz az, hogy ha összerendeltük a Model és Policy osztályokat, akkor az **authorize()** metódus is automatikusan meghívja az ide az **update()** Controller metódusba szükséges Policy metódus engedélyezési szabályát.

Ha pedig úgynevezett erőforrás (resource) Controller-ünk van, amit létrehozáskor az -r kapcsolóval hoztunk létre, és tartalmazza a 7 RESTful vezérlő metódust, akkor az automatikus metódus összerendeléseket a Controller konstruktorában is tudjuk definiálni, és nincs szükség arra, hogy a Controller metódusokban egyesével elvégezzük a **can(), cannot()** vagy **authorize()** vizsgálatokat. Tegyük ezt meg a **CategoryController** konstruktorában:

```
public function __construct() {
  // $this->middleware('auth')->except('index', 'show');
  $this->authorizeResource(Category::class, 'category');
}
```

11–16. kódrészlet: Engedélyezési eljárások hozzárendelése a 7 RESTful Controller metódushoz

Az ott meglévő **auth** Middleware így kiváltható, hiszen ennek a **CategoryPolicy** osztálynak az alkalmazása most szigorúbb engedélyezési szabályokat fogalmaz meg, mintsem az egyszerű felhasználói hitelesítésnek a megkövetelése a **create(), store(), edit(), update(), destroy()** metódusoknál. Az **index()** és **show()** metódusoknál a **CategoryPolicy** miatt így viszont már elvárás az, hogy a felhasználó be legyen jelentkezve, anélkül nem engedi megtekinteni őket a rendszer. Ha úgy döntünk, hogy a lista és megtekintés funkciókat szeretnénk engedélyezni a látogatók számára is, akkor a **CategoryPolicy** osztály **viewAny()** és **view()** metódusának paramétereit írjuk át eszerint:

```
public function viewAny(?User $user): bool
{
  return true;
}

public function view(?User $user, Category $category): bool
{
  return true;
}
```

11–17. kódrészlet: Látogatóknak is engedjük a kategóriákat tallózni

Ennek a fenti igénynek az eléréséhez mindössze arra volt szükség, amit már korábban a Gate-ek kapcsán is megjegyeztünk, hogy a Policy metódusok paramétereiben a **User** osztály példányát opcionálissá tettük úgy, hogy eléjére egy kérdőjelet tettünk.

Kiegészítésként annyit még érdemes megcsinálni a **categories.index** nézetben, hogy az ott használt **@auth-@endauth** direktíva párost átírjuk **@can-@endcan** párosra, így csak azoknak kínálja fel az alkalmazás a kategóriák szerkesztését, akik valóban jogosultak is rá a Policy alapján teljesítik az ottani **update()** metódusban definiált feltételt, tehát adminisztrátorok.

```
<td>
  @can('update', $category)
    <a href="{{ route('categories.edit', $category->id) }}">Edit</a>
  @endcan
</td>
```

11–18. kódrészlet: Szerkesztést csak annak kínálja fel az alkalmazás, aki frissítésre jogosult

Az alfejezetben végrehajtott programkód módosítások ebben a GitHub commit-ben találhatók meg.

11.2.2.2. Engedélyezés Middleware alkalmazásával

Az útvonalaknál is tudunk engedélyezést alkalmazni a **can** Middleware segítségével (a háttérben ez az egyszerű név az **Illuminate\Auth\Middleware\Authorize** Middleware-t takarja). Az útvonalakhoz tartozó **can** Middleware már azelőtt ellenőrzi az engedély meglétét, mielőtt a felhasználói kérés kiszolgálása eljutna a Controller megfelelő metódusához vagy akár a konstruktorához.

Gyakorlati szempontból még mindig maradjunk az **l10-components** projektünknél, viszont most már a kommentekkel foglalkozzunk. Változtassuk meg ezt az engedélyezési eljárást úgy, hogy nem elég a kommentek listázásához, tallózásához, létrehozásához, szerkesztéséhez, törléséhez a bejelentkezés megléte egy felhasználónál, hanem annak adminisztrátornak is kell lennie.

Hozzuk létre a **CommentPolicy** osztályt a **Comment** Model osztályhoz:

```
php artisan make:policy CommentPolicy --model=Comment
```

A **CommentPolicy** minden metódusának visszatérési értékét állítsuk be a **CategoryPolicy** osztály **create()** metódusának visszatérése alapján (**return $user->isAdmin;**), így csak az adminisztrátor felhasználók fognak tudni hozzáférni a kommentek összes funkciójához:

Megjegyzés: a **CommentPolicy restore()** és **forceDelete()** metódusai itt is törölhetők.

Regisztráljuk be az új **CommentPolicy** osztályt a rendszerbe az **AuthServiceProvider $policies** tömbjének bővítésével:

```
protected $policies = [
  // 'App\Models\Model' => 'App\Policies\ModelPolicy',
  Category::class => CategoryPolicy::class,
  Comment::class => CommentPolicy::class,
];
```

11–19. kódrészlet: CommentPolicy regisztrálása a rendszerbe

A szükséges importálásokat ne felejtsük el a fájl elején!

A **comments** erőforrás útvonalak (kvázi csoportként) a **web.php** fájlban vannak regisztrálva és jelenleg egy **auth** Middleware tartozik hozzájuk. A probléma az, hogy a **can** Middleware-t nem tudjuk a resource útvonal típus mind a 7 útvonalához egyszerre hozzáadni, így csak kisebb csoportokban van erre lehetőségünk. A szétválasztást és csoportosítást a 11–1. táblázat Policy metódusai szerint tudjuk megtenni az alábbi kódsorokkal:

```
// Route::resource('comments', CommentController::class)-
>middleware('auth');

Route::get('/comments', [CommentController::class, 'index'])
  ->middleware('can:viewAny')
  ->name('comments.index');

Route::middleware('can:create')->group(function () {
```

```
  Route::get('/comments/create', [CommentController::class, 'create'])
    ->name('comments.create');
  Route::post('/comments', [CommentController::class, 'store'])
    ->name('comments.store');
});

Route::get('/comments/{comment}', [CommentController::class, 'show'])
  ->middleware('can:view')
  ->name('comments.show');

Route::middleware('can:update')->group(function () {
  Route::get('/comments/{comment}/edit', [CommentController::class, 'edit'])
    ->name('comments.edit');
  Route::put('/comments/{comment}', [CommentController::class, 'update'])
    ->name('comments.update');
});

Route::delete('/comments/{comment}', [CommentController::class, 'destroy'])
  ->middleware('can:delete')
  ->name('comments.destroy');
```

11–20. kódrészlet: Komment erőforrás útvonalakhoz rendelt engedélyek (egyesével vagy csoportosan)

Ennél persze egyszerűbb megoldás, ha a 11–16. kódrészletben látható Controller konstruktorban végrehajtott módosítást implementáljuk, itt sokkal nagyobb a hibázási lehetőségünk és még az útvonalainkat sem kell újra elneveznünk…

Ha mégis maradunk ennél a megoldásnál, akkor egy egyszerűsítést még elvégezhetünk a kódjainkban: a **middleware('can:PolicyMetódusNeve')** kód rövidíthető így: **can('PolicyMetódusNeve', Model példány vagy osztály neve)**. Ellenben a **can()** igényli második paraméterként a Model osztály nevét (**Comment::class**) **create** Policy metódus hivatkozás esetében, vagy a Model osztály példányának nevét (**'comment'**) az **update** és **delete** Policy metódusokra való hivatkozás esetében. Ráadásul a **can()** a csoportosított útvonalaknál nem is működik, így azokat vagy szétszedjük és egyesével használjuk, vagy marad azoknál a **middleware('can:PolicyMetódusNeve')** megoldás, például így:

```
Route::get('/comments', [CommentController::class, 'index'])
  ->can('viewAny', App\Models\Comment::class)
  ->name('comments.index');

Route::middleware('can:create')->group(function () {
  Route::get('/comments/create', [CommentController::class, 'create'])
    ->name('comments.create');
  Route::post('/comments', [CommentController::class, 'store'])
    ->name('comments.store');
});

Route::get('/comments/{comment}', [CommentController::class, 'show'])
  ->can('view')
  ->name('comments.show');
```

```php
Route::middleware('can:update')->group(function () {
  Route::get('/comments/{comment}/edit', [CommentController::class, 'edit'])
    ->name('comments.edit');
  Route::put('/comments/{comment}', [CommentController::class, 'update'])
    ->name('comments.update');
});

Route::delete('/comments/{comment}', [CommentController::class, 'destroy'])
  ->can('delete', 'comment')
  ->name('comments.destroy');
```

11–21. kódrészlet: Minimális egyszerűsítés a 11–20. kódrészlethez képest a can() metódussal

Összegzésként: ismételni lehet csak a megállapításokat, hogy ennél a megoldásnál sokkal többet kell kódolni, így sokkal nagyobb a hibázási lehetőség is, mint ha a Controller konstruktorában rendelnénk hozzá a Policy metódusokat, bár nem a konkrét útvonalakhoz, hanem a Controller metódusaihoz. De ebben a megoldásban újra egyesével nevet kell adnunk az útvonalainknak, a **can()** ráadásul nem is használható csoportosításkor, hanem meg kell tartanunk a **middleware()** segédmetódust a csoportosításnál. Egyszóval, nem a Middleware-es megoldás a legkönnyebben alkalmazható, ha erőforrás (resource) útvonal csoportot szeretnénk kezelni a Policy metódusok segítségével. Inkább lehet hatékony akkor, ha egyedi útvonalaink vannak, és már itt az útvonal regisztrációjánál el szeretnénk végezni az engedély ellenőrzését, nem csak a Controller-ben.

A nézet fájloknál alkalmazott Blade direktívák, amelyeket a Gate-ek megismerése során használtunk, a Policy esetében (és a kevert megoldások esetében is) ugyanúgy alkalmazhatók.

11

Karcsúbb lett a Laravel 11-es projekt mappa az engedélyezés szempontjából is: az **app / Http / Controllers / Controller** ősosztályból hiányzik a **AuthorizesRequests** trait importálása.

Egy logikus méretcsökkentés a Controller-ekben történő engedélyezés kapcsán is megtörtént a Laravel 11-es verziójában: a Controller ősosztályból hiányzik a **AuthorizesRequests** trait importálása. Erre igazából nem is nagyon volt szükség, hiszen a Controller-jeinkben eddig sem feltétlenül úgy hajtottuk végre az engedélyeztetést, hogy **$this->authorize(…)**, hanem vagy a Controller konstruktorában vagy a metódusaiban a Policy-ket használtuk erre, vagy akár a Gate-eket így: **Gate::authorize(…)**. A Laravel keretrendszer is ez utóbbi két módszer használatát javasolja, úgyhogy logikus lépés volt kivenni a Controller ősosztályból a **AuthorizesRequests** trait használatát.

Majdnem minden Service Provider eltűnt a keretrendszerből, kivéve az **AppServiceProvider**-t. A Policy-k automatikusan *„felkutatásra kerülnek”* és élesítődnek, ahogy elkészülnek. A Gate-eket az **AppServiceProvider**-ben kell regisztrálni.

11–1. újdonság: Engedélyezési technikák a Laravel 11-ben

11.2.3. Engedélyezési technikák automatikus tesztelése

Jelenleg a **tests / Feature / CommentTest** osztályunkban lévő **test_visitors_cannot_access_comments** teszteset nem fog működni, mivel az ott definiált **/comments** elérést csak hitelesített felhasználók tudják elvégezni, különben a rendszer tovább irányítja őket a **/login** útvonalra. Módosítsuk ezt úgy, hogy a válaszban elvárt HTTP állapotkód 302 helyett 403 legyen és ne irányítsuk tovább sehova.

```
public function test_visitors_cannot_access_comments(): void
{
  $response = $this->get('/comments');
  $response->assertStatus(403);
}
```

11–22. kódrészlet: A látogatók nem érhetik el a kommentek listázását

Az alfejezetben végrehajtott módosításokat manuálisan teszteltük, különböző jogokkal, engedélyekkel bíró felhasználókkal. Automatikus tesztelések létrehozásánál érdemes az **actingAs()** metódust használni úgy, hogy az adott felhasználó éppen csak látogató, regisztrált felhasználó vagy adminisztrátor is. Ezt végül is a hitelesítések tesztelése során már hasonlóan megtettük. A tesztesetek eredményéül pedig érdemes adatbázis módosításokat, útvonal eléréseket, nézetbeli látható eredményeket megvizsgálni.

További egyetlen példát nézzünk csak meg a használatra: hozzunk létre egy új teszteset metódust a **CommentTest** osztályban, de ne felejtsük el, hogy az **isAdmin** a **users** táblában egy új mező, így ahhoz szükség van a **RefreshDatabase** trait importálására az osztályon belül.

```
use RefreshDatabase;

public function test_admin_user_can_access_comments() {
  $admin = User::factory()->create(['isAdmin' => 1]);

  $response = $this->actingAs($admin)->get('comments');
  $response->assertStatus(200);
}
```

11–23. kódrészlet: Adminisztrátor felhasználó el tudja érni a kommenteket

Ha lefuttatjuk a két teszteset ellenőrzését, akkor mindkettőre pozitív eredményt kapunk:

```
php artisan test --filter=CommentTest
```

Az alfejezetben végrehajtott programkód módosítások ebben a GitHub commit-ben találhatók meg.

11.3. Szerepkör alapú engedélyezés

Azt már eddig is megtapasztalhattuk, hogy ha a felhasználók adott szerepkörhöz, vagy felhasználói csoportokhoz tartoznak, akkor bizonyos funkcionalitásokra képesek lesznek az alkalmazásunkban.

A szerepkörök eltárolása mindig egy tervezési döntés kérdése:

1. Egy meglehetősen kis alkalmazásban a látogatókon és a regisztrált felhasználókon kívül előfordulhat, hogy már csak egy csoport van, mégpedig az adminisztrátorok. Ezt ilyenkor egy

boolean típusú mezővel (például: **isAdmin**) lehet jelezni a **users** adattáblában és attól függően, hogy igaz-e ez az érték, az adott felhasználó adminisztrátornak számít, avagy nem. Lásd az előző 11.2. alfejezetet.

2. Egy kisebb alkalmazásban a **users** táblát is bővíthetjük a szerepkört eltároló oszloppal (például: **role** mezőben valamilyen felsorolás – *enum* – típusban, ami a szerepkörök nevét tartalmazza). Ez nagyon hasonlóan tud működni, mint az első felsorolási pontban bemutatott megoldás, csak az igaz-hamis típusú mező értékének vizsgálata helyett egy konkrét szerepkör nevét keressük a feltételvizsgálat során. Erre láthatunk példát a 11.3.1. alfejezetben, amikor a Breeze hitelesítési kezdők készlettel integráljuk ezt a fajta elgondolást.
3. Történhet egy külön adattáblában (például: **roles**), és akkor a felhasználók (**users**) egy idegen kulccsal (**role_id**) kapcsolódhatnak egy-egy szerepkörhöz, csoporthoz.
4. Ha azt szeretnénk, hogy egy felhasználó több szerepkörhöz is tartozhasson, akkor egy kapcsolótáblára van szükség a **users** és **roles** táblák között (**role_user** néven, **user_id** és **role_id** külső kulcs mezőkkel).

Az első pontban említett megoldást elég részletesen áttekintettük az előző alfejezetben, most a többi pontra koncentrálunk, főleg úgy, hogy a Laravel felhasználói hitelesítéshez használható kezdőkészleteit is igénybe vesszük ehhez.

11.3.1. Laravel Breeze projekt kiegészítése paraméteres köztes réteggel

Térjünk vissza az **l10_auth_breeze** projektünkre, mivel ott már a felhasználói hitelesítés rendelkezésünkre áll a Breeze által.

Továbbá mivel a Breeze-t már használjuk, ezért *„akarva/akaratlanul”* van is már két felhasználói csoportunk. Az egyik a látogatók csoportja, ők azok, akik nincsenek regisztrálva, ezáltal bejelentkezve sem lehetnek, illetve a másik csoport az a regisztrált felhasználóké, akik be is tudnak jelentkezni az alkalmazásunkba. A regisztrált felhasználók csoportját fogjuk szerepkörök szerint szétválasztani. Lesznek adminisztrátor (**admin**) és szerkesztő (**editor**) csoportjaink, vagy nevezhetjük őket szerepköröknek is.

Mivel most arra koncentrálunk, hogy a szerepkörök alapján különböző funkcionalitásokat tudjanak elérni a felhasználóink, és nem egy teljes körű jogosultsági rendszert szeretnénk építeni, ezért a fenti felsorolás második pontját követhetjük, vagyis azt, hogy a **users** adattáblát bővítsük ki egy *enum (felsorolás) típusú mezővel*, ami a kiválasztott szerepkört kaphatja csak értékül. Hozzuk létre az ennek megfelelő migrációs fájlt:

```
php artisan make:migration add_role_column_to_users
```

Az **up()** metódusban bővítsük a **users** tábla mezőlistáját a **role** oszloppal és állítsuk nullázhatóra a mező típusát, hátha lesz olyan felhasználónk, aki egyik felsorolt szerepkörhöz sem fog tartozni (és mivel már van is egy felhasználónk a táblában, ezért enélkül – vagy a default érték beállítása nélkül – nem engedné migrálni az új mezőt az adattáblába).

```
$table->enum('role', ['admin', 'editor'])->nullable();
```

11–24. kódrészlet: Szerepkör felsorolás oszlop elhelyezése a felhasználók adattáblájában

A **down()** metódusban csak dobjuk el (**dropColumn()**) ezt az új oszlopot. Ezután migrálhatunk.

Megjegyzés: a MySQL adatbáziskezelőben a táblákhoz tudunk *enum* típusú oszlopot hozzáadni. Az SQLite adatbázis ezt egy kicsit *„trükkösebben”* oldja meg a háttérben, mivel ott egy sima *varchar* (szöveges) típusú oszlop jön létre, amihez egy ellenőrzést fűz hozzá: CHECK("role" IN ('admin', 'editor')) Tehát csak **admin** és **editor** tényleges értékek (a null-on kívül) kerülhetnek bele ebbe a mezőbe.

A **User** Model osztály **$fillable** tömbjét bővítsük ki az új **role** mezővel!

Adjunk hozzá továbbá egy új metódust, ami egy bizonyos szerepkör meglétét hivatott ellenőrizni majd:

```
public function hasRole(string $role) : bool
{
  return $this->getAttribute('role') === $role;
}
```

11–25. kódrészlet: Adott szerepkör meglétének ellenőrzését végző segédmetódus

Megjegyzés: a hármas egyenlőség jel nem csak azt ellenőrzi, hogy megegyezik-e a paraméterként kapott **$role** változó típusa (string) megegyezik-e az alapértelmezetten létező **getAttribute()** metódus visszatérési értékének típusával.

Ezután hozzunk létre két új nézetet, amelyek mindössze egy **<h1>** címsort tartalmazzanak arra vonatkozó szöveggel, hogy melyik szerepkör nézet fájljai lesznek. Például az admin szerepkör nézet fájlnak legyen ez a neve: **resources / views / admin-dashboard.blade.php** a tartalma pedig ez:

```
<h1>Admin dashboard</h1>
```

11–26. kódrészlet: Admin szerepkörű felhasználók „vezérlőpultja”

A hozzá vezető útvonalat a **routes / web.php**-ban regisztráljuk:

```
Route::get('/admin-dashboard', function () {
  return view('admin-dashboard');
})->middleware(['role:admin'])->name('admin-dashboard');
```

11–27. kódrészlet: Admin szerepkörű felhasználók „vezérlőpultjához” vezető köztes réteggel védett útvonal

A nézetnek és az útvonalnak a párját is megcsinálhatjuk ugyanígy, csak nem *„admin”*, hanem *„editor”* szerepeljen benne mindenütt.

Ezután hozzuk létre azt a Middleware-t, amely lekezeli a paraméterként kapott szerepköröket és aszerint fogja tovább engedni a felhasználókat, hogy hozzá tartoznak-e a szerepkörhöz.

```
php artisan make:middleware EnsureUserHasRole
```

A létrejövő **EnsureUserHasRole** Middleware **handle()** metódusát alakítsuk át így:

```
public function handle(Request $request, Closure $next, string $role):
Response
{
  if ( ! $request->user()->hasRole($role)) {
    abort(403);
```

```
    }
    return $next($request);
}
```

11–28. kódrészlet: Ellenőrizzük, hogy a kéréshez tartozó felhasználó hozzá tartozik-e a szerepkörhöz

Az elkészült Middleware-t regisztráljuk az útvonal Middleware-ek közé, hogy ténylegesen ellenőrzésre kerüljön az imént bemutatott feltételvizsgálat. Nyissuk meg az **app / Http / Kernel.php** fájlt és a **$middlewareAliases** tömbhöz adjuk hozzá újként az alábbit:

```
'role' => \App\Http\Middleware\EnsureUserHasRole::class,
```

11–29. kódrészlet: Szerepkör meglétét ellenőrző Middleware regisztrálása a Kernel.php-ban

A működés ellenőrzéséhez hozzunk létre két új felhasználót, mindkét szerepkörhöz egy-egy darabot. Ehhez a Tinker-t és a **UserFactory** osztályt hívjuk segítségül, de nem kell azt módosítanunk:

php artisan tinker

```
User::factory()->create(['role' => 'admin'])

User::factory()->create(['role' => 'editor'])
```

11–4. utasítások: Különböző szerepkörű felhasználók létrehozása

A létrejövő felhasználókkal (saját felhasználónevükkel – username – és a jelszavukkal, amely mindkét esetben password) be tudunk jelentkezni, és ki tudjuk próbálni a következő útvonalak elérését:

- http://127.0.0.1:8000/admin-dashboard
- http://127.0.0.1:8000/editor-dashboard

A bejelentkezett felhasználó szerepkörétől függ, hogy melyik útvonalat éri el megfelelően. Amelyik útvonalat nem éri el, ott egy 403 Forbidden HTTP státuszkódot kap a böngészőben a felhasználó.

Így tehát elértük azt, hogy a felhasználói hitelesítés után bizonyos útvonalak elérését attól tettük függővé, hogy mely felhasználói csoporthoz, szerepkörhöz tartozik az adott bejelentkezett felhasználó.

Útvonal Middleware: a Middleware-eket érintő változások itt is hatást gyakorolnak.

A **bootstrap / app.php** beállítási metódusok közül a **withMiddleware()**-be kell beilleszteni azt a kódsort, amely akkor jut érvényre, ha a látogatóink olyan útvonalat szeretnének elérni, amelyet csak adott szerepkörhöz tartozó hitelesített felhasználóknak van joguk elérni, akkor ezt egy Middleware **alias()** segédmetódusban tudjuk ellenőrizni:

```
$middleware->alias([
    'role' => \App\Http\Middleware\EnsureUserHasRole::class
]);
```

11–2. újdonság: Útvonalakat érintő köztes rétegek alkalmazása

Ha több útvonalat is szeretnénk ilyen módon ellenőrizni, akkor lehetőségünk van *csoportosítani* az útvonalainkat, például azokat, amelyek az editor szerepkörű felhasználókhoz vagy esetleg az

adminisztrátorokhoz tartoznak. Ekkor közös Middleware-t rendelünk az útvonal csoporthoz, és így közvetve a csoport útvonalaihoz is.

```
Route::group(['middleware' => ['role:admin']], function () {

  Route::get('/admin-dashboard', function () {
    return view('admin-dashboard');
  })->name('admin-dashboard');

  // 2. admin útvonal regisztrációja
  // 3. admin útvonal regisztrációja
  // ...
  // utolsó admin útvonal regisztrációja
});
```

11–30. kódrészlet: Köztes réteg hozzárendelése útvonal csoporthoz

Ha az adott útvonal elérése lehetséges több szerepkör felhasználói számára is, akkor a *„role:"* után vesszővel elválasztva kell felsorolnunk a szerepkörök nevét. Például azt csinálhatjuk, hogy amely útvonalakat az **editor** szerepkörű felhasználók elérik, azokat érjék el az **admin** szerepkörűek is.

```
Route::group(['middleware' => ['role:editor,admin']], function () {

  Route::get('/editor-dashboard', function () {
    return view('editor-dashboard');
  })->name('editor-dashboard');

  // 2. editor és admin útvonal regisztrációja
  // 3. editor és admin útvonal regisztrációja
  // ...
  // utolsó editor és admin editor útvonal regisztrációja
});
```

11–31. kódrészlet: Több szerepkör felsorolása az útvonalakhoz tartozó paraméteres köztes rétegben

Ez azonban rögtön rávilágít arra, hogy ha még több szerepkörünk lenne, akkor az útvonal csoportoknál a szerepköröket mindenféle kombináció szerint csoportosítani kellene… talán érezzük, hogy ez így biztosan nem a legmegfelelő megoldás lesz.

11.3.2. Engedélyezés a Jetstream csapatok szerepkörei, jogosultságai által

A Laravel Jetstream szerepkör kezelése a 11.3. alfejezet elején látható felsorolást még egy dologgal kiegészíti, ez pedig a csapatok kezelése. A felhasználók és a csapatok egy kapcsolótábla alapján vannak összerendelve, és ebben a **team_users** kapcsolótáblában az egyes kapcsolatoknál van definiálva a felhasználó csapatban betöltött szerepköre (szöveges formában). Így egy felhasználónál könnyedén előfordulhat, hogy az egyik csapatban adminisztrátor, a másikban szerkesztő, mivel a csapat-felhasználó kapcsolatnál van definiálva a felhasználó csapatban betöltött szerepköre. A **team_users** táblában mindenesetre nincsen a **role** oszlop *enum* típusként definiálva, hanem csak egyszerű szöveges mezőként, így nincs megszorítás arra vonatkozóan, hogy milyen szerepköröket definiálhatunk a programkódban.

A szerepköröket (például admin) és az ő jogosultságaikat (például delete) az **app / Providers / JetstreamServiceProvider** osztályban tudjuk létrehozni, módosítani, törölni. Az adott szerepköröknél látható, hogy az admin minden CRUD műveletre képes, minden jogosultsággal rendelkezik, míg az editor a törlést leszámítva tudja böngészni (kiolvasni, lekérdezni), létrehozni és frissíteni az erőforrásokat. Az API hozzáférésnél alapértelmezetten csak a kiolvasás funkció van engedélyezve.

11.3.2.1. Erőforrás elemeinek megvalósítása

Az **l10-auth-jetstream** projektben hozzunk létre egy új Model-t az -a kapcsolóval, majd ennél az erőforrásnál a hitelesített felhasználók számára engedélyezzük az **index**, **show** nézeteket (konkrét szerepkör itt nem elvárás), az editor szerepkörű felhasználók számára engedélyezzük a **create**, **store**, **edit**, **update** útvonalakat, és az admin szerepkörrel rendelkezők számára a **destroy** útvonalat és funkciót:

```
php artisan make:model Book -a
```

Tipp: Model osztályt minden szükséges kiegészítővel létre tudunk hozni egyetlen parancs kiadásával így.

Az újdonság és a fontos itt az -a kapcsoló, amely hozzáadja a következőket a projekthez:

1. **Book** Model osztály
2. **[időbélyeg]_create_books_table** migrációs fájl (az időbélyeg mindenkinél más)
3. **BookController** osztály a vezérléshez
4. **StoreBookRequest** osztály az új adat feltöltésekor validál
5. **UpdateBookRequest** osztály a meglévő adat frissítésekor validál
6. **BookFactory** adatgyár osztály könnyedén tudunk teszt adatokat létrehozni
7. **BookSeeder** adatbetöltő osztály akár több adatgyárat is képes működtetni
8. **BookPolicy** osztály az erőforráshoz való hozzáférést, engedélyezést felügyeli

Érdemes nagyjából követni a fenti felsorolást a megvalósítás során is, de röviden itt látható, hogy milyen lépéseken kell végig menni a megvalósításhoz (az egyszerűbbeket itt érdemes is végrehajtani a felsorolás áttekintése során):

1. **Book** Model osztály **$fillable** attribútumát bővítsük egy **title** és **team_id** mezővel;
   a. egy könyv mindig egy csapathoz tartozik (amelyikben létrehozzák!), egy csapathoz pedig több könyv is tartozhat (1-n kapcsolat);
      i. definiáljuk ezeket a **belongsTo()-hasMany()** kapcsolatokat a **Book** és **Team** Model osztályok között.
2. **books** táblát létrehozó migrációs fájl:
   a. adjunk hozzá egy szöveges **title** mezőt,
   b. és még egy külső kulcsot: **team_id**, amely a **teams** tábla elsődleges kulcsára mutat,
   c. majd migráljunk: `php artisan migrate`
3. **BookFactory**: az adatgyárban a következőket hajtsuk végre:

   a. a **fake()** metódusnak van egy mondat generáló metódusa, amelynek átadhatjuk, hogy hány szóból álljon, legyen például 3 szavas mondat (**sentence(3)**) a könyveink címe, ha adatgyárral hozzuk létre őket,
   b. illetve a **team_id** mezőt generálhatjuk a **Team::inRandomOrder()->first()->id** utasítással.
4. **BookSeeder**: az adatgyárat felhasználva egy futásnál hozzon létre 10 könyvet;
   a. Ha az adattáblánk, az adatgyárunk és a Seeder osztályunk megvan, akkor létre is hozhatunk teszt adatokat a **books** adattáblába:
      i. php artisan db:seed --class=BookSeeder
5. A könyvek erőforráshoz vezető útvonalak regisztrálása a **routes / web.php**-ban (11–33. kódrészlet);
6. **BookController**: a szükséges metódusok létre is jöttek benne, de ezeket majd alakítsuk át (11–34. kódrészlet);
7. A kéréseket validáló Request osztályok:
   a. **rules()** metódusaikban állítsuk be, hogy a **title** mező kitöltése kötelező, legalább 3 karakter hosszú (a **team_id** mezőt mindig a mentéskor, frissítéskor adjuk hozzá úgy, hogy az adott felhasználó aktuálisan kiválasztott csapatát mentjük el a könyvnél, így ezt külön validálni nem kell);
   b. **authorize()** metódusaikban állítsuk be, hogy **return true,** mivel alapértelmezetten hamissal térnek vissza;
8. **BookPolicy**: a metódusok ehhez is elkészültek, csak úgy, mint a **BookController**-hez, úgyhogy már csak a visszatérési értékeket kell hozzájuk meghatározni.
   a. Kezdetben mindegyik metódus visszatérhet így: **return true;** Ezzel csak azt az elvárást fogalmazzuk meg a könyveknél, hogy hitelesített felhasználónak kell lennie annak, aki hozzá szeretne férni a könyvekhez bármilyen funkcionalitás formájában.
   b. Az **AuthServiceProvider $policies** tömbjében regisztrálhatjuk a kapcsolatot, de ez az elnevezések miatt (ha nem nevezünk át semmit), akkor automatikus feltérképezésre kerül a keretrendszer által, így nem kötelező az itteni regisztráció, csak ha esetleg más metódusokat is szeretnénk használni az erőforrásnál a CRUD műveleteken kívül, akkor van szükség mindenképpen a regisztrációra. Ezt előre nem mindig lehet meghatározni, ezért talán inkább az a jobb megoldás, ha beregisztráljuk itt a kapcsolatot előre, így a későbbiekben nem érhet minket meglepetés ennek hiánya miatt.
9. Könyveket kilistázó menüpont a vezérlőpult felületen és a nézet oldalak elkészítése.

Ezen pontok nagy részét már többször megcsináltuk, így nem részletezek itt mindent, viszont, ha nem menne valakinél, akkor érdemes megnézni az alfejezet végén található GitHub commit alapján végrehajtott programkód módosításokat. Az érdekesebb részek viszont az alábbiakban ismertetésre kerülnek.

A **Book** Model osztály új elemei a kitölthető mezőkkel és a Jetstream Model specifikus kapcsolattal itt látható:

```
protected $fillable = ['title', 'team_id'];
```

```
public function team(): BelongsTo
{
  return $this->belongsTo(Jetstream::teamModel());
}
```

11–32. kódrészlet: Book Model osztály tartalma

A kapcsolat ellenkező oldalán, a **Team** Model osztályban csak egy már sokszor látott **books() : hasMany** kapcsolatot kell definiálni.

A regisztrált útvonalakat tartalmazó **routes / web.php** fájlban már előre definiálásra került a Jetstream telepítése után az a védett útvonal csoport (**group()**), amelyben a **dashboard** útvonal is található. Még ugyanebben a csoportban definiálhatjuk a könyvek erőforráshoz tartozó útvonalainkat:

```
Route::resource('books', BookController::class);
```

11–33. kódrészlet: Védett útvonal csoportba elhelyezett könyves erőforrás útvonalak

Ha ilyen rövid névvel akarunk hivatkozni a **BookController** osztályra, akkor ne felejtsük el importálni a fájl tetején az osztályt.

A **BookController** osztály konstruktora, metódusai, segédmetódusa látható alább:

```
public function __construct()
{
  $this->authorizeResource(Book::class, 'book');
}

public function index()
{
  $books = Book::where('team_id', $this->getCurrentTeamId())->get();
  return view('books.index', compact('books'));
}

public function create()
{
  return view('books.create');
}

public function store(StoreBookRequest $request)
{
  $validated = $request->validated();
  $validated['team_id'] = $this->getCurrentTeamId();

  Book::create($validated);

  return redirect()->route('books.index');
}

public function show(Book $book)
```

```
{
  return view('books.show', compact('book'));
}

public function edit(Book $book)
{
  return view('books.edit', compact('book'));
}

public function update(UpdateBookRequest $request, Book $book)
{
  $validated = $request->validated();
  $validated['team_id'] = $this->getCurrentTeamId();

  $book->update($validated);

  return redirect()->route('books.index');
}

public function destroy(Book $book)
{
  $book->delete();

  return redirect()->route('books.index');
}

private function getCurrentTeamId() {
  return auth()->user()->currentTeam->id;
}
```

11–34. kódrészlet: BookController osztály konstruktora, metódusai

A konstruktorban definiáltuk azt, hogy majd a megfelelő **BookPolicy** osztály metódusa legyenek érvényben, amikor az adott **BookController** metódusok meghívásra kerülnek (11–1. táblázat). A kilistázásnál az adott felhasználó aktuálisan kiválasztott csapatához tartozó könyvek jelennek majd meg. A **store()** és **update()** metódusokban a validálás után hozzáadjuk még az adott felhasználó aktuális csapat azonosítóját és azt mentjük el a könyvnél. A **getCurrentTeamId()** segédmetódus csak abban segít minket, hogy mindig az adott bejelentkezett felhasználó aktuálisan kiválasztott csapatának az azonosítóját adja vissza.

Az **index** nézethez való eljutási lehetőséget előbb egy navigációs linkkel biztosítani kell. Ha megnyitjuk a **resources / views / layouts / app.blade.php** fájlt, akkor láthatjuk, hogy van benne egy **navigation-menu** nevű livewire komponens. Erről mi még eddig igazából semmit sem tudunk, viszont láthatjuk, hogy van egy **navigation-menu.blade.php** fájl a nézetek között. Ha ezt megnyitjuk, már láthatjuk is benne a *„Navigation Links”* komment alatt, hogy ott van a **dashboard** útvonalhoz vezető link. Lemásolhatjuk ezt a linket és értelemszerűen átírhatjuk az attribútum értékeit így:

```
<x-nav-link href="{{ route('books.index') }}" :active="request()-
>routeIs('books.*')">
```

```
    {{ __('Books') }}
</x-nav-link>
```

11–35. kódrészlet: Könyveket listázó útvonalhoz vezető link a navigációs menüben

Ez így megfelelő, elirányít minket a link a könyves lista oldalra (ha bővebben érdekel minket a **nav-link** komponens tartalma, akkor ezt is meg tudjuk nézni a **resources / views / components** mappában). Ugyanezt a link duplázást és átszerkesztést tegyük meg a **navigation-menu.blade.php** fájl későbbi részében, ahol a *„Responsive Navigation Menu"* megjegyzés látható, és szintén a **dashboard** linket kell ott lemásolnunk.

Viszont további nézeteink még nincsenek, ezért hozzuk létre a **resources / views**-ban a **books** mappát, és ide kerülhetnek be az **index, show, create, edit** nézet fájlok. Ezeknek a fájloknak a keretes szerkezetét adhatja a **dashboard.blade.php**-ban található tartalom, pusztán a címsorban lévő szöveget kell megfelelően átírni, és majd a **welcome** komponens helyére mindig betenni a tényleges könyveinket érintő tartalmakat. Fontos még kiemelni, hogy ne csak a megszokott tartalmakat helyezzük el az index nézetben (könyv létrehozási link, könyvlistát tartalmazó táblázat, táblázat soraiban a szerkesztési és törlési műveletek linkje és gombja), hanem a **@can-@endcan** Blade direktívákkal a hozzáférési vizsgálatokat is a funkciókhoz. Ezek a nézetek a Tailwind CSS keretrendszerrel készülnek el, a teljes forráskódjukat ide nem másolom be, mivel elég átláthatatlan lenne, viszont az alfejezet végén lévő GitHub commit-ben megtalálható mindegyik nézet forráskódja, illetve itt vannak közvetlenül is: index, show, create, edit.

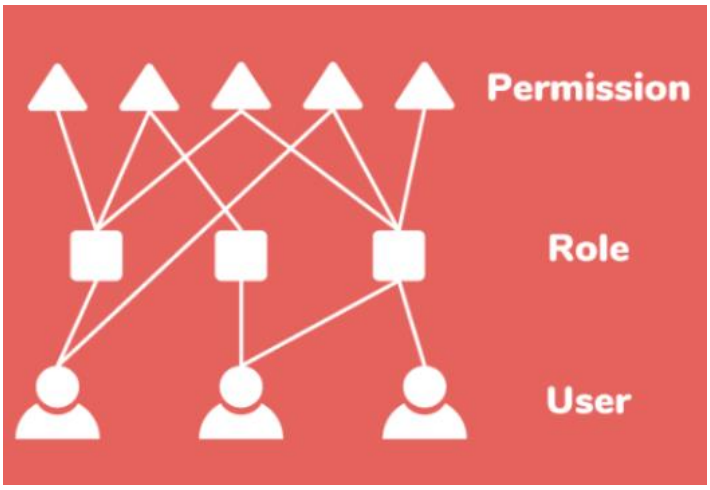
Mielőtt a **BookPolicy** osztály metódusait módosítanánk, gondoljunk végig néhány alapvetést, legjobb gyakorlatot, amelyet érdemes betartani a szerepkör alapú jogosultsági rendszereknél (RBAC[20]).

A szerepkör alapú jogosultsági rendszerek kialakítása egy nagyon jól átlátható, emiatt egyszerű és hatékony módszer arra, hogy bizonyos funkcionalitások végrehajtásához engedélyeket kössünk. Ilyenkor általában regisztrált felhasználók bizonyos csoportjaival dolgozunk, amely csoportokat szerepköröknek nevezünk.

Néhányat legjobb gyakorlat itt látható, amelyeket érdemes betartani a tervezés és megvalósítás során:

- A felhasználóknak szerepkörük (csoportjuk) van, vagy mondhatnánk úgy is, hogy szerepkörhöz tartoznak a felhasználók, például Gludovátz Attila az adminisztrátorok szerepkörrel rendelkezik.
- A szerepkörök jogosultságokkal (permission), engedélyekkel rendelkeznek, például az adminisztrátorok tudnak valamit (képesek valamire), például létrehozni egy könyvet a rendszerben.
- Az alkalmazás mindig a jogosultságokat ellenőrzi (amennyire lehetséges), és nem a szerepköröket, tehát az alkalmazás azt nézi, hogy adott felhasználó (a szerepkörén keresztül) képes-e, jogosult-e megcsinálni valamit az oldalon, végrehajthat-e adott funkcionalitást.

[20] Role-Based Access Control



11–5. ábra: Felhasználók - szerepkörök - jogosultságok (engedélyek)

Az **l10-auth-jetstream** példa alkalmazásunkban ráadásul még csapatok is vannak, mivel a Jetstream csapatokon belül határoz meg szerepköröket, és így közvetve jogosultságokat az egyes felhasználóknak.

A korábbi csapatokhoz tartozó funkcionalitások kipróbálásának és tesztelésének (10.2.2.5. alfejezet) köszönhetően ezekből áll most a rendszer:

- két csapat (egy felhasználó egy csapatot birtokol, a birtoklás itt azt jelenti, hogy neki mindent szabad a saját csapatán belül, az alapértelmezett jogosultsága: *****, ami a *mindenre* utal),
- az egyik felhasználó (Gipsz Jakab, a 2-es csapat tulajdonosa) ezen felül rendelkezik a másik csapatban editor szerepkörrel is.

A **BookSeeder** osztály futtatásának eredménye az lett, hogy a létrejött 10 könyvből 4 az 1-es csapathoz lett hozzárendelve, míg a másik 6 a 2-es számú csapathoz került.

id	title	team_id
1	Beatae mollitia accusamus illo.	2
2	Expedita ut non.	1
3	Veniam rerum omnis tempora.	2
4	Odit eaque culpa cum blanditiis.	1
5	Est maxime placeat maxime.	1
6	Temporibus dolorum in.	2
7	Eligendi distinctio adipisci eos.	1
8	Sit natus qui.	2
9	Corrupti libero sed.	2
10	Eligendi et vitae.	2

11–6. ábra: Könyvgyár működésének eredménye

Kezdjük el finomhangolni a **BookPolicy** osztály **create(), update(), delete()** metódusait (mivel a **restore()** és **forceDelete()** a *„soft delete”* tulajdonsághoz kötődnek, amely most nem releváns számunkra, ezért ezek törölhetők).

```
public function create(User $user): bool
{
  return $user->hasTeamPermission($user->currentTeam, 'create');
}
```

```
public function update(User $user, Book $book): bool
{
  $team = Team::find($book->team_id);

  return $user->hasTeamPermission($team, 'update');
}

public function delete(User $user, Book $book): bool
{
  $team = Team::find($book->team_id);

  return $user->hasTeamPermission($team, 'delete');
}
```

11–36. kódrészlet: BookPolicy create, update, delete engedélyeinek vizsgálata

Megjegyzés: a metódusok összeállításánál nagy segítséget jelentett a Jetstream hivatalos dokumentációjának a szerepköröket és engedélyeket tartalmazó része. A Jetstream most már számos segédfüggvénnyel támogatja azt, hogy hatékonyan tudjunk szerepkör alapú jogosultsági rendszert építeni vele. Ennek kapcsán mindenképpen érdemes áttekinteni ezt a dokumentáció részletet, mivel több hasznos funkcionalitás is bemutatásra kerül benne felhasználók, csapatok, szerepkörök és a jogosultságok témakörben.

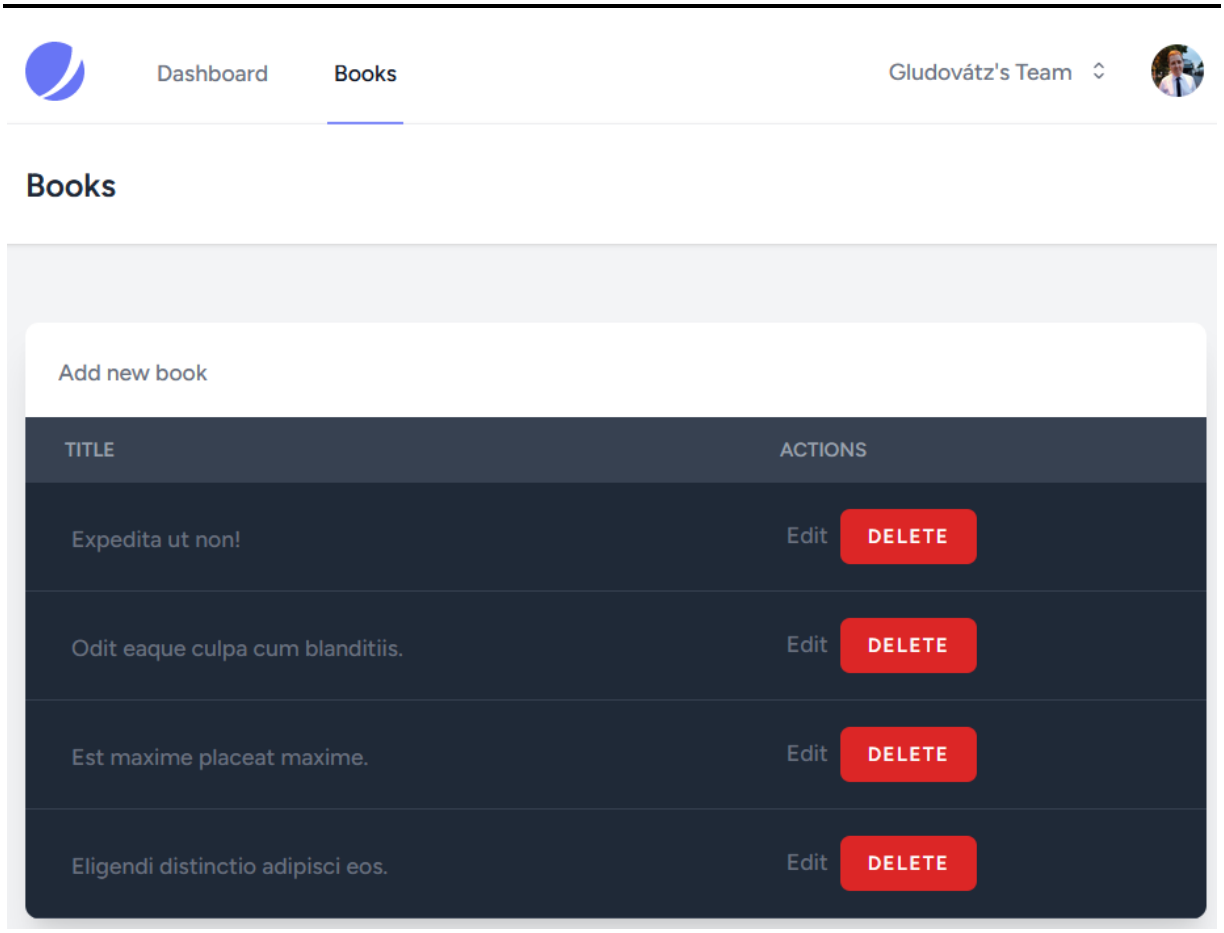
A **create** engedély vizsgálatánál lekérjük az éppen aktuális felhasználó aktuális csapatát és megvizsgáljuk, hogy annál a csapatnál rendelkezik-e a **create** jogosultsággal. Ez automatikusan hozzárendelődik a **BookController create()** és **store()** metódusaihoz, tehát csak az fér hozzá ezekhez, aki rendelkezik az adott csapatban ilyen **create** jogosultsággal.

Az **update** engedély vizsgálatánál nem az aktuálisan bejelentkezett felhasználó aktuálisan kiválasztott csapatán belül vizsgáljuk az **update** jogosultság meglétét, hanem először a szerkeszteni/frissíteni kívánt könyv *„csapatát"* kérjük le és abban vizsgáljuk meg, hogy van-e a bejelentkezett felhasználónak **update** jogosultsága. Ez automatikusan hozzárendelődik a **BookController edit()** és **update()** metódusaihoz. Ugyanezt történik a **delete** engedély esetében is, ami a **BookController destroy()** metódusának végrehajtása előtt végez ellenőrzést.

11.3.2.2. Erőforrás hozzáférésének manuális tesztelése

Mivel a **BookPolicy** osztályban kezdetben a **viewAny()** és **view()** metódusoknál beállítottunk egy **return true;** értéket, ezzel azt értük el, hogy az összes (könyvekhez tartozó) funkció eléréséhez csak hitelesített, bejelentkezett felhasználók férhetnek hozzá: ha látógatóként szeretnénk lekérni a böngészőben a **/books** útvonalat, akkor rögtön átirányít minket a rendszer a **/login** útvonalra, tehát jelzi, hogy csak bejelentkezés után érhetjük el.

A következő tesztesetnél lépjünk be az 1-es csapat tulajdonosaként az oldalra. Ez a felhasználó képes a saját csapatában elérni a könyvek listáját, új könyveket létrehozni egy űrlapon, azokat megtekinteni, továbbá szerkeszteni és törölni is tudja őket.



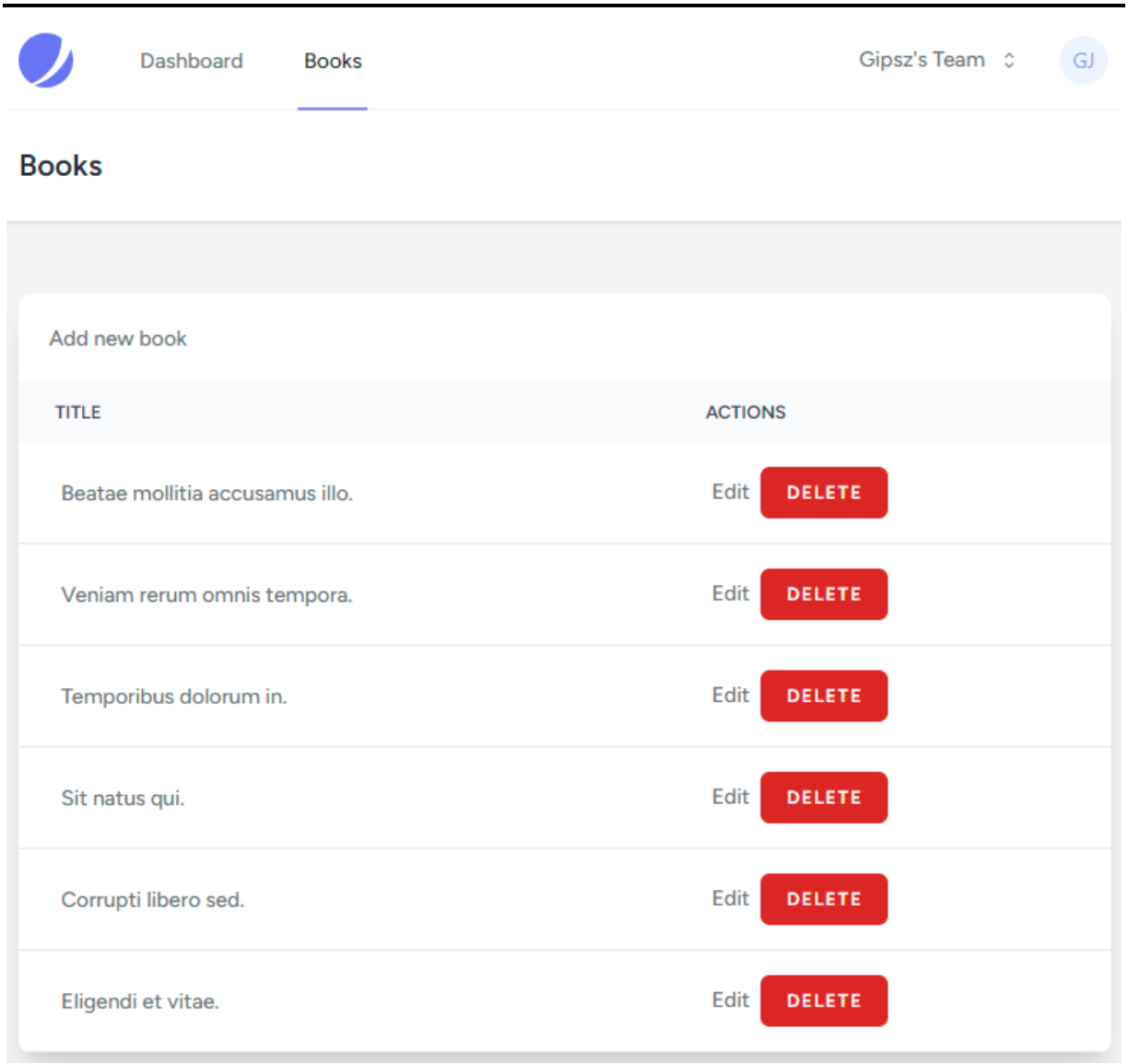
11–7. ábra: 1-es csapatnál létrehozott 4 darab könyv listája az 1-es felhasználó szemszögéből

Az 1-es csapat tulajdonosa a másik csapat könyvlistájához nem fér hozzá: nem tud átváltani a másik csapatra. Ha pedig manuálisan olyan könyvazonosítót szeretne elérni szerkesztésre (például **/books/9/edit**), amely nem az ő csapatához tartozik, akkor 403 *„This action is unauthorized."* visszajelzést kapunk. Eléri azonban megtekintésre azokat a könyveket (például **/books/9**), amelyek nem az ő csapatához tartoznak. Ha ezt ki szeretnénk javítani, akkor a **view()** metódust módosítsuk eszerint:

```
public function view(User $user, Book $book): bool
{
  $team = Team::find($book->team_id);

  return $user->hasTeamPermission($team, 'read');
}
```

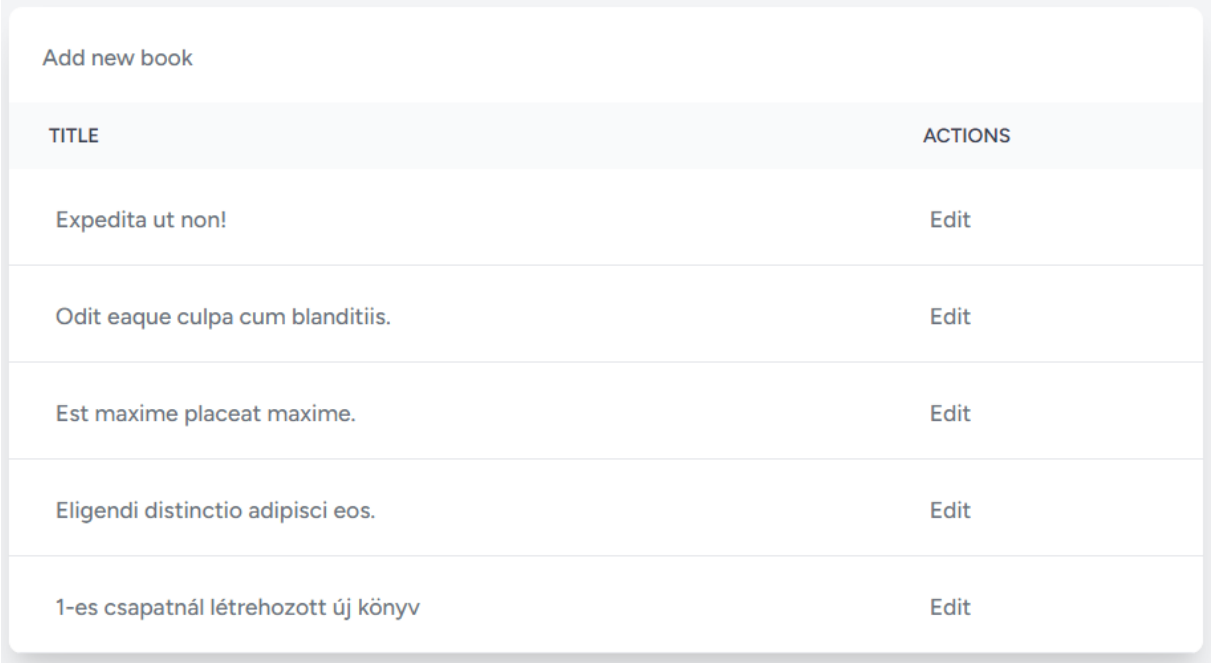
11–37. kódrészlet: Ne lehessen engedély nélkül más csapat könyvét megtekinteni

Manuális tesztelésnél, amikor egy másik felhasználóval szeretnénk belépni, akkor érdemes nyitni neki egy új privát ablakot vagy egy másik böngészőben lépjünk be vele, hogy ezen a módon tudjuk áttekinteni a lehetőségek (műveletek, nézetek) közötti különbségeket.



11–8. ábra: 2-es csapatnál létrehozott 6 darab könyv listája a 2-es felhasználó szemszögéből (a másik böngészőben a világos téma az alapértelmezett)

A 2-es csapat felhasználója (aki az 1-es csapatban editor szerepkörrel rendelkezik) képes csapatot váltani a jobb felső csapatos menüben, így mindkét csapat könyvlistáját tudja böngészni. Amikor azonban átlép az 1-es csapat könyvlistájához, akkor azt tapasztalhatja, hogy ott is képes könyvet létrehozni (11–9. ábra, a listában alul látszódik az új könyv), sőt szerkeszteni is tudja őket. Ellenben a törlési opció hiányzik. Ez abból adódik, mivel csak editor szerepkörrel rendelkezik a másik csapatnál, amelynél a **JetstreamServiceProvider** osztály alapján hiányzik a **delete** jogosultság megléte.



11–9. ábra: 1-es csapatnál létrehozott 1 új könyvet tartalmazó lista a 2-es felhasználó szemszögéből

A manuális tesztelés során tehát működött a saját csapat könyveinek kilistázása, megtekintése, szerkesztése és törlése. Annál a felhasználónál, akinél volt lehetőség a másik csapat könyveit is megtekinteni, ott szintén működött a könyv létrehozás és szerkesztés, míg a törlésre nem volt lehetősége, ahogy az az elvárásunk is volt.

Ha most a **JetstreamServiceProvider** osztály **configurePermission()** metódusában az editor szerepkörnél hozzáadnánk a **'delete'** értéket a többi jogosultság mellé, és frissítenénk, akkor egyből látszódna a táblázat soraiban a törlést lehetővé tevő gomb is.

Az alfejezetben így tehát egy összetett: csapat, szerepkör, jogosultság alapú engedélyeztetési rendszer működését ismertük meg a gyakorlatban.

Az alfejezet során elvégzett programkód módosítások ebben a GitHub commit-ben találhatók meg.

11.3.2.3. Erőforrás hozzáférések automatikus tesztelése

Az iménti manuális teszteket érdemes átültetni automatikus tesztek formájába is.

```
php artisan make:test BookTest
```

Három tesztesetet definiáljunk, miután a **RefreshDatabase** trait-et hozzáadtuk az osztályunkhoz:

```
use RefreshDatabase;

public function test_visitors_cannot_access_books()
{
  $response = $this->get('/books');
  $response->assertStatus(302);
  $response->assertRedirect('login');
}
```

```
public function test_team_owner_can_access_books()
{
  $owner = User::factory()->withPersonalTeam()->create();
  $team_id = $owner->ownedTeams()->first()->id;
  $bookTitle = Book::factory()->create(['team_id' => $team_id])->title;

  $response = $this->actingAs($owner)->get('books');

  $response->assertStatus(200);
  $response->assertSee($bookTitle);
  $response->assertSee('Add new book');
}

public function test_editor_from_other_team_cannot_delete_book()
{
  $owner = User::factory()->withPersonalTeam()->create();

  $owner->currentTeam->users()->attach(
    $editor = User::factory()->withPersonalTeam()->create(), ['role' =>
'editor']
  );

  $team_id = $owner->ownedTeams()->first()->id;
  $book_id = Book::factory()->create(['team_id' => $team_id])->id;

  $response = $this->actingAs($editor)->delete('books/' . $book_id);

  $response->assertStatus(403);
}
```

11–38. kódrészlet: BookTest osztály tesztesetei az engedélyezésre

Az első teszteset egy korábban már alkalmazott eseményt tesztel, mégpedig azt, hogy a látogatók nem érhetik el a könyveket listázó oldalt. A második teszteset ennél már bonyolultabb: adott felhasználó, aki rendelkezik (birtokol) egy csapatot, az annál a csapatnál létrehozott könyv címét láthatja az oldalon. Végül pedig a manuális tesztelésnél is hosszan ismertetett dolog egy részét láthatjuk: két felhasználó, két saját csapattal, az egyik felhasználó kap editor szerepkört a másik csapatában, viszont az oda létrehozott könyvet nem tudja törölni jogosultság hiányában.

Teszteljük a könyveket érintő teszteseteinket:

```
php artisan test --filter=BookTest
```

A teszteseteink szerencsére hibátlanul lefutnak, ezek után pedig már a meglévő példák alapján könnyedén lehet saját teszteseteket definiálni. A Jetstream is számos tesztesetet biztosít még, amelyek jó példaként szolgálhatnak az újabb tesztesetek létrehozásához, emiatt azokat is érdemes áttekinteni ismét.

Az alfejezetben végrehajtott programkód módosítások ebben a GitHub commit-ben találhatók meg.

11.3.3. Engedélyezés külső csomaggal (Spatie: Laravel-Permission)

A Laravel Breeze és a Jetstream (Fortify) kezdő készletek elsősorban a hitelesítési eljárás során hasznosak számunkra. Ahogy azt láthattuk, lehet őket használni szerepkör alapú jogosultságkezelésre is, de nem feltétlenül ez a legnagyobb erősségük.

Létezik azonban egy másik, külső csomag a Spatie szoftverfejlesztő csapat által gondozott Laravel-Permission, amely kimondottan alkalmas a felhasználó-szerepkör-jogosultság hármas hatékony kezelésére. Mivel ez a csomag már hosszabb ideje létezik, a fókuszát pedig a jogosultságkezelésen tartja, ezért ebben kifinomultabb megoldásokat találhatunk erre a témakörre, mint amit már az említett kezdőkészletek esetében mi magunk is megtettünk.

Ennek a csomagnak a használatát a blog oldalamon egy hosszabb bejegyzésfolyamban mutattam be, amelynek során egy ipari cégnél alkalmazható *karbantartás menedzsment rendszer* készült el. Ha valaki konkrétan csak a Laravel-Permission csomag használata iránt érdeklődik, akkor ezeket a blogbejegyzéseket javaslom áttekintésre.

Blog: vágjunk bele egy valós ipari projektbe, amely a karbantartás menedzseléséről szól Laravel alapokon. A projektből sokat tanulhatunk, elővéve olyan tudást, amely már ismertetésre került ebben a dokumentumban, de olyat is, amelyre itt még nem került sor. Kezdetben egy alaprendszert fogunk megvalósítani, amely már jó néhány olyan elvárásnak, funkcionalitásnak meg fog felelni, amit az ipari partner, az ügyfél követelményként fogalmazott meg.

Ez az alkalmazásfejlesztési projekt a Spatie Laravel-Permission csomagon túl a Filament csomag használatával is megismerteti az Olvasót, ami miatt szintén érdemes belevágni a blogbejegyzések feldolgozásába. A Filament egy full-stack (backend és frontend) komponensekből álló gyűjtemény a gyorsított Laravel fejlesztéshez. Gyönyörűen megtervezettek, intuitívan használhatók és teljes mértékben bővíthetőek a komponensek. Emiatt a Filament csomag használatának megkezdése is egy tökéletes kiindulópont egy Laravel alkalmazás megvalósításakor. A Filament-ről itt találunk még további információt: https://filamentphp.com/

11.4. REST API megvalósítása engedélyezéssel (Jetstream: csapat és szerepkör alapon)

REST API felületünk korábban nyilvánosan elérhető útvonalakat és funkcionalitásokat tartalmazott. Azonban ez az esetek ritkább részében szokott így lenni a való világban. Általában az API-n keresztül elérhető funkcionalitások elérését felhasználói hitelesítéshez és engedélyeztetéshez szokták kötni. Ebben az alfejezetben a meglévő API felületünk elemeit így priváttá fogjuk tenni, sőt, csapatonként és szerepkörönként keresztül is finomhangoljuk az elérhető funkcionalitások körét.

11.4.1. Előkészítés

Az Jetstream hitelesítéssel és engedélyezéssel ellátott projektünk folytatjuk továbbra is a munkát. Az erőforrások API-on keresztül történő kezelését a 7.5. alfejezet tartalmazza, míg ennek a hitelesítéssel történő kiegészítését a 10.2.2.4.4. alfejezet. Ezeket az ismereteket bővítjük most ki az engedélyezéssel és tekintjük át az API felületen keresztüli erőforrás kezelést.

A hitelesítéssel védett API erőforrás útvonalakat hozzuk létre a **routes / api.php** fájlban:

```
Route::middleware('auth:sanctum')->prefix('v1')->group(function () {
  Route::apiResource('books', BookController::class);
});
```

11–39. kódrészlet: Hitelesítéssel védett API útvonalak regisztrálása a könyvek kezeléséhez

Bár az API erőforrás **BookController**-ét még nem hoztuk létre, de az iménti útvonalak regisztrációjának fájljában importáljuk a következőt: **App\Http\Controllers\Api\V1\BookController**

Hozzuk létre a könyvhöz tartozó erőforrás fájlt, amellyel tudjuk szűrni, hogy milyen **books** adattábla mezőket lehessen a kérések során lekérdezni.

```
php artisan make:resource V1/BookResource
```

Az **app / http / Resources / V1 / BookResource.php** fájl tartalmában csak az **id**-t és a **title** mezőt adjuk vissza:

```
public function toArray(Request $request): array
{
  return [
    'id' => $this->id,
    'title' => $this->title,
  ];
}
```

11–40. kódrészlet: Könyv erőforrás szűrt mezőinek meghatározása

11.4.2. API Controller metódusai és engedélyeik

Ebben az alfejezetben áttekintjük, hogy melyik API Controller metódusnak milyen tartalmat határozzunk meg és a hozzá kapcsolódó Policy engedélyeit milyen módon változtassuk meg esetleg az API-os hozzáférés miatt.

Hozzunk létre a könyvekhez egy külön API Controller-t:

```
php artisan make:controller Api/V1/BookController --api
```

Az API Controller metódusaihoz ugyanúgy automatikusan hozzárendelődnek a Policy metódusok, mint azt korábban tapasztaltuk és áttekintettük a 11–1. táblázat alapján, csak mivel API-ról van szó, ezért az űrlapos nézetekkel visszatérő Controller metódusok (**create()** és **edit()**) itt nem játszanak szerepet. Ehhez persze hozzá kell adnunk az új Controller-hez is a konstruktort:

```
public function __construct()
```

```
{
  $this->authorizeResource(Book::class, 'book');
}
```

11–41. kódrészlet: API Controller konstruktora az engedélyezés hozzárendeléséhez

11.4.2.1. API Controller index és Policy viewAny metódusok definiálása és tesztelése

Nyissuk meg a Postman alkalmazást! Az **l10-auth-jetstream** nevű gyűjteményben lévő **get-user** kérést másoljuk le (*„Duplicate”*) és nevezzük át az új elemet erre:

- **Retrieving books** néven: GET **/api/v1/books**
- Így a kérésben benne marad az *„Authorization”* lapfülön az *„Auth Type”* kiválasztása: **Bearer Token**, illetve a *„Token”* konkrét értéke a jobb oldalon.
- A *„Headers”*-ben pedig az „Accept” kulcs és értéke az **application/json** marad benne, ami szintén továbbra is szükséges a számunkra. Emiatt is érdemes inkább duplikálni a kéréseket, mintsem újként mindig mindent újra beállítani.

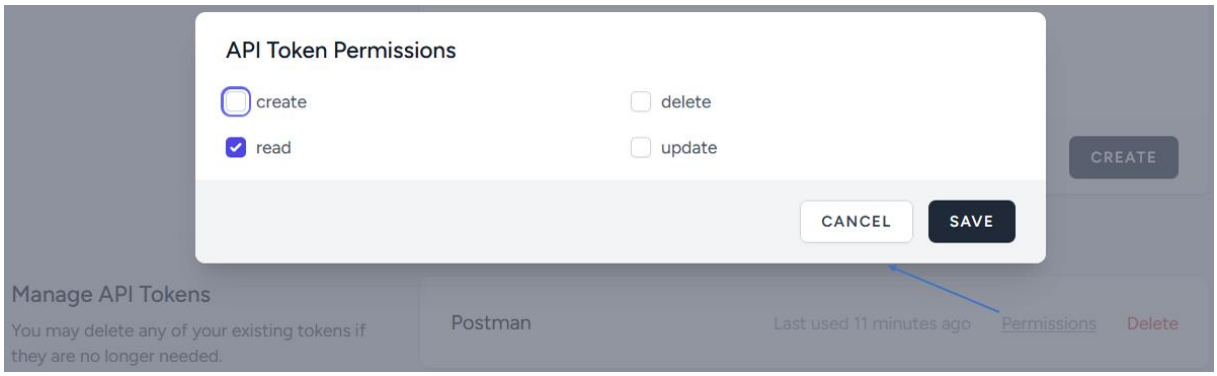
Az iménti kérés végrehajtásával az API Controller **index()** metódusát szólítottuk meg, amely visszaadta a könyvlistát JSON formátumban, mivel a **BookPolicy** osztály **viewAny()** metódusa **true** értékkel tért vissza, tehát nem volt hozzá szükséges semmilyen engedély azon túlmenően, hogy a felhasználó bejelentkezett. Ezt a *„bejelentkezést”* a token felhasználásával értük el az API felületen keresztül.

```
public function index()
{
  $books = Book::where('team_id', auth()->user()->currentTeam->id)->get();
  return BookResource::collection($books);
}
```

11–42. kódrészlet: API Controller index() metódusa

Az 1-es felhasználó token-je van jelenleg a Postman-ben, amely az 1-es csapatot birtokolja, de a 2-es csapat könyveit nem éri el. Érdemesebb inkább a 2-es felhasználóval belépni, akinek a saját 2-es csapatához teljes hozzáférése van, de az 1-esnél is van editor szerepkörű hozzáférése. A 2-es felhasználóval a weboldalon történő belépés után a 10.2.2.4.4. alfejezet szerint másoljuk ki az API token értékét és módosítsuk a Postman *„Authorization”* lapfülön a Token értékét, majd mentsünk. Ha maradunk bejelentkezve azzal a felhasználóval, akinek a másik csapat könyveihez nincsen hozzáférése, akkor az adott lekérésnél (például show), mindig 403-as Unauthorized választ fogunk kapni.

Érdemes továbbá azt is ellenőrizni, hogy az adott felhasználónál az adott API token-nel való hozzáférésnek milyen engedélyei vannak (11–10. ábra), mert hiába szeretnénk majd az adott (akár egy másik, nem feltétlenül az aktuális) csapathoz Postman segítségével könyveket létrehozni, módosítani vagy törölni, ha ezek a beállítások nem lesznek engedélyezve az adott token-nek, akkor el lesznek utasítva a kérések.



11–10. ábra: Adott csapat más alkalmazással való API token-es hozzáférésének engedélyei

11.4.2.2. API Controller show és Policy view metódusok definiálása és tesztelése

Valósítsuk meg a **show()** metódust is:

```
public function show(Book $book)
{
  return BookResource::make($book);
}
```

11–43. kódrészlet: API Controller show() metódusa

Amikor egy-egy darab könyvhöz férünk hozzá az API Controller **show(), update(), destroy()** metódusán keresztül, akkor mindig figyeljünk arra, hogy milyen felhasználóval vagyunk bejelentkezve a Postman kérésben (ezt a **get-user** kérés végrehajtásával tudjuk ellenőrizni, ha oda az *„Authorization"* részbe is ugyanazt a Token értéket másoljuk be, mint ami az érintett kérésnél szerepel). Majd aszerint, hogy kivel vagyunk bejelentkezve, és neki milyen szerepköre van abban a csapatban, ahova az adott könyv tartozik, végezzünk először manuális tesztelési műveleteket.

A Jetstream dokumentáció aktuális részének ajánlása szerint módosítsuk a **show()** Controller metódushoz tartozó **view()** Policy metódus tartalmát:

```
public function view(User $user, Book $book): bool
{
  $team = Team::find($book->team_id);

  return $user->belongsToTeam($team) &&
    $user->hasTeamPermission($team, 'read') &&
    $user->tokenCan('read');
}
```

11–44. kódrészlet: API kérések engedélyezési ellenőrzéséhez módosított view() Policy metódus

Ha olyan Jetstream-alkalmazást készítünk, amely API-támogatást és csapattámogatást is nyújt, akkor az alkalmazás engedélyezési szabályzatában ellenőriznie kell a bejövő kérés csapatjogosultságait és API-token jogosultságait is. Ez azért fontos, mert előfordulhat, hogy egy API-token segítségével elméletileg képes egy művelet elvégzésére a kérés, miközben a felhasználónak a csapatjogosultságain keresztül valójában nincs meg az adott művelet elérésének lehetősége. Így tehát az adott felhasználó jogosultságai

is ellenőrzésre kerülnek a könyv csapatában és az API token-en keresztüli ellenőrzés is megtörténik, hogy van-e engedélye az adott (például itt a megtekintési) művelet végrehajtására.

Duplikáljuk az iménti listázó kérést a Postman-ben:

- Retrieving a book néven mentsük el: GET /api/v1/books/1
- Mivel legalább 10 könyvünk van, így az elsőt biztos le tudjuk így kérni. Ha 403-as választ kapunk, akkor valószínűleg a másik csapat könyvét szeretnénk lekérni, amihez nincsen hozzáférésünk, ha nem jelentkeztünk át a Token megváltoztatásával az ő helyébe.
- Ha megfelelő a felhasználónk szerepköre és a lekért könyv csapata, akkor vissza kell kapnunk a válaszban a könyv id és title mezőjének értékét.

11.4.2.3. API Controller store és Policy create metódusok definiálása és tesztelése

Valósítsuk meg a store() metódust:

```
public function store(StoreBookRequest $request)
{
  $validated = $request->validated();
  $validated['team_id'] = auth()->user()->currentTeam->id;

   return response()->json([
    'data' => BookResource::make(Book::create($validated)),
    'message' => 'Book created'
  ], Response::HTTP_CREATED);
}
```

11–45. kódrészlet: API Controller store() metódusa validálással együtt

Importáljuk hozzá a StoreBookRequest osztályt!

Itt is módosítsuk a létrehozáshoz tartozó create() Policy metódusban lévő engedélyezési vizsgálatot:

```
public function create(User $user): bool
{
  return $user->belongsToTeam($user->currentTeam) &&
    $user->hasTeamPermission($user->currentTeam, 'create') &&
    $user->tokenCan('create');
}
```

11–46. kódrészlet: API kérések engedélyezési ellenőrzéséhez módosított create() Policy metódus

Duplikáljuk az iménti megjelenítési kérést a Postman-ben:

- Storing a book néven mentsük el: POST /api/v1/books
- A kérés *„Body”* részében a *„x-www-form-urlencoded”* fülön adjuk hozzá kulcsként a title-t, értékként egy új könyv címet, például: My new book

A kérés lefuttatása után létrejön az új könyvünk, az új azonosítóval és a megadott címmel. A Postman válasz részében láthatjuk is rögtön ezt a két BookResource által jelzett értéket.

Mivel a 2-es csapathoz tartozó felhasználónkkal vagyunk bejelentkezve, de nála az aktuális csapat azonosítója (current_team_id a users adattáblában) az 1-esnek van beállítva, ezért az új könyvet is az 1-es csapathoz hozta létre a felhasználó.

Ha az API kérésben szeretnénk megváltoztatni a *„bejelentkezett”* felhasználó aktuális csapatának azonosítóját, erre is van lehetőségünk. A Postman-ben lévő kérésben adjuk hozzá a Body-hoz a currentTeam kulcsot, ellentétes értékkel (1 vagy 2), mint amilyen csapathoz létrejött az imént létrehozott könyv.

A kérésben manuálisan megváltoztatott aktuális csapat azonosítót a BookPolicy osztály create() metódus magját változtassuk meg így:

```
$team_id = request('currentTeam') ?? $user->currentTeam->id;
$team = Team::findOrFail($team_id);

return $user->belongsToTeam($team) &&
  $user->hasTeamPermission($team, 'create') &&
  $user->tokenCan('create');
```

11–47. kódrészlet: Aktuális csapat azonosítójának megváltoztatása manuálisan az API kérés által

Ezt a manuális módosítást a korábbi kérésekhez és Policy metódus ellenőrzésekhez is beilleszthetjük. Az ellenőrzést így is végrehajtja a Policy metódus, tehát olyan felhasználó, aki nem tagja a másik csapatnak, vagy nincs oda a létrehozáshoz szükséges engedélye, akkor nem fog az neki menni, 403-as HTTP hibakódot fog kapni.

11.4.2.4. API Controller update és Policy update metódusok definiálása és tesztelése

Valósítsuk meg az update() metódust:

```
public function update(UpdateBookRequest $request, Book $book)
{
  $validated = $request->validated();

  $book->update($validated);

  return response()->json([
    'data' => BookResource::make($book),
    'message' => 'Book updated'
  ], Response::HTTP_OK);
}
```

11–48. kódrészlet: API Controller update() metódusa validálással együtt

Importáljuk hozzá az UpdateBookRequest osztályt!

Itt is módosítsuk a létrehozáshoz tartozó update() Policy metódusban lévő engedélyezési vizsgálatot:

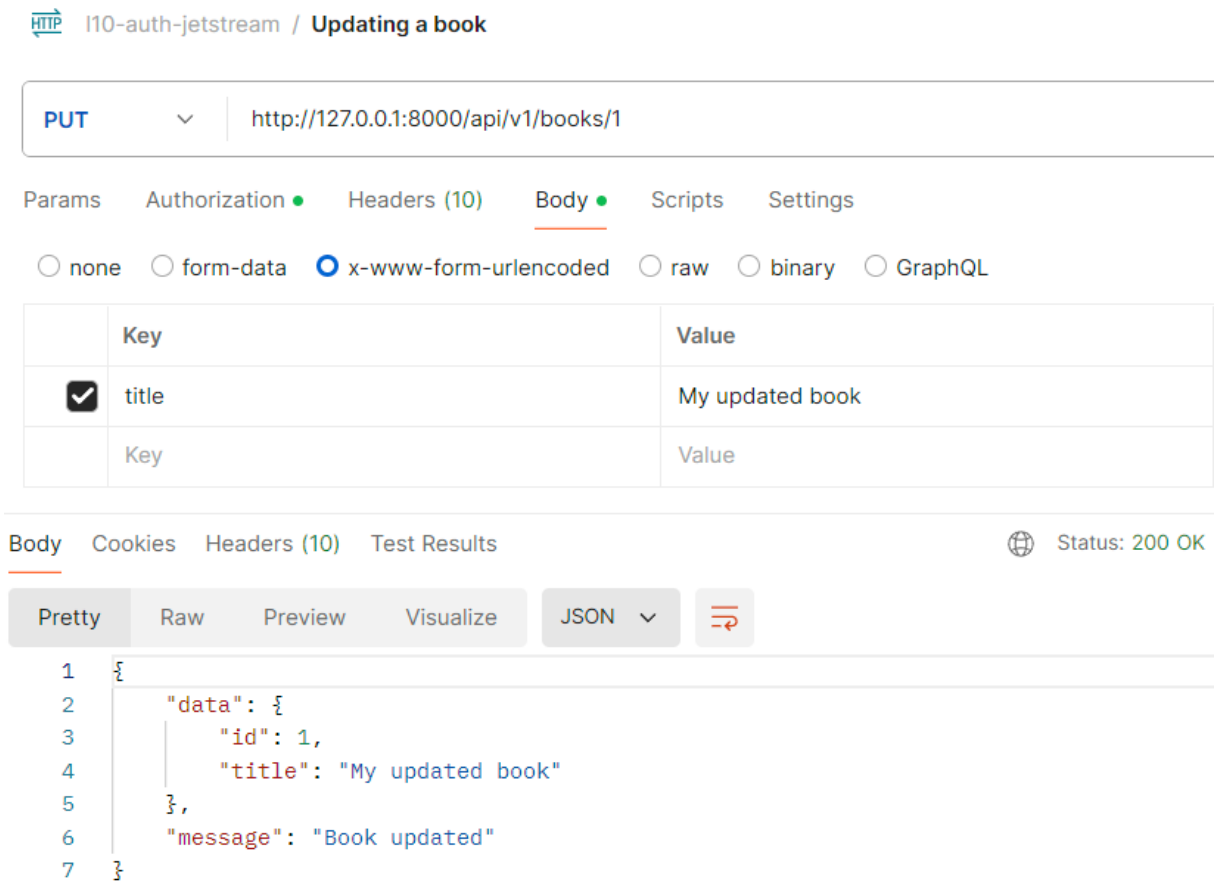
```
return $user->belongsToTeam($team) &&
  $user->hasTeamPermission($team, 'update') &&
  $user->tokenCan('update');
```

11–49. kódrészlet: API kérések engedélyezési ellenőrzéséhez tartozó update() Policy metódus módosított visszatérési értéke

Duplikáljuk a *„Retrieving a book”* (show) kérést:

- **Updating a book** néven: PUT **/api/v1/books/1**
- A kérés törzsében (*„Body”*) válasszuk az *„x-www-form-urlencoded”* lapot, és ott adjuk meg a **title** kulcshoz tartozó új értéket: **My updated book**

Az eredmény megfelelő, működik a könyv frissítése:



11–11. ábra: Sikeres könyv adat frissítés hitelesítés és engedélyezés ellenőrzése, majd validálás után

11.4.2.5. API Controller destroy és Policy delete metódusok definiálása és tesztelése

Valósítsuk meg az **destroy()** metódust:

```
public function destroy(Book $book)
{
  $book->delete();

  return response()->json([
    'message' => 'Book deleted'
  ], Response::HTTP_OK);
}
```

11–50. kódrészlet: API Controller delete() metódusa

Itt is módosítsuk a létrehozáshoz tartozó **delete()** Policy metódusban lévő engedélyezési vizsgálatot:

```
return $user->belongsToTeam($team) &&
  $user->hasTeamPermission($team, 'delete') &&
  $user->tokenCan('delete');
```

11–51. kódrészlet: API kérések engedélyezési ellenőrzéséhez tartozó delete() Policy metódus módosított visszatérési értéke

Duplikáljuk az *„Updating a book"* (update) kérést:

- **Deleting a book** néven: DELETE **/api/v1/books/1**
- A kérés törzséből törölhető a **title** kulcs és értéke is, mivel a törlésnél nincs szükség ezeknek a megadására.
- Csak olyan könyvet tud törölni a felhasználó, amelyik a saját csapatához tartozik, hiába editor szerepkörű a másik csapatban, annak a csapatnak a könyvei közül nem fog tudni törölni egyet sem.

Sikeres törlés esetén visszajelzést kapunk, hogy a könyv törlésre került, és a **books** adattáblából is törlődik az adott sor.

Az alfejezetben végrehajtott programkód módosítások ebben a GitHub commit-ben találhatók meg.

11.4.3. Erőforrás specifikus jogosultság hozzáadása és kezelése

A jelenleg szerepkörökhöz meghatározott jogosultságok meglehetősen általánosak: **read, create, update, delete**. Egy kicsivel nagyobb alkalmazás esetén már biztosan finomhangolásra szorulnak ezek a jogosultsági lehetőségek. Például, ha egy erőforráshoz definiáljuk az iménti négy jogosultságot, máris sokkal jobban testreszabhatóvá válik az alkalmazásunk. Az **app / Providers / JetstreamServiceProvider** osztályban az *admin* és *editor* szerepekhez adjuk is hozzá a következő jogosultságokat:

- **books:read**
- **books:create**
- **books:update**
- **books:delete**

Ha szeretnénk, akkor az utolsó törlést engedélyező jogosultságot továbbra se adjuk meg az *editor* szerepkörű felhasználóknak.

A weboldalon történő bejelentkezés után ezek az új jogosultság értékek rögtön megjelennek a *„Create API Token"* szekcióban, de a *„Manage API Tokens"* szekcióban lévő alkalmazás *„Permissions"* modal ablakában is (mivel a keretrendszer automatikusan felderíti őket). Ez utóbbi felugró ablakban érdemes kipipálni a négy új jogosultságot is.

Ennek megfelelően pedig a **BookPolicy** osztályban lévő összes metódusban a felsorolt jogosultságok elé írjuk oda a *„books:"* szöveget azért, hogy így pontosan erre az adott erőforrásra vonatkozóan ellenőrizze az engedélyeket a rendszer, ne csak egy általánosan meghatározott **read, create, update, delete** jogosultságokra vonatkozóan.

Az alfejezetben végrehajtott programkód módosítások ebben a GitHub commit-ben találhatók meg.

11.4.4. Engedélyezéshez kötött REST API felület automatikus tesztelése

Automatikus teszteket készíthetünk a hozzáférésekhez az iménti manuális tesztek alapján.

Hozzunk létre egy külön tesztelő osztályt az API teszteknek:

```
php artisan make:test BookApiTest
```

Az osztályban helyezzük el a **RefreshDatabase** trait-et, hogy a tesztelési adatbázisba történjenek meg a tesztek adatbázist érintő műveletei. A teszteseteket külön-külön metódusokban definiáljuk, API funkciónként legalább kettőt, hogy a pozitív és negatív eseteket is lefedjük.

11.4.4.1. Automatikus tesztek erőforrások API-on keresztüli lekérésére

A pozitív és negatív tesztekre egy-egy példa alább látható:

+ A csapat tulajdonosa le tudja kérni a könyvek listázási útvonalát és így a könyveket is.
– A látogató nem fér hozzá a könyvek listázásának útvonalához, hanem bejelentkezést vár el tőle az alkalmazás, ezért oda irányítja.

```php
// index +
public function test_owner_can_access_books()
{
  $owner = User::factory()->withPersonalTeam()->create();
  Book::factory(5)->create(['team_id' => $owner->currentTeam->id]);

  $response = $this
    ->actingAs($owner)
    ->get('/api/v1/books');

  $response->assertStatus(200);
}

// index -
public function test_visitor_cannot_access_books()
{
  $response = $this->get('/api/v1/books');
  $response->assertStatus(302);
  $response->assertRedirect('login');
}
```

11–52. kódrészlet: Pozitív és negatív teszt a könyvek API-n keresztüli lekérésére

11.4.4.2. Automatikus tesztek adott erőforrás API-on keresztüli megtekintésére

A pozitív és negatív tesztekre egy-egy példa alább látható:

+ A csapat tulajdonosa le tudja kérni saját csapatának egy könyvét (és adatait).
– Másik csapat tagja, aki az adott csapatban nincs benne, nem fog tudni hozzáférni az adott csapat könyvéhez.

```php
// show +
public function test_team_owner_can_access_books()
```

```
{
  $owner = User::factory()->withPersonalTeam()->create();
  $team_id = $owner->currentTeam->id;
  Book::factory()->create(['team_id' => $team_id]);

  $response = $this->actingAs($owner)->get('/api/v1/books/1');

  $response->assertStatus(200);
}

// show -
public function test_other_team_member_cannot_access_book()
{
  $owner = User::factory()->withPersonalTeam()->create();
  Book::factory()->create(['team_id' => $owner->currentTeam->id]);

  $foreignMember = User::factory()->withPersonalTeam()->create();

  $response = $this->actingAs($foreignMember)->get('/api/v1/books/1');

  $response->assertStatus(403);
}
```

11–53. kódrészlet: Pozitív és negatív teszt adott könyv adatainak API-n keresztüli megtekintésére

11.4.4.3. Automatikus tesztek új erőforrás API-on keresztüli létrehozására

A pozitív és negatív tesztekre egy-egy példa alább látható:

+ Az adott csapatban szerkesztő szerepkörű felhasználó képes létrehozni könyvet.
– Egyszerű (külön szerepkörrel nem rendelkező) felhasználó nem tud könyvet létrehozni idegen csapatban.

```
// store +
public function test_editor_can_create_a_book_in_another_team()
{
  $owner = User::factory()->withPersonalTeam()->create();

  $owner->currentTeam->users()->attach(
    $editor = User::factory()->withPersonalTeam()->create(), ['role' =>
'editor']
  );

  $response = $this->actingAs($editor)->post('/api/v1/books/', [
    'title' => "My first book",
    'team_id' => $owner->currentTeam->id
  ]);

  $response->assertStatus(201);
}
```

```
// store -
public function test_user_cannot_create_a_book_in_another_team()
{
  $owner = User::factory()->withPersonalTeam()->create();
  $user = User::factory()->withPersonalTeam()->create();

  $response = $this->actingAs($user)->post('/api/v1/books/', [
    'title' => "My wrong book",
    'currentTeam' => $owner->currentTeam->id
  ]);

  $response->assertStatus(403);
}
```

11–54. kódrészlet: Pozitív és negatív teszt új könyv API-n keresztüli létrehozására

11.4.4.4. Automatikus tesztek meglévő erőforrás API-on keresztüli frissítésére

A pozitív és negatív tesztekre egy-egy példa alább látható:

+ Egy szerkesztő szerepkörű felhasználó tudja módosítani a másik csapathoz tartozó könyv címét.
– Egy szerepkör nélküli felhasználó nem tudja módosítani a másik csapathoz tartozó könyv címét.

```
// update +
public function test_editor_can_update_the_book_title_in_another_team()
{
  $owner = User::factory()->withPersonalTeam()->create();

  $owner->currentTeam->users()->attach(
    $editor = User::factory()->withPersonalTeam()->create(), ['role' =>
'editor']
  );
  $owner_team_id = $owner->currentTeam->id;
  Book::factory()->create(['team_id' => $owner_team_id]);

  $response = $this->actingAs($editor)->put('/api/v1/books/1', [
    'title' => "My first book",
    'team_id' => $owner_team_id
  ]);

  $response->assertStatus(200);
}

// update -
public function test_user_cannot_update_a_book_in_another_team()
{
  $owner = User::factory()->withPersonalTeam()->create();
  $user = User::factory()->withPersonalTeam()->create();
  $owner_team_id = $owner->currentTeam->id;
  Book::factory()->create(['team_id' => $owner_team_id]);
```

```
  $response = $this->actingAs($user)->put('/api/v1/books/1', [
    'title' => "My wrong book",
    'currentTeam' => $owner_team_id
  ]);

  $response->assertStatus(403);
}
```

11–55. kódrészlet: Pozitív és negatív teszt meglévő könyv címének API-n keresztüli frissítésére

11.4.4.5. Automatikus tesztek meglévő erőforrás API-on keresztüli törlésére

A pozitív és negatív tesztekre egy-egy példa alább látható:

+ A csapat tulajdonosa tudja törölni a csapathoz tartozó könyvet.
– A szerkesztő szerepkörű felhasználó nem tudja törölni a másik csapathoz tartozó könyvet.

```
// destroy +
public function test_team_owner_can_delete_a_book()
{
  $owner = User::factory()->withPersonalTeam()->create();
  Book::factory()->create(['team_id' => $owner->currentTeam->id]);

  $response = $this->actingAs($owner)->delete('/api/v1/books/1');

  $response->assertStatus(200);
}

// destroy -
public function test_editor_from_other_team_cannot_delete_book()
{
  $owner = User::factory()->withPersonalTeam()->create();

  $owner->currentTeam->users()->attach(
    $editor = User::factory()->withPersonalTeam()->create(), ['role' =>
'editor']
  );

  $team_id = $owner->ownedTeams()->first()->id;
  $book_id = Book::factory()->create(['team_id' => $team_id])->id;

  $response = $this->actingAs($editor)->delete('/api/v1/books/' . $book_id);

  $response->assertStatus(403);
}
```

11–56. kódrészlet: Pozitív és negatív teszt meglévő könyv API-n keresztüli törlésére

Az alfejezetben végrehajtott programkód módosítások ebben a GitHub commit-ben találhatók meg.

11.5. Összegzés

A fejezet során legelőször áttekintettünk egy nagyon egyszerű gyakorlati példát az engedélyezés folyamatára azért, hogy átérezzük, hol és hogyan játszik szerepet az engedélyeztetés a hitelesített felhasználók kéréseinek kiszolgálása során.

Ezután bemutatásra kerültek a Laravel engedélyezési technikái (Gate, Policy) az elméleti hátterükkel, fontosságukkal és a gyakorlati megvalósításukkal együtt. Ezeket a technikákat az alfejezetben teszteltük is manuálisan, majd automatikusan.

Utána a szerepkör alapú engedélyezési technikákat tekintettük át elméletben, majd úgy is, hogy a Laravel hitelesítéshez használt kezdő készleteinek eszközeit segítségül hívtuk. Így egy kicsit kibővítettük a Breeze-es projektünket, majd sokkal inkább a Jetstream lehetőségeire koncentráltunk már a fejezet további részében.

Egy, az engedélyeznél gyakran használt, nagyon fontos külső csomag, a Laravel-Permission csak megemlítésre került itt, ugyanakkor a blogomon egy teljes ipari projekt megvalósításánál bemutatásra került ennek a csomagnak a használata. A projekt és a konkrét bejegyzések, amelyek az engedélyeztetéssel foglalkoznak hivatkozásra kerültek itt.

A Jetstream csapatos és szerepkörös lehetőségeinél maradtunk aztán a továbbiakban. Ez egy komplex, de teljes körű megoldást nyújt a csapatokon belüli és azokon is átnyúló, szerepkör alapú engedélyek kezelésére. A gyakorlati feladatok megoldása során aztán egy erre épülő API felületet is definiáltunk, amelynek a hozzáférési pontjait a Postman alkalmazással manuális teszteknek is alávetettük. Utána rögtön automatikus teszteket is létrehoztunk hozzájuk, amelyek pozitív és negatív szempontból egyaránt próbára tették a funkcionalitásokat.

12. A rendszer magjának további építőkövei (Core elements of the framework)

A fejezet során megismerkedünk a Laravel keretrendszer magjának fontosabb elemeivel. Ezek közé tartozik a gyűjtemények, a szolgáltatások, értesítések, események területe. Természetesen a rendszer magjának összes elemét ebben az egy fejezetben nem lehet ismertetni, ezért aki érdeklődik még azután, hogy mi történik még a *„motorháztető alatt”,* az kövesse majd a további munkásságomat a blogomon ezen a címkén keresztül.

12.1. Gyűjtemények (Collections)

Általában a programozási nyelvek nagyobb választékot kínálnak az adatok ideiglenes, memóriában történő eltárolására adatszerkezeteket tekintve. Nincs ez másként a PHP nyelvben sem. Maga a nyelv lehetőséget nyújt számunkra, hogy az adatokat tömbökben, listákban, sorokban, vermekben, hash-táblákban, vektorokban, gyűjteményekben stb. tudjuk eltárolni. Amire pedig a PHP lehetőséget ad, azt a Laravel-ben történő programozás során is bármikor alkalmazhatjuk.

Az adatszerkezetek kiválasztása azért is fontos egy-egy feladat végrehajtása előtt, mivel az alkalmazni kívánt algoritmusok nem mindegyik adatszerkezettel működnek hatékonyan együtt. Az adatszerkezetek és algoritmusok témakörének óriási a szakmai és tudományos irodalma, itt most nem is vállalkoznék arra, hogy ezeknek a mélységét ismertessem, azonban fogok hasznos tanácsokat adni egyszerűbb algoritmusok, úgynevezett programozási tételek végrehajtására a Laravel gyűjteményeihez kapcsolódó segédmetódusokkal.

A Laravel **Collections**, vagyis a gyűjtemények hatékony módot biztosítanak az adattömbökkel való munkára a Laravel-alkalmazásokban. A gyűjtemény egyfajta tömbszerű adatstruktúra, amely a natív PHP tömböt bővíti ki azért, hogy számos hasznos módszert kínáljon az adathalmazokkal való munkához.

A gyűjtemények azért fontosak a keretrendszerben, mert sok Laravel funkcionalitás az egyszerű tömbök helyett gyűjteményekkel tér vissza. Például az Eloquent segítségével az adatbázisból történő lekérdezés az eredményeket **Collection** objektumként adja vissza, ami maga a gyűjtemény, az adathalmaz. Ezután a keretrendszer lehetővé teszi még számunkra az Eloquent, Query Builder és a gyűjteményt lekérdező és manipuláló *metódusok láncolását*. A gyűjtemények ereje a Laravel számos szolgáltatásában megmutatkozik, és ezek által is tűnik annyira hatékonynak a keretrendszer működése.

A gyűjtemények használatának legfőbb előnyei:

- A gyűjtemények elemeit könnyen végig tudjuk iterálni (végig tudunk lépkedni rajtuk) a **foreach** ciklus segítségével, ez annak köszönhető, hogy a Laravel Collections implementálja a PHP **Iterable** interface-ét.
- Tömörebb, átláthatóbb, kifejezőbb kódot tudunk írni az adathalmazok manipulálásához.
- A gyűjtemények módosítása során alapértelmezés szerint nem kerülnek ténylegesen módosításra az eredeti gyűjtemények, hanem a rendszer egy új gyűjteménnyel tér vissza a módosítás eredményeként, amely az eredetire nincs hatással. Például, ha egy számokat tartalmazó

gyűjtemény minden elemét megszorozzuk egy számmal, akkor az eredeti gyűjtemény nem feltétlenül változik meg, így a gyűjtemények kezelése során előforduló lehetséges hibák száma is minimális lesz.

- Segédmetódus készlet: a Laravel rengeteg segédmetódust biztosít az adatok különböző módokon történő lekérdezéséhez és manipulálásához. Jó néhányat mi is meg fogunk ismerni példákon keresztül, de itt van az összes elérhető metódus listája, ha pedig az adott metódus nevére rákattintunk, akkor a lapon belül odairányít minket a weboldal egy rövid leíráshoz és példához, hogy rögtön egyértelművé váljon, mit tud az adott metódus (bár a nevük is meglehetősen kifejező);
- Metódusokat képesek leszünk láncolni, ezáltal egymás után több műveletet is végrehajtunk az adathalmazon azért, hogy aztán végül a kívánt eredményt kapjuk meg akár egyetlen kódsor begépelésével. Ezáltal egy jól olvasható kódot is kapunk.
- Ami a szerver oldalon tömbszerű működést jelent, az a kliens oldal felé, amikor API-t használunk már JSON-nak tűnik, de ezt mindenfajta konvertálás nélkül végzi el a rendszer, így a használata a programozók életét még inkább megkönnyíti.

A gyűjtemények a Laravel keretrendszer magjában vannak, élnek és működnek, így nincs szükség semmilyen további csomag telepítésre a használatukhoz. Léteznek persze olyan csomagok (például: Spatie: Laravel collection macros), amelyek még több szolgáltatással egészítik ki az amúgy is meglehetősen jól használható gyűjteményeket. Ezeket igény szerint telepíthetjük, ha valamilyen fontos funkció hiányozna nekünk, amelyet a csomag pedig nyújt számunkra.

A gyűjtemények témaköre nem feltétlenül újdonság a számunkra, hiszen amikor például adatbázisból kértünk le több adatsort, akkor az egy gyűjteményként érkezett vissza hozzánk, és tudtuk azt kezelni, lekérdezni különböző módokon. A nézeteknél is használtuk már őket, amikor egy adathalmazt adtunk át a nézetnek és végeztük el a felsorolást bennük a **@foreach** Blade direktívával, vagy amikor a **@selected** direktívát használtuk, és benne a feltételt **pluck()** és **contains()** gyűjtemény segédmetódusokkal valósítottuk meg. Az útvonalaknál és a Controller-ekben is használtunk gyűjteményeket, amelyeket az **only()** és **except()** metódusokkal manipuláltunk. A gyakorlati példák megvalósítása során további gyűjteményeket kezelő segédmetódusok használatával ismerkedünk meg ebben az alfejezetben

A gyakorlásra leggyakrabban a Tinker eszközt fogjuk használni utasítások segítségével. Ehhez indítsuk el például az **l10-components** projektünket, majd ott a Tinker-t.

12.1.1. Gyűjtemények létrehozása

Gyűjteményeket manuálisan a **collect()** segédmetódussal hozhatunk létre. Nem érzékeny a különböző típusok eltárolására, ahogy egy PHP-s tömb sem az. Az alábbiakban néhány példát tekinthetünk meg utasításokon keresztül, majd utána rögtön a lefutás eredményével.

```
$collection = collect();

$collection = collect([1, 2, 3]);
```

12–1. utasítások: Gyűjtemények létrehozása, elemek hozzáadása

Először egy üres gyűjteményt hoztunk létre, majd egy tömböt alakítottunk gyűjteménnyé, így sokkal több segédmetódus lesz elérhető hozzá. Mindez látható az alábbi ábrán is:

```
> $collection = collect();
= Illuminate\Support\Collection {#6190
    all: [],
  }

> $collection = collect([1, 2, 3]);
= Illuminate\Support\Collection {#6188
    all: [
      1,
      2,
      3,
    ],
  }
```

12–1. ábra: Gyűjtemények létrehozása

Gyűjteményt készíthetünk asszociatív tömbökből is, majd akár azokat is bővíthetjük.

```
$products = collect([ ["name" => "apple", "price" => 10], ["name" => "cherry", "price" => 8], ["name" => "banana", "price" => 11], ["name" => "mango", "price" => 7] ]);
```

12–2. utasítások: Gyűjtemény készítése asszociatív tömbből

Az utasítás eredménye a következő ábrán látható:

```
= Illuminate\Support\Collection
    all: [
      [
        "name" => "apple",
        "price" => 10,
      ],
      [
        "name" => "cherry",
        "price" => 8,
      ],
      [
        "name" => "banana",
        "price" => 11,
      ],
      [
        "name" => "mango",
        "price" => 7,
      ],
    ],
  }
```

12–2. ábra: Gyűjtemény készítése asszociatív tömbből

Ahogy említésre került, az adatbázis lekérdezések, legyenek azok akár az Eloquent, akár a Query Builder által, (speciális Eloquent) gyűjteményekkel térnek vissza, ha több elemet (Model objektumot) tartalmaz az eredmény halmaz.

```
$categories = App\Models\Category::all();
```

12–3. utasítások: Adatbázis lekérdezés eredménye is (kibővített) gyűjtemény

Az utasítás eredménye a következő ábrán látható:

12–3. ábra: Model objektumok gyűjteménye az eredménye egy adatbázis lekérdezésnek

12.1.2. Gyűjtemény metódusok használatának esetei

Ebben az alfejezetben számos példát áttekintünk arra vonatkozóan, hogy az egyes gyűjtemény segédmetódusok hogyan és miképpen segítik a velük való munkát.

12.1.2.1. Elemek hozzáadása

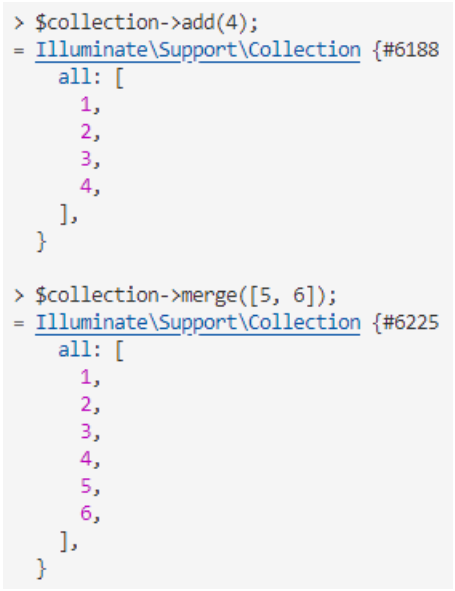
A létrehozott gyűjteményhez elem(ek)et is hozzáadhatunk.

```
$collection->add(4);
```

```
$collection->merge([5, 6]);
```

12–4. utasítások: Elem hozzáadása a gyűjteményhez

Az **add()** (vagy a **push()**) segédmetódussal egy elemet tudunk hozzáadni a gyűjteményhez, a **merge()** segédmetódussal gyűjteményünket egy tömbbel fésültük össze, így eredményül egy bővített gyűjteményt kaptunk. Mindez látható az alábbi ábrán is:



12–4. ábra: Gyűjtemények létrehozása és további adatok hozzáadása

A **merge()** segédmetódus az eredeti gyűjteményt nem módosítja, így az továbbra is az 1, 2, 3, 4 elemekből fog állni, viszont visszaad a **merge()** egy új gyűjteményt, ami az imént felsoroltakon kívül az 5 és 6 elemeket is tartalmazza. Ha a kibővített gyűjteménnyel szeretnénk tovább dolgozni, akkor az utasítást értékül kell adni egy másik változónak, így a generált gyűjteményt kapná meg értékül az új elem, egyelőre azonban maradjunk a szűkebb (4 elemű) gyűjteményünknél.

Megjegyzés: az utasítások eredményeit a továbbiakban nem jelenítem meg, hacsak nem valamilyen nagyon különleges dolgot jelezne számunkra a rendszer. Emiatt is érdemes az utasításokat mindenkinek magának végrehajtania, hogy lássa is az eredményeket a magyarázatokon túlmenően.

12.1.2.2. Gyűjtemény elemeinek elérése, törlése, léptetése

Konkrét helyen lévő gyűjtemény elemhez való hozzáférés a **first()** és **last()** segédmetódusokkal lehetséges, így kaphatjuk meg az első és utolsó gyűjtemény elemet.

```
$collection->first();

$products->last();
```

12–5. utasítások: Kiemelt jelentőségű gyűjtemény elemekhez való hozzáférés és lekérés

A gyűjtemény ugyanúgy indexelhető, mint a tömb, tehát az első gyűjtemény elem indexe **0**, az utolsóé pedig a **gyűjtemény mérete - 1**. Ha adott indexű elemhez szeretnénk hozzáférni a gyűjteményben, akkor a **get()** segédmetódus is használható erre.

```
$categories[4];

$collection->get(3);

$products->get(2);
```

12–6. utasítások: Adott gyűjteményelemek lekérése

A hozzáadást már megtekintettük, de törölni is tudunk a gyűjteményből az indexe alapján a **forget()** segédmetódus segítségével. A **pop()** segédmetódus a gyűjtemény utolsó elemét fogja törölni, ha a metódus paraméterében egy számot kap meg, akkor annyi elemet töröl a gyűjtemény végéről, amennyit paraméterül kapott.

```
$products->add( ["name" => "grape", "price" => 5 ] );

$products->forget(4);

$collection->push(100);

$collection->pop();
```

12–7. utasítások: Új gyűjtemény elemek hozzáadása majd törlése

A gyűjteményen végig tudunk haladni (iterálni) ciklusok nélkül is, ehhez az **each()** segédfüggvényt kell használnunk. A Laravel gyűjteményekben az **each()** metódus sokoldalú megközelítést kínál. Megismétli a gyűjtemény minden egyes elemét (például a **$products** gyűjteményben a **$product** elemek), és végrehajt egy adott műveletet, az alábbi példában egy egyszerű kiíratást. Ez a módszer kényelmes módot biztosít arra, hogy minden elemen műveleteket hajtsunk végre anélkül, hogy kifejezetten ciklust használnánk.

```
$products->each( function ($product) { echo $product["name"] . ": " . $product["price"] . "\n"; });
```

12–8. utasítások: Ciklus működtetése az each() segédmetódussal

12.1.2.3. Matematikai és adatbázis lekérdezés eredményeknél használható segédmetódusok

Ebben az alfejezetben leginkább azokra az utasításokra érdemes gondolni, amelyet a legtöbb programozási nyelv alapértelmezetten támogat megvalósítással, de az adatbázis lekérdezések (SELECT) is tartalmaznak ilyen úgynevezett aggregáló függvényeket: **min(), max(), sum(), avg(), count().** Mondhatnánk, hogy már ezek is algoritmusok, amelyeket a gyűjtemény adatain hajtunk végre, de a későbbiekben azért ennél bonyolultabbakat is fogunk majd csinálni, ámde mégis egyszerű módon.

```
$collection->min();

$collection->max();

$collection->sum();

$collection->avg();

$collection->count();
```

12–9. utasítások: Aggregáló függvények meghívása a számokat tartalmazó gyűjteményen

Így megkapjuk a korábban definiált számokat tartalmazó gyűjtemény minimumát (1), maximumát (4), az elemek összegét (10), átlagát (2.5) és a darabszámát (4).

Az asszociatív tömbünkből készített gyűjteményen és az adatbázis lekérdezés eredményén is értelmezhetjük ezeket a metódusokat, például így:

```
$products->pluck("price")->avg();

$categories->count();
```

12–10. utasítások: Termékek átlagos árának lekérése

Ezzel a termékek átlagos árát (9) kapjuk vissza, illetve utána a kategóriák darabszámát (8).

Ha az alapvető algoritmusmintákat, vagyis az úgynevezett programozási tételeket tekintjük, akkor ezekkel az egyszerű függvényekkel néhányat már képesek is vagyunk megvalósítani:

- minimumkiválasztás (legkisebb elemet adja vissza a gyűjteményből),
- maximumkiválasztás (legnagyobb elemet adja vissza a gyűjteményből),
- összegzés (a gyűjteményben lévő összes elem összegét adja vissza).

12.1.2.4. Kiválogatás, szűrés, csoportosítás, rendezés

Ha a gyűjtemény bizonyos elemeire van szükségünk, akkor azokat kiválogatással vagy szűréssel tudjuk kinyerni. A kiválogatás a **filter()** függvénnyel valósítható meg, a szűrésre – csak úgy, mint az SQL lekérdezések vagy az Eloquent lekérdezések esetén – a **where()** függvénnyel történhet meg.

```
$collection->filter( function ($item) { return $item % 2 == 0; } );

$collection->filter( fn ($item) => $item % 2 == 0 );

$categories->filter( fn($category) => Illuminate\Support\Str::startsWith($category->name, 'd') );
```

12–11. utasítások: Kiválogatás: páros elemek (hosszabban és rövidebben)

Az iménti első két utasítás ugyanazt hajtja végre, csak az első hosszabban van leprogramozva (megfogalmazva), míg a második rövidebben. Az **$item** változó mindig egy gyűjteménybeli elemet képvisel, ahogy a **filter()** végigmegy a gyűjteményen. A harmadik utasítással a kategóriákat szűrjük meg, méghozzá úgy, hogy a nevüket vizsgáljuk, és azt nyerjük ki így, amelyik **d** betűvel kezdődik (az én kategóriáimnál ebből van több, így látható érdemi eredmény). A szöveges vizsgálatokban az **Illuminate\Support\Str** osztály segít minket. Több tucat további segédmetódussal támogat minket a Laravel keretrendszer (a PHP nyelvi elemeit kiegészítve), ha *string*-ekkel szeretnénk hatékonyan dolgozni. A segédmetódusok listája itt érhető el.

```
$filteredProducts = $products->filter( fn ($item) => $item["price"] > 9 );

$products->where( "price", '>', 9 );
```

12–12. utasítások: Szűrések filter() és where() függvénnyel

A **$products** gyűjtemény nem változik meg a szűrés hatására, de eredményül egy szűrt gyűjteményt ad vissza, amely így (**$filteredProducts**) már csak azokat az értékeket tartalmazza, amelyek a szűrési feltételnek megfelelnek.

A **where()** metódussal is tudtuk szűrni a gyűjtemény elemeit.

Általában elmondható, hogy amilyen metódusokat megismertünk az Eloquent kapcsán, mint például a **groupBy()** csoportosításra, vagy a **sort(), sortDesc(), sortBy(), sortByDesc()** stb. metódusok pedig rendezésre használhatók, azok a gyűjteményeknél is mind-mind használhatók. A teljesség igénye nélkül néhány példát tekintsünk meg az alábbiakban:

```
$collection->sortDesc();

$products->sortBy("name");

$products->sortByDesc("price");
```

12–13. utasítások: Gyűjtemények különböző rendezései

Az első utasítás a számokat rendezi csökkenő sorrendbe, a második ABC sorba rendezi a termékeket nevük szerint, a harmadik utasítás a termékeket ár szerint csökkenő sorrendbe rendezi.

12.1.2.5. Eldöntés és optimista eldöntés (mind eldöntés)

Az eldöntés tételt a **some()** segédmetódussal tudjuk megvalósítani. Ez azt vizsgálja, hogy a gyűjtemény legalább egy eleme megfelel-e a meghatározott feltételnek, ha igen, akkor igaz eredményt ad vissza, ha egy sem felel meg neki, akkor hamis eredményt ad vissza. A Laravel-ben a **some()** metódus egy fedő metódusként szolgál a **contains()** metódushoz, de ugyanúgy működnek.

```
$collection->contains( function ($value, $key) { return $value > 2; });

$collection->contains( fn ($value) => $value < 5 );

$collection->some( fn ($value) => $value > 2 );

$products->some( fn ($value, $key) => $value["price"] == 8 );
```

12–14. utasítások: Eldöntés tétel alkalmazása a contains() és some() metódusokkal

Az első utasítás megvizsgálja, hogy van-e 2-nél nagyobb szám a gyűjteményben, a második utasítás azt vizsgálja, hogy van-e 5-nél kisebb, csak rövidebben implementálva. Ezekhez a **contains()** metódust használtuk, de utána a **some()** metódus is meghívásra került, a feltételek itt is adottak voltak, az utolsónál az adott termék árát vizsgáltuk, hogy van-e 8 egységbe kerülő.

Az optimista eldöntés tételt az **every()** segédmetódussal tudjuk megvalósítani. Ekkor minden gyűjtemény elemre elvárás, hogy megfeleljenek a feltételnek, ha nem így lenne, akkor hamis eredményt ad vissza, de ha így van, akkor igazat.

```
$collection->every( fn ($value) => $value > 2 );
```

12–15. utasítások: Optimista eldöntés vizsgálata (minden elemre vonatkozik a feltétel)

12.1.2.6. Másolás, sorozatszámítás (összegzés)

A **map()** metódus iterálja a gyűjteményt, és minden egyes új értéket átad a megadott visszahívásnak (callback). A callback szabadon módosíthatja az elemet, és visszaadhatja azt új elemet, így egy új gyűjteményt alkotva a módosított elemekből. Ezzel a metódussal egy másolást is megvalósíthatunk: az esetek többségében valamilyen kalkulációt végzünk a gyűjtemény elemeivel, és azt adjuk át egy új gyűjtemény új elemeinek. Hosszabb utasítást több sorban is bemásolhatunk a Tinker felületére, akkor is helyesen értelmezni fogja egy Enter megnyomása után.

```
$multiplied = $collection->map(function (int $item, int $key) {
  return $item * 2;
});
$squaredNumbers = $collection->map(fn ($number) => $number * $number );
```

12–16. utasítások: Gyűjtemény elemek manipulálása a map() metóduson belül, az eredmény átadása egy új gyűjteménynek

Az első, több soros utasítás egy új gyűjteményt hoz létre úgy, hogy hozzáadogatja a **map()** minden egyes lefutásakor az adott gyűjtemény elem dupláját. A **$key** érték az elemhez tartozó kulcsot jelöli, ha van ilyen kulcs (használata és jelölése opcionális). A második utasításban a gyűjtemény elemeiből négyzetszámokat generálunk, és azt adjuk hozzá egy új gyűjteményhez (**$squaredNumbers**).

A következő gyűjtemény segédmetódus, a **reduce()** a sorozatszámításnál hasznos, amikor valamiket összegezni szeretnénk, például: ha egy számlát áttekintünk, és a számlasorokban szereplő értékeket szeretnénk összeadni egyesével, hogy kijöjjön a számla végösszege.

```
$total = $collection->reduce(function (?int $carry, int $item) {
  return $carry + $item;
});
$totalPrice = $products->reduce( fn ( $carry, $product ) => $carry + $product["price"] );
```

12–17. utasítások: Sorozat elemeinek összegzése a reduce() segédmetódussal

Az első, több soros utasítás a **$collection** számokat tartalmazó gyűjtemény elemeit adja össze. a **$carry** (a metódus első) paramétere arra szolgál, hogy az iteráció során ebben tároljuk el az összegzés aktuális értékét, tehát mint ha a számla sorainak részösszegét tárolnánk el ebben, és az új értékeket mindig hozzáadjuk, végül pedig kijön egy végösszeg. Utána a második utasítás egy rövidített megfogalmazása ennek, ahol a gyűjteményben meglévő termékek árának összegét számoljuk így ki, és adjuk vissza egy **$totalPrice** változó értékének.

12.1.3. Metódusok összefűzése

A gyűjteményeket manipuláló segédmetódusok összefűzése rendkívül hatékony tud lenni, emiatt sokszor alkalmazzuk a való életben történő szoftverfejlesztés során. A programkódot is meglehetősen olvashatóvá

tudja tenni, amivel így a végső eredmény kvázi *„felolvasható”,* hogy hogyan alakult ki, és jött létre a kezdeti gyűjteményből a manipulálások után az új gyűjtemény.

Az alábbiakban olyan utasítássorozatot láthatunk, amely először megszűri a számokat tartalmazó gyűjteményt azáltal, hogy páros-e, ha pedig igen, akkor megszorozza 3-mal az adott elemet, amit visszaad egy új gyűjteménynek eltárolásra. Utána a páratlan számokat szűrjük ki és szorozzuk meg néggyel a példa kedvéért, az így kialakult új gyűjteménynek pedig vesszük a legutolsó elemét, ami a végeredménye lesz ennek az összefűzésnek.

```
$items = collect([ 1,2,3,4,5,6,7,8,9,10 ]);

$filteredNumbers = $items->filter(function ($item) { return $item % 2 == 0; })->map(function ($item) { return $item * 3; });

$items->filter(fn ($item) => $item % 2 == 1 )->map(fn ($item) => $item * 4 )->last();
```

12–18. utasítások: Gyűjteményt manipuláló segédmetódusok összefűzése (példák)

Ha nem lenne egyértelmű, hogy miképp és hogyan működik a metódusok összefűzése, akkor a tanulási és gyakorlási fázisban megtehetjük azt is, hogy szépen felépítjük egymás után az összefűzéseket, és megfigyeljük azt, mi történik az adott új metódus hozzáfűzése után.

```
$items->filter(fn ($item) => $item % 2 == 1 );

$items->filter(fn ($item) => $item % 2 == 1 )->map(fn ($item) => $item * 4 );

$items->filter(fn ($item) => $item % 2 == 1 )->map(fn ($item) => $item * 4 )->last();
```

12–19. utasítások: Gyűjteményt manipuláló segédmetódusok összefűzése lépésről-lépésre a jobb megértéshez

Mivel a Tinker-t az **l10-components** projektünkben használjuk, ezért itt rendelkezésünkre állnak a projekt elemei, például a kategóriákat már többször használtuk a példák során. Nézzünk meg ennek kapcsán néhány metódus összefűzést, amelyek hasznosak lehetnek a munkáink során:

```
$posts = App\Models\Post:: get();

$posts->pluck('tags');

$posts->pluck('tags')->collapse();

$posts->pluck('tags')->collapse()->pluck('name');

$posts->pluck('tags')->collapse()->pluck('name')->unique();
```

12–20. utasítások: Eloquent-es gyűjtemény manipulálása, lekérése

Az első utasítással lekérjük az összes blogbejegyzést, ebben semmi újdonság nincsen. A második utasítás egy nagyobb gyűjteményt ad vissza, amiben szintén további beágyazott gyűjtemények vannak, amik tartalmazzák a Tag objektumokat, amelyek a blogbejegyzésekhez tartoznak, hiszen minden blogbejegyzésnek több Tag-je is van (vagy lehetségesen van). A harmadik utasítással a **collapse()** metódus bejárja azokat a Tag-eket, amelyek kapcsolódnak blogbejegyzéshez, és kiemeli ezeket az eredményül

kapott gyűjtemény legfelsőbb szintjére. Utána az újabb **pluck()** hívással megkapjuk ezeknek a blogbejegyzéshez kapcsolódó Tag-eknek a neveit. Végül pedig a **unique()**-val egy olyan hívást hajtunk végre, hogy minden blogbejegyzéshez kapcsolódó Tag-nek a neve csak egyszer szerepeljen az eredményül kapott gyűjteményben.

```
$posts->pluck('tags')->collapse()->pluck('name')->unique();

$posts->pluck('tags.name');

$posts->pluck('tags.*.name');

$posts->pluck('tags.*.name')->collapse()->unique();
```

12–21. utasítások: Segédmetódusok (pluck()) összevonása

Az iménti utasításokban azt láthatjuk, hogy mivel a **pluck()** metódus hívás az eddigi végső utasításban duplán történt meg, ezért ezt össze szerettük volna vonni. A **pluck('tags.name')** azonban nem adott jó eredményt, csak null értékeket adott vissza, mivel a beágyazott gyűjtemények struktúrájában nem mentünk le a megfelelő szintre. A **pluck('tags.*.name')** viszont már jó lesz nekünk, megfelelő beágyazási szintre jutunk. Utána már következhet – a megfelelő **pluck()** összevonásnak köszönhetően – a **collapse()** és **unique()** metódusok hozzáfűzése.

Ez az összetett példa is talán érzékeltetni tudja, hogy mennyire hasznos tud lenni az Eloquent gyűjteményeknél is a segédmetódusok használata összefűzéssel.

Tipp: Ami a Tinker-ben működik, annak a programkódjainkban is működnie kell (és fordítva is igaz ez)!

Ezért nyugodtan használjuk bonyolultabb metódusok összefűzésére először a Tinker-t, aztán ha úgy ítéljük meg, hogy az eredmények már jók, akkor átültethetjük azt a programkódjainkba. Ha a Tinker helyett ott a programkódjainkban szeretnénk teszteléseket végrehajtani, akkor használhatjuk a **dump()** vagy a **dd()** metódusok hozzáfűzését ehhez a metódus összefűzési lánchoz, így rögtön láthatóvá válik, hogy milyen aktuális eredményhalmazt vagy -értéket ad vissza a metódusok egymás utáni végrehajtásának összessége.

12.1.4. Gyűjtemények kezelésének hatékonysága

Nagy gyűjteményekkel való munka során érdemes figyelni a lekérdezés gyorsaságára és a memóriabeli korlátozásokra. Szükség esetén meg kell fontolnunk az adatok feldarabolásának és a szűrésének (ezt már alkalmaztuk korábban) a lehetőségét.

A gyűjtemények lehetővé teszik, hogy a nagyobb adathalmazokat darabokban dolgozzuk fel anélkül, hogy a teljes adathalmazt betöltené a memóriába a rendszer. Az olyan módszerek, mint a Laravel-es **chunk()** és a **lazy()** lehetővé teszik annak meghatározását, hogy hány rekorddal dolgozzunk egyszerre.

Amikor a **get()**-tel kértünk le adatokat az Eloquent vagy a Query Builder segítségével, akkor az egyben betöltötte az adatokat a memóriába, amely gyorsan történt meg, de nagy adathalmazok esetén nagy erőforrás (memória) terheléssel járt. A **get()** metódus a PHP-s **fetchAll()** funkciót használja a háttérben.

Ezért ez akkor hatékony, ha kisebb adathalmazokkal dolgozunk. A **get()** metódus használatára már néztünk korábban példákat többször (például a 12–6. utasítások).

A **chunk()** metódus a **get()** metódushoz hasonlóan a **fetchAll()** metódust használja az adatok kinyerésére. Azonban ahelyett, hogy az összes rekordot egyszerre hívná le és töltené be a memóriába, az adatokat kisebb, kezelhető *„darabokra"* bontja. Ez a megközelítés drámaian csökkenti a memóriafogyasztást, különösen nagyobb adathalmazok esetében, mivel a megadott darabméret alapján korlátozza az egyszerre lekérdezett rekordok számát. Annak ellenére, hogy lassabb, mint a **get()**, a **chunk()** olyan esetekben, ahol a memóriamegtakarítás kiemelkedően fontos, kiválóan használható.

```
$items->chunk(3);
```

12–22. utasítások: Gyűjtemény feldarabolása a kevesebb memóriahasználatért

Óriási gyűjteményeink egyelőre nincsenek, de a példa jól érthető így is: eredményül gyűjtemények gyűjteményét kapjuk vissza, méghozzá a 10 darab számot 4 darab gyűjteményre bontja így a rendszer, az első háromban 3 elem van (ezt határoztuk meg a **chunk()** paraméterében), az utolsóban 1 darab elem van.

```
= Illuminate\Support\Collection {#7092
    all: [
      Illuminate\Support\Collection {#7371
        all: [
          1,
          2,
          3,
        ],
      },
      Illuminate\Support\Collection {#7380
        all: [
          3 => 4,
          4 => 5,
          5 => 6,
        ],
      },
      Illuminate\Support\Collection {#7381
        all: [
          6 => 7,
          7 => 8,
          8 => 9,
        ],
      },
      Illuminate\Support\Collection {#7382
        all: [
          9 => 10,
        ],
      },
    ],
  }
```

12–5. ábra: A chunk() metódus futtatásának eredménye egy gyűjteményen

A további segédmetódusok működésben nagyon hasonlóak a **chunk()**-hoz erőforrás kezelésben, viszont az adatok betöltése nem azonnal történik meg, hanem csak részletekben. A **cursor()** metódus egy LazyCollection példányt ad vissza. A **cursor()** metódus a PHP **fetch()** függvényét használja a rekordok egyesével történő lekérdezésére az adatbázis táblájából. Ennek eredményeképpen kevesebb memóriát használ, mint a **get()**, így nagyobb adathalmazok kezelésénél hatékonyabb. Ennek azonban a sebesség az

ára: a rekordok egyenkénti iterációja miatt lassabb. A **cursor()** fontos előnye a konzisztencia a feldolgozás során esetlegesen változó adathalmazok feldolgozásakor. Itt láthatunk példát a **cursor()** használatára.

A Laravel 8-ban bevezetett (és azóta is használt) **lazy()** metódus a **chunk()**-hoz hasonlóan kis szegmensekben kér le rekordokat. Ahelyett azonban, hogy visszahívást (callback) igényelne, a **lazy()** PHP generátorokat[21] használ. Ez egyszerűsített szintaxist eredményez, mivel egy LazyCollection-t ad vissza a kényelem érdekében. A **lazy()** módszer egyensúlyt teremt a hatékony memóriahasználat és az olvashatóság között. Itt láthatunk a használatára példát.

12.1.5. Gyűjtemények… gyűjtemények mindenütt…

Ahogy az említésre került, a gyűjtemények a Laravel keretrendszerben mindenhol jelen vannak és előnyeiknek köszönhetően mindenhol hasznos, könnyen programozható megoldásokat eredményeznek. Ebben az alfejezetben az adatlekérés, nézet működtetés és az API hívások kezelése során megjelenő gyűjteményekre, illetve azok kezelésére irányul a fókusz.

12.1.5.1. Eloquent és Query Builder: adatbázisból lekért adatok manipulálása

Már sokszor találkoztunk ezzel a területtel, még ha nem is hangsúlyoztuk ki akkor, hogy gyűjteményekkel dolgozunk: amikor az adatbázisból több rekordot lekérünk, akkor az eredményhalmazt egy gyűjteményként kapjuk vissza. Így nem csak egy objektumot vagy objektumok halmazát kapjuk meg, ami már nem csak egy egyszerű tároló, hanem a gyűjtemény maga rendelkezésünkre bocsát több tucat olyan metódust, amelyekkel tudjuk formálni és lekérdezni a megkapott adathalmazt vagy egyes részeit.

Minden olyan adatlekérő módszer, amely az eredményét az **Illuminate\Database\Eloquent\Collection** osztály példányaiként adja vissza, az egy speciális gyűjteményként kezelhető. Ezek a lekérdezések (amikor **get()** metódussal zárult egy adatlekérés) a Laravel alapvető gyűjteményeinek egy olyan kiterjesztett halmazával térnek vissza, amelyek a több tucat elérhető metóduson *túl*, még Eloquent-specifikus metódusokat is elérhetővé tesznek a fejlesztők számára. Ezeknek a metódusoknak a listája és néhány egyszerű példa rájuk itt érhető el. A **get()** metódushívást már rengetegszer alkalmaztuk, amikor adatokat kértünk le a 6.2.2.5. alfejezettől kezdődően, illetve az **l10-components** projektünk példa útvonalai között is számos olyan szerepel, amelyek ilyen módon kérték le az adatbázistáblák tartalmait.

12.1.5.2. Nézetek: gyűjteményeket kezelő Blade direktívák

A nézetekben már többször használtuk a **@foreach** Blade direktívát arra, hogy végig lépkedjünk a gyűjtemények elemein. Ezen kívül azonban léteznek még más gyűjtemény bejárási ciklusok is. Használhatjuk például a **@for-@endfor** párost, amelyben a ciklusváltozót először inicializálnunk kell, majd feltételnek megfeleltetnünk és léptetnünk is kell. Emiatt ez egy picit kényelmetlenebb, mint az egyszerű

[21] A generátor lehetővé teszi, hogy olyan kódot írjunk, amely foreach segítségével végigmegy egy adathalmazon anélkül, hogy egy tömböt kellene létrehoznia a memóriában, ami miatt túllépheti a memóriakorlátot, vagy jelentős mennyiségű feldolgozási időt igényelne a generálás. Ehelyett írhatunk generátorfüggvényt, amely ugyanolyan, mint egy normál függvény, azzal a különbséggel, hogy a generátor ahelyett, hogy egyszer térne vissza, annyiszor adhat vissza, ahányszor csak szükséges, hogy az iterálandó értékeket megadja. Így a memóriahasználatot a töredékére csökkenthetjük le, mint ha nem generátor függvényként valósítanánk meg a működést.

@foreach-es bejárás, viszont, ha az elemnek a gyűjteményben elfoglalt helye is érdekes számunkra, akkor a ciklusváltozó ennek megtalálásában segíthet minket (*megjegyzés:* most már a @foreach-ben is a $loop változó segítségével tudjuk követni a ciklus lefutásának paramétereit: hányadszor fut le, mennyi van még hátra stb., ehhez lásd a következő bekezdést). Ilyen feladat lehet például maximális vagy minimális elem helyének megtalálása (ha a nézetben szeretnénk ezzel foglalkozni az üzleti logikai elemek helyett).

A $loop változón keresztül a gyűjtemény első ($loop->first) és utolsó ($loop->last) eleme is lekérhető a @foreach cikluson belül, vagy éppen az @if-fel történő feltételvizsgálatnál lehetnek ezek hasznosak. A $loop->index változóval a ciklus aktuális lefutásának számlálóját érjük el, mint a @for-nál a ciklusváltozóval valósítható meg. A ciklusváltozó további hasznos paraméterei itt érhetők el.

Rendelkezésünkre áll még a @forelse-@endforelse páros is, amit szintén használtunk már (például a korábbi 7–31. kódrészletben). Ez nagyon hasonlóan működik, mint a @foreach, de vele szemben annyi pluszt jelent a számunkra, hogy ha a nézetnek átadott gyűjteményben egyetlen elem sincs (üres), akkor a záró direktíva elé beszúrhatunk egy @empty direktívát és ott megadhatjuk neki, hogy mi jelenjen meg akkor, ha a nézetben megkapott gyűjtemény nem tartalmaz egyetlen elemet sem.

Az imént említett ciklusokon kívül rendelkezésünkre áll még az elöl tesztelő (a ciklusba való belépéshez feltételt szabó) ciklus, amit a @while-@endwhile direktívák fognak közre.

A ciklusokban végezhetünk még feltételvizsgálatot is a ciklusmag további részeinek figyelmen kívül hagyására: @continue(feltétel), vagy a ciklusból való kilépésre is: @break(feltétel). A működésük nagyon hasonló az @selected-hez, mivel itt is igazából egy @if(feltétel)-@continue-@endif direktíva kombináció működik, illetve az @if(feltétel)-@break-@endif van a háttérben, csak az egyszerűsítés miatt használhatjuk önmagukban őket.

A gyűjteményeket a @php-endphp direktívákban is használhatjuk a nézet fájlok tetején, de inkább ne itt a nézet fájlokban akarjuk elhelyezni az üzleti logikát érintő kalkulációkat, hiába ad rá lehetőséget a rendszer. Próbáljuk betartani és követni az MVC tervezési minta ajánlásait és szabályait!

12.1.5.3. API válaszokban a gyűjtemények

Az API kérések elküldése után a válaszokat gyakran JSON objektumban (vagy objektumok tömbjében) kaptuk meg a Postman alkalmazásban. Ezek is igazából gyűjtemények, hiszen az adott Eloquent Model objektumok összességét, vagy a Resource::collection() utasítás eredményét (például a korábbi 7–61. kódrészletben), amikor az adott API Controller osztály index() metódusán keresztül küldünk vissza válaszként egy-egy gyűjteményt a kérésekre.

A gyűjtemények használatának nagy előnyei az API hívások megválaszolásánál is jelentkeznek, például a segédmetódusok használatával formálhatjuk, átalakíthatjuk az adatbázisból lekért adatokat még szerver oldalon, és úgy adhatjuk azt vissza a kliens oldalnak, hogy már *„késznek"* tekinthető az adatok feldolgozása és előkészítése a kliens oldali felhasználásra.

12.2. Architekturális koncepciók

Ebben az alfejezetben áttekintjük azokat a rendszer architektúrát érintő elemeket, amelyek a Laravel keretrendszer magjának részét képezik. Sok minden a Laravel-ben úgy tűnik, hogy nem is igazán értjük

még, miért, de mágikusan, magától működik. Ez azért van, mert a rendszer magja megfelelően, hibamentesen és gyorsan dolgozza fel a felhasználói kéréseket, és nekünk, programozóknak kevésbé kell foglalkoznunk azzal, hogy mi történik a rendszer magjában, egyszerűen csak használjuk, és örülünk neki, hogy könnyedén működnek a dolgok. A rendszer magjának elemeit aránylag ritkán kell programoznunk, ellenben a rendszer biztosít olyan felületeket, amelyek segítségével mi magunk, a saját (vagy más külső csomagok) szolgáltatásait tudjuk implementálni a rendszerbe a plusz funkcionalitások működtetéséhez. A rendszer magjának működésével érdemes tisztában lenni azért, hogy ne csak azt gondoljuk, itt valamilyen csoda miatt működnek a dolgok, hanem programozóként lássuk is át, mi miért történik úgy, ahogy azt mi elvárjuk. Emiatt ez az alfejezet talán egy kicsit nehéznek tűnhet elsőre, de ha nem érthető, akkor érdemes újra és újra végig gondolni, kipróbálni a példákat, és értelmezni őket.

12.2.1. Felhasználói kérések életciklusa

Nézzünk egy kicsit a Laravel alkalmazásunk mélyébe! Az alkalmazás architektúrája szempontjából, ha azt egy webszerver (például Apache vagy nginx) futtatja, akkor a **public / index.php** lesz az alkalmazás belépési pontja. Innen indul el a felhasználói kérések kiszolgálása.

Ha megnyitjuk ezt a fájlt, akkor azt látjuk benne, hogy először megvizsgálja, karbantartás alatt áll-e a rendszer, ha nincs, akkor folytatódhat tovább a feldolgozás. Betöltésre kerülnek azok a fájlok, osztályok, amelyek a rendszer részét képezik, így nem kell őket manuálisan betöltenünk.

Utána létrehoz az alkalmazásunkból egy példányt. Az alkalmazás magja a Kernel, ezen keresztül áramolnak a kérések a rendszerben. A Kernel-re gondolhatunk egy nagy fekete dobozként is, amelybe például HTTP kéréseket küldünk be, és HTTP válaszokat fogunk visszakapni tőle. Attól függően, hogy milyen típusú kérés érkezik (webes, API, console-os), a rendszer magja le fogja kezelni a kérést, amire aztán majd egy annak megfelelő választ fog adni. A Kernel ezen felül definiál és inicializál egy indítási beállításokat tartalmazó halmazt, amelyben meghatározásra kerülnek hibakezelési, naplózási és az alkalmazás környezethez kapcsolódó beállítások. Továbbá a Kernel felelős azért, hogy ha megfelelőek a kérések, akkor átjussanak a köztes rétegeken, mint például a munkamenet (hitelesítési és engedélyezési) ellenőrzésen, CSRF token ellenőrzésen és így tovább.

```php
require __DIR__.'/../vendor/autoload.php';

$app = require_once __DIR__.'/../bootstrap/app.php';

$kernel = $app->make(Kernel::class);

$response = $kernel->handle(
    $request = Request::capture()
)->send();

$kernel->terminate($request, $response);
```

```php
// Register the Composer autoloader...
require __DIR__.'/../vendor/autoload.php';

// Bootstrap Laravel and handle the request...
(require_once __DIR__.'/../bootstrap/app.php')
    ->handleRequest(Request::capture());
```

12–6. ábra: public / index.php kiemelt része a Laravel 10-ben (balra) és 11-ben (jobbra)

A Laravel 10-ben még láthatjuk a **Kernel** osztályt (**app / Http / Kernel.php**), de a Laravel 11 már elrejti előlünk! *Megjegyzés:* nagyon-nagyon sok Laravel fejlesztő ezt a változtatást eléggé negatívan fogadta a 11-es verziójú keretrendszer megjelenésekor, így nem tudni, hogy ezzel a negatív változtatással a jövőben történik-e még valami.

Ezután következnek a Service Provider-ek, amelyek a keretrendszer különböző komponenseit jelentik. Adatbázis kezelésre, adatérvényesítésre, útvonal kezelésre stb. használ a rendszer Service Provider-eket. Ezeket először regisztrálni kell a **register()** metódusuk segítségével, majd utána bekötésre kerülnek a rendszerbe a **boot()** metódusuk által (ezt a két metódust mindegyik Service Provider osztály kötelezően tartalmazza). Azért regisztrálunk mindent, majd kötünk be mindent, mert a bekötésnél már ellenőrzésre kerül az is, hogy a függőségek megfelelnek-e (nincsenek hiányzó elemek) a működéshez. Ilyen Service Provider-eket már mi is használtunk, például az útvonalak kezelésénél a **RouteServiceProvider**-t, vagy a külső csomagok kapcsán a **Folio** (3.5. alfejezet), **Jetstream** vagy a **Fortify** csomagok (10.2.2.6. alfejezettől kezdve) Service Provider osztályait. Emiatt már tapasztalhattuk azt, hogy lényegében a Laravel által kínált minden fontosabb funkciót egy Service Provider indít el és állít be. Mivel a Laravel keretrendszer által kínált számos funkciót ők indítják el és állítják be, ezért a Laravel teljes indítási folyamatának legfontosabb részei a Service Provider-ek.

Amikor a Service Provider-eket beregisztrálta a rendszer és elindultak, utána következik az útvonalak meglétének (regisztrálásának) ellenőrzése például a **routes / web.php**-ban, és folytatódik a kérés kiszolgálásának folyamata a köztes rétegekkel, Controller metódusokkal stb. De ezeket mi már mind áttekintettük, megtanultuk, csak most a Service Provider-ek, kvázi *„egy kezdeti lépésként”* bekerültek ebbe a folyamatba, amit például a 3.4. alfejezettől (3–5. ábra) kezdve lépésről-lépésre bővítettünk az éppen megismert elemekkel. Az ábra legutóbb a 10.1.2. alfejezetben került ismertetésre a köztes réteg elemekkel történő kibővítés során (10–6. ábra).

De nem csak egy *„új”* első lépéssorozat került be a felhasználói kérések feldolgozásának és kiszolgálásának folyamatába, hanem a végére is, ha újra rátekintünk a **public / index.php** fájlra (12–6. ábra bal oldala – Laravel 10), akkor a rendszer magja, a Kernel a feldogozott kérésre adott választ visszaküldi (**send()**) a felhasználó böngészőjének, tipikusan egy nézet fájl formájában.

Így ér véget a felhasználói kérés feldolgozásának és kiszolgálásának folyamata a Laravel keretrendszerben.

12.2.2. Service Container működése

A Laravel Service Container kezeli az objektumok és függőségeik példányosítását. Egyszerűbben fogalmazva, a Service Container egy olyan konténer, amely tartalmazza és kezeli az összes olyan objektumot és szolgáltatást, amelyre az alkalmazás működése során szüksége van. A Service Container felelős ezen objektumok és függőségeik létrehozásáért, majd *„befecskendezi”* (injection) őket az alkalmazás kódjába, amikor szükség van rájuk.

A Service Container pontosan azt jelenti tehát, amit a neve is takar, ez egy konténer, amiben szolgáltatások vannak. Betehetünk és segítségül hívhatunk belőle szolgáltatásokat, ahogy éppen szeretnénk. Egy szolgáltatás bármi lehet, gondolhatunk egy komplexebb algoritmusra, egy adatreprezentációra, egy gyűjteményre, osztályra… gyakorlatilag bármit eltárolhatunk a konténerben, amit majd később segítségül szeretnénk hívni valaminek a megvalósításához a Laravel projektünk egy másik részén. A hangsúly talán az utolsó kifejezésen van, tehát valamit beteszünk egy konténerbe valahol, aztán azt a valamit az alkalmazás egy másik részén fogjuk tudni használni, anélkül, hogy meghívnánk az

osztály konstruktorát (vagy például a *„setter”* metódusát) előtte. Talán mi magunk sem tudtuk eddig, de már rengetegszer használtuk a Service Container-t anélkül, hogy tudtunk volna róla, vagy manuálisan kellett volna kapcsolatba kerülnünk a konténerrel.

Vegyük csak az alábbi példát:

```
public function show(Category $category)
{
  return view('categories.show', compact('category'));
}
```

12–1. kódrészlet: CategoryController show() metódusa a Service Container megértéséhez

Ez az **l10-components** projektünk **app / Http / Controllers / CategoryController.php** fájlból vett **show()** metódus, amely a paraméterében megkapja a **Category** osztály egy példányát. Mindezt úgy kapja meg, hogy nem kellett az osztályt példányosítanunk a **show()** metódus magjában, vagy a konstruktora segítségével beállítanunk a **$category** példány attribútumait. Ezt mind elvégezte helyettünk a Laravel azáltal, hogy csak beírtuk az osztály nevét az adott helyen. Ezt a technikát **dependency injection**-nek nevezi a szaknyelv, amikor osztályfüggőségeket használunk anélkül, hogy konkrétan példányosítanánk őket a konstruktoruk használatával vagy felparamétereznénk őket egy *„setter”* metódusuk segítségével.

A Service Container használatának előnyei:

1. *Függőségkezelés:* a Service Container segít kezelni az alkalmazás függőségeit, megkönnyítve ezzel a kódbázis karbantartását és frissítését. A függőségeknek az alkalmazás kódjába történő befecskendezésével moduláris és karbantartható maradhat a kód, így könnyebben tudunk új funkciókat hozzáadni és hibákat javítani.
2. *Tesztelhetőség:* a Service Container megkönnyíti az alkalmazáskód tesztelését, köszönhetően a függőségek mockolására (szimulációra) való képességének. A Service Container használatával a mock (szimulációs) objektumok injektálásával a kódba, izoláltan tesztelhetjük az alkalmazás logikáját, anélkül, hogy a külső függőségek miatt aggódni kellene.
3. *Kód újrafelhasználhatóság:* a Service Container elősegíti a kód újrafelhasználhatóságát, mivel lehetővé teszi, hogy a szolgáltatásokat egyszer definiálja, majd az egész alkalmazásban használja őket. Ez azt jelenti, hogy olyan kódot írhatunk, amely több projektben vagy modulban is használható, csökkentve ezzel a fejlesztési időt és növelve a kód hatékonyságát.
4. *Skálázhatóság:* a Service Container megkönnyíti az alkalmazás skálázását, lehetővé téve, hogy szükség szerint új szolgáltatásokat és függőségeket adjon hozzá. Ez azt jelenti, hogy gyorsan hozzáadhatunk új funkciókat az alkalmazásához anélkül, hogy aggódni kellene a meglévő kód megváltoztatása miatt.

A legtöbb esetben azonban erre a Service Container-re konkrétan nincsen szükség, hiszen a dependency injection-nek vagy a Facade-oknak (lásd majd a 12.2.4. alfejezetet) köszönhetően úgy építhetünk fel egy komplett Laravel-alkalmazást, hogy soha semmit nem kell manuálisan bekötnünk a konténerbe, vagy kinyernünk valamit onnan.

Előfordulhat mégis két olyan eshetőség, amikor a bekötést és kinyerést manuálisan kell elvégeznünk:

1. Ha olyan osztályt írunk, amely egy interface-t valósít meg, és azt szeretnénk, hogy ha beírjuk az interface nevét egy útvonalba vagy egy osztály konstruktorába, akkor meg kell mondanunk a konténernek, hogy hogyan oldja fel azt az interface-t.
2. A másik egy inkább valós életbeli szituációra vonatkozik: ha olyan Laravel csomagot írunk, amelyet más Laravel fejlesztőkkel szeretnénk megosztani, akkor szükség lehet a csomag szolgáltatásainak konténerbe való regisztrálására és bekötésére.

Kisebb alkalmazások fejlesztése esetén nem nagyon lesz szükségünk arra, hogy mi magunk manuálisan kössünk be (binding: a **bind()** vagy **singleton()** metódussal) a rendszerbe osztályokat vagy szolgáltatásokat, majd utána azokat kinyerjük (resolving: a **make()** vagy **resolve()** metódussal) belőle, de a továbbiakban megtekintjük ennek a működését és folyamatát, hogy jó szoftverfejlesztőként a dolgok mélyére lássunk.

12.2.2.1. Service Container használata

Hozzunk létre egy új Laravel 11-es projektet, amiben ki tudjuk próbálni a Service Container és majd a saját Service Provider működtetését.

```
laravel new l11-framework-core-elements
```

Hozzuk neki létre a GitHub repository-t is, hogy utána nyomon követhető legyen!

Utána hozzunk létre egy **Imdb** osztályt, amely például arra szolgálhat, hogy a híres filmes adatbázis oldalhoz hozzáférjünk API-on keresztül, de ezeket a funkcionalitásokat persze most nem fogjuk megcsinálni (csak szimuláljuk), mert nem ez a lényeg, amire koncentrálni fogunk.

```
php artisan make:class Services/Imdb
```

A létrehozott fájlban, kódoljunk egy kicsit *„régimódi PHP-s programozóként”*, és töltsük fel az alábbi tartalommal:

```php
<?php

namespace App\Services;

use Illuminate\Http\Request;

class Imdb
{
  public function __construct()
  {
    //
  }

  public function getPoster($movieId)
  {
    return "actual poster URL";
  }
}
```

```
class MovieController
{
  public function index(Request $request)
  {
    $imdb = new Imdb();

    $poster = $imdb->getPoster($request->get('poster'));
    echo $poster;
  }
}
```

12–2. kódrészlet: Imdb szolgáltató osztály és MovieController felhasználó osztály

Ez a kód talán itt az **Imdb.php** fájlban működhetne is, természetesen, ha a **getPoster()** metódus fel lenne töltve megfelelő tartalommal. De ha ettől eltekintünk, akkor mi lehet még a gond ezzel az implementációval? Gondoljunk csak arra, hogy ha ezt az **Imdb** osztályt máshol is szeretnénk használni az alkalmazásban, akkor mit kellene tenni? Esetleg az **Imdb** osztálynak még további függőségi viszonyai is vannak, amelyeket át kellene adnia? Ezekre a kérdésekre adott válaszok rávilágítanak arra, hogy ez a tervezési módszer biztosan nem lesz megfelelő így, mert a kezelés és karbantartás gyorsan nehézkessé tud válni ebben a szituációban. Emiatt is találták ki a dependency injection-t, amely sokat tud segíteni nekünk ebben a helyzetben. *Megjegyzés:* az injektálást már alkalmaztuk is, amikor a **Request** osztályt használtuk a **MovieController index()** metódusában.

Alapértelmezetten a Laravel lehetővé teszi a számunkra, hogy ezt a függőségi injektálást szabadon alkalmazzuk. Mindössze annyit kell tennünk, hogy beírjuk az injektálni kívánt osztály nevét, itt most a Controller metódusába, de működne úgy is, ha egy Middleware-ben, vagy majd a későbbiekben eseményeknél használnánk ugyanígy.

Módosítsuk a **MovieController index()** metódusát így:

```
public function index(Request $request, Imdb $imdb)
{
  $poster = $imdb->getPoster($request->get('poster'));
}
```

12–3. kódrészlet: Dependency injection a MovieController index() metódusában

Az **Imdb** osztály egyszerű begépelésével a Controller metódusunkban a Laravel automatikusan megkeresi az **Imdb** osztályt és függőségeit, megpróbál létrehozni egy új példányt az osztályból, és átadja azt a vezérlő metódusának. Ha a **MovieController** összes metódusában szükség lenne az **Imdb** függőség felhasználására, akkor érdemes a **MovieController** konstruktorában elhelyezni az **Imdb** függőséget az alábbiak szerint, és akkor nem lesz szükség arra, hogy minden egyes metódusba injektáljuk a függőséget:

```
class MovieController
{
  protected Imdb $imdb;

  public function __construct(Imdb $imdb)
  {
```

```
    $this->imdb = $imdb;
  }

  public function index(Request $request)
  {
    $poster = $this->imdb->getPoster($request->get('poster'));
  }
}
```

12–4. kódrészlet: Függőség felhasználása a Controller összes metódusában

Ehhez nincs is szükség más egyébre, mivel ez a zero configuration resolution gondoskodik arról, hogy ez számunkra ilyen könnyedén működjön.

Mivel egy *„elképzelt, szimulált"* API hívásokat lekezelő osztályról beszélünk az **Imdb** osztály esetén, ezért könnyedén előfordulhat, hogy a távoli szolgáltatás használatához szükségünk lehet egy API kulcsra, amivel azonosítani tudjuk magunkat az **Imdb** számára, és így a kéréseinkre érdemi válaszokat adhatna. Ezt az API kulcsot az **Imdb** osztály konstruktorába tölthetnénk be az alábbiak szerint:

```
class Imdb
{
  public function __construct(protected $apiKey)
  {
    //
  }
```

12–5. kódrészlet: API kulcs átadása az Imdb osztály konstruktorának

Ha most megpróbálja a keretrendszer injektálni ezt az **Imdb** osztályt a **MovieController** vezérlőnkbe, akkor a Laravel nem fog tudni mit kezdeni az **$apiKey** konstruktorban lévő paraméterrel, és kivételt fog dobni, mivel nem tud létrehozni egy új példányt az osztályból. A tényleges kipróbáláshoz helyezzük át a **MovieController** osztály tartalmát innen egy tényleges Controller fájlba:

```
php artisan make:controller MovieController
```

A tartalma legyen akkor ez:

```
<?php

namespace App\Http\Controllers;

use App\Services\Imdb;
use Illuminate\Http\Request;

class MovieController extends Controller
{
  protected Imdb $imdb;

  public function __construct(Imdb $imdb)
  {
    $this->imdb = $imdb;
```

```
  }

  public function index(Request $request)
  {
    $poster = $this->imdb->getPoster($request->get('poster'));
    echo $poster;
  }
}
```

12–6. kódrészlet: MovieController osztály az MVC tervezési minta szerinti helyén: app/Http/Controllers/Moviecontroller.php

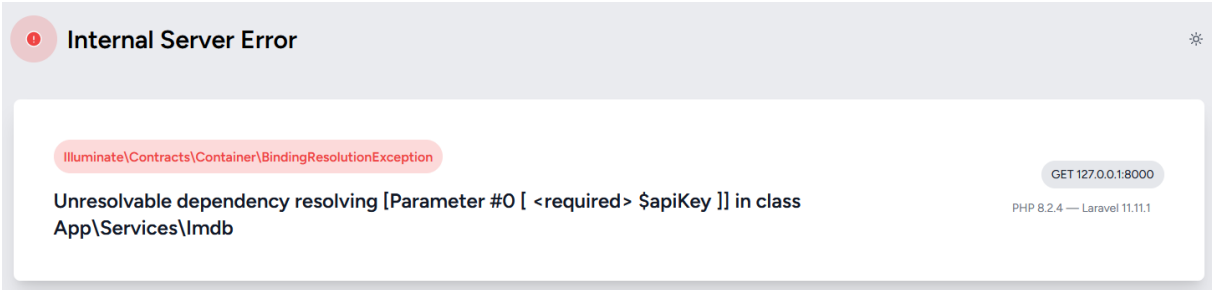
Az **Imdb.php** fájlból ezzel egyidejűleg kivehető a **Request** osztály importálása is.

Hozzuk létre neki a **routes / web.php** fájlban a hozzá vezető útvonalat is:

```
Route::get('/movies', [MovieController::class, 'index']);
```

12–7. kódrészlet: Útvonal regisztrálása a MovieController index() metódusa felé

Ha most lekérjük az útvonalat a böngészőben, akkor megkapjuk a várt hibát:



12–7. ábra: Osztály feloldási hiba a Laravel-ben

12.2.2.2. Bekötés a Service Container-be (eltárolás)

A feloldási hiba abból adódik, hogy a Laravel próbálja megtalálni a Service Container-ben, hogy van-e ilyen API kulcs elhelyezve, de nem találja, ezért egy kis segítséget szeretne kérni. A feloldási hiba megoldásához általában manuálisan regisztrálnunk kell az osztályt kulccsal együtt a Service Container-be. Ezt a **app / Providers** mappájában lévő **AppServiceProvider.php** fájlban tehetjük meg, méghozzá ennek is a **register()** metódusában kell egy szolgáltatást regisztrálni.

A meglévő Service Provider-eket kétféle módon is el tudjuk érni, mielőtt manuálisan bekötnénk a segítségükkel valamilyen új szolgáltatást:

1. **app()** segédmetódussal
2. **$this->app** tulajdonságon keresztül

```
public function register(): void
{
  $this->app->bind(Imdb::class, function ($app) {
    return new Imdb(config('services.imdb.key'));
  });
}
```

12–8. kódrészlet: app / Providers / AppServiceProvider register() metódusában regisztráltunk saját szolgáltatást

A **bind()** metódussal végeztük el a szolgáltatás bekötését úgy, hogy a konstruktorba elhelyeztünk egy beállítást: **config('services.imdb.key')**. Ez a beállítás azt jelenti, hogy a **config()** segédfüggvénnyel a Laravel a **config** mappájában keresi a **services.php** fájlt (ezt meg is találja, benne olyan példákkal, mint a Slack vagy a Postmark alkalmazások és hozzáférési kulcsaik, még szintén tényleges értékek nélkül). Majd abban az ott meglévő gyűjteményben keresi az **imdb** gyűjteményt, és benne a **key** kulcshoz tartozó értéket. Ez természetesen még nincsen meg, hiszen ezt mi magunknak kellene oda elhelyeznünk. Tegyük is ezt meg egy véletlen karaktersorozat értékkel:

```
'imdb' => [
  'key' => env('IMDB_KEY', 'asdfqewr')
]
```

12–9. kódrészlet: Példa IMDB kulcs a config / services.php fájlban

12.2.2.3. Kinyerés a Service Container-ből

Módosítsuk a **MovieController** osztály **index()** metódusának magját, és adjuk hozzá a Service Container-be bekötött osztály kiíratását úgy, hogy a **resolve()** segédmetódust használjuk a kinyeréshez:

```
dump(resolve('App\Services\Imdb'));
```

12–10. kódrészlet: Service Container-be elhelyezett osztály példány kinyerése és kiíratása

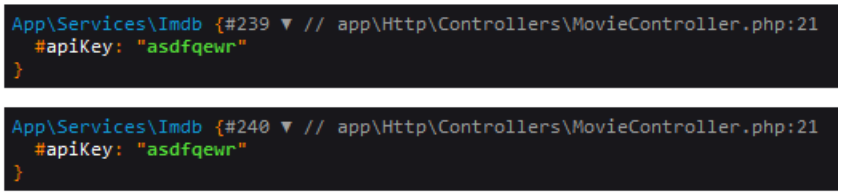
Eredményül megkapjuk most már a feloldási hiba helyett az osztály egy példányát a megadott alapértelmezett API kulccsal:

12–8. ábra: Service Container-ből kinyert osztály példány kinyerésének és kiíratásának eredménye

Ha duplán nyerjük ki, vesszük ki a konténerből, akkor az osztálypéldányt azonosító szám különböző lesz (#239 és #240, de mindenkinél más számok lesznek, a lényeg a *különbözőség*) már mást fog mutatni:

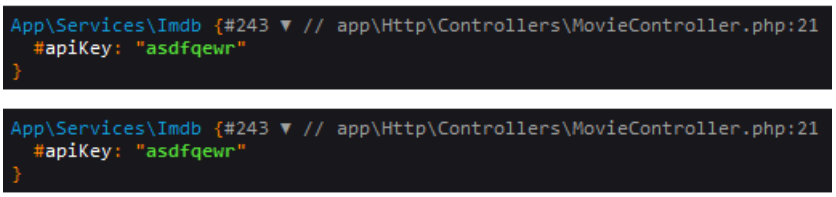
```
dd(resolve('App\Services\Imdb'), resolve('App\Services\Imdb'));
```

12–11. kódrészlet: Service Container-be elhelyezett osztály kinyerése és kiíratása



12–9. ábra: bind() bekötés miatt két különböző példány kinyerése a Service Container-ből

Ez azért van, mert **bind()**-dal kötöttük be az osztályt a Service Container-be, ezért ha duplán nyerjük ki, akkor más-más osztálypéldányt fog visszaadni nekünk a rendszer. Ez nem mindig hasznos, mivel gyakran inkább ugyanarra a konténerbe tett objektumra lenne szükségünk. Változtassuk meg **singleton()**-ra a **bind()**-os bekötést, és kérjük le újra a böngészőnkben az útvonalat, mivel ekkor már ugyanazt a példányt kapjuk meg kétszer (#243 és #243, de mindenkinél más szám lesz a két szám, a lényeg az *egyezőség*).



12–10. ábra: singleton() bekötés miatt ugyanannak a példánynak a dupla kinyerése a Service Container-ből

Így tehát bekötöttünk egy osztályt a Service Container-be, majd utána annak példányait kinyertük a konténerből és felhasználtuk ott, ahol szükségünk volt rá.

Az alfejezet során végrehajtott programkód módosítások ebben a GitHub commit-ben találhatók meg.

12.2.3. Service Provider működése

A Service Provider-ek a keretrendszer különböző komponenseinek, például az adatbázis kezelésnek, a felhasználói adatérvényesítésnek (validation) és az útválasztásnak (routing) stb. az indításáért felelősek. Az indítás alatt azt értjük, hogy elvégezzük a bekötéseket a Service Container-be, a rendszer élesíti az eseménykezelőket, a köztes rétegeket felállítja és még az útvonalakat is regisztrálja.

A Service Provider-ek azok a központi elemek, ahol az alkalmazásunkat be tudjuk állítani. Gondoljunk csak vissza arra, amikor a **JetstreamServiceProvider**-ben a szerepköröket és a jogosultságaikat állítottuk be, és utána a rendszer aszerint végezte el az engedélyezéseket. Az persze egy csomag szolgáltatását állította be, de az alaprendszerben több tucat Service Provider működik, amelyek a Laravel 10-ben a **config / app.php 'providers'** kulcs alatti gyűjteményében kerültek regisztrálásra a rendszerbe, míg a Laravel 11-ben ezeket már jobban elrejti előlünk a keretrendszer (ahogy a 10–5. újdonság is rávilágított már erre korábban).

```
'providers' => [

    /*
     * Laravel Framework Service Providers...
     */
    Illuminate\Auth\AuthServiceProvider::class,
    Illuminate\Broadcasting\BroadcastServiceProvider::class,
    Illuminate\Bus\BusServiceProvider::class,
    Illuminate\Cache\CacheServiceProvider::class,
    Illuminate\Foundation\Providers\ConsoleSupportServiceProvider::class,
    Illuminate\Cookie\CookieServiceProvider::class,
    Illuminate\Database\DatabaseServiceProvider::class,
    Illuminate\Encryption\EncryptionServiceProvider::class,
    Illuminate\Filesystem\FilesystemServiceProvider::class,
    Illuminate\Foundation\Providers\FoundationServiceProvider::class,
    Illuminate\Hashing\HashServiceProvider::class,
    Illuminate\Mail\MailServiceProvider::class,
    Illuminate\Notifications\NotificationServiceProvider::class,
    Illuminate\Pagination\PaginationServiceProvider::class,
    Illuminate\Pipeline\PipelineServiceProvider::class,
    Illuminate\Queue\QueueServiceProvider::class,
    Illuminate\Redis\RedisServiceProvider::class,
    Illuminate\Auth\Passwords\PasswordResetServiceProvider::class,
    Illuminate\Session\SessionServiceProvider::class,
    Illuminate\Translation\TranslationServiceProvider::class,
    Illuminate\Validation\ValidationServiceProvider::class,
    Illuminate\View\ViewServiceProvider::class,

    /*
     * Package Service Providers...
     */
    App\Providers\FortifyServiceProvider::class,
```

12–11. ábra: A regisztrált Service Provider-ek (l10-components projekt)

Megjegyzés: én nem feltétlenül értek egyet azzal, hogy a Laravel 11-ben a keretrendszer számos dolgot elrejtett előlünk különböző megfontolásokból. Talán ezért sem baj, ha a Laravel 10-ben látjuk, hogy hogyan is működött ez a Service Provider regisztráció az alaprendszerben, és konkrétan milyen Service Provider-ek kerültek alapértelmezetten regisztrálásra egy friss (Fortify csomaggal kibővített) webes projektbe.

Ahogy azt már korábban csináltuk, a Service Provider-eket Laravel 10-ben a **config / app.php 'providers'** gyűjteményébe kell elhelyezni, míg a Laravel 11-nél a **bootstrap / providers.php** fájlban lévő gyűjteménybe.

12.2.3.1. Saját Service Provider megvalósítása

Tekintünk egy nagyon egyszerű példát: az 5 legjobb filmet mindig meg szeretnénk mutatni a felső sávban a filmes nézeteinkben, mindezt úgy, hogy nem adjuk át ezeket a filmeket külön-külön minden ilyen nézetnek.

12.2.3.1.1. Előkészítés

A feladat megoldásához hozzunk létre egy **Movie** Model osztályt, a hozzá kapcsolódó migrációs fájlt és adatgyárat.

```
php artisan make:model Movie -mf
```

A **Movie** Model osztályban adjuk hozzá a **$fillable** mezőhöz a **title** és **score** mezőket. A migrációs fájlban is hozzuk létre ezt a két mezőt: a **title** legyen *string*, a **score** legyen *float* típusú. Az adatgyárban a **title** legyen egy 4 szóból álló mondat, míg a **score** értéket generáljuk a **fake()->randomFloat(1, 1, 10)** metódussal, amelynek az első paramétere a maximális tizedesjegyek számára utal, a második paraméter a minimum, a harmadik paraméter a maximum film értékelést jelenti. A **database / seeders / DatabaseSeeder** osztályban a **run()** metódushoz adjuk hozzá a filmes adatgyárunkat, hogy hozzon létre 100 darab filmet. Majd futtassuk a migrálást és seed-elést:

```
php artisan migrate --seed
```

Laravel 11-ben dolgozunk, úgyhogy utána ellenőrizhetjük is a **database / database.sqlite** fájlban, hogy benne van-e a 100 darab film és az értékelési pontszámok is olyanok-e, mint amilyet elterveztünk.

Tables
- cache
- cache_locks
- failed_jobs
- job_batches
- jobs
- migrations
- movies
- password_reset_tokens
- sessions
- sqlite_sequence
- users

	id	title	score	created_at	updated_at
1	1	Sit consequatur odio.	7.8	2024-06-24 08:20:03	2024-06-24 08:20:03
2	2	Delectus voluptatem alias rerum et id.	8	2024-06-24 08:20:03	2024-06-24 08:20:03
3	3	Voluptas dicta beatae cumque magnam.	3.4	2024-06-24 08:20:03	2024-06-24 08:20:03
4	4	Enim sunt nesciunt explicabo.	1.3	2024-06-24 08:20:03	2024-06-24 08:20:03
5	5	Suscipit minus repellendus ut.	6.5	2024-06-24 08:20:03	2024-06-24 08:20:03
6	6	Iste asperiores deserunt delectus.	6.7	2024-06-24 08:20:03	2024-06-24 08:20:03
7	7	Corporis aspernatur ut cum dolores.	4.6	2024-06-24 08:20:03	2024-06-24 08:20:03
8	8	Mollitia sed sed.	6.4	2024-06-24 08:20:03	2024-06-24 08:20:03
9	9	Ea consequuntur sunt nobis consequatur aut.	7.2	2024-06-24 08:20:03	2024-06-24 08:20:03
10	10	Deleniti voluptatem sunt.	5.2	2024-06-24 08:20:03	2024-06-24 08:20:03
11	11	Ut possimus et.	8.4	2024-06-24 08:20:03	2024-06-24 08:20:03
12	12	In doloremque distinctio repudiandae sit quo.	2.9	2024-06-24 08:20:03	2024-06-24 08:20:03
13	13	Tenetur similique sint fugit dolorum vero.	8.6	2024-06-24 08:20:03	2024-06-24 08:20:03

12–12. ábra: Migrált és teszt adatokkal feltöltött movies adattábla (VSCode-ban SQLite Viewer-es kiterjesztéssel készült)

A **MovieController** már a rendelkezésünkre áll, azt bővítsük ki egy **show()** metódussal, illetve a hozzá vezető útvonalat is hozzuk létre a **routes / web.php**-ban (**/movies/{movie}** és a **MovieController show** metódusa legyen a célja).

A **MovieController index()** és **show()** metódusa is egy-egy nézettel térjen vissza, amelyek a **movies.index** és **movies.show** nézetnek küldjék az adatokat. Ezek a nézetek még nincsenek meg, úgyhogy hozzuk létre őket, akár utasítással is (bár annyit nem nyerünk vele, mivel a nézet fájloknak nincsen részletes szerkezete, viszont a megfelelő helyre fogja betenni őket a rendszer úgy, hogy a **movies** mappát is létrehozza nekik):

```
php artisan make:view movies.index
```

```
php artisan make:view movies.show
```

A **movies.index** nézet táblázatosan jelenítse meg a filmeket (**title** és **score** oszlop nevekkel) és a filmek címei legyenek linkek, amelyek az adott film **show** nézetére vihetik át a felhasználót. A **movies.show**

nézetben elég kiírni a film címét és értékelésnek pontszámát, plusz legyen benne egy link, ami visszaviheti a felhasználót a filmes index oldalra.

Az előkészítés során semmilyen új dolgot nem csináltunk, de ha mégse ment volna valamelyik kód megírása, akkor ezt a GitHub commit-et érdemes áttekinteni hozzá.

12.2.3.1.2. Használat

A célunk tehát az, hogy mindkét nézet felső sávjában a legjobb 5 filmet soroljuk fel, így megmutatjuk, hogy a Service Provider által a nézetek között megosztott adat használható egy *„globális"* változó értékének lekérésével. Így, ha valaki a mi általunk létrehozott Service Provider-t használja majd, akkor nem lesz szüksége különösebb beállításokra, hanem csak egyszerűen működtetni tudja.

```
php artisan make:provider MovieServiceProvider
```

Ennek hatására a Laravel 11-ben nem csak a **MovieServiceProvider.php** fájl jött létre az **app / Providers** mappában, hanem a regisztrációja is automatikusan bekerült a **bootstrap / providers.php** fájlba. Laravel 10-nél ez utóbbit még manuálisan magunknak kellett megtennünk. A létrejött **MovieServiceProvider.php** fájlban először foglalkozzunk a **register()** metódussal. Ezzel tudunk bekötni szolgáltatásokat a Service Container-be. Mi most még egyszerűen csak az adatkapcsolatot létrehozó szolgáltatást tegyük ide bele:

```
public function register(): void
{
  $this->app->singleton(Connection::class, function ($app) {
    return new Connection(config('database.default'));
  });
}
```

12–12. kódrészlet: Adatkapcsolat szolgáltatás regisztrálása a Service Container-be

Importáljuk hozzá a fájl tetején az **Illuminate\Database\Connection** osztályt!

A Service Provider **boot()** metódusa az, ahol a keretrendszer vagy más szolgáltatók által regisztrált szolgáltatásokkal kapcsolatba léphetünk. Itt adhatunk hozzá további funkciókat, például eseménykezelőket vagy köztes rétegeket, vagy akár regisztrált szolgáltatásokat is módosíthatunk.

```
public function boot(): void
{
  view()->composer('*', function ($view) {
    $view->with('top5Movies', Movie::orderByDesc('score')->take(5)->get());
  });
}
```

12–13. kódrészlet: Top 5 film lekérése és megosztása a nézetek számára

Importáljuk hozzá a **Movie** Model osztályt!

Ezáltal az összes nézetünkben tudjuk használni ezt a **$top5Movies** változót, ami egy gyűjtemény.

Szúrjuk be ezt a **movies.index** és **movies.show** nézetünk elejére is:

```
<aside>
```

```
  <h1>Top 5 movies:</h1>
  <ul>
    @foreach ($top5Movies as $topMovie)
      <li><a href="{{ route('movies.show', $topMovie->id)}}">{{ $topMovie-
>title }} ({{ $topMovie->score }})</a></li>
    @endforeach
  </ul>
</aside>
```

12–14. kódrészlet: Top 5 filmet listázó nézet részlet

Ha már két nézetben is ugyanazt a kódot használjuk fel, akkor egyből ugorjon be nekünk az a lehetőség, hogy ezt akár egy komponensbe vagy egy *include* fájlba is kiszervezhetjük, amit aztán felhasználunk / importálunk a megfelelő helyen a nézet fájljainkban. Válasszunk egy lehetőséget a kettő közül (én az előbbit választottam) és hajtsuk is végre!

Ezt a **$top5Movies** változót tehát nem a Controller metódusain keresztül juttattuk el a nézeteknek, hanem a Service Provider, illetve azon keresztül az úgynevezett View Composer szolgáltatás volt ebben a segítségünkre. Ez pontosan abban segít minket, mint amire mi is használtuk: ha van egy olyan adat, amelyet több nézet között is meg szeretnénk osztani, akkor ezt a funkcionalitást kell megvalósítanunk egy Service Provider-ben, így elég egyetlen helyen megvalósítanunk, és utána több helyen felhasználhatjuk az eredményt, adatot.

A bemutatott példában egy saját Service Provider segítségével osztottunk meg adatokat a nézeteink között, de akár különböző validációkat is végrehajthattunk volna, vagy további kulcs-érték párokat is megoszthattunk volna a nézetekkel, a lehetőségeink elég szélesek ennek használata során.

Ezen kívül érdemes visszatekinteni a korábban használt **Folio**, vagy **Jetstream** / **Fortify** Service Provider-ekre, és megnézni azt, hogy a mások által készített csomagokban hogyan és mire használják a Service Provider-eket.

Továbbá a Laravel 10-ben még jobban bele tudunk nézni a Laravel alaprendszer Service Provider-einek kódjaiba is, ezekből is sokat tanulhatunk. Például Laravel 10-nél a **RouteServiceProvider** osztályt, Laravel 11-nél a **bootstrap / app.php withRouting()** metódus definícióját érdemes áttekinteni tanulás és megértés céljából. Erre lehetőségünk van, mivel a Laravel keretrendszer nyílt forráskódú és le tudunk fúrni a rendszer magjáig, amikor az adott kódokat böngészgetjük. Nem utolsó sorban, azért is érdemes ezt megtenni, mert sokat lehet tanulni mások által írt kódok olvasásából is! A rendszer magjának kódjai, illetve a külső csomagok forrásai pedig jól olvasható módon kerültek implementálásra és a fájlokban elhelyezett kommentek is sokat segítenek nekünk a megértésben.

Az alfejezet során elvégzett programkód módosítások ebben a GitHub commit-ben érhetők el.

12.2.4. Facade használata

Most már van egy alapvető tudásunk a Service Container-ek és Service Provider-ek témaköreiről, folytathatjuk az ismerkedést a keretrendszer magjával. Tudjuk, hogy bármit beköthetünk a Service Container-be, és utána azokat le is kérhetjük. Amire itt most koncentrálni fogunk, az az, hogy ez a Service Container már a webes alkalmazás létrejöttekor sem üres, tartalmaz dolgokat, amik előre definiáltan

benne vannak, és használni is tudjuk őket, ezek a Facade-ok (nem igazán tudok rá jó magyar megfelelőt, úgyhogy inkább ezt az angol kifejezést használjuk a továbbiakban is). A Laravel Facade-ok szolgáltatnak egy egyszerű, kényelmesen használható statikus felületet a keretrendszer alapvető komponenseihez, amelyek a Service Container-ben alapból megtalálhatóak. Ezáltal a Laravel számos olyan Facade-ot tartalmaz, amelyek hozzáférést biztosítanak a keretrendszer szinte minden funkciójához. A Facade-ok egy tömör, megjegyezhető szintaxist biztosítanak, amely lehetővé teszi a Laravel funkcióinak használatát anélkül, hogy hosszú osztályneveket kellene megjegyezni.

Mi magunk is használtunk már Facade-okat néhányszor, például a **Route**, **DB**, **Validator**, **Auth** vagy a **Gate** Facade már többször a segítségünkre volt a munkáink során.

A Laravel összes Facade osztálya az **Illuminate\Support\Facades** névtérben található meg, így ezekhez bármikor és bárhol könnyen hozzáférhetünk. Mi legelőször a Route Facade-dal kerültünk kapcsolatba az útvonalak regisztrációja során, majd a Query Builder-rel való ismerkedés során találkoztunk a **DB** Facade-dal (6–9. kódrészlet), és kértük le a segítségével a **posts** adattábla tartalmát egy útvonal visszatéréseként a **web.php**-ban.

12.2.4.1. Segédfüggvények

A Facade-ok kiegészítéseként a Laravel számos globálisan, tehát bárhol alkalmazható segédfüggvényt kínál a számunkra, amelyek még egyszerűbbé teszik a munkánkat, amikor a Laravel funkcionalitásait szeretnénk használni. Segédfüggvényt is már rengeteget használtunk, elég ha csak a **view(), route(), redirect(), compact(), request(), config()** stb. függvényekre gondolunk, és ez a lista még korántsem volt teljes, ugyanis az itt érhető el, témák szerint csoportosítva.

12.2.4.2. Első használati példa: nézet (Facade és segédmetódus háttere)

De maradjunk annál a segédfüggvénynél, amellyel munkánk során legelőször találkoztunk, és használunk azóta is, ez a **view()**. Nézzük meg, hogy hogyan kapcsolódik ez a Facade-ok témaköréhez, mi a kapcsolat köztük, és a háttérben hogyan működik. Nyissuk meg a **routes / web.php**-t és nézzük meg a kezdőoldalunk útvonalát, majd módosítsunk a visszatérési értékén az alábbi kódrészlet szerint:

```
use Illuminate\Support\Facades\View;

Route::get('/', function () {
  // return view('welcome');
  return View::make('welcome');
});
```

12–15. kódrészlet: Kezdőoldal meghatározása a View Facade segítségével

Ekkor persze importálnunk kell a megfelelő **Illuminate\Support\Facades\View** osztályt. Ha most megnézzük a főoldalt a böngészőnkben, akkor ugyanazt az eredményt kapjuk. A két módszer tehát funkcionálisan ugyanaz, nincs egyiknek sem gyorsabb végrehajtása (esetleg csak a leütött karakterszám az, ami a segédfüggvény használata mellett szólhat). Mindkét megoldás (a **view()** és a **View::make()** Facade is) nagyon jól tesztelhető a kérésnek megfelelő válaszban való kereséssel, például az **assertSee()** metódussal.

Ha mégis a Facade-os változatnál maradunk és utánajárunk *(„mélyére fúrunk")* annak, hogy hogyan is működhet ez a **make()** metódus (**vendor / laravel / framework / src / Illuminate / Support / Facades / View.php**), akkor kicsit összezavarodhatunk, ugyanis a **View** osztálynak nincsen **make()** metódusa. Egyetlen egy statikus függvény látszódik az osztály magjában, ez pedig a **getFacadeAccessor()**. Az osztály felett viszont ott van a direktívák között, hogy melyik metódusok használják ezt, és látszódik, hogy az **\Illuminate\Contracts\View\View** interface **make()** metódusa is köztük van. Magának a **make()** metódusnak az implementációja a **Illuminate / View / Factory.php**-ben található meg.

vendor > laravel > framework > src > Illuminate > View > Factory.php > PHP > Factory

```
14   class Factory implements FactoryContract
141      /**
142       * Get the evaluated view contents for the given view.
143       *
144       * @param  string  $view
145       * @param  \Illuminate\Contracts\Support\Arrayable|array  $data
146       * @param  array  $mergeData
147       * @return \Illuminate\Contracts\View\View
148       */
         4 references | 0 overrides
149      public function make($view, $data = [], $mergeData = [])
150      {
151          $path = $this->finder->find(
152              $view = $this->normalizeName($view)
153          );
154
155          // Next, we will create the view instance and call the view creator for the view
156          // which can set any data, etc. Then we will return the view instance back to
157          // the caller for rendering or performing other view manipulations on this.
158          $data = array_merge($mergeData, $this->parseData($data));
159
160          return tap($this->viewInstance($view, $path, $data), function ($view) {
161              $this->callCreator($view);
162          });
163      }
```

12–13. ábra: View Facade-hoz tartozó make() metódus megvalósítása

Ha ezt és az osztályához tartozó konstruktort megnézzük, akkor látjuk, hogy néhány paramétert elvárna a konstruktor, viszont nekünk ezzel nem kell foglalkoznunk, mi a saját kezdőoldal útvonalunkból csak simán meghívjuk a **View::make('…')** metódust, és végre is hajtja nekünk ezt a rendszer, annyira egyszerűen, amennyire mi azt csak szeretnénk.

Az **Illuminate / Support / Facades** mappán belül számos olyan osztály elérhető, amelyek a rendszer alapfunkcionalitásához tartoznak: **App**, **Artisan**, **Auth**, **Blade**, **Cache**, **Config**, **DB**, **Event**, **File**, **Lang**, **Log**, **Mail**, **Notification** stb. Ebből is látszódik, hogy ezek a Facades mappában megtalálható elemek mind egy-egy olyan statikus felületet jelentenek, amelyen keresztül elérhetőek a rendszer alapvető komponensei és funkcionalitásai.

12.2.4.3. Második használati példa: kérés (Facade és segédmetódus feloldása a Service Container-ből)

Regisztráljunk a rendszerbe egy új útvonalat a 12–15. kódrészlet logikája alapján.

```
use Illuminate\Support\Facades\Request;
```

```
Route::get('/request-test', function () {
  // return request('name');
  return Request::input('name');
});
```

12–16. kódrészlet: Felhasználói bemenetet feldolgozó útvonal

Ezzel a link eléréssel tesztelhető: http://127.0.0.1:8000/request-test?name=Attila

Ha ismét megpróbálunk utánajárni ennek a **Request** osztálynak (**vendor / laravel /framework / src / Illuminate / Support / Facades**), akkor látható ismét, hogy nincs neki ilyen **input()** metódusa, csak a már imént is látott **getFacadeAccessor()**, ami hivatkozza az összes olyan metódust, ami az osztály felett meg van jelölve és ott már köztük van az **input()** is.

Az összes többi Facade is így működik, ez az egy **getFacadeAccessor()** metódus van bennük, és az osztályok felett a hivatkozott metódusok találhatók meg. Az adott visszatérési értékek pedig azoknak a metódusoknak a nevei, amelyeket mi már sokszor használtunk korábban és csak *„segédmetódusnak"* hívtuk őket. Ezek azok a visszatérési kulcsok, amelyek be vannak kötve a Service Container-be, és ezért tudjuk olyan könnyedén meghívni és használni őket szinte bárhol a projektünkön belül. A segédmetódusok teljes listája az alfejezet korábbi részében hivatkozásra került.

Itt tehát a segédmetódusok háttérműködésébe nyerhettünk betekintést, de folytassuk az ismerkedést még ugyanitt a **web.php** fájlban, mivel itt is be tudunk kötni kulcs-érték párokat a Service Container-be.

```
app()->bind('my-key', function () {
  return "This is my key.";
});
```

12–17. kódrészlet: Kulcs-érték pár bekötése a Service Container-be

Majd indítsuk el a Tinker-t és próbáljuk meg kinyerni ezt az értéket a kulcs alapján:

```
resolve('my-key');
```

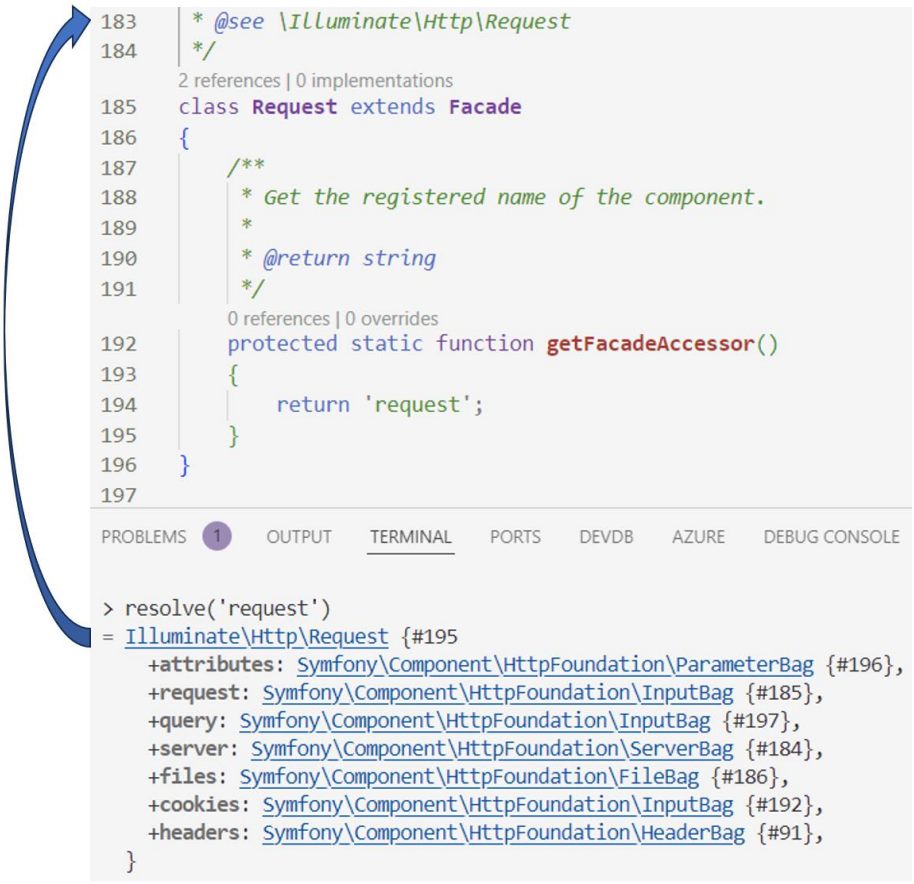
12–23. utasítások: Adott kulcsnak megfelelő értéket a Service Container-ből kinyerő utasítás

Válaszul visszakapjuk a *„This is my key"* szöveget. A Facade-oknál ez a terminológia hasonló: ők a **getFacadeAccessor()** metódus visszatérési értékét kötik be a Service Container-be, ebből adódóan ők le is kérdezhetők akár a Tinker-ben is:

```
resolve('request');
```

12–24. utasítások: request() segédmetódusnak megfelelő osztályt a Service Container-ből kinyerő utasítás

Válaszul megkapjuk az **Illuminate\Http\Request** osztály egy példányát. Az eredmény azt jelzi, hogy melyik *„alaposztályra"* utal a **Request** kérés (a **@see** direktíva mutatja mindig a **class** felett - 12–14. ábra).



12–14. ábra: request (segédmetódus) kulcsnak megfelelő osztály kinyerése a Service Container-ből

Magában az **Illuminate\Http\Request** osztályban nincsen **input()** metódus megvalósítva, azt az **InteractsWithInput** trait-en keresztül tudja használni ez az osztály.

Itt tehát megnéztük, hogy hogyan tárolódnak a Facade-hoz és a segédfüggvényhez kapcsolódó elemek a Service Container-ben. A megismert technika az összes többi Facade-ra és a kapcsolódó segédfüggvényre ugyanígy érvényes.

12.2.4.4. Harmadik használati példa: fájl (Facade és dependency injection)

A **File** Facade segítségével tudjuk olvasni a fájlokat, megnézni, hogy írhatók-e, a szülőkönyvtárát tudjuk módosítani, a fájl nevét, kiterjesztését tudjuk írni, olvasni, szóval gyakorlatilag minden fájlműveletet meg tudunk ennek segítségével csinálni. Ehhez pedig nincs is másra szükségünk, mint a **files()** segédmetódust használjuk (aminek a neve az **Illuminate\Support\Facades\File** osztály **getFacadeAccessor()** metódus visszatérési értéke is, így tehát a Service Container-ben lévő egyik kulcs is). Minden fájlokkal végezhető művelet ott van elhelyezve az osztály feletti referencia gyűjteményben, nyugodtan böngészhetjük őket és megpróbálhatjuk kitalálni, hogy melyik mire is való, mert szerencsére meglehetősen beszédes mindegyik ilyen referencia neve.

Folytassuk a példát azzal, hogy megpróbáljuk bekérni a Laravel projektünk belépési pontját, a **public** mappában lévő **index.php** fájlt. Olvassuk ezt be úgy, hogy használjuk a **files()** segédmetódust, vagyis a háttérben lévő **File** Facade-ot, aminek köszönhetően a rendszer tartalmazza a **files** kulcsot a Service Container-ben. Ismét a Tinker-ben próbáljuk ki ezt:

```
File::get(public_path('index.php'));
```

12–25. utasítások: File Facade használata a public / index.php fájl tartalmának kinyerésére

Visszaadja nekünk a **public** mappában lévő **index.php** fájl tartalmát.

Kiegészítés: vegyük észre, hogy itt is alkalmaztunk egy segédmetódust, ami a **public_path()**, ez a Laravel projektünk **public** mappájának elérési útvonalát adja vissza. De használhattuk volna a **base_path()** metódust is, ami a Laravel projekt *„gyökerét”*, illetve annak elérési útját adja vissza, azon belül pedig még el kellene navigálnunk a **public** mappába, hogy elérjük az **index.php**-t, például így:

File::get(base_path('public/index.php'));

A **public_path()**-hoz hasonlóan használható lenne bármelyik Laravel projekten belüli főmappa hivatkozása is, például: **app_path(), database_path(), resource_path()** stb.

A kitérő után folytassuk a **files** kulcs kinyerésével a Tinkerben, mivel ez is megtalálható a Service Container-ben:

```
resolve('files');
```

12–26. utasítások: request() segédmetódusnak megfelelő osztályt a Service Container-ből kinyerő utasítás

Ez visszaadja nekünk a hivatkozott osztályt: **Illuminate\Filesystem\Filesystem**

Most már tudjuk, hogy ez az az osztály, ami szerepel a **File** Facede feletti **@see** direktíva után. Ha bármilyen problémába ütközünk egy segédmetódus használatánál, akkor lekérhetjük a Service Container-ből, hogy melyik osztály van mögötte, és annak hogyan is működik az adott metódusa, mint például itt a **File::get()**.

Próbáljuk ki a kódunkban egy útvonalon keresztül a **File** Facade használatát!

```
Route::get('/file-facade-get-content', function () {
  dd(File::get(public_path('index.php')));
});
```

12–18. kódrészlet: File Facade segítségével a public / index.php tartalmát kiíratő útvonal

Így kiírásra kerül a **public / index.php** fájl tartalma. De megtehetjük ezt úgy is, hogy nem a **File** Facade-ot használjuk, hanem a dependency injection-t hívjuk segítségül:

```
use Illuminate\Filesystem\Filesystem;

Route::get('/file-di-get-content', function (Filesystem $file) {
  dd($file->get(public_path('index.php')));
});
```

12–19. kódrészlet: Dependency Injection-nel létrehozott objektummal lekért fájl tartalmát kiíró útvonal

Ehhez persze importálnunk kellett a hivatkozott osztályt, ahogy az az iménti kódrészletben látszódik is.

Működik így is! Azért mert a **File::get()** és a **$file->get()** gyakorlatilag ugyanazt a működést hajtja végre a háttérben (bár más-más importálást igényelnek). A Facade-os megoldással viszont nem kell létrehoznunk semmilyen objektumot az útvonal callback függvényének paraméterlistájában (**$file**), nem kell végrehajtanunk dependency injection-t, mégis egy rendkívül egyszerű és olvasható szintaxison keresztül férünk hozzá bármilyen fájlhoz (jelen példában a fájlt kezeljük, de a terminológia más erőforrásra is alkalmazható).

12.2.4.5. Facade-ok használatának hátránya

A hátrányuk a nagyon könnyű használhatóságukból adódik. Ha most megnézzük a **routes / web.php**-t, akkor azt láthatjuk, hogy még csak néhány útvonalunk van, de már be van importálva jó néhány Facade is. Ezáltal jó nagyra tudnak nőni az osztályok, ha túl sok Facade-ot használunk bennük/hozzájuk. Az osztály a túl nagyra növekedéssel könnyen megsértheti az Objektumorientált Programozás egyik alapelvét (*„Vonatkozások szétválasztása"*, Separation of concerns) vagy a SOLID elvek közül az *„Egy felelősség elvét"* (Single Responsibility Principle), mivel a felelősségi körök túl tágak lennének így. Ha azt érezzük, hogy már túl nagy lenne az osztályunk és több mindenért lenne felelős, akkor bontsuk fel több kisebb osztályra őket inkább és valósítsuk meg köztük a kapcsolatokat.

Tehát érdemes lehet az általunk megírt osztály konstruktorában meghatározni a Facade-os eléréseket, mert akkor látszódik, hogy mire is akarjuk használni ezt az osztályt. Ha túl sok ilyen Facade-ot szeretnénk használni a megírt osztályunkban, akkor esetleg túlságosan bonyolulttá, vagy kevésbé hatékonnyá válhat az osztály működése, és érdemesebb lenne inkább szétbontani *„alosztályokra"* ezt a *„főosztályt"*, majd megvalósítani az alosztályok együttműködését. Ha nem a konstruktorban jelezzük a Facade-ok használatának igényét, hanem a további metódusokban használjuk a Facade-okat, akkor esetleg nem látjuk át annyira, hogy hogyan lehetne szétdarabolni ezt a nagy és bonyolult főosztályt. Ez azonban majd a későbbi gyakorlat és tapasztalatszerzés során lesz igazán hasznos, a mostani alfejezet leginkább csak a működés megértését hivatott segíteni, és itt a végén csak egy utalást szerettem volna tenni erre a veszélyre is.

Végezetül még érdemes áttekinteni azt a táblázatot, amely a Facade-okat, a hivatkozott osztályokat és a Service Container-be bekötött kulcsokat (és így kvázi a segédmetódusok nevét) tartalmazza. Alább látható egy részlet a táblázatból:

Facade	Class	Service Container Binding
App	Illuminate\Foundation\Application	`app`
Artisan	Illuminate\Contracts\Console\Kernel	`artisan`
Auth	Illuminate\Auth\AuthManager	`auth`
Auth (Instance)	Illuminate\Contracts\Auth\Guard	`auth.driver`
Blade	Illuminate\View\Compilers\BladeCompiler	`blade.compiler`

12–15. ábra: Facade-ok osztály referenciája és a Service Container-ben lévő kulcsok (részlet)

Az alfejezetben végrehajtott programkód módosítások ebben a GitHub commit-ben találhatók meg.

12.3. Értesítések (Notifications)

Az értesítések a webes alkalmazások szerves részét képezik, mivel egy élő alkalmazásban mindig történik valami, amiről a felhasználót vagy az adminisztrátort érdemes rövid üzenetben tájékoztatni. Az értesítések küldése többféle formában is megtörténhet a Laravel keretrendszer támogatásával. Ezekből a lehetőségekből fogunk megismerni néhányat ebben az alfejezetben.

Gyakorlati szempontból maradunk a Laravel 11-es projektünknél, ami eddig nem túl bonyolult, filmes / IMDB-s elemeket tartalmaz. Az értesítések, amelyeket definiálni fogunk benne valamilyen filmes megjelenésről, a premierről fognak szólni.

12.3.1. Értesítések létrehozása, jellemzői és küldésének lehetőségei

Mint oly sok minden mást is, az értesítéseket is egy osztály jelképezi a Laravel keretrendszerben. Az értesítés osztályok az **app / Notifications** mappába fognak bekerülni, és érdemes őket egy artisan paranccsal létrehozni, például így:

```
php artisan make:notification MoviePremier
```

Ha megvizsgáljuk a létrejött osztály tartalmát, akkor azt láthatjuk, hogy van benne egy **Queueable** trait, amellyel – a nevéből adódóan – fogjuk majd tudni az értesítéseket várakoztatni vagy várólistára helyezni. Ehhez viszont egy segédet (munkást) kellene beállítani, hogy felügyelje a sorokban lévő üzenet kérések feldolgozását. Ilyen sorok könnyedén kialakulhatnak, főleg, ha harmadik féltől származó szolgáltatást veszünk igénybe például a több száz vagy ezer e-mail vagy sms értesítés kiküldéséhez. Az értesítések sorba rendezését és feldolgozását viszont ebben az alfejezetben nem részletezzük tovább, inkább magára a folyamatra koncentrálunk, hogy hogyan lehet megvalósítani a gyakorlatban az értesítés létrehozását és a felhasználókhoz történő eljuttatását.

Van az osztályban továbbá három metódus:

1. Az első a **via()** amelyben azt tudjuk meghatározni, hogy hogyan értesítsük a felhasználót, milyen módon vagyis csatornán keresztül, például e-mailben, SMS-ben vagy valamilyen belső, adatbázisban tárolt üzenettel az alkalmazásban. Viszont minden ilyen csatornának létre kell hozni egy segédfüggvényt a fájlban. Az e-mail küldésre már meg is van a metódus a fájlban arról, hogy hogyan nézzen ki, vagyis mit tartalmazzon az e-mail. Ha például SMS-t szeretnénk küldeni, akkor azt is ezen a nagyon jól olvasható és értelmezhető API-n keresztül kell megvalósítani.
2. A **toMail()** metódus határozza meg, hogy milyen részei legyenek például az e-mail-nek, amikor tájékoztatjuk a felhasználót valamilyen esemény bekövetkezéséről.
3. A **toArray()** pedig az alkalmazáson belüli adatbázisban tárolt értesítésnek a tartalmi szerkezetét lesz képes eltárolni.

Értesítést kétféle módon tudunk küldeni:

1. **Notifiable** trait használatával a **notify()** metódus segítségével. Ez a trait a **User** Model osztályunkban alapértelmezetten importálva van.

2. **Notification** Facade segítségével, annak leggyakrabban a **send()** metódusával.

12.3.2. Értesítés küldésének előkészítése

A filmes alkalmazásunkat használhatjuk az értesítésekkel való ismerkedés során is. Ehhez szükségünk lesz még egy új mezőre a **movies** adattáblában és a **Movie** Model osztály **$fillable** mezőjében: **premier_date**, ami dátum típusú lesz az adattáblában, hozzuk ehhez létre a migrációs fájlt, majd hajtsuk végre a migrálást!

```
php artisan make:migration add_premier_date_to_movies_table
```

Az új mező definíciója a migrációs fájlban:

```
$table->date('premier_date')->after('title')->nullable();
```

Megjegyzés: az **after()** az SQLite-ban nem működik: hiába van ott, ugyanúgy az oszloplista végére szúrja be az új mezőt.

A **MovieController** osztály már tartalmaz egy **show()** és **index()** metódust is. Most viszont a **create()** és **store()** metódusra lesz szükségünk. Ezzel párhuzamosan az útvonalaknál a meglévő kettőt töröljük ki, és helyezzük el a **web.php**-ban a **movies** erőforrás útvonalát, így mind a 7 RESTful útvonal, nevekkel együtt regisztrálásra kerül.

A **MovieController create()** metódusa a **movies.create** nézettel térjen vissza. A létrehozandó **movies / create.blade.php** fájl tartalmazzon egy nagyon egyszerű űrlapot, amellyel új filmeket tudunk felvinni az adatbázisba:

```
<form action="{{ route('movies.store')}}" method="post">
  @csrf
  <label for="title">Title:</label><br>
  <input type="text" id="title" name="title"><br>
  <label for="premier_date">Premier date:</label><br>
  <input type="date" name="premier_date" id="premier_date"><br>
  <label for="score">Initial score:</label><br>
  <input type="number" name="score" id="score" value="1"><br>
  <input type="submit" value="Save">
</form>
```

12–20. kódrészlet: Egyszerű űrlap film létrehozásához

Megjegyzés: itt most az alkalmazás szépségére (sablon, komponensek, design), továbbá a felhasználói hitelesítésre, engedélyezésre, az űrlap adatok validációjára stb. nem helyezünk hangsúlyt, csak az értesítések szempontjából vizsgáljuk a folyamatot, de természetesen egy valós alkalmazásnál nem tekinthetünk el az imént felsorolt folyamat részektől.

A létrehozó űrlap ezen a linken érhető el: http://127.0.0.1:8000/movies/create

A **store()** metódusba fogjuk elhelyezni az üzenetküldést, amihez a **Notification** Facade-ot használjuk először.

```
public function store() {
```

```
    Notification::send(User::first(), new MoviePremier());
}
```

12–21. kódrészlet: Űrlap elküldése után értesítést küldünk az első felhasználónak a film létrehozásáról

Ezzel gyakorlatilag értesítést küldünk a legelső felhasználónak, az értesítés típusát pedig a **send()** metódus második paramétere határozza meg. Ha a begépeléskor nem importálta volna a kódszerkesztő az hivatkozott osztályokat, akkor tegyük meg mi magunk manuálisan.

12.3.3. E-mail értesítés elküldése

Teszteljük is a film létrehozását, és nézzük meg, hogy mi történik a **store()** metódus végrehajtása során!

Az űrlap elküldésének hatására látszólag nem történik semmi, egy minimális idejű töltés után egy üres lapot fogunk kapni. Eltárolást még nem végeztünk, így az új film nem is kerülhetett be a **movies** táblába. A **MoviePremier** osztály **toMail()** metódusa hajtódott végre, amely egy *„e-mail üzenetet"* akart kiküldeni az 1-es számú teszt felhasználónknak. Mi viszont még nem állítottunk be semmilyen e-mail szolgáltatót, viszont ha megnyitjuk az **.env** fájlt, akkor azt láthatjuk benne, hogy a **MAIL_** előtagú beállítások pontosan erre valók. A **MAIL_MAILER** beállítás értéke jelenleg **log** (*megjegyzés:* attól függően, hogy milyen verziójú Laravel-ben kezdünk el dolgozni, ennek a paraméternek az értéke lehetett volna **smtp** is alapértelmezetten, ekkor azonban valószínűleg hibát kaptunk volna a film létrehozási teszt során, úgyhogy legyen **log** az értéke), úgyhogy nyissuk meg a **storage / logs / laravel.log** fájlt! Ha a fájl vége felé görgetünk, akkor már meg is találjuk a *„kiküldött"* e-mail-ünk forrását:

storage > logs > laravel.log

```
[2024-06-25 08:59:33] local.DEBUG: From: Laravel <hello@example.com>
To: test@example.com
Subject: Movie Premier
MIME-Version: 1.0
Date: Tue, 25 Jun 2024 08:59:33 +0000
Message-ID: <dfff924378ed94f120d86ef1a9dd2ac0@example.com>
Content-Type: multipart/alternative; boundary=8UWgpZmB

--8UWgpZmB
Content-Type: text/plain; charset=utf-8
Content-Transfer-Encoding: quoted-printable

Laravel: http://localhost

# Hello!

The introduction to the notification.

Notification Action: http://127.0.0.1:8000

Thank you for using our application!

Regards,
Laravel

If you're having trouble clicking the "Notification Action" button, copy and paste the URL below
into your web browser: [http://127.0.0.1:8000](http://127.0.0.1:8000)

© 2024 Laravel. All rights reserved.
```

12–16. ábra: Kiküldött (log-ban eltárolt) e-mail forrása

Az e-mail üzenetben láthatjuk, hogy a *From* értéke az **.env** fájlban megtalálható **MAIL_FROM_ADDRESS** változó értéke. A *To* az 1-es számú felhasználó e-mail címe. Továbbá a **MAIL_HOST** és **MAIL_FROM_NAME** paraméterek értékei is bekerültek az e-mail törzsébe, a **MoviePremier** osztály **toMail()** metódusában meghatározott értékeken, szövegezésen túl.

Ez tehát így működik, bár még eléggé *„fapados"* a megoldásunk.

Bővítsük a folyamatot és hajtsuk végre a következőket:

- mentsük el az új filmet az adatbázisba,
- irányítsuk vissza a filmes listázó oldalra a látogatót a **store()** metódus végén,
- a létrehozónak adjunk visszajelzést, hogy a film premier beállítása megtörtént, és erről üzenetet küld a rendszer az 1-es számú felhasználónak,
- az üzenet szövegezését egy kicsit pontosítsuk.

A módosított **store()** metódus a **MovieController**-ben:

```
public function store()
{
  Movie::create(request()->all());
  Notification::send(User::first(), new MoviePremier());
  return redirect(route('movies.index'))->with('message', 'Movie created.');
}
```

12–22. kódrészlet: Új film eltárolása, értesítés kiküldése, film létrehozásának jelzése

Az filmes listázó (**movies.index** nézet) oldal legtetejére szúrjuk be ezt:

```
@session('message')
  <div>{{ session('message') }}</div>
@endsession
```

12–23. kódrészlet: Egyszer felvillanó üzenet helye a filmes listázó oldalon

A **MoviePremier** osztály **toMail()** metódusában a visszatérési értéket módosítsuk erre:

```
public function toMail(object $notifiable): MailMessage
{
  return (new MailMessage)
    ->line('Movie created.')
    ->action('Movies\' list', url('/movies'))
    ->line('Thank you for using our application!');
}
```

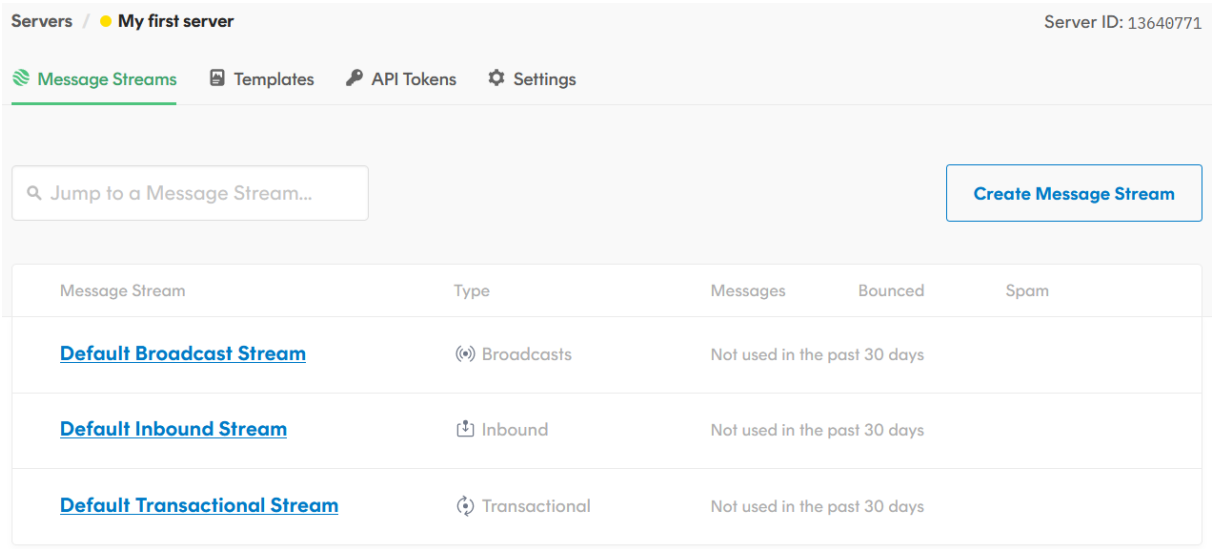
12–24. kódrészlet: Új üzenettörzs a film létrehozásáról

Egy újabb film létrehozásával tesztelhetjük a funkcionalitásokat: a film létrejön, létrehozás után rögtön elirányít minket a rendszer a filmes listázó oldalra, és felül látható az egyszer megjelenítésre kerülő üzenet, az e-mail pedig ugyanúgy bekerül a **laravel.log** fájl végére, már az új szövegezéssel.

12.3.3.1. Levelező kiszolgáló használata

A Laravel korábbi verzióinál a Mailtrap.io szolgáltatót támogatta alapértelmezetten a rendszer, de a 10 és 11 verzióknál már a Postmark szolgáltatót és driver-ét. Alapértelmezett címszó alatt azt értem itt, hogy a **config / mail.php** fájlban a **postmark**-hoz van külön beállítás, de természetesen bármilyen nyilvános vagy privát SMTP szolgáltató szóba jöhet, amihez van hozzáférésünk, azt fogjuk tudni használni a levelek kiküldésére. Kezdetben itt a Postmark-ot használjuk, de mivel annak a válaszideje rendkívül hosszú volt a tesztelések során (~90 másodperc), ezért a későbbi példákban a Mailtrap lesz ismét használatba véve.

Használjuk először a Postmark-ot, menjünk fel az oldalára, regisztráljunk, majd jelentkezzünk be! Rögtön a szerverek felülete fogad minket, és létre tudunk hozni egy újat, hogy el tudjuk különíteni majd a leveleinket ügyfelek, esetleg környezetek között: kattintsunk a *„Create Server"* gombra! Adjunk a szervernek nevet (My first server), és választhatunk neki színt is, ha több szerverünk lesz, akkor a könnyebb megkülönböztetés miatt lehet ez hasznos. A szerver típusa viszont már fontosabb beállítás, mivel ennek a megváltoztatása később nem lehetséges. Válasszuk a **Live**-ot, de most még csak tesztelésre használjuk a szerverünket a tanulási fázisban. A folyamat zárásaként nyomjuk meg a *„Create Server"* gombot! Utána létre is jön már az új szerverünk:



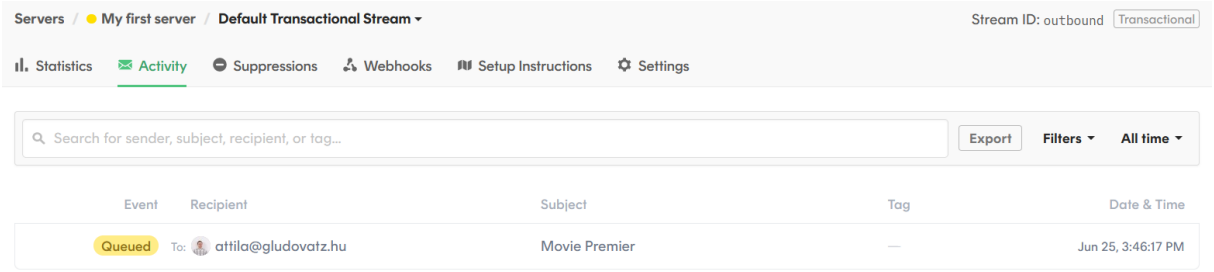
12–17. ábra: Saját levelező szerver a Postmark szolgáltató oldalon

Válasszuk ki a *„Default Transactional Stream"* értéket a táblázatból, majd kattintsunk a megjelenő ablakban a *„Settings"* lapfülre. Tekintsük meg az *„SMTP"* szekciót az oldalon, ez tartalmazza az SMTP szerver nevét (címét), a hozzáférési felhasználónevet és a jelszót. Másoljuk le, és illesszük be ezeket az értékeket a webes alkalmazásunk **.env** fájljába.

- MAIL_MAILER értékét írjuk át **smtp**-re a log-ról
- MAIL_HOST=smtp.postmarkapp.com
- MAIL_PORT=2525
- MAIL_USERNAME és MAIL_PASSWORD értékeit másoljuk ki a Postmark szerver beállítási oldaláról (elvileg ugyanaz a két érték és jó hosszúak)

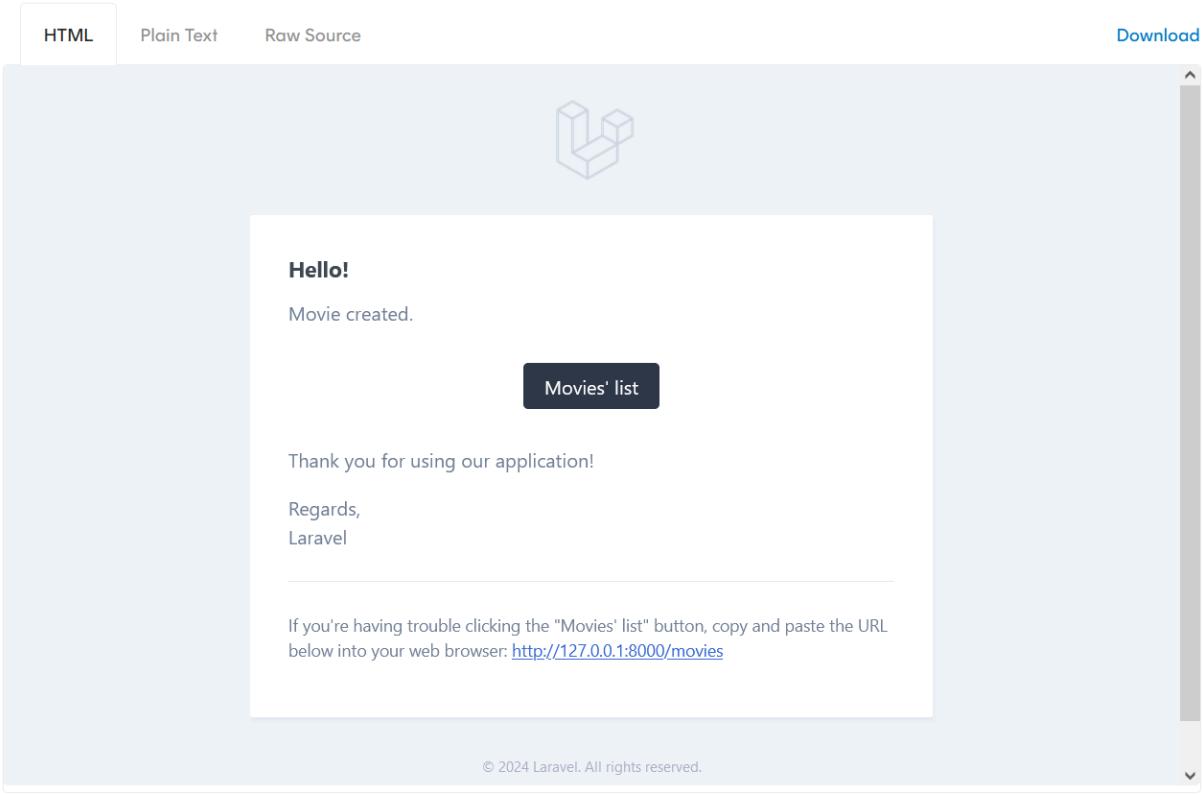
Regisztráció (és a szerver létrehozás) után 24 óráig teszt fázisban van a felhasználónk (legalábbis ezt írja az oldal, viszont én azt tapasztaltam, hogy már napok óta ugyanolyan teszt fázisban vagyok, mint az elején), így vannak bizonyos korlátozások, amelyeket be kell tartani ahhoz, hogy e-mailt tudjunk küldeni a szolgáltató segítségével. Példa a korlátozásra: a címzett címének ugyanazt a tartományt (domain) kell használnia, mint a küldő címének. A küldő e-mail címe az **.env** fájl MAIL_FROM_ADDRESS paraméter értéke, a címzett pedig az 1-es számú felhasználónk e-mail címe, ami text@example.com a seed-elés következtében. Emiatt a két domain-t állítsuk be azonosra (maradhat az example.com, esetemben mindkettőt átírtam gludovatz.hu-ra).

Az új film sikeres feltöltése után néhány pillanattal (~1,5 percbe is beletelt) megjelent az *„Activity”* lapfülön az elküldött e-mail:



12–18. ábra: E-mail értesítés a Postmark postaládában

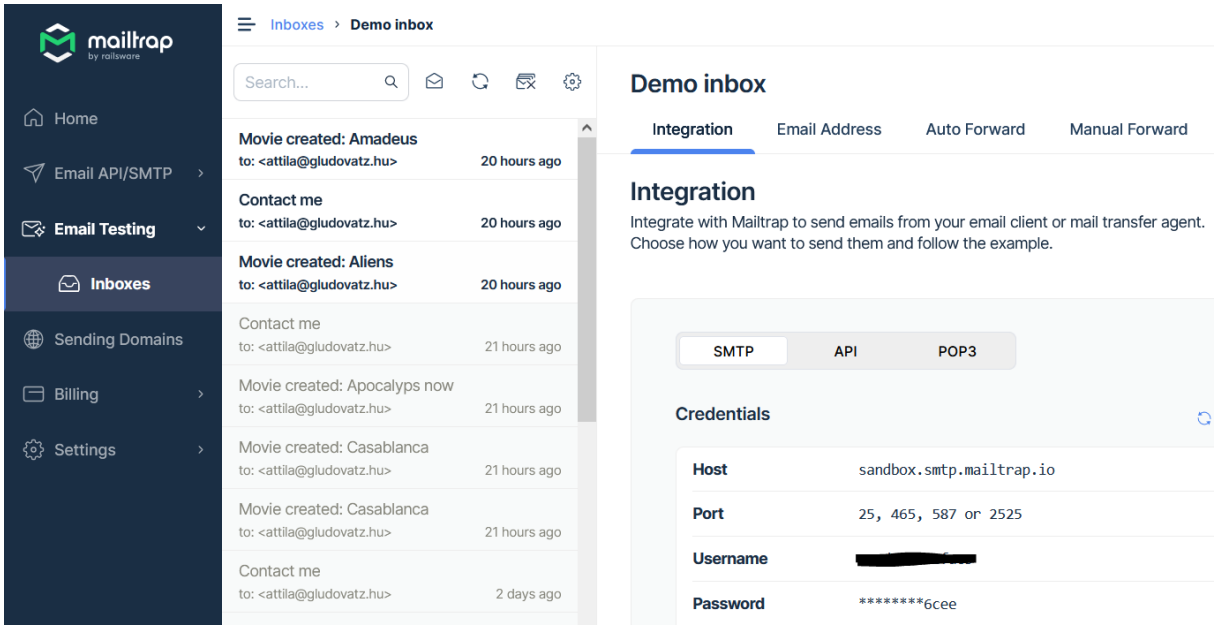
A Postmark oldalon a levél tárgyára rákattintva láthatóvá váltak a részletesebb információk, és maga a HTML levél is:



12–19. ábra: E-mail értesítés HTML formában

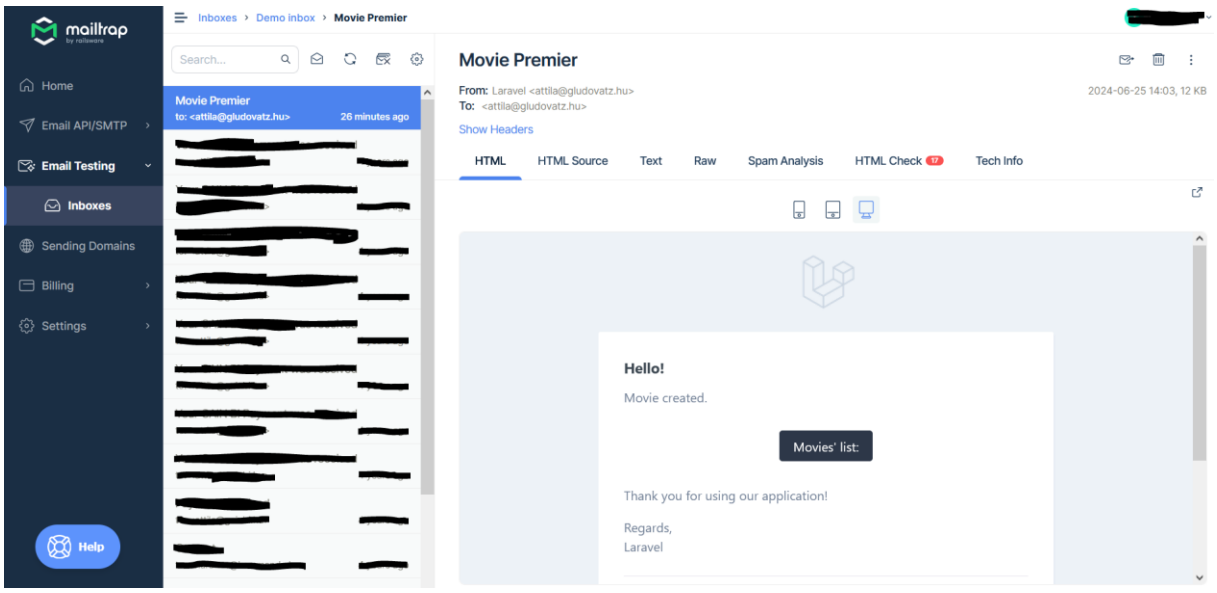
Ezek a szolgáltatók nem konkrétan azért vannak, hogy e-mail-eket küldjünk velük, bár arra is alkalmasak, hanem azért hasznosak nekünk, mert látjuk, hogy működik az e-mail küldő funkció a saját alkalmazásunkban. Innentől kezdve pedig az iménti megjegyzés alapján is, csak átállítjuk a megfelelő paramétereket az **.env** fájlban és már működik is már jól kipróbált, letesztelt éles e-mail küldés is.

Ha a Mailtrap.io kiszolgálóra váltunk, ami szintén ingyenes (én korábban regisztráltam oda és használtam is már), akkor az **.env** fájl beállításainál csak a **MAIL_HOST** (sandbox.smtp.mailtrap.io), **MAIL_USERNAME**, **MAIL_PASSWORD** értékeket kell átírni és már működik is minden ugyanúgy.



12–20. ábra: A létrehozott Demo inbox SMTP beállításai (jobb alul) beírhatók az .env fájlunkba

A levél ott azonnal megjelenik a postaládában, nem kell rá várakozni. Mailtrap kiszolgálónál így látszódik a levél:



12–21. ábra: Levelezés a Mailtrap kiszolgálónál, példa e-mail ebbe a postaládába is megérkezett

Ezért fejlesztés és tesztelés szempontjából a Mailtrap-et javaslom (2024-ben), mivel sokkal gyorsabban érkezik meg az az e-mail, amit a Laravel alkalmazásunkkal küldünk ki, így a munkánkat is meggyorsítja ez.

12.3.3.2. E-mail testre szabása

Azt már megvalósítottuk, hogy a **MoviePremier** Notification osztályban **toMail()** metódusban megváltoztattuk a levél szövegét, tartalmát, de természetesen ennél sokkal több beállítási lehetőségünk van még a levéllel kapcsolatban.

12.3.3.2.1. Levélküldés paramétereinek módosítása

A levélküldésnél lehetőségünk van módosítani a következőket a **toMail()** metódus visszatérési értékének metódusláncolatával:

- a feladó adatait (név, e-mail cím),
- a levél tárgyát,
- módosíthatjuk a köszöntést,
- mellékleteket fűzhetünk hozzá (a fájl mellékletekre különböző módokon reagálnak a szolgáltatók, érdemes ennek először utánanézni, mielőtt ezzel próbálkoznánk).

Ezek közül néhányat ki is próbálhatunk, például így:

```
public function toMail(object $notifiable): MailMessage
{
  return (new MailMessage)
    ->from('attila@gludovatz.hu', 'Attila Gludovatz')
    ->greeting('Hi ' . \App\Models\User::first()->name . '!')
    ->subject('Movie created')
    ->line('New movie was created!')
    ->action('Movies\' List', url('/movies'))
    ->line('Thank you for using our application!');
}
```

12–25. kódrészlet: Levél és a küldési paraméterek testre szabása

Az új film létrehozásakor már ez a módosított levél fog megérkezni a Mailtrap postaládánkba.

12.3.3.2.2. Adatátadás a Controller-től a Notification példánynak

Adatot is tudunk átadni az értesítésnek: a **MovieController store()** metódusát módosítsuk úgy, hogy kinyerjük először a létrejövő film azonosítóját, majd átadjuk azt a **MoviePremier** osztály konstruktorának:

```
public function store() {
  $movie = Movie::create(request()->all());
  Notification::send(User::first(), new MoviePremier($movie->id));
  return redirect(route('movies.index'))->with('message', 'Movie created.');
}
```

12–26. kódrészlet: Adat átadása a Controller-től a Notification példánynak

A **Notification** osztályban ezt egy változóban tároljuk el, majd a változónak a konstruktor segítségével adjunk értéket:

```
protected $movieId;

public function __construct($movieId)
{
  $this->movieId = $movieId;
}
```

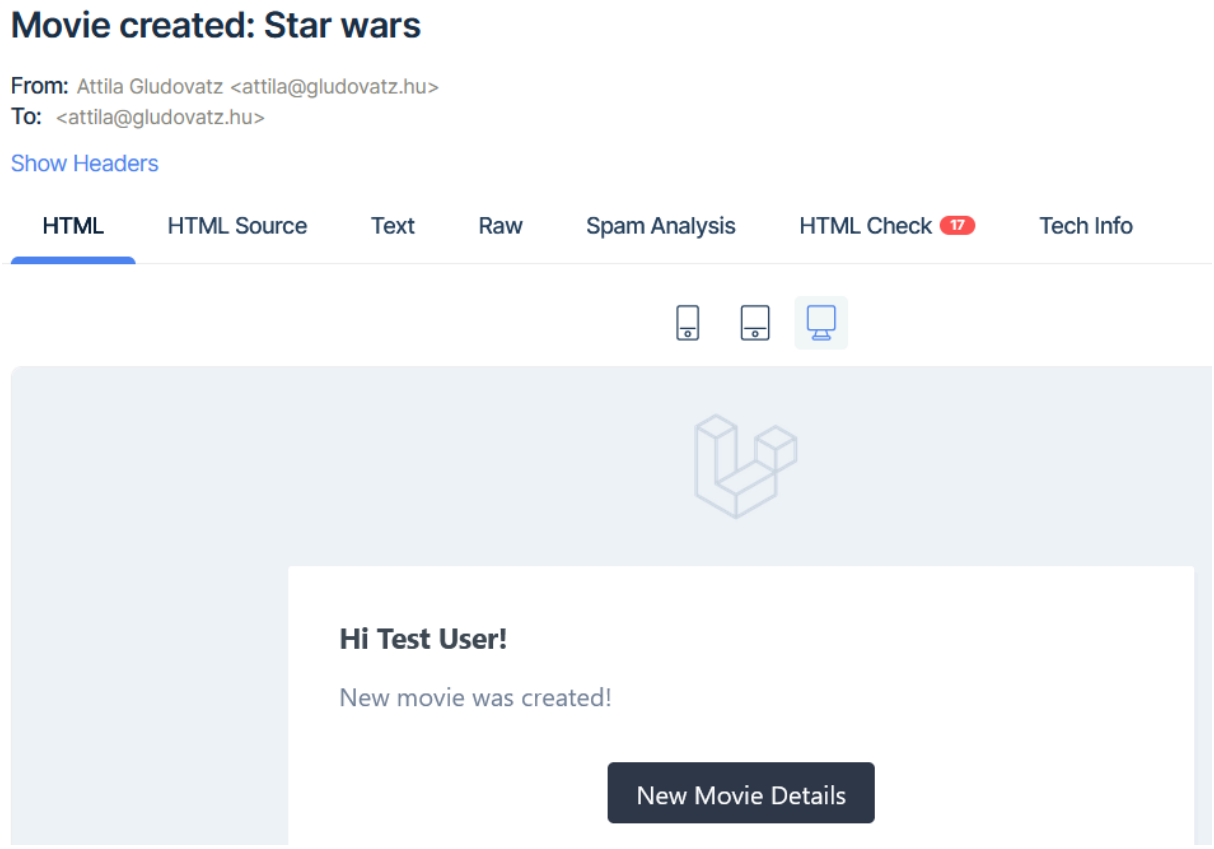
12–27. kódrészlet: Tagváltozónak értékadás a Notification osztály konstruktorában

Ezután ismét módosítsuk a **toMail()** metódus visszatérési értékét és helyezzünk el benne további adatokat az új filmmel kapcsolatban:

```
public function toMail(object $notifiable): MailMessage
{
  return (new MailMessage)
    ->from('attila@gludovatz.hu', 'Attila Gludovatz')
    ->greeting('Hi ' . \App\Models\User::first()->name . '!')
    ->subject('Movie created: ' . Movie::find($this->movieId)->title)
    ->line('New movie was created!')
    ->action('New Movie Details', url('/movies/' . $this->movieId))
    ->line('Thank you for using our application!');
}
```

12–28. kódrészlet: Adatok hozzáadása a levélhez

Az új film létrehozása után a kiküldött levél már így néz ki a postaládában:



12–22. ábra: Kiküldött levél az adatokkal

Fent a levél tárgya tartalmazza a film címét (*„Star wars”*), a *„New Movie Details”* gomb megnyomása pedig az új film adatlapját (**movies.show**) nyitja meg.

12.3.3.2.3. Levél kinézetének módosítása: Blade, HTML, CSS, Markdown

Lehetőségünk van a levél kinézetének módosítására is, ehhez mindössze publikálni kell az értesítési sablonokat, amelyek utána már teljes mértékben testre szabhatók. Adjuk ki a következő utasítást:

```
php artisan vendor:publish --tag=laravel-notifications
```

Ennek hatására átmásolódik egy nézet fájl a Laravel forráskódjából a **resources / views / vendor / notifications** mappába: **email.blade.php**

A fájlt megnyithatjuk és láthatjuk, hogy Blade direktívákkal és komponensekkel van elhelyezve benne a tartalom. Látható például, hogy minden tartalom egy **mail::message** komponensben van benne: a köszöntés, a bevezető sorok, az akció gomb, ami a levél címzettjét az adott oldalra irányítja el stb. Ez a **mail::message** komponens azonban még nem látszódik, így nem is lehet különösebben testre szabni. Adjuk ki a következő utasítást:

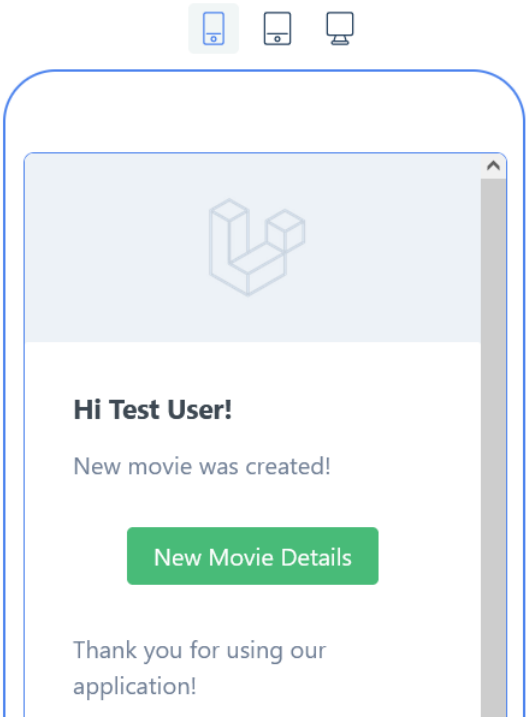
```
php artisan vendor:publish --tag=laravel-mail
```

A hatására szintén a **resources / views / vendor** mappa bővült: bekerült egy **mail** mappa, amelynek **html** almappájában ott van a **message.blade.php**. Erre a komponens fájlra hivatkozott az **email.blade.php**. Maga a **message.blade.php** tartalmazza a **mail::layout** komponenst, amely (**layout.blade.php**) fájl már ténylegesen HTML és CSS kódokat tartalmaz. A HTML-en és a beágyazott CSS kódokon kívül publikálásra került itt a **themes** mappában egy **default.css** fájl, ami a további stílusszabályokat tartalmazza. Ha ebben megváltoztatjuk a stílusok szabályait, akkor a következő levélküldésnél már az új szabályok szerint fogja felépíteni a rendszer a levél tartalmát és kinézetét. Ha elkezdünk a színekkel játszani, például a gombnál, ami a filmhez vezeti a felhasználót, akkor láthatjuk, hogy alapból **primary** a gomb osztálya, ez pedig a **default.css**-ben ilyen stílusszabályokat jelent:

```
.button-primary {
  background-color: #2d3748;
  border-bottom: 8px solid #2d3748;
  border-left: 18px solid #2d3748;
  border-right: 18px solid #2d3748;
  border-top: 8px solid #2d3748;
}
```

12–29. kódrészlet: A button-primary stílus szabályai a default.css fájlban

Természetesen ezek a háttér és keret színértékek megváltoztathatók szabadon, de a gomb stílusára az értesítés szintjével (**level**) is tudunk hatni, például, ha a küldésnél a **toMail()** metódusban a visszatérési értékhez hozzáadjuk az **error()** vagy **success()** segédmetódust. Itt most a film létrehozása sikeres lesz, úgyhogy adjuk hozzá a **success()**-t, de nyilván ha azt érzékelnénk egy vizsgálat során, hogy sikertelen volt a létrehozás, akkor az **error()** metódust kellene belefűzni a metódus láncolatba. Utána hozzunk létre egy újabb filmet és már a **.button-success** osztálynak megfelelően zöld hátterű gombot fogunk kapni:



12–23. ábra: Sikeres film létrehozáskor a gomb színe zöldre változik (CSS módosítás nélkül)

Megjegyzés: a Mailtrap-en lehetőségünk van arra is, hogy a reszponzívitást teszteljük, mivel gombok segítségével tudjuk kiválasztani, hogy telefonos, táblagépes vagy nagyobb monitoros nézetben szeretnénk megjeleníteni a levél tartalmát. A levél alapértelmezetten reszponzív, így minden felbontásban (szélességben) megfelelő kinézettel fog rendelkezni.

Bár a HTML, CSS és a Blade direktívák ismeretével felvértezve teljesen testre tudjuk szabni a kiküldésre váró levelek kinézetét, sablonját, de lehetőségünk van arra is, hogy úgynevezett *markdown-okkal* építsük fel a levél szerkezetét. A markdown levelekben is használhatunk Blade komponenseket, direktívákat, de a markdown-os szintaktikát[22] is.

Hozzunk létre egy új markdown-os e-mail értesítőt:

```
php artisan make:notification ContactMe --markdown=mail.contact
```

A **ContactMe** Notification osztályban a **toMail()** metódust adjuk meg eszerint:

```php
public function toMail(object $notifiable): MailMessage
{
  return (new MailMessage)
    ->markdown('mail.contact')
    ->subject('Contact me');
}
```

12–30. kódrészlet: Markdown üzenet küldési paraméterei

[22] Markdown szintaktika ismertető weboldal: https://www.markdownguide.org/basic-syntax/

Az imént kiadott utasítás hatására nem csak az új **Notification** osztály jött létre, hanem vele együtt egy nézet fájl is, ami a **resources / views / mail / contact.blade.php** lett. Figyeljünk a névkonvenciókra, amit az utasításban meghatároztunk: **mail.contact**, az gyakorlatilag a nézet elérési útja is lett. Ha megnyitjuk ezt a nézetet, ami a levél sablonját adja meg, és markdown formázások használhatók benne, de ugyanígy találunk benne Blade direktívát, komponenst és HTML kódot is. Korlátozás tehát nincsen, inkább csak a lehetőségek közül választhatunk. Minimálisan módosítsuk ezt a **contact.blade.php**-t és benne a gomb komponenst:

```
<x-mail::button :url="'http://attila.gludovatz.hu/blog'">
```

12–31. kódrészlet: Kiküldendő e-mailben lévő gomb a blog oldalamra irányítja a felhasználót

A **MovieController store()** metódusához adjuk hozzá ezt az új értesítést is az előző után:

```
Notification::send(User::first(), new ContactMe());
```

12–32. kódrészlet: Új markdown értesítés küldése film létrehozás esetén

Ezekután visszatekintve a **resources / views / vendor / notifications / email.blade.php**-ra már az is tartalmazott markdown formázásokat, elég csak a **#**-tel kezdődő sorokat megnézni.

Kipróbálás után azt tapasztaljuk, hogy ez az e-mail küldés is ugyanolyan jól működik, mint a korábbiak, és a benne lévő gomb is megfelelően a blog oldalra irányítja a felhasználót kattintás után.

Még egyetlen egyszerűsítést hajtsunk végre a **MovieController store()** metódusában:

```
// Notification::send(User::first(), new ContactMe());
User::first()->notify(new ContactMe());
```

12–33. kódrészlet: User notify() metódussal üzenünk

Így még beszédesebb és érthetőbb, itt már nincs is szükség a **Notification** Facade-re, működik nélküle is (ha a másik üzenetküldést is átalakítjuk így, akkor a **Notification** Facade importálása ki is vehető a fájlból). Ugyanis, ha megnézzük a **User** Model-t, akkor abban láthatjuk, hogy tartalmazza a **Notifiable** trait-et, ami arra utal, hogy a felhasználókhoz lehet értesítéseket rendelni, és mi pont ezt használtuk most ki. Ha visszakövetjük a **Notifiable** trait forrását, akkor láthatjuk a **RoutesNotifications.php**-ban, hogy háromféle módon tudjuk értesíteni a felhasználót (**notify(), notifyNow(), routeNotificationFor()**). Az első kettő között időbeli különbség van, a **notifyNow()** azonnal kiküldi az értesítést, a **notify()** berakja a sorba, és ha esetleg ott több értesítés vár még a sorára, kiküldésre, akkor annak a sornak a végére rakja az új értesítést.

Az alfejezetben végrehajtott programkód módosítások ebben a GitHub commit-ben találhatók meg.

12.3.4. Adatbázis értesítések

Az adatbázis-értesítési csatorna az értesítési információkat egy adatbázis táblában (**notifications**) tárolja. Ez a tábla olyan információkat tartalmaz, mint az értesítés típusa, valamint egy JSON adatszerkezetet, amely az értesítés további adatait leírja.

12.3.4.1. Előkészítés

Kezdjük el kialakítani ehhez a környezetet: elsőként mindenképpen szükségünk van egy adattáblára az adatbázisban, ami eltárolja az értesítéseket. Futtassuk a következő parancsokat:

php artisan make:notifications-table

php artisan migrate

Így létre is jött az adatbázisunkban az új **notifications** adattáblánk. A struktúráját megvizsgálva láthatjuk, hogy az értesítéseknek van típusa, belső leíró adata a **data** mező (*JSON* formátumban egyszerű *text*-ként) és időbélyegei. A háttérben természetesen elkészült a **notifications** táblához tartozó migrációs fájl is, amit ha megnézünk, akkor látjuk, hogy az elsődleges kulcs az egy univerzális egyedi azonosító lesz, ami egy egyedi véletlen karaktersorozatot fog jelenteni minden adatsornál, továbbá még egy fontos időbélyeg van benne, ez a **read_at** mező, ami azt mutatja meg, hogy mikor olvasta el a felhasználó az értesítést. Ha ebben a mezőben konkrét érték (időbélyeg) szerepel, akkor az azt jelenti, hogy a felhasználó elolvasta már az értesítést.

12.3.4.2. Értesítés küldése és eltárolása adatokkal

Ha ezek után visszatérünk a **MoviePremier.php** fájlunkhoz, akkor a **via()** metóduson belül beállíthatjuk, hogy ne csak e-mail kerüljön kiküldésre, hanem adatbázisba is mentsük az értesítést. Ilyen egyszerűen tehetjük ezt meg:

```
public function via(object $notifiable): array
{
  return ['mail', 'database'];
}
```

12–34. kódrészlet: MoviePremier értesítés via() metódus visszatérése kibővítve a database értékkel

Egy újabb film létrehozásával tesztelhetjük, majd nézzük meg, hogy mi került be a **notifications** adattáblába:

- a **type** mező értéke az értesítési osztály nevét kapta meg: **App\Notifications\MoviePremier**
- a **notifiable_type** azt mutatja, hogy milyen típusú címzettet kellett értesíteni: felhasználót, vagyis **App\Models\User**
- a **notifiable_id** azt mutatja, hogy a címzett típusából melyik azonosítójút (az 1-es számú felhasználót) értesítettük.

A **_type** és **_id** postfixű mezők összetartoznak, együttesen határoznak meg egy **morph()** típusú kapcsolatot, ezek a mezők arra utalnak, hogy ez a **notifications** tábla nem egy másik konkrét táblával, hanem a **notifable_type**-ban meghatározott Model (adattábla) **notifiable_id** sorszámú elemével (adatsorával).

A **data** mező jelenleg még üres, és az időbélyegek is megfelelő értékeket kaptak (hiszen a felhasználó még nem nézte meg az értesítését). A **data** mező értéke azért üres, mert a **MoviePremier** osztályunkban a **toArray()** funkció visszatérési értéke is üres. Egészítsük ki ezt a megoldást úgy, hogy…

1. hozzáadunk a **MoviePremier** osztályhoz egy új változót: **$budget**
2. ennek értéket adunk a konstruktorral,
3. majd a **toArray()** visszatérési tömbjéhez a **$movieId** és **$budget** mezők értékeit is megadjuk.

```
protected $movieId;
protected $budget;
```

```
public function __construct($movieId, $budget)
{
  $this->movieId = $movieId;
  $this->budget = $budget;
}

public function toArray(object $notifiable): array
{
  return [
    'movieId' => $this->movieId,
    'budget' => $this->budget,
  ];
}
```

12–35. kódrészlet: Kiegészítő adatok hozzáadása az adatbázis értesítéshez

Utána pedig a **MovieController store()** metódusában módosítsuk az értesítést küldő utasítást, és egészítsük ki egy véletlen (kódba beégetett, de akár az űrlapban egy új mezőt is definiálhatunk hozzá, itt a **store()**-ban pedig a **request()** segédmetódussal kinyerhető) pénzösszeg értékkel:

```
Notification::send(User::first(), new MoviePremier($movie->id, 1234));
```

12–36. kódrészlet: Értesítés kiegészítése az új film azonosítójával és a készítési költségvetés (büdzsé) értékével

Az új film felvitelével tudjuk tesztelni az új funkcionalitást, tegyük is ezt meg!

Utána azt tapasztaljuk, hogy a **notifications** tábla új sorának **data** mezőjébe bekerült egy JSON objektum: **{"movieId": 118, "budget": 1234}**, ami az új film azonosítója (változó szám) és a költségvetés értéke.

12.3.4.3. Értesítések megtekintése

A hitelesített felhasználónak mutassuk meg az értesítéseit, majd utána csak a még nem olvasott értesítéseit! Ehhez azonban szükség van a projektünkben egy hitelesítési rendszerre, mert az 1-es számú felhasználó kapta eddig az értesítéseket, viszont bejelentkezni még nem tud a webes alkalmazásunkba a felhasználó. Telepítsük hozzá például a Breeze-t (a 10.2.1.1. alfejezet alapján). *Fontos:* arra figyeljünk, hogy ez gyakorlatilag kitörli a **routes / web.php** fájlunkat, így másolatként mentsük ki belőle az útvonalakat, amit a Breeze telepítése utána majd vissza tudunk illeszteni a fájlba.

Telepítés után már be tudunk jelentkezni a kezdőoldalon *(Log in)* az 1-es felhasználóval (a **users** táblában látható e-mail címmel és a **password** jelszóval).

Ezután hozzuk létre azokat az útvonalakat a **web.php**-ban lévő **middleware('auth')** útvonal csoportba, amelyek majd az értesítéseket mutatják a bejelentkezett felhasználónak.

```
Route::get('/notifications',
[\App\Http\Controllers\UserNotificationController::class, 'index'])-
>name('notifications.index');
```

```
Route::get('/notifications/{notification}',
[\App\Http\Controllers\UserNotificationController::class, 'show'])-
>name('notifications.show');
```

12–37. kódrészlet: Felhasználói értesítések listája

Ez a Controller osztály viszont még nem létezik, úgyhogy hozzuk létre:

php artisan make:controller UserNotificationController

Írassuk ki a bejelentkezett felhasználó értesítéseit az **index()** metódusban:

```
public function index()
{
  dd(auth()->user()->notifications);
}
```

12–38. kódrészlet: Bejelentkezett felhasználó értesítéseinek kiíratása

Ha felkeressük ezt az oldalt (http://127.0.0.1:8000/notifications), akkor megkapjuk, hogy ez egy **DatabaseNotificationCollection**, amely gyűjtemény **DatabaseNotification** elemeket tartalmaz. Ebből aztán kinyerhetőek az attribútumai, amelyből mi most a **data**-ra koncentrálunk. A felhasználónak meg szeretnénk mutatni, hogy milyen új film került be a rendszerbe, és annak mekkora volt a költségvetése.

Ha megnézzük a **vendor / laravel / framework / src / Illuminate / Notifications / DatabaseNotification** osztályt, akkor láthatjuk benne, hogy a **data** attribútum értékei átalakításra (kasztolásra) kerülnek:

```
protected $casts = [
  'data' => 'array',
  'read_at' => 'datetime',
];
```

12–39. kódrészlet: DatabaseNotification osztály átalakított mezői

A **data** mező értékét tehát kezelhetjük tömbként. Állítsuk össze a nézetnek küldeni kívánt adathalmazt az **UserNotificationController index()** metódusában:

```
public function index()
{
  $notifications = [];

  foreach (auth()->user()->notifications as $notification)
  {
    if($notification->data != null)
    {
      $notifications[] = [
        "id" => $notification->id,
        "movieTitle" => Movie::find($notification->data["movieId"])->title,
        "budget" => $notification->data["budget"]
      ];
    }
  }
```

```
  return view('notifications.index', [
    'notifications' => $notifications
  ]);
}
```

12–40. kódrészlet: Adattal rendelkező értesítések átküldése az értesítések lista nézetnek

Hozzuk létre a **resources / views / notifications / index.php** fájlt! A tartalma legyen a következő:

```
<h1>Notifications</h1>
<h2>New movie premiers</h2>
<ul>
  @foreach ($notifications as $notification)
    <li>
      <a href="{{ route('notifications.show', $notification["id"]) }}">
        Title: {{ $notification["movieTitle"] }}, Budget: {{
$notification["budget"] }}
      </a>
    </li>
  @endforeach
</ul>
```

12–41. kódrészlet: Adattal rendelkező értesítések felsorolása

Ha ráfrissítünk a böngészőben a **/notifications** útvonalra, akkor most már meg kell kapnunk azt az értesítést, ami a legutóbb hozzáadott filmet és a költségvetését tartalmazza.

Az értesítés maga egy link is, amire ha rákattintunk, akkor a **notifications.show** útvonal felé irányítja a felhasználót. Ezt az útvonal kérést a **UserNotificationController show()** metódusa kezeli le úgy, hogy az értesítést *„olvasottá”* teszi, majd át is irányítjuk a film adatlapjára.

```
public function show($id)
{
  $notification = DatabaseNotification::find($id);
  $notification->markAsRead();

  return to_route('movies.show', [
    'movie' => $notification->data["movieId"]
  ]);
}
```

12–42. kódrészlet: Értesítés megtekintése: olvasottá teszi utána a film adatlapjára továbbít

A Controller osztály előtt importáljuk az **Illuminate\Notifications\DatabaseNotification** osztályt.

A **to_route()** segédfüggvény új, ezzel még nem találkoztunk, így gyakorlatilag a **redirect(route())** dupla segédfüggvény használatot tudjuk egyszerűsíteni egy függvénnyel.

Ha most rákattintunk a **/notifications** oldalon az új filmre, akkor be fogja hozni a film adatlapját, a **notifications** adattáblában pedig az adott értesítés sorának **read_at** mezőjébe be fog kerülni az az időbélyeg (UTC időzóna szerint alapértelmezetten), amikor rákattintottunk az értesítés linkjére az oldalon.

Ezek után érdemes lenne nem az összes értesítést megjeleníteni az adott felhasználónak, hanem csak azokat, amelyeket nem olvasott el korábban (vagyis nem kattintott rá az adott értesítésre). Ehhez mindössze annyit kell tennünk, hogy a **UserNotificationController index()** metódusában, amikor a **foreach**-ben végig megyünk a bejelentkezett felhasználó értesítésein, akkor inkább a nem olvasott értesítésein (**unreadNotifications**) menjünk végig:

```
foreach (auth()->user()->unreadNotifications as $notification)
```

12–43. kódrészlet: Végig haladunk a bejelentkezett felhasználó nem olvasott értesítésein

Ennek megfelelően a **notifications.index** nézetet is átalakíthatjuk **@foreach** helyett lehet **@forelse** és az **@empty** részébe beírhatjuk, hogy éppen nincsen új üzenete a felhasználónak.

```
@forelse ($notifications as $notification)
  <li>
    <a href="{{ route('notifications.show', $notification["id"]) }}">
      Title: {{ $notification["movieTitle"] }}, Budget: {{
$notification["budget"] }}
    </a>
  </li>
@empty
  <li>You have no unread notifications at this time.</li>
@endforelse
```

12–44. kódrészlet: Olvasatlan értesítések megtekintése, ha nincs ilyen, akkor azt is jelezzük

Ez így működik is, visszakapjuk, hogy nincsen olvasatlan értesítésünk!

Még egyetlen dolgot végezzünk el: ha látható legalább egy olvasatlan értesítése a felhasználónak itt az értesítések oldalon, akkor legyen egy link arra vonatkozóan, hogy mindegyik értesítést olvasottá tehetjük egy kattintással.

Először adjuk hozzá az új hitelesítéssel védett csoportba az útvonalat, ügyelve arra, hogy a **notifications.show** útvonal elé kerüljön még be ez:

```
Route::get('/notifications/mark-as-read-all',
[\App\Http\Controllers\UserNotificationController::class, 'markAsReadAll'])
  ->name('notifications.mark-as-read-all');
```

12–45. kódrészlet: Mindegyik üzenet olvasottá tételének útvonala

Ha megnézzük, hogy a korábban egy **DatabaseNotification** példányra alkalmazott **markAsRead()** metódus honnan származik, akkor láthatjuk, hogy a **DatabaseNotificationCollection** osztály tagja (ami a **Collection** osztályból van kiterjesztve). Tehát ez a **markAsRead()** metódus nem csak egy-egy elemre hívható meg, hanem akár egy egész gyűjteményre is, ami nekünk most tökéletes, hiszen akkor csak ezt az egy sort kell meghívni a **foreach** ciklussal való léptetés és olvasottá tétel helyett:

```
public function markAsReadAll()
{
  auth()->user()->unreadNotifications->markAsRead();
  return to_route('notifications.index');
```

```
}
```

12–46. kódrészlet: Bejelentkezett felhasználó olvasatlan értesítéseinek olvasottá tétele

Készítsünk egy linket a **notifications.index** nézetbe, ami erre az új útvonalra irányítja a felhasználót kattintáskor! Szúrjuk be a nézet fájl végére ezt a feltételes elágazást:

```
@if (count($notifications) > 0)
  <a href="{{ route('notifications.mark-as-read-all')}}">Mark as read all
notifications</a>
@endif
```

12–47. kódrészlet: Minden olvasatlan üzenetet olvasottá tevő link

A linkre való kattintással az összes nem olvasott értesítése olvasottá válik a felhasználónak.

Az alfejezetben végrehajtott programkód módosítások ebben a GitHub commit-ben érhetők el.

12.3.5. Összegzés és további értesítések: SMS, Slack, Teams

Ezzel tehát megismertük az értesítések témakörét: létrehoztuk őket, kiküldtük az értesítéseket e-mailben egy-egy levelező szolgáltató segítségével. A levelek különböző paramétereit, szövegezését és kinézetét is testre szabtuk. Végül az adatbázis értesítéseket is elkezdtük használni, kilistáztuk a meglévő értesítéseket a bejelentkezett felhasználónak, majd lehetőséget adtunk arra, hogy olvasottá tegye őket egyesével és csoportosan is.

A Laravel-ből gyakorlatilag bármilyen csevegő alkalmazásba lehet értesítéseket küldeni, amelynek megfelelő felülete van az értesítések fogadásához. A Laravel dokumentációja konkrétan a telefonos SMS küldést említi még, illetve a Slack alkalmazásba küldött üzenetküldést részletezi, de lehetőségünk van például a Microsoft Teams-be is üzenetet küldeni. Ezekről a blog oldalamon, az #Értesítések (Notifications) címkét követve lehet a továbbiakban információt szerezni.

12.4. Események (Events)

A Laravel eseményei egy egyszerű megfigyelői minta (observer) implementációját biztosítják, amely lehetővé teszi, hogy feliratkozzon és figyeljen az alkalmazáson belül bekövetkező különböző eseményekre. Az eseményosztályok jellemzően az **app / Events** könyvtárban, míg a figyelők az **app / Listeners** könyvtárban tárolódnak.

Az események kiválóan alkalmasak arra, hogy a webes alkalmazásunk funkcióit, lehetőségeit szétválasszuk, mivel egyetlen eseménynek több figyelője is lehet, amelyek egymástól függetlenek.

Az eseménykezelés fogalmát és működését számos programozási nyelvből és keretrendszerből ismerhetjük, itt most mi megvizsgáljuk a Laravel-es megoldását.

Kezdésnek nézzük meg, hogy milyen események és figyelőik vannak a rendszerbe regisztrálva alapértelmezetten (az utasítást a későbbiekben is használjuk a kilistázásra, mivel a saját eseményeinket és figyelőiket is tartalmazni fogja a kapott eredménye):

```
php artisan event:list
```

```
Illuminate\Auth\Events\Registered ..................................................
⇂ Illuminate\Auth\Listeners\SendEmailVerificationNotification
Illuminate\Database\Events\QueryExecuted ...........................................
⇂ Illuminate\Foundation\Exceptions\Renderer\Listener@onQueryExecuted
Illuminate\Foundation\Events\LocaleUpdated .........................................
⇂ Closure at: /vendor/nesbot/carbon/src/Carbon/Laravel/ServiceProvider.php:57
Illuminate\Queue\Events\JobProcessed ...............................................
⇂ Closure at: /vendor/laravel/framework/src/Illuminate/Foundation/Exceptions/Renderer/Listener.php:33
Illuminate\Queue\Events\JobProcessing ..............................................
⇂ Closure at: /vendor/laravel/framework/src/Illuminate/Log/Context/ContextServiceProvider.php:38
⇂ Closure at: /vendor/laravel/framework/src/Illuminate/Foundation/Exceptions/Renderer/Listener.php:33
```

12–24. ábra: Laravel keretrendszer alapértelmezett eseményei és alattuk a figyelői (Laravel 11)

Itt például a legelső azt jelenti, hogy amikor egy felhasználó regisztrál a rendszerbe (esemény), akkor egy e-mail értesítés kerül kiküldésre számára (figyelő), hogy erősítse meg a regisztrációját. Ez tipikusan egy *„ha ez történik - akkor utána az legyen"* (if-then) kapcsolat, vagyis, ha megtörténik az esemény, akkor utána ez váltsa ki a lekezelését.

12.4.1. Saját események és figyelők létrehozása és regisztrálása

Továbbra is dolgozzunk a mozifilmes alkalmazásunkkal, és gondoljunk rá úgy, hogy filmet ezentúl csak felhasználó tölthet majd fel a rendszerbe. Amikor filmfeltöltés történik, akkor még mindig a *„kitüntetett szerepű"* 1-es felhasználó fog értesítést kapni róla, de az is kapjon róla értesítést, aki feltöltötte a filmet a rendszerbe. Továbbá, ha egy felhasználó legalább kettő filmet feltölt az alkalmazásba, akkor kapni fog egy rangot, mint aktívan tevékenykedő felhasználó, mivel hozzájárult az oldal adatainak bővüléséhez. Talán érezhető, hogy több mindennek kell már megtörténnie egyetlen film feltöltésekor, ilyenkor jó lenne ezeket a funkcionalitásokat összekötni, vagyis ahhoz az eseményhez kötni, ami kiindulópontja a további történéseknek.

Az eseményt és a figyelőjét is artisan paranccsal fogunk tudni létrehozni:

```
php artisan make:event MovieCreated
```

```
php artisan make:listener SendPremierNotification --event=MovieCreated
```

A parancsok működnek paraméter nélkül is, de akkor a rendszer egy prompt-on (kérdésen vagy kérdéseken) keresztül szeretné megtudni, hogy mi legyen az adott esemény neve, illetve a figyelőnél még, hogy melyik esemény bekövetkezését figyelje, reagálja le.

Az első utasítás hatására az **app** mappában létrejön egy új **Events** mappa, benne pedig a **MovieCreated.php** fájl és az osztálya. De nem csak egy üres osztály jött létre, hanem az Event-ekre jellemző sablon szerint hozta létre nekünk a rendszer. Jó sok osztályt be is importált itt nekünk a rendszer és néhányat trait-ként alkalmaz az osztályon belül. De ezek közül a legtöbb az üzenet szóráshoz (broadcasting) kapcsolódik, amire nekünk első körben nem lesz szükségünk, csak akkor, ha valamilyen szerver oldali eseményt szeretnénk lekezelni úgy, hogy az összes kliensnek jelzést küldünk róla. Erre most nincs szükség, szóval a **broadcastOn()** metódust törölhetjük az osztályból, ezzel együtt az **InteractsWithSockets** trait-et is törölhetjük. Így az importálások többségére nincsen szükség, a VSCode el is halványítja ezeket, mivel az osztályban nem használjuk ezek (**Channel, InteractWithSockets,**

PresenceChannel, PrivateChannel, ShouldBrodcast) részeit, funkcionalitásait. Ami így megmaradt az már sokkal-sokkal egyszerűbb, átláthatóbb.

Tehát még egyszer, egy Event osztály egy eseményt reprezentál a rendszerünkben azzal az adattal, amit itt mi hozzáadunk, legyen az bármi. Például így alakítsuk át a MovieCreated osztály konstruktorát:

```
public function __construct(public Movie $movie)
{
  //
}
```

12–48. kódrészlet: MovieCreated esemény osztály konstruktora

Ez automatikusan létrehozza a publikus $movie tagváltozót, vagyis Movie Model osztály példányt ebbe az osztályba, és rögtön értéket is ad a mezőinek a konstruktor, amikor majd az esemény kiváltásra kerül.

Laravel 10 esetén az app / Providers mappában van az EventServiceProvider.php és benne az EventServiceProvider osztály, ami a kiindulópontja a Laravel teljes eseménykezelésének. Ebben található meg a $listen tömb, amely egy kulcs-érték párokból álló készletet reprezentál. A kulcsok az események, az értékek pedig a lekezelőik.

```
protected $listen = [
  Registered::class => [
    SendEmailVerificationNotification::class,
  ],
];
```

12–49. kódrészlet: EventServiceProvider-ben az esemény-figyelő összerendelése (Laravel 10)

A Laravel 10-ben tehát még manuálisan adtuk meg az esemény-figyelő összerendelést. A Laravel 11-ben ugye már nem láthatók ezek a rendszer magjához tartozó Service Provider osztályok. A Laravel 11 feltérképezi az app / Events és az app / Listeners könyvtárakat, és megkeresi az együttműködő eseményeket és figyelőiket. Ha mégis az alapértelmezettől eltérő mappákban szeretnénk tárolni őket, akkor a bootstrap / app.php-ban tudjuk megadni, hogy milyen mappát kellene figyelni az eredeti helyett. Ezt a ->withEvents(discover: […]) tömbben helyezhetjük el (a kipontozott részbe kell beírni a mappa elérési útját).

Ha artisan parancs segítségével hoztuk létre az eseményeket és figyelőiket, akkor az alapértelmezett mappákba kerültek be és sikeresen feltérképezi majd őket a keretrendszer.

12.4.2. Esemény kiváltása és lekezelése

Lépjünk vissza most a MovieController store() metódusához! Legelőször kommenteljük ki az értesítéseket küldő sorokat a metódusban, majd a fő logikát próbáljuk megvalósítani azzal, hogy az eseményt kiváltjuk/elindítjuk a film létrehozása után. Ezt két módon tehetjük meg:

1. az esemény osztály statikus dispatch() metódusával,
2. az event() segédmetódussal, paraméterként az esemény osztály egy új objektumával.

Helyezzük el a kódsorokat és válasszuk a kettő közül az egyiket (az alábbi példakódban látható még mindkettő, kommentelés nélkül, utána pedig egy javaslat olvasható a választásról):

```
MovieCreated::dispatch($movie);
event(new MovieCreated($movie));
```

12–50. kódrészlet: Esemény kiváltási módszerekre példák

Ez a két kódsor egyenértékű egymással. Ha az elsőt választjuk, és próbálunk lefúrni a dolgok mélyére, akkor a **Dispatchable** trait-ben látható, hogy van a **dispatch()** metódus, ami pontosan az **event(...)** segédmetódussal tér vissza, szóval teljesen ugyanaz mindkét megoldás, de az első egy kis kerülőúton jut el ugyanahhoz az **event()** segédmetódus híváshoz, mintsem ha rögtön azt használnánk. Úgyhogy most maradjunk annál a megoldásnál, amikor mi közvetlenül használjuk az **event()** segédmetódust, az első sor kikommentelhető. *Megjegyzés:* a **Dispatchable** trait-ban megtalálhatók még a **dispatchIf()** és **dispatchUnless()** metódusok, amelyek feltételvizsgálathoz kötik az esemény kiváltását, ezek szintén hasznosak lehetnek a jövőben.

Ez most gyakorlatilag azt jelenti, hogy a fő logika meghívása, vagyis az esemény kiváltása ezzel adott is lesz. Jeleztük a rendszer felé, hogy megtörtént a film létrehozása és a premierjének meghatározása. Most ezt valahogy le kellene kezelni:

1. egy jelzést kell küldeni az érintettek felé:
    a. a létrehozónak,
    b. az 1-es felhasználónak,
2. továbbá a létrehozó felhasználó rangját kellene megadni azáltal, hogy hány filmnél van ott létrehozóként (ha legalább kettő filmnél létrehozó, akkor megkapja a rangot).

Ezek a folyamatok viszont már mind a figyelő (listener), vagyis az eseménykezelő részhez tartoznak.

A **MovieController** és metódusainak elérését, vagyis a 7 **resource** útvonalat helyezzük át a **web.php**-ban abba az útvonal csoportba, ami az **auth** köztes rétegen vezeti át a kéréseket (a **UserNotificationController** útvonalai mellé/után). Így csak a hitelesített felhasználók férnek hozzá a filmekhez.

Az alkalmazás bővítése során egy előkészítéssel kezdhetünk, adjuk hozzá az eddig simán beégetett értéket tartalmazó **budget**-et a **movies.create** űrlapos nézethez, a **Movies** Model osztály **$fillable** mezőjéhez és egy új migrációs fájlt is generáljunk neki, úgy hogy a **budget** egy integer típusú nullázható mező: **$table->integer('budget')->nullable();**

A filmekhez (**movies** adattábla) adjuk hozzá a létrehozó felhasználót külső kulcsként, és persze a Model osztályokhoz (**Movie** és **User**) definiáljuk a megfelelő kapcsolatokat (**belongsTo()** és **hasMany()**), a Movie-nál pedig a **$fillable** mezőjéhez is adjuk hozzá a **user_id**-t.

A tapasztalati pont (feltöltött filmek száma) mezőt ne hozzuk létre a **users** adattáblában, mivel az a pontszám kiszámítható lesz majd abból, hogy hány filmnél volt létrehozó az adott felhasználó. Hozzuk létre a migrációs fájlt:

```
php artisan make:migration add_user_id_to_movies_table
```

Az új migrációs fájl tartalma pedig legyen ez:

```
public function up(): void
{
  Schema::table('movies', function (Blueprint $table) {
    $table->foreignId('user_id')->nullable()->constrained()-
>cascadeOnDelete()->cascadeOnUpdate();
  });
}

public function down(): void
{
  Schema::table('movies', function (Blueprint $table) {
    if (env('DB_CONNECTION') !== 'sqlite')
      $table->dropForeign(['user_id']);

    $table->dropColumn('user_id');
  });
}
```

12–51. kódrészlet: movies adattáblában a létrehozó felhasználó külső kulcsként

Megjegyzés: a **cascadeOnUpdate()** és **cascadeOnDelete()** metódusok csak könnyebb olvashatóságot biztosítanak az **onUpdate('cascade')** és **onDelete('cascade')** paraméteres metódusokhoz.

Mivel a Laravel 11-ben alapértelmezetten SQLite-tal dolgozunk, ezért a migrációs fájl **down()** metódusában figyeljünk arra, hogy nem tudjuk eldobni SQLite-ban a külső kulcs kényszert, csak magát az oszlopot. De most migráljunk!

A **MovieController store()** metódusában bővítsük a **user_id** értékkel a film eltárolását:

```
$movie = Movie::create(
  array_merge(
    request()->all(),
    ['user_id' => auth()->id()]
  )
);
```

12–52. kódrészlet: Film eltárolásánál a létrehozót is elmentjük

Tinker segítségével hozzunk létre egy új felhasználót, akit majd az új filmek feltöltésénél alkalmazhatunk:

```
App\Models\User::factory(['email' => 'uploader@example.com'])->create();
```

12–27. utasítások: Felhasználó létrehozása film feltöltéshez fix e-mail címmel

Kezdjük az esemény lekezelését a **SendPremierNotification** osztályban. *Megjegyzés:* értesítést lehetne küldeni úgy, ahogy már korábban is megtettük a **MovieController store()** metódusában, de itt most nem erre helyezzük a hangsúlyt, hanem az esemény lekezelési folyamatát szeretnénk végrehajtani és azt átlátni, ezzel együtt pedig a felelősségeket is szétválasztjuk az átláthatóbb működés érdekében. Általában

elmondható, hogy a keretrendszerben egy-egy probléma megoldására nem csak egy megoldás létezik, szerencsére. Így itt most megismerünk egy másik utat a megoldás felé!

A **SendPremierNotification** osztályban létrejött a **handle()** metódus, a paramétere pedig a **MovieCreated** esemény osztály objektuma. Ez az, amivel gyakorlatilag Laravel 11-ben összerendeljük az eseményt a figyelőjével, Laravel 10-ben még külön is jelezni kellett ezt a kapcsolatot az **EventServiceProvider** osztály **$listen** gyűjteményében (ha egy eseményhez több figyelő is tartozik, akkor azokat egyszerűen vesszővel elválasztva felsorolhatjuk ott). Laravel 11-ben, ha egy eseményhez több figyelő is tartozik, akkor ez egyszerűen úgy működik, hogy az adott figyelő osztályban lévő **handle()** metódus ugyanazt az esemény osztály objektumot kapja meg paraméterként, és akkor ahhoz lesz automatikusan hozzárendelve. Laravel 11-ben tehát ez az összerendelési folyamat sokkal egyszerűbb, kényelmesebb, de elérhetjük Laravel 10-ben is ugyanezt az automatikus feltérképezési (esemény-figyelő összerendelési) funkcionalitást úgy, hogy megnyitjuk az **EventServiceProvider**-t és a benne lévő **shouldDiscoverEvent()** metódus visszatérési értékét hamisról igazra állítjuk. Így már automatikusan fog menni a feltérképezés, ugyanúgy, mint a Laravel 11-ben.

Megjegyzés: attól függetlenül, hogy a Laravel 11-ben nincs már **EventServiceProvider**, ott is lehet manuálisan regisztrálni eseményeket és figyelőiket – bár kérdés, hogy mennyire van szükség manuális regisztrációra, ha betartjuk a szabályokat, és a megadott módon rendeljük össze az eseményt a figyelőjével. Az **app / Providers / AppServiceProvider** osztály **boot()** metódusában definiálhatnánk a manuális összerendelést (az **Event** itt egy Facade, és a többi osztályt is importáljuk hozzá):

```
Event::listen(
  MovieCreated::class,
  SendPremierNotification::class,
);
```

12–53. kódrészlet: Esemény és figyelőjének manuális összerendelése

Térjünk vissza az esemény egyik lekezelőjére, amely üzenetet küld két felhasználónak is. A megvalósításnál használjuk már azt a tudást, amit ennek a fejezetnek az elején, a gyűjtemények kapcsán megtanultunk, és alkalmazzuk az **each()** gyűjtemény segédmetódust.

```
public function handle(MovieCreated $event): void
{
  collect([User::first(), User::find($event->movie->user_id)])->each(
    fn ($user) =>
    $user->notify(new MoviePremier($event->movie->id, $user->id, $event-
>movie->budget))
  );
}
```

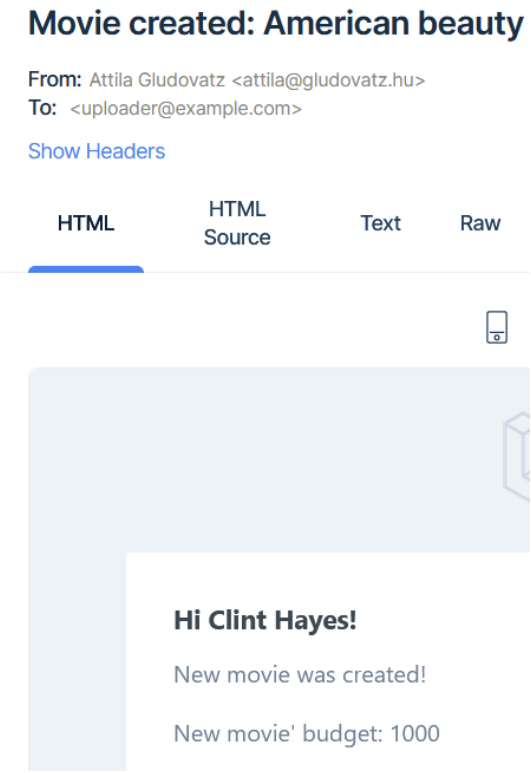
12–54. kódrészlet: MovieCreated esemény egyik figyelője, lekezelője

Viszont ahhoz, hogy ez működjön még módosítanunk kell a **MoviePremier** értesítési osztályt is, mivel most már nem csak az 1-es számú felhasználónak, hanem a film létrehozójának is küldünk értesítést, plusz a költségvetést is jelezzük az e-mail-es értesítőben. Ezeket a módosításokat az alábbiak szerint végezzük el:

```
protected $notifiableUserName;

public function __construct($movieId, $notifiableUserId, $budget)
{
  $this->movieId = $movieId;
  $this->notifiableUserName = User::find($notifiableUserId)->name;
  $this->budget = $budget;
}

public function toMail(object $notifiable): MailMessage
{
  return (new MailMessage)
    ->from('attila@gludovatz.hu', 'Attila Gludovatz')
    ->greeting('Hi ' . $this->notifiableUserName . '!')
    ->subject('Movie created: ' . Movie::find($this->movieId)->title)
    ->success()
    ->line('New movie was created!')
    ->line('New movie\' budget: '. $this->budget)
    ->action('New Movie Details', url('/movies/' . $this->movieId))
    ->line('Thank you for using our application!');
}
```

12–55. kódrészlet: MoviePremier értesítési osztály bővítése a felhasználóval és a költségvetéssel

Így az eseménykezelőből érkező adatok a két értesítés kiküldésénél már illeszkedik az értesítés tartalmához, mivel e-mail-enként különbözik a **greetings()** metódus tartalma, hiszen itt köszöntjük az adott felhasználót: az 1-es számút és a létrehozót. A film feltöltésnél bejelentkezett uploader@example.com példa felhasználó is megkapja így már az e-mail értesítést a nevére a megadott költségvetéssel együtt:



12–25. ábra: Értesítés a filmet létrehozó felhasználónak

A feltöltött videók alapján a felhasználók *„tapasztalati pontokat gyűjtenek”*, így elit rangot tudnak elérni az alkalmazásban. Az elit rang a **users** táblában jelenjen meg egy *boolean* értékként, ami alapértelmezetten hamisra legyen állítva. Ezt az új mezőt egy új migrációs fájllal tudjuk beszúrni a **users** táblába:

```
php artisan make:migration add_elite_to_users_table
```

Az **up()** metódusba: **$table->boolean('elite')->default(0);** a **down()**-ban pedig csak dobjuk el ezt a mezőt. Utána migráljunk!

Ezt a mezőt nem kell a **User** Model osztályunkban kitölthetővé tenni, mert majd az esemény lekezelőjével fogjuk beállítani, nem manuálisan a felhasználótól érkező űrlapos adatok segítségével.

Hozzunk létre egy másik eseménykezelőt, amely szintén a **MovieCreated** esemény bekövetkezésére reagál majd:

```
php artisan make:listener ConditionalEliteRankAddition --event=MovieCreated
```

A létrejövő eseménykezelő **handle()** metódusa:

```
public function handle(MovieCreated $event): void
{
  $user_id = $event->movie->user_id;
  $xp = Movie::where('user_id', $user_id)->count();

  if ($xp == 2) {
    User::where('id', $user_id)->update(['elite' => true]);
```

```
    }
}
```

12–56. kódrészlet: Elite rang megszerzése, ha már két filmfeltöltése van

Megjegyzés: a feltételvizsgálatnál, ha már több filmet töltöttünk fel tesztelésnél, mint 1, akkor érdemes az egyenlőség vizsgálat helyett nagyobb egyenlőre írni, de az éles programban elég lenne egyszer frissíteni az elit rangra a felhasználót, nem kell minden egyes újabb feltöltésnél update-elni az állapotát ugyanarra.

Így a felhasználónak a második filmfeltöltése során ez az eseményfigyelő is aktiválódik és a felhasználó elite státuszba kerül.

Ez az erőforrás létrehozási esemény egy nagyon kitüntetett szerepű esemény, mivel sokszor előfordul egy alkalmazás életciklusában, hogy valamilyen erőforrást létrehozunk. Ekkor megtehetjük azt, hogy ne feltétlenül az erőforráshoz tartozó Controller **store()** metódusában váltsuk ki az eseményt, hanem lehetőségünk van magában a Model osztályban, a **$dispatchesEvents** tömbben hozzáadni az eseményeket. Próbáljuk is ezt ki, majd teszteljük a megoldását, hogy ugyanúgy működik-e. Töröljük a **MovieController store()** metódusából az **event()** kiváltását, aztán a **Movie** Model osztályhoz adjuk hozzá a következőt:

```
protected $dispatchesEvents = [
  'created' => MovieCreated::class,
];
```

12–57. kódrészlet: Movie erőforrás létrehozásakor automatikusan kiváltódó esemény a Model osztályban

Ha most vagy bármikor máskor bővíteni szeretnénk az eseményhez köthető lekezelési logikák sorát, akkor csak egyszerűen létrehozunk egy új figyelő osztályt, ahogy legutóbb is, és a **handle()** metódusában megírjuk a további *„ mellékhatások"* kódjait, például, ha az adott felhasználónak a premierre akarunk egy ingyenjegyet is küldeni, vagy csak egy sorsolásról értesíteni, akkor semmi egyebet nem kell tennünk, hiszen minden más össze van kötve a rendszerünkben.

Az eseményekhez tartozó automatikus tesztek létrehozásához itt találunk segítséget, főleg az **assertDispatched()** és **assertNotDispatched()** metódusok lehetnek segítségünkre.

Az alfejezetben nem lett minden programkód módosítás részletezve, mivel már sokszor megcsináltunk bizonyos műveleteket, de ha valami nem működne tökéletesen, akkor ebben a GitHub commit-ben megtalálható az összes programkód hozzáadás és változás, amit itt elvégeztünk.

Laravel Reverb: ez az egyik legújabb csomag a Laravel fejlesztőitől. Segítségével valós idejű üzenetküldő rendszert tudunk építeni, amely lehetővé teszi számunkra, hogy figyeljük a backend oldalon történő eseményeket és reagáljunk rá frontend oldalon.

A Laravel Reverb további funkciói:

- valós idejű értesítéseket készíthetünk új üzenetekről, kommentekről, jelentésekről vagy rendszer eseményekről,
- valós idejű chat alkalmazást építhetünk,
- valós idejű vezérlőpultokat (dashboard) építhetünk élő adatokkal,

- értesíthetjük a felhasználókat a rendszer változásairól, úgy, mint például egy webshop esetében: új termékek, akciók stb.
- élő rendszert építhetünk úgy, hogy a felhasználókat nyilvántartjuk, ki van online és ki nincsen,
- és még sok minden más…

A Laravel Reverb hivatalos oldala itt érhető el.

12–1. újdonság: Laravel Reverb és funkciói

12.4.3. Model megfigyelők eseményei

Az Eloquent Model osztályok számos eseményt kiváltanak, amelyek lehetővé teszik a számunkra, hogy a Model életciklusának következő fontos pillanataiba, momentumaiba bekapcsolódjunk:

- lekérés (**retrieved**): amikor egy létező Model lekérdezésre kerül az adatbázisból.
- új Model objektum / adatsor létrehozáskor:
  - létrehozás előtt (**creating**)
  - létrehozás befejezésekor (**created**)
- meglévő Model objektum / adatsor frissítésekor:
  - frissítés előtt (**updating**)
  - frissítés befejezésekor (**updated**)
- új Model objektum létrehozásakor és meglévő frissítésekor is meghívásra kerülnek, akkor is, ha az objektum attribútumai nem kerültek frissítésre:
  - eltárolás előtt (**saving**)
  - eltárolás befejezésekor (**saved**)
- törlés előtt (**deleting**)
- törlés befejezésekor (**deleted**)
- soft delete befejezésekor (**trashed**)
- fizikai törlés előtt soft delete tulajdonság megléte esetén (**forceDeleting**)
- fizikai törlés befejezésekor soft delete tulajdonság megléte esetén (**forceDeleted**)
- helyreállítás előtt (**restoring**)
- helyreállítás befejezésekor (**restored**)
- egy Model objektum másolatának elkészítése előtt (**replicating**)

A **-ing** végződésű eseménynevek a Model objektumban bekövetkezett változások bekövetkezése előtt, míg a **-ed** végződésű események a Model objektumban bekövetkezett változások tárolása után történnek meg.

Ezek tehát szintén olyan kitüntetett szerepű esemény bekövetkezési momentumok, amit például már a 12–57. kódrészletben is láttunk (**created**). Az ott látott Model osztálybeli **$dispatchesEvents** az imént felsorolt objektum életciklusbeli momentumokat képes meghatározni, és azokhoz esemény osztályokat tud rendelni. Annyi csak az elvárás, hogy minden Model esemény osztály konstruktorának meg kell kapnia az érintett Model osztály egy példányát: ott a **MovieCreated** esemény osztálynál ez pontosan megvalósult.

Ha nem szeretnénk konkrét esemény osztályt definiálni, akkor lehetőségünk van arra is a Model osztályban, hogy a **booted()** metódusában definiáljunk momentumokhoz végrehajtási metódust. Például, ha kikommentezzük a **$dispatchesEvents**-ben lévő **created** momentumot és a hozzá tartozó **MovieCreated** eseményt, de ott a **Movie** Model osztályban létrehozzuk a **booted()** metódust a következő struktúrával:

```
protected static function booted(): void
{
  static::created( function (Movie $movie) {
    dd($movie);
  });
}
```

12–58. kódrészlet: Új film létrehozása után végrehajtódó esemény: a film adatainak kiíratása

Akkor az új film létrehozásánál azt fogjuk tapasztalni, hogy létrejön a film, és az eltárolás befejezés után most az e-mail üzenetek kiküldése helyett egy egyszerű kiíratása történt meg a létrejövő **$movie** objektumnak. Ide a **booted()** metódusba több másik fontos momentum lekezelés is bekerülhet, de arra figyeljünk, hogy túl sok mindent ne akarjunk megoldani itt, mert akkor átláthatatlanná válhat a működése.

Mivel az **-ing** végződésű eseményneveknél még nem történik meg az adott esemény (például a **creating**-nél az eltárolás az adattáblában az adott erőforrás elem), ezért ebben a fázisban még végezhetünk ellenőrzéseket az adott értékekkel kapcsolatban, vagy kiegészíthetjük őket, ha például éppen a bejelentkezett felhasználó azonosítóját szeretnénk hozzáfűzni az elmenteni kívánt adatstruktúrához, mint például a 12–52. kódrészletben megtettük a **user_id**-val. Így már látható, hogy azt inkább **creating** eseményhez kellene hozzákötni, tegyük is ezt meg, de előbb egyszerűsítsük le a **MovieController store()** metódusában a film létrehozást, csak ennyi maradjon ott (a merge() megvalósítása helyett):

```
Movie::create(request()->all());
```

12–59. kódrészlet: Film létrehozása az űrlapról érkező adatokkal (majd a creating eseménynél kiegészítjük a user_id-val)

Ha csak így hagynánk, akkor a rendszer nem mentené el az új film létrehozóját, mert a **user_id** mező hiányozna a filmeknél (**movies** táblában), NULL érték kerülne be a **user_id** mező helyére az új film adatsorában. A mezőt a migrációs fájlban nullázhatóvá tettük, mivel voltak már adatok a táblában, és ha kötelezően kitöltendő mező lett volna, akkor nem működött volna a hozzáadás (default érték nélkül).

A **Movie** Model osztály **booted()** metódusában kikommentelhetjük az ott lévő **created()**-et, aztán pedig adjuk hozzá a **creating()** metódust.

```
static::creating(function (Movie $movie) {
  $movie->user_id = auth()->id();
});
```

12–60. kódrészlet: Filmhez a létrehozó felhasználó azonosítójának hozzáadása a létrehozás előtt

Így már a Laravel programlogikája szerint történt meg a felhasználói azonosítóval való kiegészítése az új filmnek, és nem egy ódivatú PHP-s megoldással. A film létrehozásánál bekerül a létrehozó felhasználó **id**-ja az új film adatsorába.

12.4.4. Model megfigyelő osztályok (Observers)

Ha egy adott Model-nél sok eseményre kell figyelni, akkor egyszerűbb, ha a figyelőket egyetlen osztályba csoportosítjuk össze. A megfigyelő osztályok (Observer) metódusainak nevei tükrözik a fent ismertetett momentumokat, Eloquent eseményeket, amelyekre figyelni kell.

Egy paranccsal létre tudunk hozni egy ilyen *„megfigyelő"* osztályt:

```
php artisan make:observer MovieObserver --model=Movie
```

Az utasítás hatására létrejön az osztály az **app / Observers** mappában.

Az MovieObserver osztály azonban még nincsen automatikusan összekötve a Movie Model osztállyal. Az összekötést kétféleképpen valósíthatjuk meg:

1. A Model osztály neve felett jelöljük (importálásokkal együtt), hogy melyik osztály figyeli az eseményeit (12–61. kódrészlet).
2. Az **app / Providers / AppServiceProvider** osztály **boot()** metódusában van meghatározva, hogy melyik osztály figyeli az eseményeit. Laravel 10 esetén az **EventServiceProvider boot()** metódusába kell elhelyezni a 12–62. kódrészletet.

```php
use App\Observers\MovieObserver;
use Illuminate\Database\Eloquent\Attributes\ObservedBy;

#[ObservedBy([MovieObserver::class])]
class Movie extends Model
```

12–61. kódrészlet: Observer és Model osztály összerendelése (1. módszer)

Az **AppServiceProvider**-es (vagy Laravel 10-nél az **EventServiceProvider**-es) manuális összerendelés így nézne ki:

```php
public function boot(): void
{
  Movie::observe(MovieObserver::class);
}
```

12–62. kódrészlet: Observer és Model osztály összerendelése (2. módszer)

A megfelelő működéshez importálni kell az osztály tetején a **Movie** és **MovieObserver** osztályokat, azonban mi most maradjunk az 1. módszernél, mert az is tökéletesen megfelelő a számunkra.

Az új osztály mindegyik metódusa paraméterként megkapja az érintett Model osztály példányát, ugyanakkor az összes eseményhez tartozó metódus nem került bele alapértelmezetten az osztályba, de ezeket manuálisan is hozzáadhatjuk, ha például a **creating()** esemény lekezelőjét szeretnénk a Model osztály **booted()** metódusából áthelyezni ide az Observer osztályba.

De most mi helyezzük el a **MovieObserver** osztályba a **retrieve()** metódust, ami szintén nem szerepel még benne az artisan parancs általi létrehozás után.

```php
public function retrieved(Movie $movie): void
```

```
{
  dd($movie);
}
```

12–63. kódrészlet: Film lekérésekor aktiválódik az Observer retrieved() metódus

Ha most valamelyik film adatlapjára megpróbálnánk eljutni a filmeket kilistázó oldalról, akkor a kiíratást kapnánk meg, ami a Model objektum minden adatát mutatja. Gyakorlásként esetleg érdemes kipróbálni, hogy a film adatlap megtekintéseket számláljuk ennek a metódusnak a segítségével: egy új egész számot tartalmazó **movies** adattábla mezőben.

Érdemes áttekinteni, hogy melyik Observer metódus mikor váltódik ki, illetve mik a lefutási sorrendek (ezek az Observer metódusok felsorolásánál látszódnak):

Esemény	Model osztály vagy példány metódusok	Observer metódusok
Lekérés	Model::findOrFail($id) vagy a $model objektum első hozzáférésekor	retrieved
Létrehozás	Model::create()	saving, creating | created, saved
Frissítés	Model $model->update()	saving, updating | updated, saved
Törlés	Model $model->destroy()	deleting | deleted
Helyreállítás	Model $model->restore()	retrieved, restoring, saving, updating | updated, saved, restored

12–1. táblázat: Események és az őket kiváltó Model osztály/példány metódusok: Observer metódusok (függőleges vonallal került jelzésre a tényleges esemény bekövetkezése)

Ha nincsen **saving()** vagy **saved()** metódusok, akkor ezek nélkül, de a megadott sorrendben hajtódnak végre a **creating()-created()**, az **updating()-updated()**, vagy a **retrieved()-restoring()-updating()-updated()-restored()** metódusok. Fontos még megemlíteni, hogy az **updating()** és **updated()** metódusok csak akkor hajtódnak végre, ha volt olyan mező, aminek az értéke megváltozott.

A tranzakciókat különleges módon kell kezelni, mint például a 9–23. kódrészlet esetében, ha Observer **created()** metódust szeretnénk használni hozzá, akkor az Observer osztálynak implementálnia kell a **ShouldHandleEventsAfterCommit** interface-t, a többit utána a keretrendszer már elvégzi helyettünk, hogy a működés megfelelő legyen.

Lehetőségünk van továbbá arra is, hogy bár definiálásra, majd összerendelésre került az Observer a Model osztállyal, de azt szeretnénk, hogy bizonyos műveleteknél ne hajtódjanak végre az adott eseménynek megfelelő Observer metódusok automatikusan, akkor használhatjuk így: **$model->saveQuietly(), $model->updateQuietly(), $model->deleteQueietly()** stb. metódus hívásokat.

Az alfejezetben végrehajtott programkód módosítások ebben a GitHub commit-ben találhatók meg.

12.5. Összegzés és továbblépés

Ebben a fejezetben a Laravel magjának részeivel foglalkoztunk. Ide tartoztak a gyűjtemények (collections), architekturális szempontból a Service Container, Service Provider, Facade elemek, aztán az értesítések (notifications), azok közül is az e-mail-es és alkalmazáson belüli adatbázis értesítések, események megismerése, és a Model-hez kapcsolódó speciális momentumok bekövetkezésének eseményei.

Természetesen, a Laravel keretrendszer magjához sokkal több elem tartozik hozzá még ezeken kívül is, mint például az üzenetszórás (broadcasting), a gyorsítótár (cache) kezelése, a sorok és feladatok ütemezése (queues and task scheduling) stb.

Ezeket a további, eddig nem érintett és a már érintett témákat mélységükben a blog oldalamon fogom a későbbiekben ismertetni, az itt jól megszokott gyakorlati szemszögből. Érdemes a blogon ehhez követni a #Rendszer magja (System's Core) címkét, hogy ne maradjunk le semmiről.

13. Kódminőség ellenőrzése teszteléssel és elemzéssel, hibafeltárás (Code quality control)

Az eddigi (és a további) fejezetek során is nagy hangsúly volt a megfelelő kódminőségen, amelynek a részét képezik az alábbiak:

- a programkód olvashatósága, érthetősége,
- az elkészült kódok manuális és automatikus tesztelése,
- a lefutások hatékony végrehajtása,
- és a kapott eredmények értelmezése.

Tehát a *„csináljunk többet kevesebből”* (kódsorokat tekintve) elv mindig a fókuszban volt, mert ebben a keretrendszer rettentően támogat is minket.

Ebben az áttekintő fejezetben a kódminőséggel kapcsolatos technikákról, eszközökről olvasható egy összefoglalás, illetve egy meghívás, amellyel kitekinthetünk majd a világnak erre a részére, de már a blogomon és további szakirodalmakon keresztül. Hiszen tekinthetjük ezt a területet egy külön szakmának, amikor a kézhez kapott programkódról (vagy éppen a saját kódjainkra kívülállóként tekintve) ítéletet kell mondanunk a megvalósítások *minőségéről*.

Ez a fejezet egyáltalán nem vállalkozik arra, hogy Robert C. Martin Clean Code könyvét újra definiálja vagy feldolgozza, mivel az önmagában is egy kerek egész mű, és én magam is többször javasoltam olvasásra. Inkább a Laravel keretrendszer szempontjából ad ez a fejezet útmutatást a megfelelő kódminőség elérésére.

13.1. Programkód minőség (Code quality)

A jó minőség meghatározása az időben változó folyamat is lehet, mert ami régen jó volt, az manapság már nem biztos, hogy elegendő. Fontos, hogy kövessük az irányokat, irányelveket, szabványokat, mert azok segíthetik a fejlődésünket a minőség szempontjából. A minőség fenntartásához hasznos, ha olyan terméket (akár beszélhetünk szoftverről is termékként) készítünk, ami *karbantartható, moduláris* – tehát a részei cserélhetők úgy, hogy a főbb működésben nem történik változás. De érdemes szem előtt tartani azt is, hogy mindig lehet jobb és optimálisabb kódot írni, a kérdés mindig, hogy megéri-e nekünk vagy a vevőinknek?

A minőséggel megalapozzuk a profitunk növekedését:

- ha növeljük a minőséget, akkor nő az ügyfél elégedettsége, így nőhet a piaci részesedésünk is,
- szabványos folyamatok alkalmazásával nő a termelékenységünk, továbbá csökkennek az eladott termékek árai, ami ismét az ügyfél nagyobb elégedettségéhez és magasabb piaci részesedéshez vezethet.

A *minőségbiztosítás* szisztematikus megközelítése a termelési-fogyasztási folyamat megfelelőség-szabályozási elemének:

- minőség értékelési eljárások szabályozása,

- folyamatok technológiájának szabályzása:
    - szabályozás: rögzítés (pl. szabványok), monitorozás, értékelés, tanúsítás, beavatkozási lépésekkel,
    - folyamatok: tevékenységek expliciten rögzített lépésekkel.

Szoftverek esetén nem feltétlenül látható a működésük (és eredményük), ezért a hibák felderítése nehezebb. A szoftverfejlesztési folyamat épít a történelmileg előtte lévő folyamatok működésére, mint például a gyártás során alkalmazott technikák felhasználására (nem véletlen, hogy a manapság népszerű módszertanok a japán Toyota gyárból érkező hagyományokra, szabályok betartására és szakszavak alkalmazására alapoznak).

A minőség javítása előtt fel kell mérni a munkakörülményeket és a szoftver környezetet is, vagyis meg kell vizsgálni, hogy mik adottak és miken lehet fejleszteni:

- határidők,
- költségvetés,
- csapat, csapatmunka,
- vevők, ügyfelek és elvárásaik,
- esetleg függőség egyéb fejlesztő cégektől vagy szoftveres keretrendszerektől, csomagoktól.

A szoftver minőségbiztosítás kiterjed a következőkre:

- forráskód érthetősége, olvashatósága, és milyen szabályrendszer szerint fejlesztünk,
- fejlesztési eljárások, munkaszervezés,
- adatok mennyisége és minősége,
- dokumentációkészítési irányelvek.

A *szoftverminőség-biztosítás* felügyeli a szoftverfejlesztési és -karbantartási folyamatot. Emellett foglalkozik a funkcionális követelményekkel, határidőkkel és a költségkerettel. Ezekkel a tevékenységekkel a szoftverprojekt minél korábbi fázisában a hibák okainak kizárását, a hibák felderítését és azok kijavítását célozzák meg.

A világ legnagyobb ezen az IT szakterületen működő szabványosítási szervezete, az IEEE[23] a *szoftverminőséget* úgy határozza meg, hogy a szoftverminőség annak a foka, hogy az elkészült rendszer mennyire felel meg a specifikált követelményeknek, ezzel egyidejűleg a megrendelő elvárásainak és igényeinek. A szoftver minőségbiztosítás pedig egy tervezett és szisztematikus tevékenység annak biztosítására, hogy megfelelő bizonyosságot nyerjünk arról, hogy egy termék megfelel meghatározott műszaki követelményeknek. Továbbá a tevékenységek halmaza egy termék fejlesztési folyamatának értékelésére, minősítésére.

Sokszor bármennyire is igyekszünk, az elkészült rendszerben lesz hiba, a hibamentességet nem tudjuk garantálni, csak azt, hogy a rendszerben biztosan van hiba. A hibák a szoftver helytelen működését eredményezik, amik főleg a fejlesztők hibájából, tévedéséből adódhatnak. További hibaokok lehetnek még:

[23] Institute of Electrical and Electronics Engineers

alapból hibásak voltak a követelmények, kommunikációs hiba a megrendelő és a fejlesztő között, szándékos eltérés a specifikációtól, logikai tervezési problémák, kódolási hibák, elégtelen tesztelési eljárások.

A szoftver készítése során felléphetnek szintaktikai (elírások) és szemantikai (programkód logikai) hibák is. Ezek adódhatnak tudásbeli hiányosságból, figyelmetlenségből, szervezési gondokból, vagy éppen a kapcsolódó programok, csomagok hibáiból. A szoftver működése és működtetése során felléphetnek hardveres, hálózati, beállítási, kezelői hibák, vagy éppen adódhatnak a nem megfelelő környezet kialakításából, esetleg az adatok sérüléséből.

A működési (megfelelési és hiba elkerülési) irányelvekből 11 darab van, amelyek McCall szerint[24] ebbe a 3 kategóriába sorolhatók:

1. Termék működési faktorok: korrektség, megbízhatóság, hatékonyság, integritás, használhatóság;
2. Termék revíziós faktorok: karbantarthatóság, rugalmasság, tesztelhetőség;
3. Termék átviteli faktorok: hordozhatóság, újra felhasználhatóság, együttműködési képesség.

Ha ezeknek az irányelveknek az elkészült alkalmazásunk megfelel, akkor jó eséllyel a szoftver minősége is magas szintet fog képviselni.

13.1.1. Minőség ellenőrzés és javítás elvek alapján

Az alfejezet megismerése által egy rövid áttekintést kapunk az objektumorientált tervezési alapelvekről és a SOLID elvekről, amelyeket érdemes betartani mindenfajta programtervezés és -megvalósítás során.

13.1.1.1. Objektumorientált tervezés alapelvei

Az objektumorientált tervezés alapelvei (Object-Oriented Design Principles) a tervezési mintáknál magasabb absztrakciós szinten írják le, milyen a *jó* program. A tervezési minták (13.1.3. alfejezet) ezeket az alapelveket valósítják meg egy elég magas absztrakciós szinten. A tervezési mintákat megvalósító programokat az alapelvek megtestesüléseként tekinthetjük. Az itt összefoglalt alapelveket természetesen úgy is alkalmazhatjuk, hogy nem ismerjük (vagy csak egyszerűen nem alkalmazzuk) a tervezési mintákat. Az objektumorientált tervezési alapelvek abban segítenek, hogy több, általában egyenértékű programozói eszköz (például öröklődés és objektum-összetétel) közül kiválasszuk azt, amelyik jobb kódot eredményez.

A tapasztalat az, hogy lehet programozni ezen alapelvek ismerete nélkül, vagy akár tudatos megszegésével, csak nem érdemes. Mint ahogy a Laravel-t is lehet úgy programozni, hogy szembe megyünk a szabályaival, névkonvencióival, ami által még mindig működő kódhoz juthatunk, de sokkal bonyolultabb módon. Így könnyebben is hibázhatunk, ezért nem érdemes szembe menni a szabályokkal.

Ha rugalmatlan, nehezen változtatható vagy kevésbé karbantartható programot írunk, akkor a jövőbeli énünk (és kollégáink) életét keserítjük meg, hiszen, ha egy változtatást kell elvégezni, akkor az ezáltal nehézkessé válik. Inkább érdemes a jelenben több időt rászánni a tervezésre és fejlesztésre, és biztosítani, hogy a jövőben könnyebb legyen a változások kezelése. Ezt adja számunkra az alapelvek betartása.

[24] Szoftver minőségi faktorok:
https://www.tutorialspoint.com/software_quality_management/software_quality_management_factors.htm

Az objektumorientált tervezési alapelvekről röviden:

1. Ne ismételd önmagad (Don't Repeat Yourself, DRY): nem csak a programkódnál alkalmazzuk, hanem az adatbázis sémáknál, teszt terveknél és még a dokumentációnál is. *„Egy rendszeren belül a tudás minden darabkájának egyetlen, egyértelmű és megbízható reprezentációval kell rendelkeznie",* ha ez így van, akkor a DRY elvet betartottuk, így a rendszer egy elemének a módosítása nem igényli a rendszer más, a módosított elemmel kapcsolatban nem lévő részek megváltoztatását. Ehhez persze fontos, hogy felismerjük az ismétlődő részeket, amelyhez a gyakorlat fog hozzásegíteni minket, hogy észrevegyük ezeket. Az ismétlődő kódrészleteket leggyakrabban egy metódusba szoktuk kiszervezni. Ha nem így teszünk, és megsértjük a DRY alapelvet, akkor a kódunk WET *(„Write Everything Twice"* vagy *„We Enjoy Typing"),* vagyis nedves lesz, nem pedig száraz (DRY).
2. Kerüljük a felesleges bonyodalmakat (Keep It Simple Stupid, KISS): tiszta, könnyen érthető, egyszerű megoldásokra törekedjünk mindig. Ne bonyolítsuk el feleslegesen a dolgokat!
3. Demeter törvénye: ennek betartásával könnyen karbantartható program kódbázishoz jutunk, hiszen az objektumok kevésbé függenek egy más objektum belső felépítésétől és állapotától. Az objektumok felépítése így sokkal könnyebben módosítható, akár a hívó szerkezetének módosítása nélkül. Adott objektumnak ne kelljen a kelleténél jobban ismernie egy másik objektum belső működését azért, hogy egy harmadik objektum szolgáltatásait igénybe vegye.
4. Vonatkozások szétválasztása (Separation of concerns) elve szerint egy programot vagy osztályt lehetőleg úgy próbáljunk meg felbontani különböző részekre, hogy az egyes részek külön-külön vonatkozásokat (concern) vagy felelősségi köröket fedjenek le. A vonatkozás ebben a kontextusban az információk összességét jelenti, amely befolyásolja a programkód működését. Például ne akarjuk az adathalmaz lekérését az adatbázisból egy osztályban megvalósítani a naplózással. Az átfedési arány az, amit minimalizálnunk kell a funkcionalitások megvalósítása során. Természetesen ebben is a nagyobb gyakorlat lesz az, ami segíteni fog majd minket.
5. A felelősségek hozzárendelésének általános mintái (General Responsibility Assignment Software Patterns, GRASP): szintén kapcsolódik a felelősségek hozzárendelésének meghatározásához, amihez itt egy minta gyűjteményt kapunk.
6. GoF (Gang of Four) alapelvek: további 23 tervezési mintát határoz meg. Például:
    a. Interface-re programozzunk, ne pedig konkrét implementációra! Ez nagyobb szabadságot ad, mivel egy változás következtében nem kell más osztályokat megváltoztatnunk, elég csak az interface-t módosítani.
    b. Használjunk objektum-összetételt (kompozíció) öröklődés helyett, ha lehet! A tartalmazási kapcsolatban (rész-egész) az egész lesz a felelős a részének az életciklusáért (létrehozás, megszüntetés). Például egy bolti vásárlásnál a végén megkapott blokkon szereplő elemek függjenek a blokktól magától, ha a blokk törlésre kerül, akkor a vásárlás részletei is törölhetők.

Itt most csak nagyon röviden kerültek összefoglalásra az objektumorientált programozás tervezési alapelvei, de a további szakirodalmakon túl, PHP programozási és Laravel-es szemszögből: Jeffrey Way a

Laracasts oktatási oldalon készített egy videósorozatot arról, hogy a PHP-ban, hogyan kell és érdemes alkalmazni az objektumorientált tervezési alapelveket, amit megtekintésre ajánlok.

13.1.1.2. SOLID elvek

Az objektumorientált tervezési alapelvek betartásával már jóval hatékonyabb kódbázishoz juthatunk, azonban történtek további fejlesztések is a kódminőség javítása érdekében az idők során. De probléma is van az objektumorientált tervezési elvek betartásával: az osztályok túlságosan egymásra épülnek, használják egymást, tehát a függőség nagy köztük. Így ez megnehezíti a kód újraszervezési és tesztelési folyamatokat. Ezért is jöttek létre a SOLID elvek (13–1. táblázat), amelyek betartásával az objektumorientált elvek betartásának hátrányai kiküszöbölhetők. Ugyanakkor az objektumorientált tervezési alapelvek és a SOLID elvek között gyakran átfedéseket is felfedezhetünk.

S	Single Responsibility Principle	Egy felelősség elve	Egy osztály vagy modul egy és csak egy felelősséggel rendelkezzen (azaz: egy oka legyen a változásra).
O	Open-Closed Principle	Nyílt/zárt elv	Egy osztály vagy modul, legyen nyílt a kiterjesztésre, de zárt a módosításra.
L	Liskov substitution principle	Liskov helyettesítési elv	Minden osztály legyen helyettesíthető a leszármazott osztályával anélkül, hogy a program helyes működése megváltozna.
I	Interface segregation principle	Interface elválasztási elv	Több specifikus interface jobb, mint egy általános.
D	Dependency inversion principle	Függőség megfordítási elv	A kódod függjön absztrakcióktól, ne konkrét implementációktól.

13–1. táblázat: SOLID elvek

A SOLID elvekről röviden (hosszabb leírás itt is található hozzájuk):

- S: Az egyszeres felelősség elve azt mondja ki, hogy minden osztálynak egyetlen felelősséget kell lefednie, de azt teljes egészében. Ami nem tartozik a felelősségi köréhez, azt el kell távolítani belőle.
- O: egy program forráskódja legyen nyitott a bővítésre, de emellett legyen zárt a módosításra.
- L: azt írja elő, hogy a leszármazott osztályok példányainak úgy kell viselkedniük, mint az ősosztály példányainak, vagyis a program viselkedése nem változhat meg attól, hogy az ősosztály egy példánya helyett a jövőben valamelyik gyermekosztályának egy példányát használjuk.
- I: az elv azt mondja ki, hogy egy sok szolgáltatást nyújtó osztály fölé el kell helyezni interface-eket, hogy minden kliens, amely használja az osztály szolgáltatásait csak azokat a metódusokat lássa, amelyeket ténylegesen használ.

- D: az elv azt mondja ki, hogy a magas szintű komponensek ne függjenek alacsony szintű implementációs részleteket kidolgozó osztályoktól, hanem épp fordítva, a magas absztrakciós szinten álló komponensektől függjenek az alacsony absztrakciós szinten álló modulok.

A SOLID elvek nem eredményezik automatikusan azt, hogy a kód mindörökké karbantartható lesz. Ahhoz emberi erőt is bele kell tenni. Egy ügyféltől érkező módosítási kérésnél meg kell állni, hogy gyorsan *„összedrótozzuk"* működőre az alkalmazást. Ha már ott van az összedrótozás, akkor venni kell a fáradtságot, és szét kell szedni a korábbi trehány munka eredményét. Ha olyan ügyfél kérés jön, amire nem számítottunk, és hirtelen szét kell szednünk valamit, ami eddig együtt volt, nem ússzuk meg a refaktorálást. Ebben azonban rengeteget segít, ha tesztelést, teszt-vezérelt fejlesztést használtunk.

Gyakorlati szempontból annyit érdemes még hozzátenni, hogy könnyedén előfordulhat egy programozói munkakört betöltő álláspályázat egyik állomásaként, hogy a SOLID elvek szerint kell átalakítani (refactoring) egy megkapott *„rossz"* kódbázist.

A SOLID elvekről további magyarázatokkal és konkrét példákkal szolgál az ELTE Informatikai Karának egyik oktatási weboldala is.

Itt most csak nagyon röviden kerültek összefoglalásra a SOLID elvek, de a további szakirodalmakon túl, PHP programozási és Laravel-es szemszögből: Jeffrey Way a Laracasts oktatási oldalon készített egy videósorozatot arról, hogy a PHP-ban és Laravel-ben (bár már meglehetősen régi sorozat), hogyan kell és érdemes alkalmazni a SOLID elveket.

13.1.2. Szabványok

A szabvány (standard) egy olyan dokumentum, amely adott termék jellemzők vagy folyamatok rögzített (valamely hivatalosan is elismert szervezet által jóváhagyott) normáit, követelményeit tartalmazza. A szabványosítás a felhasználó és a fogyasztó érdekében végzett szabályozó, egységesítő tevékenység. Szabványokat írtak szolgáltatásokra és vonatkozó műszaki követelményeire, továbbá a minőségi feltételeinek a meghatározására is.

Mi is követjük a szabályokat, hiszen a World Wide Web Consortium (W3C) világhálóra és web-re vonatkozó szabványait követjük a fejlesztés során. Az egyes szoftverek működése, viselkedése nehezen írható le általános szabványokkal, de maga a fejlesztési folyamat már jobban szabványosítható, azonban a nem megfelelő vagy rossz szabványosítás jelentősen csökkentheti a termelékenységet, a fejlesztés hatékonyságát.

A szabványosítás mindemellett számos előnnyel jár:

- Magas szintű fejlesztési, majd utókövetési (karbantartási és működtetési) tevékenység biztosítása.
- Megfelelő koordináció biztosítása a kapcsolódó fejlesztői csapatok között.
- Megfelelő kommunikáció és együttműködés a fejlesztési folyamat külső résztvevőivel.
- Hatékonyabb kommunikáció biztosítása a megrendelő és a fejlesztők között, így kevesebb lesz a félreértés, felesleges munka.
- Tanúsítás, tanúsítványok (certificate), audit alapjainak a biztosítása.

A megfelelő minőségű szabványosítás anyagi előnyökkel is járhat mind a megrendelő, mind a fejlesztő vonatkozásában.

A szoftver minőségbiztosításban alkalmazott szabványok kapcsolódhatnak a szoftverhez, a projekthez és a speciálisabb szoftverprojekt működtetési és felügyeleti területhez is. A következő szabványok kapcsolódnak ezekhez a területekhez:

- Számítás tudományi szabványok (pl. IEEE Std610.12:1990)
- Minőségbiztosítási / minőség menedzsment szabványok (pl. ISO/IEC/IEEE 90003:2018)
- Projekt menedzsment szabványok (pl. IEEE Std 1058.1-1987)
- Rendszer mérnökségi szabványok (pl. ISO/IEC WD 15288)
- Biztonsági szabványok (pl. IEC61508)
- Termék szabványok (pl. ISO/IEC 14598)
- Folyamat szabványok (pl. ISO/IEC 12207:1995)

Jellemzően a folyamatok, és nem a termékek konzisztens minősége biztosítható a szabványok követése által.

A szoftver minőségbiztosításban alkalmazott szabványok típusai lehetnek tanúsítási szabványok (vagy más szóval tanúsítványok) és értékelési szabványok. A tanúsítványok a fejlesztő szervezet számára lehetővé teszik, hogy a szoftvertermék vagy karbantartási szolgáltatás minőségének konzisztens fenntartásának képességét demonstrálja. Lehetővé teszik továbbá a minőségbiztosítási rendszer javítását is. Az értékelési szabványok segítenek a fejlesztő cégnek (csapatnak, egyénnek) az önértékelési eszköz alkalmazásában a fejlesztési projekt minőségére vonatkozóan. Lehetővé teszi így a fejlesztési és karbantartási folyamatok javítását. De ezen túlmenően a potenciális beszállító cégek között is elősegítheti a kiválasztást.

Bár ritka, hogy valaki ezekre a faktorokra koncentrálna konkrétan a fejlesztés vagy az üzemeltetés során, mint amiket ezek a szabványok megfogalmaznak. De ha átolvassuk és magunkévá tesszük az esszenciájukat, akkor a későbbiekben funkcionalitás, megbízhatóság, teljesítmény, működés, biztonság, kompatibilitás, működtetés szempontjából is jobb minőségű kódokat írhatunk egyénileg vagy csapatban.

13.1.3. Tervezési minták a Laravel-ben

A tervezési minták drámaian javíthatják a kódbázis általános minőségét, teljesítményét és karbantarthatóságát a Laravel alkalmazások készítésekor. A tervezési minták olyan hasznos *„erőforrások"*, amelyek megfelelő és kipróbált válaszokat kínálnak a szoftverfejlesztés tipikus problémáira. Bár a tervezési mintáknak számos előnye van, fontos megjegyezni, hogy nem jelentenek *„bolondbiztos"* megoldást minden problémára. Mielőtt eldöntenénk, hogy melyik tervezési mintát használjuk, fontos, hogy gondosan értékeljük az alkalmazásunk sajátos, egyedi követelményeit.

Több szakértő a tervezési minták megrögzött vagy szigorúan elvárt használatát inkább tehernek, egy olyan kényszernek érzi, amely a programozók alkotói szabadságát korlátozza. Ők amellett vannak, hogy sokkal fontosabb betartani az objektumorientált tervezési irányelveket és a SOLID elveket, mintsem a tervezési mintákat erőltetni minden körülmények között.

Ugyanakkor azt nem szabad elvitatni a tervezési mintáktól, hogy számos előnnyel rendelkezik a használatuk, például jobb kód újra felhasználhatóságot, skálázást, kódszervezést és karbantarthatóságot eredményeznek.

A Laravel keretrendszer alapvetően az MVC tervezési mintára (2.3. alfejezet) épül, de emellett még számos tervezési mintát alkalmaztunk már a fejlesztéseink során:

- Factory tervezési minta, amikor adatgyárakat építettünk.
- Singleton tervezési minta, amikor a Service Container-be szolgáltatásokat kötöttünk be és nyertünk ki onnan.
- Facade-ok alkalmazása is egy tervezési minta része.
- Observer tervezési minta az események figyelőinek megvalósításánál fordult elő.

Ami szintén működik a Laravel-ben, de mi nem használtuk őket:

- Strategy tervezési minta, amely főleg a gyorsítótár (cache) menedzselésénél játszik szerepet.
- Adapter tervezési mint, amely akkor hasznos, ha a Laravel-ben külső API-ok szolgáltatásait szeretnénk igénybe venni.
- Repository tervezési minta, de 2024-ben ennek a mintának a használatát Povilas Korop a Laravel daily projekt atyja már nem is támogatja, mivel az Eloquent-ek használata annyira hatékony, hogy erre, a véleménye szerint nincsen szükség.

A tervezési mintáknak, használatuknak, előnyeiknek/hátrányaiknak óriási szakirodalma van. Ha egyet lehetne javasolni azért, hogy többet tudjunk meg a tervezési mintákról, akkor ez a weboldal nagyon jól, érthetően magyarázza el, ábrák segítségével a működésüket. Illetve egy könyvajánló még a témához: *„A Brain-Friendly Guide", Head First, Design Patterns* könyv is meglehetősen jól, ábrákkal illusztrálva magyarázza el a tervezési minták működését, előnyeit, hátrányait.

13.2. Tesztelés

Tesztelésre azért van szükség, hogy a kifejlesztett szoftverben meglévő hibákat még az éles üzembehelyezés előtt megtaláljuk, így növeljük a termék minőségét és a megbízhatóságát. Abban gyakorlatilag mindig biztosak lehetünk, hogy a tesztelés előtt van hiba, abban viszont soha nem lehetünk biztosak, hogy tesztelés után már nem marad benne hiba (általában marad). A program azon funkcióit érdemes minél alaposabban tesztelni, melyeket a felhasználók a leggyakrabban használnak.

Gyakori félreértés szerint a tesztelés csak tesztek futtatásából áll. Valójában a tesztelés egy folyamat, amely számos különböző tevékenységet foglal magába: teszttervezés, tesztfelügyelet és -irányítás, tesztelemzés, műszaki teszttervezés, teszt megvalósítása, tesztvégrehajtás és tesztlezárás.

A tesztelés célja az, hogy a rendszert és a fejlesztési folyamatot is javítsuk. Közvetlen céljai közé tartozik, hogy a követelményeket teljesítse, a felhasználók elvárásainak megfeleljen az elkészült alkalmazás. Ezenkívül a programhibák megelőzésében és felfedésében, a szoftver aktuális minőségének meghatározásában is segít.

A tesztelés alapelvei a következők:

1. A tesztelés hibák jelenlétét jelzi: a tesztelés képes felfedni a hibákat, de azt nem, hogy nincs hiba az alkalmazásban.
2. Nem lehetséges kimerítő teszt: minden bemeneti kombinációt nem lehet letesztelni, ezért a magas kockázatú és magas prioritású részeket teszteljük.
3. Korai teszt: érdemes a tesztelést az életciklus minél korábbi szakaszában elkezdeni, minél hamarabb találunk meg egy hibát, annál olcsóbb javítani (akár már a specifikációnál).
4. Hibák csoportosulása: a tesztelésre csak véges időnk van, ezért a tesztelést azokra a modulokra kell koncentrálni, ahol a hibák a legvalószínűbbek, illetve azokra a bemenetekre kell tesztelnünk, amelyre valószínűleg hibás a szoftver (például a szélsőértékek).
5. A féregirtó paradoxon: ha az újra tesztelések során mindig ugyanazokat a teszteseteket futtatjuk, akkor egy idő után ezek már nem találnak több hibát.
6. A tesztelés függ a körülményektől: másképp tesztelünk egy atomerőműnek szánt programot és egy programozási beadandó feladatot.
7. Hibátlan rendszer téveszméje: ha a megrendelő nem elégedett a szoftverrel akkor azt felesleges tesztelni.

A munkáink során folyamatosan manuális és automatikus teszteléseket végeztünk a fejlesztett projektek kapcsán. A manuális tesztek részét képezték a következő ellenőrzések:

- webes funkcionalitások működésének megfelelősége (többféle böngészővel),
- API funkcionalitások működésének megfelelősége (böngészővel és a Postman alkalmazással),
- a grafikus felületekre kisebb hangsúlyt fektettünk, de több sablont építettünk össze a Laravel alkalmazásinkkal, illetve a frontend oldal elemeinek illeszkedését vizsgáltuk a backend-hez képest (többféle böngészővel).

A tesztelés témakörében érdemes elmélyülni akár szoftverfejlesztőként is, mert másképp tekintünk utána a kódjainkra, és már a tervezés folyamatában olyan meglátásokat, ötleteket ad hozzá a rendszerünkhöz, amelyek segítségével sokkal hatékonyabban működő alkalmazásokat készíthetünk. Az ISTQB[25] és a hazai HSTQB[26] szervezet olyan tanúsítványok megszerzését teszi lehetővé, amelyek által profi tesztelőkké is válhatunk. De ha még nem is szerezzük meg maguknak a papírokat, akkor is érdemes elmélyedni azokban a tananyagokban, amelyeket a vizsgák elvégzéséhez nyújtanak ezek a szervezetek.

13.3. Statikus tesztelési technikák

A statikus tesztelés egy olyan vizsgálat, amit a forráskódon vagy a létrejött binárisokon (lefordított állományokon) végzünk, ha nem fut a program. Ide tartozik a dokumentáció felülvizsgálata is [4].

Statikus technikákkal könnyen megtalálhatóak azok a kódsorok, ahol null referencián keresztül akarunk metódust hívni. Ugyanezt elérni dinamikus teszteléssel (13.4. alfejezet) nagyon költséges, hiszen 100%-os kód lefedettség kellene hozzá.

[25] International Software Testing Qualifications Board, https://www.istqb.org/
[26] Hungarian Software Testing Qualifications Board, https://hstqb.org/

A statikus tesztelési technikák előnye, hogy nagyon korán alkalmazhatóak, már akkor is, amikor még nincs is futtatható verzió. Így hamarabb lehet velük hibákat találni és kevésbé költséges a hibajavítás.

A statikus tervezési technikák két fajtája:

1. A felülvizsgálat a kód, illetve a dokumentáció, vagy ezek együttes manuális átnézését jelenti.
2. A statikus elemzés a kód, illetve a dokumentáció automatikus vizsgálatát jelenti, ahol a statikus elemzést végző segédeszköz megvizsgálja a kódot (illetve a dokumentációt), hogy bizonyos szabályoknak megfelel-e.

Az alábbiakban áttekintjük ezeket részletesebben, illetve a statikus kódelemzést már a Laravel-ben használható eszközökkel ismerjük meg a gyakorlatban.

13.3.1. Felülvizsgálati technikák

A felülvizsgálat azt jelenti, hogy manuálisan átnézzük a forráskódot és gyanús részeket keresünk benne. Ezzel szemben áll a statikus elemzés (13.3.2. alfejezet), ahol szoftverekkel nézetjük át automatikusan a forráskódot. A felülvizsgálat legismertebb típusai [4]:

- Informális felülvizsgálat (csoporton belüli):
  - egy tapasztalt programozó nézi át a csoportban (**review**),
  - páros programozás, valaki írja a kódot, a másik figyeli, hogy vét-e hibát (**pair programming**),
  - kód újra szervezés (**refactoring**).
- Átvizsgálás (házon belüli): célja, hogy mások is átlássák az általunk írt kódrészletet, kritikai megjegyzéseikkel segítsék a kód minőségének javítását:
  - váll feletti átnézés (**over-the-shoulder review**),
  - forráskód átnézés (**code review**),
  - kód átvétel (**code acceptance review**),
  - körbeküldés (**pass-around**),
  - csoportos átnézés (**team review**),
  - felület átnézés (**interface review**),
  - kód prezentálás (**code presentation**).
- Technikai felülvizsgálat (külsős szakérő bevonásával rövid idejű): erre általában akkor kerül sor, amikor nem vagyunk elégedettek a szoftver működésének teljesítményével és meg kell keresnünk azokat a szűk keresztmetszeteket, amelyek a lassú működést leginkább okozhatják.
- Inspekció (külsős szakérő bevonásával hosszú idejű): a probléma megoldásához általában nem elég csak a szoftver egy részét megvizsgálni, hanem az egész kódbázist, az adatbázissal együtt vizsgálni kell, hogy kiderüljön a hiba és annak az oka.

A felsorolt fogalmak pontos megértéséhez ajánlom Ficsor Lajos, Dr. Kovács László, Krizsán Zoltán és Dr. Kusper Gábor által írt Szoftvertesztelés című jegyzetet, amely itt érhető el elektronikus formában.

13.3.2. Statikus elemzési eszközök a Laravel-ben

A statikus elemzés fehérdobozos teszt, hiszen szükséges hozzá a forráskód. A statikus elemzés azért hasznos, mert olyan hibákat fedez fel, amiket más tesztelési eljárással nehéz lenne megtalálni. Például kiszűrhető segítségével minden null referencia hivatkozás, ami az alkalmazásban lekezeletlen kivétel dobásához vezethet, ha benne marad a programban. Az összes null referencia hivatkozás kiszűrése dinamikus technikákkal (pl. komponens teszttel vagy rendszerteszttel) nagyon sok időbe telne, mert 100%-os kódlefedettséget kellene elérnünk.

A statikus elemzés azt használja ki, hogy az ilyen tipikus hibák leírhatók egyszerű szabályokkal, amiket egy egyszerű kódelemző (**parser**) gyorsan tud elemezni [4].

Az alábbi statikus kódelemzési eszközöket nem egymás helyett, hanem egymás mellett érdemes alkalmazni. Mindegyik más-más szemszögből vizsgálja meg az alkalmazásunk kódjait, és így az eredmények által hasznos információkhoz juthatunk a kódjaink minőségéről.

13.3.2.1. Enlightn

Az Enlightn eszköz vizsgálja a programkódot 1. teljesítmény (performance), 2. megbízhatóság (reliability) és 3. biztonság (security) szempontjából. Ezzel a statikus kódelemzési eszközzel (az ingyenes verziójával) 67 tesztet futtathatunk (a fizetős PRO verzióval 131 tesztet) az imént említett három kategóriában.

Válasszuk ki bármelyik eddigi projektünket, vagy akár hozzunk létre egy teljesen újat, és telepítsük az eszközt!

```
composer require enlightn/enlightn --dev
```

Utána számos beállítást elvégezhetünk, majd futtathatjuk a statikus kódelemzést:

```
php artisan enlightn
```

Egy friss telepítésű Laravel 11-es webes alkalmazásnál az elemzés során mutatja a tesztelt funkcionalitásokat, majd mutatja még azt is, hogy mi bukik el, mi megy át, végül egy összesítő táblázatot jelenít a számunkra:

```
Report Card
===========

+----------------+-------------+-------------+-----------+-----------+
| Status         | Performance | Reliability |  Security |     Total |
+----------------+-------------+-------------+-----------+-----------+
| Passed         |   10  (56%) |   26  (93%) | 16  (76%) | 52  (78%) |
| Failed         |    2  (11%) |    2   (7%) |  3  (14%) |  7  (10%) |
| Not Applicable |    6  (33%) |    0   (0%) |  2  (10%) |  8  (12%) |
| Error          |    0   (0%) |    0   (0%) |  0   (0%) |  0   (0%) |
+----------------+-------------+-------------+-----------+-----------+
```

13–1. ábra: Kódelemzés eredménye az Enlightn eszközzel

13.3.2.2. Psalm plugin

A Psalm plugin statikus elemzési eszköz a Laravel projektekben típustámogatással működtethető. A célja az, hogy minél több típusokkal kapcsolatos hibát találjon, ezáltal növelve a fejlesztők termelékenységét

és az alkalmazás megfelelőségét. Ezzel az eszközzel olyan hibákat találhatunk, amelyekhez nem kell külön teszteseteket írnunk.

Telepítsük egy Laravel 10-es projektünkbe, például a korábbi l10-auth-breeze projektbe:

```
composer require --dev psalm/plugin-laravel -W
```

A telepítési leírás nem említi a -W kapcsolót, azonban enélkül nálam a csomagok verzióinak problémái miatt nem tudott települni. Laravel 11-ben sajnos ez a kapcsoló sem segített, nem tudott települni az eszköz.

Hozzuk létre a Psalm beállítási fájlt:

```
./vendor/bin/psalm –init
```

Engedélyezzük a plugin-t:

```
./vendor/bin/psalm-plugin enable psalm/plugin-laravel
```

Futtassuk a Psalm statikus kódelemző eszközt:

```
./vendor/bin/psalm
```

Az eredmény pedig itt látható:

```
ERROR: UnusedClosureParam - database/factories/UserFactory.php:37:39 - Param attributes is never referenced in this method (see https://psalm.dev/188)
        return $this->state(fn (array $attributes) => [


ERROR: UnusedClass - database/seeders/DatabaseSeeder.php:8:7 - Class Database\Seeders\DatabaseSeeder is never used (see https://psalm.dev/075)
class DatabaseSeeder extends Seeder


------------------------------
21 errors found
------------------------------
44 other issues found.
You can display them with --show-info=true
------------------------------
Psalm can automatically fix 4 of these issues.
Run Psalm again with
--alter --issues=PossiblyUnusedMethod --dry-run
to see what it can fix.
------------------------------

Checks took 63.50 seconds and used 505.516MB of memory
Psalm was able to infer types for 93.7984% of the codebase
```

13–2. ábra: Psalm Laravel plugin statikus kódelemző eredmény listája (részlet)

Az eszköz talált hibákat, néhányat automatikusan ki is javított közülük. Emellett jelzi, hogy mennyi ideig dolgozott (kb. 1 percig), mennyi memóriát használt a munkához, valamint, hogy a kódbázis ~94%-ának típusait tudta meghatározni, kikövetkeztetni.

13.3.2.3. Pint

A Pint a PHP Coding Standards Fixer-re épül, és egyszerűvé teszi annak biztosítását, hogy a kód stílusa tiszta és konzisztens maradjon. A Pint automatikusan települ minden új Laravel alkalmazással, így azonnal elkezdhetjük használni. Alapértelmezés szerint a Pint nem igényel semmilyen beállítást, és a Laravel kódolási stílusát követve kijavítja a kódban lévő kódstílusbeli problémákat.

Futtassuk a következő utasítással:

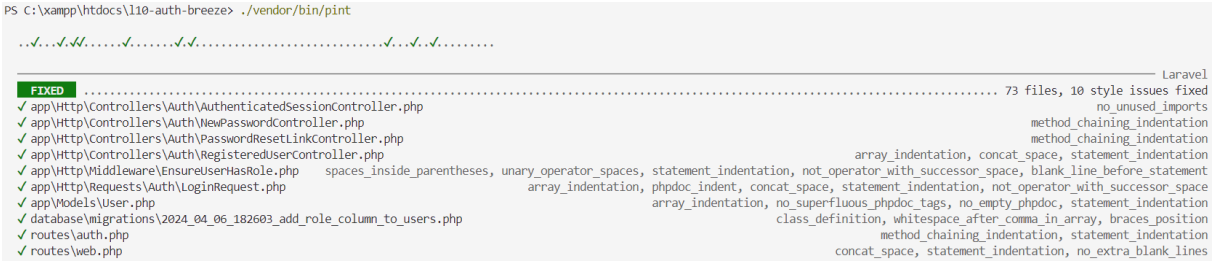
./vendor/bin/pint

A Pint egy teljesen friss Laravel 11-es alkalmazásban:

13–3. ábra: Pint futtatásának eredménye egy friss Laravel 11-es projektben

Itt minden rendben van!

Ha visszatérünk a **l10-auth-breeze** projektbe, akkor ott is futtassuk le:



13–4. ábra: Pint futtatásának eredménye egy már fejlesztett, módosított Laravel 10-es projektben

Itt már talált problémákat, amelyeket aztán jelez is nekünk, hogy kijavította őket az adott fájlokban! Például törölt olyan importálásokat fájlokból, amelyek ott nem kerültek felhasználásra az adott osztályban. Ezek a javítások könnyedén nyomon követhetők, ha használjuk a verziókezelést a projektünkben.

13.3.2.4. Larastan

A Larastan a kódban található hibák megtalálására összpontosít. A hibák egész csoportjait észleli, még mielőtt teszteket írnánk a kódhoz. Statikus tipizálást ad a Laravel-hez a fejlesztői termelékenység és a kódminőség javítása érdekében. Támogatja a Laravel legtöbb funkcionalitását, és felfedezi a hibákat a kódunkban.

A Larastan az alkalmazás kódját úgy vizsgálja, hogy megfelel-e az általa támasztott szabályszerűségeknek.

Laravel 10-ben (v10.48.14) és 11-ben (v11.13.0) nem működik a telepítése csomagkonfliktusok miatt. Csak 9-es főverziójú Laravel-ben működik egyelőre az eszköz, úgyhogy abban érdemes kipróbálni (én megtettem, és kaptam is vissza néhány hibajelzést a programkódról).

13.4. Dinamikus tesztelési technikák

A dinamikus tesztelés egy olyan vizsgálat, amit akkor használunk, ha fut a program: funkciók tesztelése, memóriaelemzés, sebezhetőség vizsgálat, teljesítmény vizsgálat. A dinamikus teszteléssel könnyen észrevehetőek a programlogikában vétett hibák. A dinamikus tesztelési technikák a tesztelendő rendszer futtatását igénylik. Laravel esetében ezek a **feature** és **unit** tesztek lesznek főleg. Az alábbi táblázatban megtekintjük azt, hogy mit érdemes tesztelni egy unit tesztben:

Conformance	Az érték **megfelel** az elvárt formátumnak?
Ordering	Az értékek a megfelelő **sorrendben** vannak?
Range	Az érték az elvárt **minimum** és **maximum** érték között van?
Reference	**Hivatkozik-e** kód külső erőforrásra, amire nincs közvetlen ráhatása?
Existence	**Létezik** az érték? (Pl. nem lehet null)
Cardinality	Az értékek **számossága** megfelelő?
Time	Minden a megfelelő sorrendben futott le, az **elvárt idő** alatt?

13–2. táblázat: CORRECT: mit teszteljünk egy unit tesztben?

Egy unit teszt legyen gyors, független (többi teszttől függetlenül futtatható), ismételhető (akárhányszor futtatjuk, mindig ugyanazt az eredményt adja), önellenőrző (mindig egyértelműen eldönthető, hogy hibás vagy helyes), alapos (a tesztek minden fontos esetet lefednek, de nem a 100%-os kódlefedettség a cél).

A tesztek alanyai vagy céljai lehetnek a következők:

- funkcionalitás,
- képesség,
- tranzakció (összetett műveletsor),
- minőségi jellemző elérése,
- valamilyen strukturális elem.

Tesztesetek (**Test cases**) meghatározása az alábbi összetevőkből áll:

- Végrehajtási előfeltételek (**preconditions**): egy teszteset végrehajtása esetén a rendszert egy megadott kezdő állapotba kell hozni.
- Bemeneti értékek halmaza: megadott input értékek halmazával végre kell hajtatni a tesztelt elemet.
- Elvárt eredmény: majd a teszt futásának eredményét össze kell hasonlítani az elvárt eredménnyel.
- Végrehajtási utófeltételek (**postconditions**): ellenőrizni kell, hogy a végrehajtás után a rendszer az elvárt állapotba került-e.

Egy teszteset célja egy meghatározott vezérlési út végrehajtása a tesztelendő program egységben, vagy egy meghatározott követelmény teljesülésének ellenőrzése.

A kódminőség meghatározásához a teszt lefedettség (**Test coverage**) mérőszám is segítségünkre lehet. Ez a számszerű értékelése annak, hogy a tesztelési tevékenység mennyire alapos, milyen a minősége, illetve

időben hol tart a tesztelés, mennyi erőforrás szükséges még a fejlesztés hátra lévő szakaszához. A tesztelés a fejlesztési folyamat fontos (és erőforrás igényes, tehát költséges) része, folyamatosan keressük a folyamat előrehaladásának a mérési lehetőségeit [4].

Az egyes tesztelési technikák más és más lefedettségi mérőszámokat alkalmaznak. A teszt lefedettség számszerűsítése alkalmas a tesztelési tevékenység értékelésére az alábbi *szempontok* szerint:

- Lehetőséget ad a tesztelési tevékenység *minőségének* mérésére.
- Lehetőség biztosít arra, hogy *megbecsüljük, mennyi erőforrást kell még a fejlesztési projekt hátralevő idejében tesztelési tevékenységre fordítani.*

A lefedettségi mérőszámok tehát arra nézve adnak információt, hogy milyen készültségi szinten áll a tesztelési tevékenység, és a tesztelési terv részeként meghatározzák, hogy milyen feltételek esetén tekinthetjük a tevékenységet késznek.

A Laravel-ben történő fejlesztés során mi is képesek vagyunk meghatározni a kód lefedettségi mérőszámot, ehhez azonban szükség van az Xdebug vagy a PCOV eszköz telepítésére. Utána futtatható a projektjeinkben a parancs:

```
php artisan test --coverage
```

Ez általában egy [0,1] értékkel jellemzett mérőszám. A tesztelés teljes, ha a mérőszám pontosan 1, de a gyakorlatban ez ritkán elérhető:

- Irreálisan nagy teszteset halmazzal érhető el, ezért inkább annak csak minél jobb megközelítésére törekedhetünk.
- 100%-os lefedettség sem jelenti azt, hogy minden hibát megtaláltunk.
- Mivel a különböző lefedettségi mérőszámok más és más szempontból értékelik a tesztelés alaposságát, célszerű többet is használni: utasítások lefedettsége, ágak (döntések) lefedettsége, utak lefedettsége stb.

Négy alapvető dinamikus tesztelési technikát különböztetünk meg általában [4]:

1. *Specifikáció alapú technikák:* black-box technika, teszteseteket közvetlenül a rendszer specifikációjából (modelljéből) vezetik le, nem kell ismerni a tesztelendő elem belső felépítését. Ide tartozik az ekvivalencia partícionálás, amikor a tesztelési osztályok egy-egy elemét vizsgáljuk meg, és mondunk ítéletet ezáltal az egész csoportra vonatkozóan. Határérték analízis során a bemeneti értékek csoportjainak határait vizsgáljuk, hogy megfelelően reagál-e rájuk az alkalmazásunk. Az ok-hatás analízis során döntési táblát építünk, és az irreális eseteket próbáljuk kiszűrni a működésben. A véletlenszerű adatok generálásával is ellenőrizhetjük az alkalmazás helyes működését. A használati esetek sorra vétele pedig alapvető elvárás az alkalmazás helyes működésének ellenőrzésekor.
2. *Modell alapú technika:* Közvetlenül az UML[27] modellből vezeti le a teszteseteket, gyakorlatilag az előző eset speciális változata (az előzőnek egy formalizáltabb változata).

[27] Unified Modeling Language, vagyis egységes modellező nyelv

3. *Struktúra alapú technikák:* a kód ismeretében határozzák meg a teszteseteket (white-box technika). Bonyolult, erőforrás igényes feladat. Például a vezérlési folyamat gráf alkalmazása egy ilyen technika.
4. *Gyakorlat alapú technikák:* a fejlesztői tapasztalaton, tudáson alapul: hiba becslés, felderítő tesztelés tartozik ezek közé.

A továbbiakban azokat az eszközöket tekintjük át, amelyek segítségünkre voltak, vannak és lesznek, amikor automatikus, dinamikus teszteknek akarjuk alávetni a Laravel-es alkalmazásainkat.

13.4.1. PHPUnit

A PHPUnit egy programozó-központú tesztelési keretrendszer a PHP számára. Ez az xUnit architektúra egy példánya a unit tesztelési keretrendszerek számára.

A PHPUnit azon az elképzelésen alapul, hogy a fejlesztőknek képesnek kell lenniük arra, hogy gyorsan megtalálják a hibákat az újonnan elkészült kódjukban, és biztosítsák, hogy ez nem okoz problémás működést a kódbázis más részein. Más egységtesztelő keretrendszerekhez hasonlóan a PHPUnit is állításokat (**assert** utasításokat) használ annak ellenőrzésére, hogy a tesztelt komponens vagy egység az elvárásoknak megfelelően viselkedik.

Az egységtesztelés célja a program minden egyes részének elkülönítése, és annak bizonyítása, hogy az egyes részek helyesek. Ennek eredményeképpen az egységtesztek a fejlesztési ciklus korai szakaszában találják meg a problémákat.

A munkáink során folyamatosan ezt az eszközt használtuk arra, hogy automatikus teszteket írjunk az alkalmazásaink működésének teszteléséhez. A következőket teszteltük a segítségével:

- útvonalakat (3.7. alfejezet),
- nézeteket (4.6. alfejezet),
- adatbázis műveleteket (7.6. alfejezet),
- validációs szabályokat (9.3. alfejezet),
- funkcionalitásokat a hitelesítési kezdő készleteknél (10.2. alfejezet),
- engedélyezési technikákat (11.2.3. alfejezet).

13.4.2. PEST

A Laravel 11 megérkezésével az addig PHPUnit-ot használó keretrendszer már a Pest tesztelési keretrendszert használja alapértelmezetten a tesztesetek végrehajtásához. Ez nem azt jelenti, hogy a PHPUnit már nem alkalmazható, pusztán azt, hogy kaptunk még egy – elvileg jobb – eszközt a kezünkbe. A Pest az egyszerűséget helyezi a fókuszába. Aprólékosan megtervezték a működését azért, hogy visszaadja a tesztelés örömét (korábban már utaltunk arra, hogy a tesztek írása nem a leginnovatívabb tevékenység, könnyen unalmassá válhat), de a Pest keretrendszer a használatával vissza szeretné adni a tesztelés örömét.

Mindezt úgy éri el, hogy beszédes és színes hibaüzeneteket szolgáltat, lehetővé teszi a stressztesztek végrehajtását, jó a dokumentációja. Rendelkezésünkre bocsát beépített kód lefedettségi jelentéseket, párhuzamos végrehajtások lehetőségét is.

A Laravel keretrendszer alapértelmezetten rendelkezésünkre bocsátja már a Pest keretrendszer használatát, ha azt választottuk a telepítéskor. Illetve, ha figyelmesek vagyunk a Laravel dokumentációjának olvasása során, akkor észrevehetjük, hogy a tesztesetek most már nem csak PHPUnit keretrendszer specifikus formában elérhetőek, hanem Pest specifikus tesztek is elérhetők ugyanott mindössze egy lapfül váltással. Tekintsük meg például az értesítések tesztelésének dokumentálását!

PHPUnit példa

```php
<?php

namespace Tests\Feature;

use App\Notifications\OrderShipped;
use Illuminate\Support\Facades\Notification;
use Tests\TestCase;

class ExampleTest extends TestCase
{
    public function test_orders_can_be_shipped(): void
    {
        Notification::fake();
```

Pest példa

```php
<?php

use App\Notifications\OrderShipped;
use Illuminate\Support\Facades\Notification;

test('orders can be shipped', function () {
    Notification::fake();
```

13–5. ábra: Automatikus teszteset létrehozása ugyanarra a célra PHPUnit és Pest keretrendszerek felhasználásával

Az összehasonlításból látszódik, hogy a Pest teszt gyakorlatilag osztály deklaráció nélkül hoz létre tesztelési metódust és mindezt sokkal rövidebben (*megjegyzés:* a példa képek nem mutatják végig a teszteket, de ugyanazt a funkcionalitás részt hasonlítjuk így össze).

Gyakorlás céljából megtehetjük a következőket:

- az ebben a fejezetben létrehozott Laravel 11-es projektünkben hozzunk létre egy Pest unit tesztet és futtassuk,
- vagy ha már profibbnak érezzük magunkat, akkor az előző PHPUnit alfejezetben megtalálható felsorolásban lévő hivatkozásokat tartalmaznak azokra az alfejezetekre, ahol PHPUnit segítségével hoztunk létre automatikus teszteket. Ezeknek a teszteknek a megvalósítását átültethetjük Pest alapúra.

Egyelőre most csak az egyszerűbb feladattal foglalkozzunk! Telepítsük a Pest-et a függőségeivel együtt, ha a Laravel telepítésénél nem ezt választottuk automatikus tesztelési keretrendszernek (előtte távolítsuk el a PHPUnit-ot):

```
composer remove phpunit/phpunit

composer require pestphp/pest --dev --with-all-dependencies
```

Inicializáljuk a projektben a Pest-et, amivel létrehozunk a **tests** mappában egy **Pest.php** beállítási fájlt:

```
./vendor/bin/pest –init
```

Utána már futtathatjuk is a tesztelést:

./vendor/bin/pest

Ezzel az ExampleTest tesztek sikeresen lefutnak, de hozzunk létre saját unit tesztesetet is:

php artisan make:test SumTest -u

A tests / Unit / SumTest.php fájlban adjuk meg ezt a tartalmat:

```
<?php

function sum(int $a, int $b)
{
  return $a + $b;
}

test('sum', function () {
  $result = sum(1, 2);

  expect($result)->toBe(3);
});
```

13–1. kódrészlet: Összeadás metódus tesztelése Pest keretrendszer segítségével

Megjegyzés: itt a sum() metódus nem egy másik osztály metódusát teszteli, hanem ugyanott került megvalósításra a sum() is, de ez most csak az egyszerűség miatt volt így.

Futtassuk a tesztet:

php artisan test tests/Unit/SumTest.php

Az eredmény pozitív, sikeresen lefut az első tesztünk a Pest keretrendszer segítségével.

13.4.3. Laravel Dusk

A Laravel Dusk egy olyan tesztelési eszköz, amely egyszerűen használható böngészőbeli automatizmusok végrehajtására és tesztelésre. Mindehhez csak egy Google Chrome kiterjesztésre van csak szükségünk, és utána már futtathatjuk is a teszteseteket, akár végignézhetjük a tesztek lefutásának látványos lépéseit.

Munkánk során már találkoztunk vele és használtuk a Laravel Dusk eszközt. Mi ezeket hajtottuk végre vele:

- a nézetek elemeinek meglétét ellenőriztük, továbbá az űrlap elküldését automatizáltuk tesztesettel (8.4. alfejezet),
- majd a szerver és kliens oldali validációs szabályokat ellenőriztük (9.3.3. alfejezet).

Ezen kívül rávilágítottunk olyan funkcionalitásaira, amelyek hasonlítottak, de minimálisan különböztek is a PHPUnit-os tesztelések végrehajtásától.

13.5. Összegzés és továbblépés

A fejezet során megismerkedtünk a programkód minőségét javító alapelvekkel és technikákkal. A statikus tesztelési eljárások közül a felülvizsgálati technikák csak felsorolásra, illetve hivatkozásra kerültek, de a

statikus elemzési eszközök közül kipróbáltunk néhányat a gyakorlatban, amelyek a Laravel keretrendszer bizonyos verzióihoz illeszkedtek. Ezekkel az eszközökkel felfedezhetjük a programkódjainkban elrejtőző hibákat, és ki is javíthatjuk őket manuálisan, vagy az eszköz kijavítja nekünk automatikusan, így a használatuk mindenképpen javasolt egy projekt fejlesztése során.

A továbbiakban a dinamikus tesztelési technikákról tudhattunk meg információkat. A PHPUnit eszközt az összes munkánk során aktívan használtuk, de a Laravel 11-nél már a PEST tesztelési keretrendszer is tökéletesen használható, sőt, alapértelmezetten már ezt adja nekünk a rendszer telepítésnél. Általában megállapítható az, hogy aki a PHPUnit keretrendszer használatát elsajátította, annak nem lesz problémája a PEST alkalmazásával sem, mivel a bonyolultabb eszközről áttérni az egyszerűbbre, az mindig egy könnyebb folyamat. Böngésző tesztelésre a Laravel Dusk eszköz alkalmazható, de korábban a gyakorlatban már ezt is többször használtuk.

Blog: A kódminőség kapcsán fontos téma még az alkalmazás fejlesztéséhez használt dokumentáció készítése, főleg akkor, amikor egy API felületet is biztosít az alkalmazás, amellyel más fejlesztők, programok is hozzáférhetnek a saját programunkhoz. Ilyen esetben különösen fontos egy jól felépített dokumentáció elérhetősége.

A programkódok javítását hibák felfedésével pedig a debug-olás művelettel tudjuk megtenni, ebben a Telescope csomag segít minket a Laravel-ben.

A tudásanyag eléréséhez érdemes követni a blogomon a #Kódminőség (Code Quality) címkét, és az ott fellelhető újabb bejegyzéseket.

A cél tehát az volt, és mindig az, hogy az alkalmazásaink minősége javuljon, amelyet az itteni és a hivatkozott, ajánlott tudás birtokában és felhasználásával el tudunk érni a jövőben.

14. Webalkalmazás publikálása, közzététele (Build & deploy)

A fejezet során áttekintjük, hogy hogyan lehet a Laravel keretrendszer támogatásával készített webes alkalmazásainkat elérhetővé tenni a nyilvánosság számára. Végtére is, ez a webes alkalmazások egyik alapvető célja. A bemutatás során egy hazai, magyarországi nyilvános tárhely és domain szolgáltató lehetőségeit vesszük igénybe. Utána pedig felköltözünk a felhőbe és a Microsoft Azure szolgáltatásainak támogatásával tesszük közzé az alkalmazásunkat.

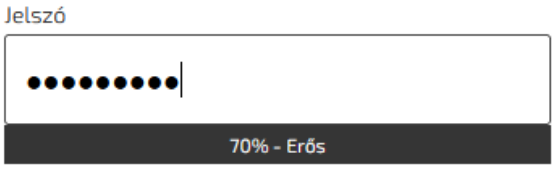
14.1. Nyilvános tárhely és domain szolgáltatások Laravel projektekhez (Public hosting and domain services for Laravel projects)

Számos webhely kiszolgáló létezik, köztük magyarok is. Mi pedig a Laravel projektjeinket szeretnénk futtatni, úgyhogy szükségünk van *„speciálisabb"* szolgáltatásokra is a webszerver lehetőségein kívül, mint például PHP fordító, adatbáziskezelő rendszer (például MySQL), esetleg SSH, Composer stb. Itt elérhető egy top lista a magyar tárhelyszolgáltatásokról, amelyet évről-évre frissítenek. Ezek közül a Nethely használatát vesszük végig úgy, hogy egy Laravel projektet telepítünk fel oda. A Nethely egy ingyenes tárhely és domain szolgáltató, amely a diákok, hallgatók, tanulni vágyó fejlesztők számára kiváló lehetőséget nyújt a Laravel projektek közzétételéhez, már az ingyenes csomagjával is: Nginx webszervert ad folyamatos felügyelettel, az éppen aktuálisan legfrissebb PHP verziót (8.3, de a régebbieket is lehet használni). A többi, számunkra fontos szolgáltatására és eszközére pedig ebben az alfejezetben még ki fogunk térni.

14.1.1. Nyilvános tárhely és domain szolgáltatás létrehozása a Nethelyen

Készítsük el az ingyenes tárhelyünket és domain-ünket is az alábbi útmutató alapján:

1. Készítsük el a Nethely weboldalon a felhasználónkat a regisztrációs oldalon: https://www.nethely.hu/ugyfelszolgalat/regisztracio - igyekezzünk egy erős jelszót választani, a rendszer segít minket a megadásakor, hogy éppen mennyire erős az aktuálisan beírt jelszavunk.



14–1. ábra: Nethely regisztráció során a jelszó erősségére vonatkozó élő információ

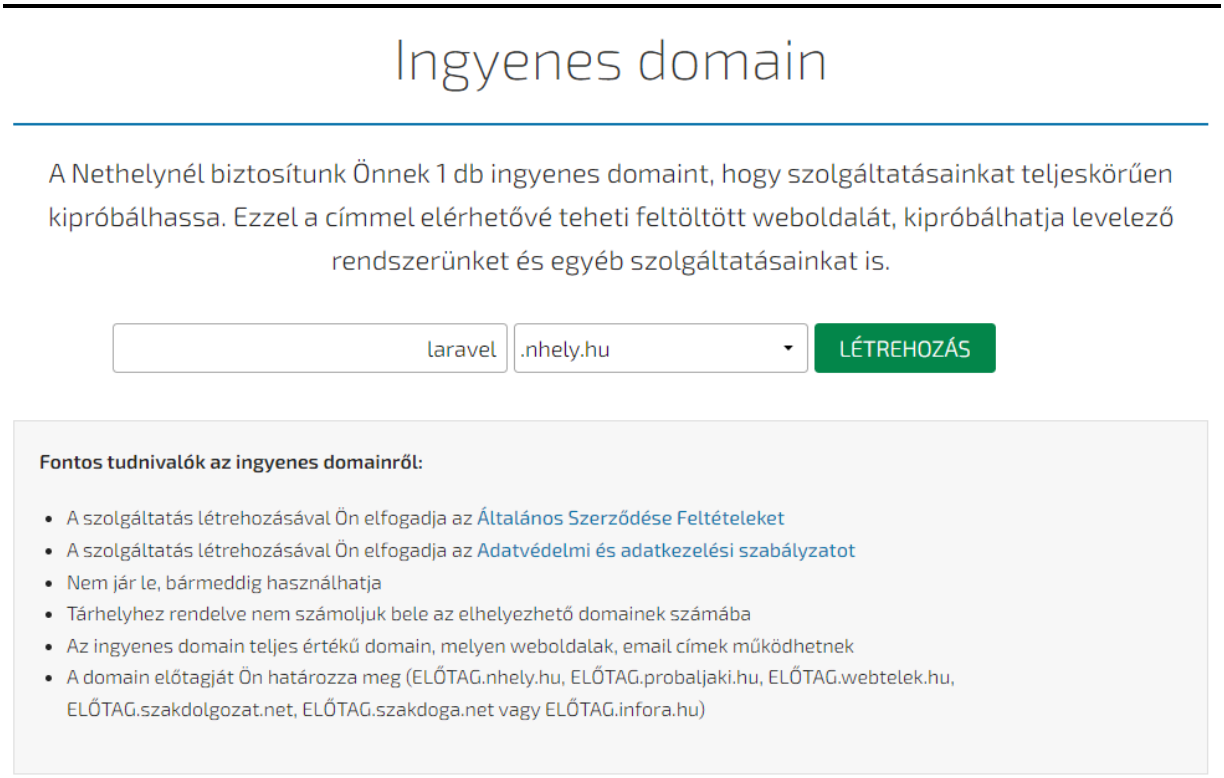
2. A regisztrációt meg kell erősíteni, a megadott e-mail címre érkezni fog egy levél, amelyben egy link van. Ha rákattintunk, akkor regisztráció utáni oldalra jutunk, és ott a fiókunkat hitelesíteni kell.

14–2. ábra: Regisztrációs folyamat sikeresen lezárult a fiók hitelesítésével

3. A Tárhely szekcióban válasszuk ki az *„Ingyenes tárhely"* funkciót, ez kezdetben nekünk megfelelő lesz. Majd kattintsunk a *„Bekapcsolás"* gombra, és már el is készül a tárhelyünk.

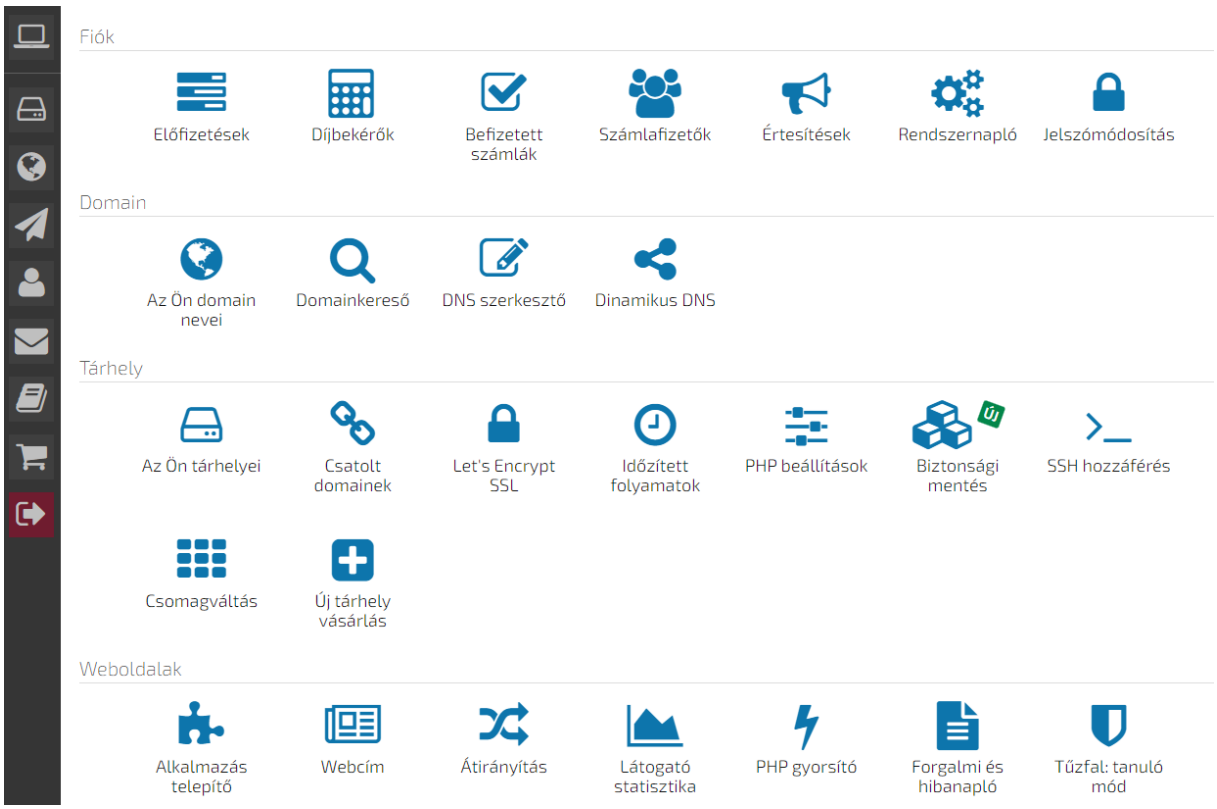
14–3. ábra: Ingyenes tárhely jellemzői és a felhasználási feltételei

4. Utána hozzuk létre az ingyenes domain-ünket! Válasszunk egy nekünk (és a céljainknak) megfelelő domain-t, és kattintsunk a *„Létrehozás"* gombra! Ha foglalt már a domain, akkor erről visszajelzést kapunk, és egy másikat kell választanunk.



14–4. ábra: Ingyenes domain létrehozás és a felhasználási feltételei

5. A domain létrehozása után egyből a vezérlőpult felületre érkezünk meg.



14–5. ábra: Tárhely és domain szolgáltatásaink vezérlőpultja (részlet)

Az ingyenes tárhely és domain szolgáltatásnál sajnos nem érhetők el olyan kényelmi funkciók és eszközök használata, mint az SSH-n keresztüli becsatlakozás az alkalmazásunkhoz, vagy a Composer csomagkezelő

lehetőségei. De ez nem probléma a mi esetünkben, mivel ezek nélkül is meg fogjuk tudni oldani a Laravel webes alkalmazásunk közzétételét. *Megjegyzés:* ha érdeklődik az Olvasó a fizetős SSH és Composer használati lehetőségekről a Nethelyen, akkor ennek a kényelmesebb folyamatnak a bemutatását megtalálhatja a blogomon.

14.1.2. Laravel projekt közzététele a Nethelyen

A közzétételi folyamat bemutatásához létrehozhatunk egy új Laravel projektet, vagy egy már meglévőt publikálhatunk ki a tárhelyünkre és domain-ünkre. A továbbiakban egy új Laravel projekt publikálása kerül ismertetésre.

14.1.2.1. Új projekt Laravel létrehozása

Lépjünk be a terminal-unk segítségével (akár a VSCode-ban) abba a mappába, ahol a projektjeink vannak! Utána adjuk ki a projekt létrehozó utasításunkat: ha a legfrissebb verziót akarjuk telepíteni, akkor a `laravel new projektnév` paranccsal, ha egy régebbi verziót szeretnénk létrehozni, akkor a `composer create-project` parancsát használhatjuk hozzá.

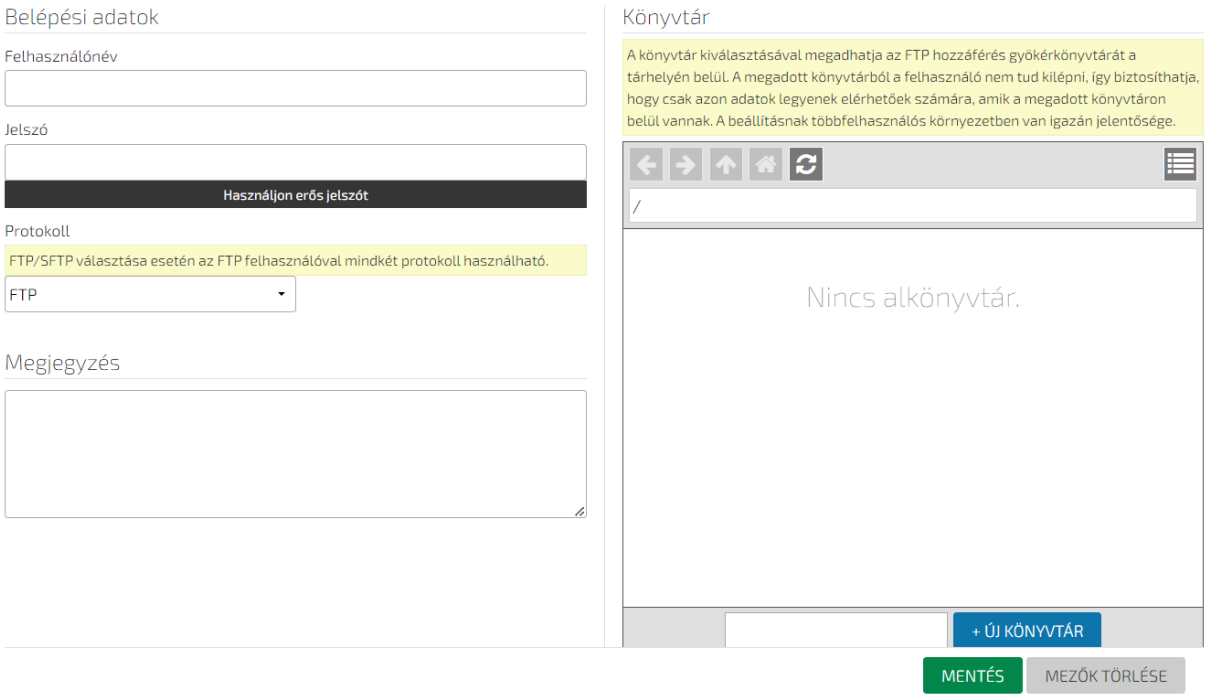
```
laravel new l11-nethely
```

Ha létrejött a projekt, akkor haladhatunk tovább a közzétételi folyamattal.

14.1.2.2. FTP hozzáférés létrehozása a Nethelyen

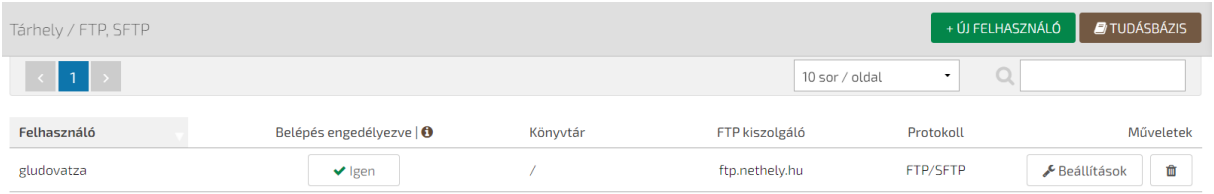
A Nethely vezérlőpultján (14–5. ábra) keressük meg az *„FTP / SFTP hozzáférések”* menüpontot és kattintsunk rá! Hozzunk létre egy új FTP[28] hozzáférést, az *„Új FTP hozzáférés létrehozása”* gomb megnyomásával eljutunk az érintett űrlapra.

[28] File Transfer Protocol



14–6. ábra: FTP hozzáférés létrehozásának űrlapja

Adjuk meg a FTP hozzáféréshez tartozó felhasználó nevét és jelszavát, ezzel fogunk majd belépni az FTP kapcsolat létrehozására alkalmas programmal a Nethelyen lévő tárhelyünkre. Igyekezzünk minél erősebb jelszót megadni! Protokollnak válasszuk az FTP/SFTP lehetőséget, hogy a későbbiekben bármelyikkel tudjunk majd csatlakozni, igény szerint. Mást nem szükséges megadni, úgyhogy ezután elmenthetjük.

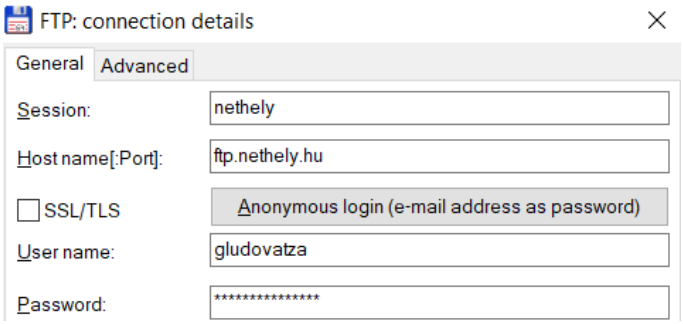


14–7. ábra: Sikeres FTP hozzáférés létrehozás után megjelenik a listában az újonnan létrehozott felhasználó

14.1.2.3. Becsatlakozás FTP-n keresztül a Nethelyes tárhelyre

Ha minden sikerült, akkor telepítsük a Total Commander alkalmazást is, vagy bármilyen egyéb alkalmazást, amely erre a feladatra használható.

A Total Commander elindítása után a menüsorban az FTP ikon megnyomásával egy kis ablak (*„Connect to ftp server”*) ugrik fel, ahol egy új kapcsolatot (*„New connection”*) hozhatunk létre. Adjunk egy nevet a session-nek, a Host name legyen **ftp.nethely.hu**, a User name az imént megadott FTP hozzáférésünk, amit létrehoztunk a Nethelyen, illetve a Password a hozzá tartozó jelszó.



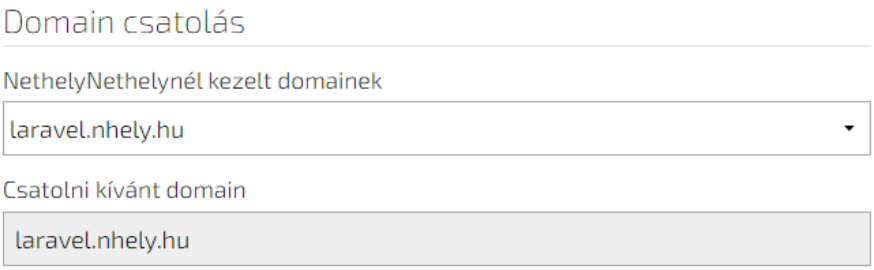
14–8. ábra: Session beállítása a Total Commander programban

Elmenthetjük a Session-t, elfogadhatjuk az automatikusan felajánlott lehetőséget ekkor, illetve elmenthetjük a jelszavunkat is hozzá. Mentés után kétszer kattinthatunk az új Session nevére, és bejön a távoli tárhelyen lévő mappa/fájlstruktúra, ez kezdetben nyilvánvalóan üres lesz.

Keressük ki az imént létrehozott Laravel projektünket (**l11-nethely**) és kezdjük el felmásolni a tartalmát a távoli tárhelyre (minden mappa és fájl kijelölése után F5 funkcióbillentyű megnyomásával például).

14.1.2.4. Webalkalmazás elérésének beállítása a Nethelyen

Hozzuk be a Nethely vezérlőpultját! Keressük meg a *„Webcím”* menüpontot, kattintsunk rá, majd nyomjuk meg a *„Domain csatolás”* gombot. Van már domain-ünk, így kattintsunk az *„Új domain csatolása”* gombra. Válasszuk ki a korábban létrehozott domain-ünket.

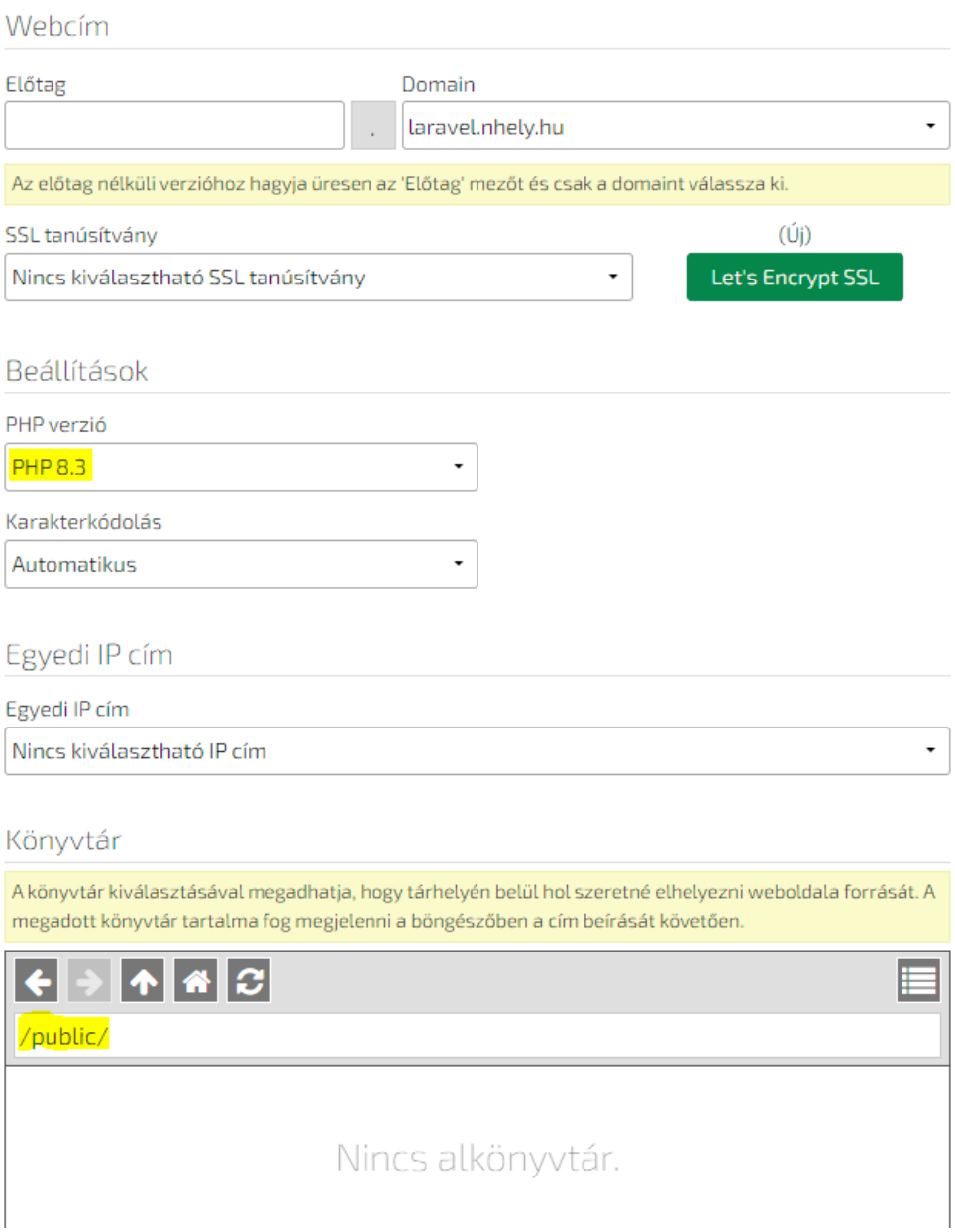


14–9. ábra: Domain csatolás

Mentés után megjelenik a listában a csatolt domain-ünk és néhány pillanat múlva visszajelzést kapunk arról, hogy beállításra került a csatolt domain-ünk.

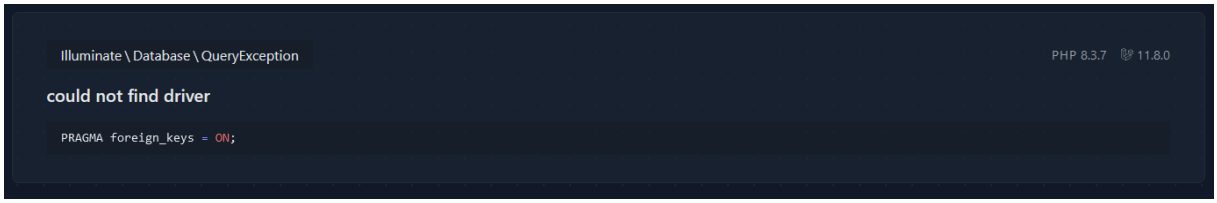
Ha sikerült, vissza kell menni a *„Webcím”* menüpontra, és létre kell hozni a webcímet, a gomb megnyomásával megkapjuk a hozzá kapcsolódó űrlapot (14–10. ábra).

Nagyon fontos, hogy Laravel 10 és 11-es verzió esetén is a PHP 8.x-et válasszuk ki (aktuálisan legfrissebb a 8.3), és a könyvtáron belül a **/public** mappába legyünk, mivel a Laravel projektek kiszolgálása az itteni **index.php** fájlon, mint belépési ponton keresztül történik meg.



14–10. ábra: Új webcím létrehozásának űrlapja

Mentés után, néhány pillanat múlva elkészül a webcímünk a megadott beállításokkal és rögtön böngészhetjük is az alkalmazásunkat, azonban kezdetben hibát kapunk:



14–11. ábra: PHP az SQLite kiterjesztés hiánya miatti hiba

A probléma az SQLite adatbázis eléréséhez szükséges PHP kiterjesztés engedélyezésének hiányából adódik. A Nethely vezérlőpultján keressük meg a *„PHP beállítások”* menüpontot, és kattintsunk rá. Utána a verziót kell kiválasztanunk, amelynél válasszuk azt, amit a webcím létrehozási űrlapon (14–10. ábra)

kiválasztottunk. Ezután megkapjuk a PHP beállítási felületet. A PHP modulok szekcióban engedélyezzük az SQLite3 és a PDO SQLite nevű modulokat, és mentsük el a beállításokat (14–12. ábra).

14–12. ábra: PHP modulok (kiterjesztések) engedélyezése/letiltása (részlet)

Ezután működni fog az alkalmazásunk! Ezen a címen elérhető: http://laravel.nhely.hu/

A Nethely tárhely szolgáltatóhoz tartozó közzétételi folyamat FTP/SFTP-n keresztül valósult meg. Ezt a módszert a 14.3.2. alfejezetben is alkalmazhatnánk, amikor a Microsoft Azure felhőszolgáltatásába publikáljuk a webes projektünket, de ott majd egy másik folyamattal ismerkedünk meg.

A Nethely tárhely szolgáltató Laravel projekt közzététel szempontjából fontos funkcióinak ismertetéséhez, kipróbálásához, teszteléséhez egy korábban nálam Szakdolgozatot készítő hallgató, **Dalos Donát** volt a segítségemre.

14.1.3. Módosítások érvényre juttatása a környezeti változókban és a programkódban

A környezetbeli és programkódbeli változásokat is manuális módosítással, illetve másolással-beillesztéssel kell majd érvényre juttatnunk, kvázi verziókezelés nélkül tudjuk ezt megtenni. Ez a *„kényelmetlenség”* az ingyenesség ára!

14.1.3.1. Adatbáziskezelő rendszer lecserélése

Lokálisan hozzuk létre az **l11_nethely_db** nevű adatbázist! VSCode-ban hajtsuk végre az **.env** fájl módosítását az **sqlite** kapcsolattípusról a **mysql**-esre! Utána migráljuk az adatszerkezeteket az adatbázisba adatok seed-elésével együtt!

```
php artisan migrate --seed
```

Egy *„Test User”* nevű felhasználó fog létrejönni mindössze a **users** adattáblába.

Az adatbázis tábláit és adatait exportáljuk egy SQL fájlba például a phpMyAdmin eszközzel, amit majd importálni fogunk a Nethelyen.

MySQL adatbáziskezelő rendszer használatához a Nethely is biztosít phpMyAdmin felületet a vezérlőpult *„Adatbázisok”* szekciójában. Ezen keresztül könnyedén tudjuk importálni a lokálisan létrehozott

adattáblákat az előtte exportált adatbázisunkból. Ez a műveletsor ingyenes szolgáltatás esetén is tökéletesen tud működni. Ha fizetős szolgáltatást használunk, akkor alkalmazhatjuk az SSH segítségével a migrációs fájlokat tábla létrehozásra, a seeder fájlokat pedig az adatok betöltésre.

De előbb hozzuk létre a Nethelyen az adatbázist! A Nethelyen csak egy adatbázist tudunk létrehozni ingyenesen, kevesebb a mozgásterünk ott. A vezérlőpulton válasszuk ki az *„Adatbázisok kezelés"* menüpontot, majd az *„Új SQL adatbázis létrehozása"* gomb megnyomásával létrehozhatjuk az adatbázisunkat. Előtte azonban szükség van az adatbáziskapcsolat paramétereinek megadására. Az adatbázis neve ugyanaz, mint a felhasználónak a neve, amivel csatlakozni tudunk majd az adatbázisunkhoz. A jelszó legyen erős! Az adatbázis típusa legyen MySQL, bár a Laravel-nek a PostgreSQL-lel sem lenne gondja, ugyanúgy tudná kezelni. A karakterkódolás pedig UTF-8 maradhat. Mentés után létre is jön az adatbázisunk.

Adatbázis	Felhasználó	Típus	Méret	Karakterkódolás
gludovatza	gludovatza	MySQL	0 MB	UTF-8

14–13. ábra: Nethelyen létrejött a MySQL adatbázis

Az adatbázis távoli elérése és a publikus phpMyAdmin felülete csak fizetős csomagokban elérhető.

Kattintsunk a létrehozott adatbázisunk sorában a phpMyAdmin gombra! Válasszuk ki az új adatbázisunkat a bal oldali panelben. Utána a jobb oldali panelen válasszuk az Importálás menüpontot a vízszintes menüben. Az imént exportált **.sql** fájl kitallózása után kattinthatunk is az *„Importálás"* gombra. Sikeres lefutás esetén létre is jönnek a Nethelyes adatbázisban a lokálisan is meglévő tábláink, tartalmaikkal.

14.1.3.2. Programkód módosítása

A MySQL adatbázis eléréséhez a szerveren lévő **.env** fájlt kell módosítanunk, ahogy azt tettük lokálisan is.

```
DB_CONNECTION=mysql
DB_HOST=127.0.0.1
DB_PORT=3306
DB_DATABASE=gludovatza
DB_USERNAME=gludovatza
DB_PASSWORD=titkos
```

14–1. kódrészlet: Nethelyen lévő .env fájl DB_ prefixű attribútumok értékei (a szükségesek módosítandók, sárga hátterűek)

Ha valamelyik beállítást elrontjuk, akkor a weboldalunk nem fog helyesen betöltődni.

Illuminate \ Database \ QueryException

SQLSTATE[HY000] [1045] Access denied for user 'gludovatza'@'localhost' (using password: YES)

```
SELECT * FROM `sessions` WHERE `id` = AKPBdM5OY5NAYlenLGXN20Ri2xuzIyiKV4OpETyC limit 1
```

14–14. ábra: Elrontott kapcsolódási adatokkal hibát kapunk az oldalunkon

Ha megfelelően adjuk meg az adatokat, akkor viszont tökéletesen és hibátlanul betöltődik a Laravel projektünk kezdőoldala.

Tipp: bár az ingyenes regisztrációnkkal SSH-t nem tudunk használni, így nem tudjuk a terminal-ban már megszokott `php artisan` kezdetű parancsokat kiadni, de egy kicsit tudunk trükközni. Ha megnyitjuk az **app / Providers / AppServiceProvider.php** fájlt a Nethelyen szerkesztésre, akkor a **boot()** metódusába elhelyezhetünk `artisan` parancsot, amely aztán végrehajtódik, amikor valaki az alkalmazást betölti a böngészőben (behozza az oldalát). A példa kedvéért csak egy olyan parancsot hajtsunk végre, ami a beállítások törlését eredményezi a gyorsítótárból, ha valamilyen hibás beállítás *„beragadna"*, és nem tudnánk kieszközölni a kapott hibaüzenet frissülést, akkor ezt a parancsot például érdemes alkalmazni.

```php
use Illuminate\Support\Facades\Artisan;
//...
public function boot(): void
{
  Artisan::call('config:clear');
}
```

14–2. kódrészlet: Artisan parancs hívása a programkódon belül

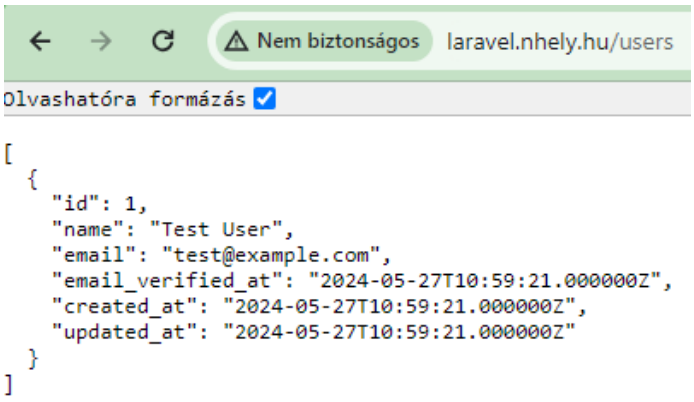
Ha pedig ott szeretnénk például migrálni seed-eléssel együtt, akkor csak írjuk át a *„config:clear"* részt *„migrate --seed"*-re.

Egy új útvonal regisztrálásával az adatbázis hozzáférést és lekérdezést is ellenőrizhetjük direktben. Nyissuk meg helyben a VSCode-ban a **routes / web.php** fájlt és adjunk hozzá egy új útvonalat:

```php
use App\Models\User;
//…
Route::get('/users', function () {
  return User::all();
});
```

14–3. kódrészlet: Adatbázisban létező felhasználókat listázó útvonal regisztrálása

Ha helyben működik a **/users** útvonal lekérése és megkapjuk a böngészőben a *„Test user"* adatait, akkor másoljuk fel a **web.php** fájlt a Nethelyen lévő tárhelyünkre, írjuk felül az ott lévő **web.php** fájl tartalmát. Ezután a Nethelyen lévő címen is meg kell kapnunk az adatbázisban lévő felhasználóinkat.



14–15. ábra: Adatbázisban létező felhasználók listázása az útvonal elérésével

Ezekből a példákból látható volt, hogy hogyan kell megváltoztatni a Nethelyen lévő Laravel alkalmazásunk környezeti változóit (például egy új adatbáziskiszolgáló és adatbázis hozzáadásával), és a programkódunkat hogyan tudjuk manuálisan módosítani, felülírni az újabb funkciók megvalósításához.

14.2. Felhőszolgáltatások (Cloud Computing services)

A felhőszolgáltatások alkalmazásához előbb egy rövid bemutatást tekintünk át azért, hogy utána egy konkrét szolgáltató segítségével előbb a webes alkalmazásunk adatbázisát (adatszolgáltatását), utána pedig az egész szoftverprojektet publikáljuk és futtassuk a felhőben.

14.2.1. Felhőszolgáltatások általános jellemzői

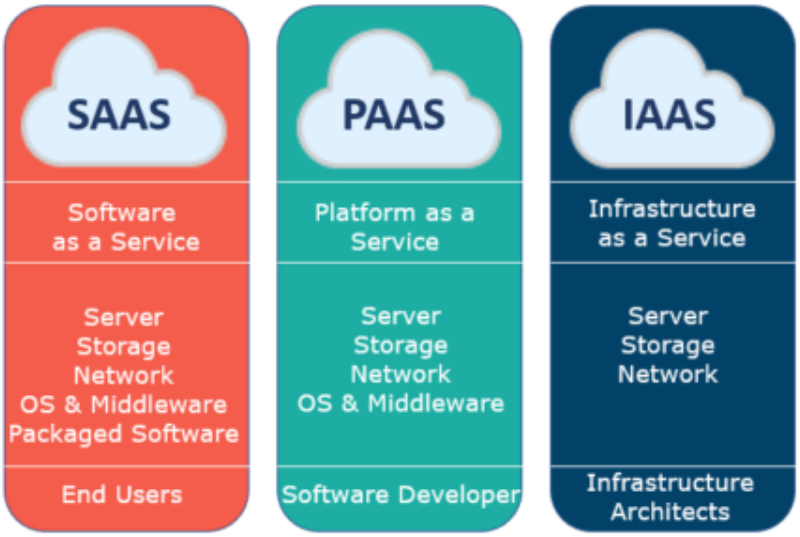
2010-es évektől napjainkig kedvelt informatikai ágazat a Cloud Computing (Felhőalapú számítástechnikaként is fordíthatnánk). A neves informatikus John McCarthy nevéhez fűződik, hogy a technológiát az áramhoz és vízhez hasonlóan közüzemi szolgáltatásként akarta értékesíteni.

Egyik fontos tulajdonsága, hogy a szolgáltatások nem csak egy fizikai eszközön, hanem a szolgáltató eszközein elosztva, a felhasználóktól rejtve működnek. A szolgáltatásokat a felhasználók publikus felhő esetében az interneten keresztül, míg privát felhő esetén a helyi hálózaton vagy szintén az interneten keresztül érhetik el. Fő jellemzője még az önkiszolgálás, a felhasználók dönthetnek a számítási kapacitás igénybevételéről és mértékéről. A hálózati tárolás vagy szerver idő igénylés a szolgáltatótól függetlenül, automatikusan, interakció nélkül történik. Továbbá még rugalmasság jellemzi, hiszen az aktuális igénybevétel alapján változtathatják a kapacitást.

A felhő alapú rendszerek automatikusan vezérlik és optimalizálják az erőforrásaikat a szolgáltatás típusának és igényeinek megfelelően. Az erőforrás-felhasználás jól megfigyelhető, ellenőrizhető és pontosan mérhető, ezzel biztosítva átláthatóságot a felhasználók számára. Ehhez szorosan kapcsolódik még, hogy a felhőszolgáltatásoknál az ügyfél sosem egy fix havidíjat, hanem az általa használt erőforrások arányában fizet. Az egyik legfontosabb tulajdonsága a közös erőforrás használata, ami lehet memória, sávszélesség vagy akár tárhely is. Nagy mennyiségű fizikai és virtuális erőforrásokat gyűjtenek össze, hogy azonos időben a sok ügyfél összes igényét ki tudják szolgálni.

14.2.2. Felhőszolgáltatások fajtái

Az *„as a Service"* kifejezés a felhőalapú számítástechnikai szolgáltatásokat jelenti, amelyeket egy harmadik fél biztosít a felhasználóknak. A Cloud Computing minden típusa arra törekszik, hogy felhasználóinak kevesebb helyszíni hálózatot és fizikai eszközt kelljen üzemeltetni.



14–16. ábra: Felhőszolgáltatások fajtái, fizikai és szoftveres eszközei, felhasználó típusai

14.2.2.1. SaaS (Software as a Service), vagyis Szoftver, mint Szolgáltatás

A szoftverfrissítéseket, hibajavításokat, beállításokat és az általános szoftver karbantartásokat a szolgáltató kezeli. A végfelhasználók a szolgáltatói alkalmazásokat egy webböngészőn (irányítópulton vagy API-n) keresztül érik el, ezért nincs szükség az alkalmazások telepítésére minden egyes számítógépen. SaaS típusú szoftverek az online levelezők (például Yahoo mail, GMail), a számviteli rendszerek, az elemző és kommunikációs eszközök, valamint a tárolási kapacitást, másnéven felhőtárhelyet kínáló szolgáltatások (például Dropbox, OneDrive, Google Drive). A szolgáltatásokat nem minden esetben csak számítógépes alkalmazáson keresztül lehet használni, gondoljunk akár a GMail-re, amihez bármilyen webböngészővel hozzá tudunk férni, és el tudjuk olvasni üzeneteinket. A felhőtárhelyek esetén szintén láthatjuk, hogy bármilyen okos eszközről el tudjuk érni az ott tárolt adatainkat.

A SaaS jó lehetőség lehet azon kisvállalkozások számára, akiknek nincs külön alkalmazottja a szoftverek telepítésére és frissítésére, esetleg nem akarnak telepíteni és beállítani olyan alkalmazásokat, amelyeket csak időszakosan fognak használni. A felhasználók a szolgáltatásokért cserébe díjat fizetnek, ezt nevezzük licencdíjnak, amit fizethetnek havonta vagy akár évente is.

Összességében a SaaS időt és karbantartási tevékenységet takaríthat meg a felhasználóknak, ez viszont – a licencdíjon kívül – a biztonságuk kockáztatásába vagy a teljesítményük rovására történhet meg, ezért fontos, hogy megbízható és jó hírnévvel rendelkező szolgáltatót válasszanak maguknak.

14.2.2.2. PaaS (Platform as a Service), vagyis Platform, mint Szolgáltatás

Akkor beszélünk platformról, mint szolgáltatásról, amikor a szolgáltató biztosítja a mögöttes platformot (operációs rendszert), illetve a teljes hálózatot, tárolást, kiszolgálást, mindezeket akár virtuálisan is. Ezt internetkapcsolaton keresztül nyújtja a felhasználóknak.

A PaaS főként a fejlesztők és programozók számára hasznos, mert lehetővé teszi saját alkalmazásaik fejlesztését és futtatását anélkül, hogy egy infrastruktúrát vagy platformot kellene kiépíteniük és karbantartaniuk. Ez a módszer egy keretrendszert biztosít a fejlesztők számára webes alkalmazásaik felépítésére és testre szabására. A fejlesztők beépített szoftverösszetevőket használnak, ezzel csökkentve az írandó kód mennyiségét.

A felhasználóknak, vagy jelen esetben a fejlesztőknek elegendő lefejleszteniük az alkalmazást és azt a felhőbe telepíteni. Nem kell foglalkozniuk a szoftverek frissítésével és hardvereszközök karbantartásával, a fejlesztéshez szükséges környezet biztosítva lesz számukra a szolgáltató által. PaaS rendszert kínál például az AWS Elastic Beanstalk, a Heroku vagy a Red Hat OpenShift.

14.2.2.3. IaaS (Infrastructure as a Service), vagyis Infrastruktúra, mint Szolgáltatás

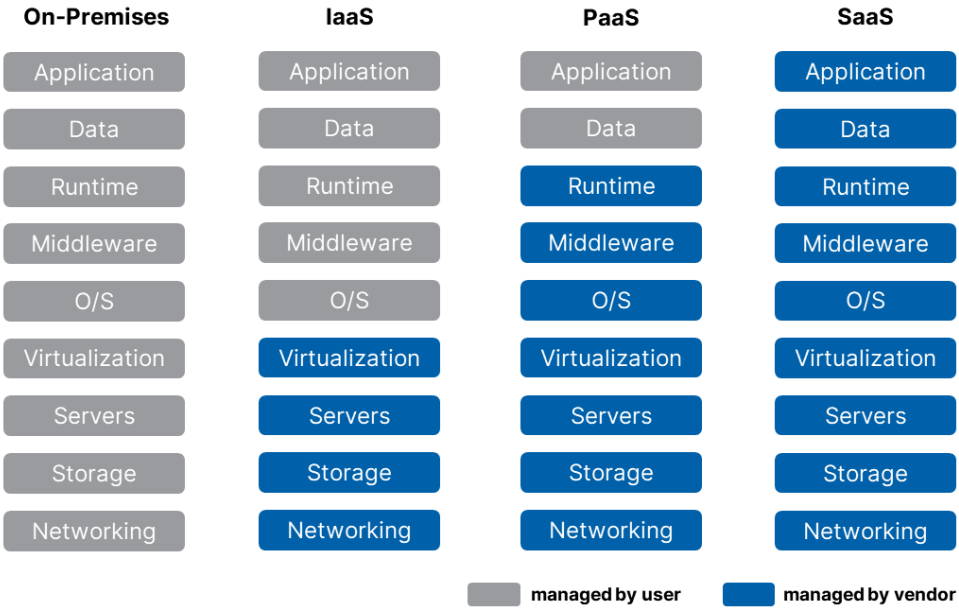
Az IaaS egy olyan szolgáltatás, ahol a felhasználók egy harmadik fél (magán) infrastruktúráját, hardvereit tudják kibérelni. Interneten keresztül tárolási és virtualizációs szolgáltatásokat nyújt nekik a szolgáltató. Fontos tulajdonsága, hogy a felhasználók nem havidíjat vagy licencdíjat kell fizetniük, hanem a felhasznált erőforrással arányos összeget. Alacsony szolgáltatás fenntartási díj jellemzi, és nincsenek karbantartási költségek, ezért megfizethető lehetőséget kínál bármely kisebb cég számára is.

Ez már jóval komplexebb üzemeltetési beállításokat tesz számunkra elérhetővé, így itt már magunknak kell figyelnünk, beállítanunk a futtató szerver operációs rendszerét, a tárhelyét menedzselnünk kell, a tűzfalat beállítani stb.

Akkor érdemes egy IaaS-t igénybe venni, amikor semmilyen berendezést, hardvert vagy szervert sem szeretnénk fenntartani, de a szoftvereink feletti felügyeletet szeretnénk megőrizni. A feladat ellátására egy virtuális gépet kell bérelnünk, ezen pedig az általunk választott alkalmazásokat és operációs rendszert tudjuk használni. Használatával gyors, rugalmas fejlesztési és tesztelési környezetet tudunk kialakítani. Az infrastruktúrát bármikor tudjuk növelni vagy csökkenti, továbbá konkrétan annyi erőforrásért fizetünk, amennyit használunk.

A hardveres részt a szolgáltató távolról biztosítja, így a felhasználóinak nem kell semmilyen karbantartási vagy frissítési feladatot ellátniuk, ezt a szolgáltató elvégzi majd helyettünk. Az infrastruktúrát egy programozási felületen (API-n) vagy egy irányítópulton keresztül érhetjük el.

A rendszernek azonban hátránya a megbízhatósági és biztonsági problémák, amelyek a megosztott, több bérlő által használt infrastruktúrából erednek. Ezek azonban elkerülhetőek, ha megbízható, szilárd múlttal és hírnévvel rendelkező szolgáltatót választunk. IaaS szolgáltatás nyújt az Amazon Web Services, a Microsoft Azure és a Google Cloud.



14–17. ábra: Szolgáltatások fajtái és felügyeletük

Az összefoglaló ábra megmutatja, hogy melyik szolgáltatás típus miket foglal magába és ki (a felhasználó vagy a szolgáltató) felügyeli az egyes részeket fajtánként. Ez az áttekintés megmutatja nekünk, hogy melyik területeket felügyeli és menedzseli maga a felhasználó (user - szürke), vásárló, ügyfél, nevezhetjük többféleképpen, illetve azt, hogy melyiket menedzseli a *„szállító"* (vendor - kék), vagyis felhő szolgáltató. A bal oldali oszlop azt mutatja, amikor a felhasználó mindent a saját hardveres és szoftveres környezetében futtat, és mindenért saját maga felelős. Minél inkább jobbra tekintünk az oszlopok között, annál inkább a szolgáltató átveszi rólunk a terhet, de ebből adódik, hogy egyre többe is kerül.

14.2.3. Telepítési modellek

A szolgáltatások fajtái mellett létezik még egy csoportosítási mód a felhőszolgáltatások megkülönböztetésére, ezeket a csoportokat nevezhetjük a telepítési modelleknek. Ebben a csoportosítási módszerben a hozzáférhetőség (a szolgáltatásokat ki, kinek és hogyan osztja meg) kerül előtérbe, így történik meg a kategorizálás. Az Amerikai Nemzeti Szabványügyi Intézet négy telepítési modellfajtát különböztet meg.

14.2.3.1. Nyilvános felhőszolgáltatás (Public Cloud)

A legismertebb és leggyakrabban alkalmazott modell a nyilvános felhő. A hétköznapi emberek számára is könnyen elérhető, de az erőforrások a szolgáltatók tulajdonában vannak, ők birtokolják és működtetik az infrastruktúrát saját telephelyeiken (világszinten működtetett szerverparkokban). A felhasználók általában interneten keresztül érik el, de ezen modell esetében nincs rálátásuk az infrastruktúrára, és ellenőrizni sem tudják annak elhelyezkedését.

Nyilvános felhők esetében a platform megosztás miatt kisebb a biztonság, a rugalmasság és az irányíthatóság. A legismertebb nyilvános felhőszolgáltatók és szolgáltatásaik: Microsoft Azure, Google Cloud, Amazon Web Services (AWS), Alibaba Cloud, Salesforce, Digital Ocean.

14.2.3.2. Magán felhő (Private Cloud)

Ezt a fajta felhő infrastruktúrát kizárólag egy szervezet/ügyfél használhatja. Az üzemeltetés megoldható úgy, hogy csak a felhasználó szervezet maga kezeli, de ez is működhet harmadik fél által. Itt is igaz, hogy az infrastruktúra lehet fizikailag a felhasználó telephelyén vagy akár azon kívül is. A magánfelhő általában magasfokú biztonsággal rendelkezik, így kiváló választás lehet azoknak felhasználóknak, akiknél fontos szempont az adataik biztonsága. Emellett még nagyfokú rugalmasság és irányíthatóság jellemzi.

14.2.3.3. Hibrid felhő (Hybrid Cloud)

A hibrid szó itt is kettő vagy több dolog keverékére utal, jelen esetben általában a nyilvános és a magán felhő erősségei vegyülnek. A felhők megtartják jellegzetes tulajdonságaikat, valamint szabványosított és szabadalmaztatott technológiák kapcsolják őket össze a megfelelő működés érdekében. A hibrid felhő lehetővé teszi az adatok és az alkalmazások hordozhatóságát. Rugalmasság és többféle telepítési lehetőség jellemzi. Hátrányt jelent azonban a hálózat összetettsége és a megfelelőségi problémák.

Hibrid felhőnél a vállalatok érzékeny adataikat magán felhőben tárolják, míg a támogatott szolgáltatásaikat a nyilvános felhőben, így adataikat ellenőrzésük alatt tudják tartani.

14.2.3.4. Közösségi felhő (Community Cloud)

A közösségi felhő típusú infrastruktúrát több szervezet megosztottan használja, tehát az adott csoport közös érdekeit szolgálja. Ilyen érdekek lehetnek a közös küldetés, a biztonsági követelmények, az előírások és megfelelőségi szempontok. A menedzselést elvégezheti akár a felhasználó szervezet is, de ezt a feladatot kiadhatják egy harmadik félnek is. Az infrastruktúra fizikai elhelyezkedése lehet a felhasználó telephelyén vagy azon kívül is.

14.2.4. Felhőszolgáltatások előnyei és hátrányai

Mielőtt feltelepítenénk a szolgáltatásainkat a felhőbe, érdemes áttekinteni, hogy milyen előnyökkel és hátrányokkal járhat a felköltözési folyamat és az ottani működés, működtetés.

14.2.4.1. Előnyök

A *„házon belüli"* adattárolás és kezelés jelentős pénzbe kerülhet egy cégnek, ugyanis minden egyes szerver megvásárlásának, szoftveres felszerelésének, üzemeltetésének akár óriási költsége is lehet. Felhőszolgáltatások igénybevétele esetén nincs szükség induló tőkeberuházásra, korszerű informatikai eszközök beszerzésére. Bármilyen eszköznél előfordulhat, hogy elromlik (ez lehet hardveres probléma vagy emberi hiba miatt is). Ezek a problémák és költségek megszűnnek felhőalapú számítástechnika alkalmazásakor. A karbantartások összegei bele vannak kalkulálva az infrastruktúra költségeibe, ez pedig fel van osztva az összes ügyfél között (nyilván mi a szolgáltatások igénybevételének arányában fizetünk ezért). A Global Cloud Services Market jelentése[29] szerint a felhőszolgáltatásokat használó szervezetek évente több, mint 20%-ot takarítanak meg a működési költségeiken a 2024-es előrejelzések szerint. Ha

[29] Worldwide cloud service spending to grow by 20% in 2024, https://www.canalys.com/newsroom/worldwide-cloud-q4-2023 (2024. 02. 26.)

kevesebb idő és figyelem megy el az infrastrukturális problémák megoldásra, akkor már az tiszta nyereség a vállalat számára.

A felhőalapú szolgáltatásoknak nagy hatása van a cég dolgozóira is. Ha a szervereket egy házon belüli informatikai csapat kezeli az a költségek növekedéséhez vezethet. Ez a többletköltség csökkenthető felhőszolgáltatások alkalmazásával. A szolgáltató sok feladatot (például karbantartás, biztonsági mentés) levesz a cég alkalmazottainak válláról. Felhőalapú tárolás esetén az adatok elosztásra kerülnek adatközpontok között, így sokkal kisebb az adatvesztés lehetősége. A szinkronizálás lehetővé teszi az adatok gyors összekapcsolását, azok frissítését, de felhő esetén szükségtelenné válik a manuális szinkronizálás (az adatok tükrözése), mert itt mindig pontosan tudjuk, hogy az adataink biztonságban vannak (persze, ha plusz biztosítást szeretnénk adatainknak, akkor azt plusz szolgáltatásként és így költségként, igénybe tudjuk venni). Az adatvesztés bármekkora méretű vállalat számára katasztrófát okozhat. A felhőalapú adattárolás biztonságosabb egy helyi adatközponttal szemben. A felhőtárhelyek mellett szóló nagy előny, hogy nincsen egyetlen meghibásodási pont. Az adatokról több szerverre készül biztonsági mentés, így meghibásodás esetén a szervezet adatai biztonságban maradnak.

Egy nem tervezett leállás pénzügyi hatásait előre nem lehet (vagy nagyon nehéz) megbecsülni. Bekövetkezése súlyosan ronthatja a vállalat hírnevét, főleg, ha az ügyfeleket is érint. Ezen leállásoknak fő okai általában rendszer- vagy emberi hibák, pedig mindkettő elkerülhető lenne felhőszolgáltatások alkalmazásával.

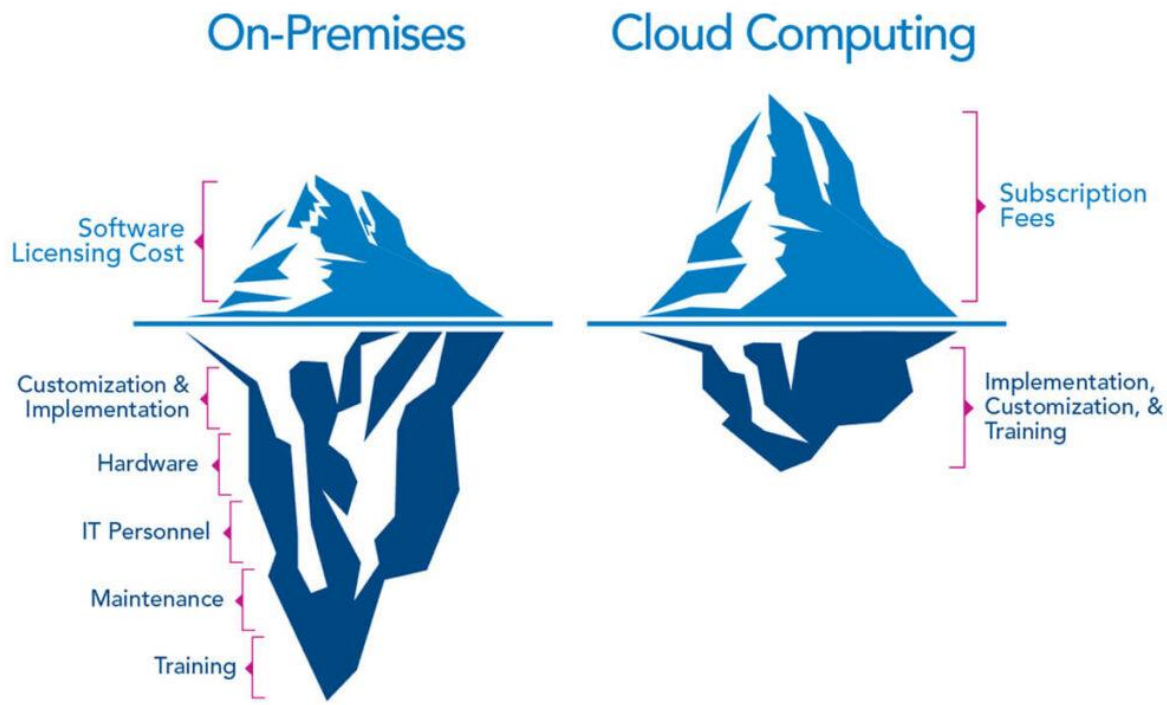
A vállalkozások fellendülésében nagy szerepet játszik a *digitális együttműködés* lehetősége. Ez nagy előnyt jelent számukra hisz az IT szektor alkalmazottai akár otthonról is megfelelően el tudják látni a munkájukat. A felhőalapú számítástechnika lehetővé teszi, hogy a dolgozók hatékonyabban tudjanak együttműködni egy projektben, hiszen több alkalmazott valós időben tudja megtekinteni vagy módosítani a fájlokat. A dokumentumok felhőben való elérése segít, hogy mindenki a megfelelő verzión dolgozzon és az elavult változat ne kerüljön át a helyi források közé.

A növekedés egyik nagy kihívása a *skálázhatóság megőrzése*. Ha a vállalat úgy gondolja, hogy gyorsan növekszik, és több tárhelyre van szüksége, akkor két lehetőség közül választhat. Az első, hogy a szervezet újabb és/vagy több berendezést vásárol, majd azokat telepíti, üzemelteti, illetve a karbantartáshoz szükséges csapatot felveszi, foglalkoztatja. Vagy a másik lehetőség, hogy felkeresnek egy felhőszolgáltatót, és bérelnek tőlük kapacitást. A *szolgáltatások költségei könnyen kiszámíthatók*, valamint a befektetési kockázatok is kiküszöbölhetőek.

A szerverek és berendezések egyik jelentős tulajdonsága, hogy fizikailag helyet foglalnak és speciális környezetet igényelnek és használnak (villamos energiafelhasználás, hűtési költségek, személyi jogosultságok a beléptetéshez – kapuk, sorompók, zárak stb.). A felhőalapú számítástechnika lehetővé teszi, hogy munkaterületet vagy egyéb kényelmi, biztonsági szolgáltatást szabadítsanak fel, közben pedig szükségtelenné teszi a jövőbeli bővítések megtervezését.

A felhőszolgáltatások legnagyobb előny tehát a költségcsökkentés, de azért arra nem érdemes sajnálni a pénzt, hogy jól határozzuk meg a saját erőforrás igényünket és keretünket, amivel kényelmesen és biztonságosan tudunk gazdálkodni a jövőben. Ekkor érvényesül leginkább a költségcsökkentés

paramétere. A felhőbe való költözéshez vegyük igénybe szakemberek segítségét, így sok bosszúságtól megkímélhetjük magunkat a jövőben.



14–18. ábra: Költségek összehasonlítása helyben üzemeltett IT részleg és a felhőszolgáltatások használata esetén (Forrás)

14.2.4.2. Hátrányok

Bár a felhő használata bizonyos területeken segíthet a költségcsökkentésben, de mielőtt egy vállalat áttérne rá fontos meggyőződniük arról, hogy van-e értelme a felhőszolgáltatások használatának. Első körben meg kell vizsgálni a szervezeten belüli összes rendszert, majd a rendelkezésre álló információk alapján egy megfelelő tervet kell készíteni. Ehhez a kulcs a rendszerek elemzése és azok két kategóriába való sorolása. Az első kategória a rendszer olyan részeit tartalmazza, amelyeket át kellene helyezni a felhőbe, a másodikba pedig azok a komponensek tartoznak, amelyeknek helyben kell maradniuk. Ha ezeket sikeresen megismerték, akkor ki tudják számolni a felhőbe költözés részletes költségvetését.

A helyi kiszolgálókról a felhőalapú adatközpontokba való átköltözés a szervezet számára egy aránylag egyszerű folyamat. Ellenben a más felhőszolgáltatóhoz vagy egy helyi kiszolgálóhoz való költözés már nem ennyire könnyű. Ez a folyamat meglehetősen költséges lehet, és a feltételek gyakran a felhőszolgáltatónak kedveznek. Szerződéskötés előtt szükséges megismerni a felhő adatközpontból való kiköltözés folyamatát, és fontos a szerződésben leszögezni a határidőket, a pénzügyi részét és magát a folyamat részleteit is.

Mivel a felhőstruktúra a szolgáltató birtokában van a vállalkozások gyakran nem rendelkezhetnek kellő ellenőrzéssel a szolgáltatás felett. Ebben segíthet a szolgáltató véghasználói licencszerződése (röviden EULA). Minden felhő alapú szolgáltató lehetővé teszi a szervezet számára az ellenőrzést, de az infrastruktúra megváltoztatását nem feltétlenül. A szolgáltató felhasználóinak egy szolgáltatási szint szerződést (angolul SLA = Service Level Agreement) ad, amely segít, hogy az ügyfelek tájékozódjanak arról, mit tehetnek meg és mit nem a szolgáltatással kapcsolatban. A szerződés magába foglalja a saját

eszközök elhelyezését a felhőszolgáltató adatközpontjában. Illetve megkapják a listát azon berendezésekről, amelyekhez karbantartás igénye esetén hozzáférést kaphatnak.

A felhőalapú számítástechnika egyik hátránya a szolgáltatók különbözőségéből adódik. Ez általában másik kiszolgálóhoz való átköltözéskor jelentkezik. Ha a költözést nem kezelik megfelelően, akkor az adatok sérülékenységnek vannak kitéve, amely adatvesztéshez vezethet. Egy jó felhőszolgáltató rendelkezik kellő szakértelemmel, hogy biztonságosan, adatvesztés és sérülés nélkül tudja a szervezet adatait átköltöztetni a két kiszolgáló között.

Az adatok felhőbe való átköltöztetése jelentős kommunikációs késéssel járhat, ezért egy biztonsági mentés készítése hosszabb ideig eltarthat, mint egy helyi hálózat esetében. Teljes biztonsági mentés esetén a későbbi mentések sokkal kevesebb időt fognak igénybe venni. Ha egy teljes szervert kell visszaállítani, akkor az általában több időt vesz igénybe, de ezen sebességbeli különbségek elhanyagolhatóak megfelelő szolgáltató választásával.

A felhő másik hátránya, hogy teljes mértékben a folyamatosan elérhető, stabil és nagy sávszélességű internetkapcsolatra támaszkodik. Ha megszakad a kapcsolat az adatközponttal, akkor nem lehet elérni az adatokat, de ez semmilyen adatvesztéssel vagy -sérüléssel nem fenyeget. A felhőalapú számítástechnika semmiben sem különbözik egy webalapú eszköztől, mert az adatok és a funkciók eléréséhez szükség van internetkapcsolatra. Ha a biztonsági mentéseket munkaidőben készítik (amikor az emberek intenzívebben használják az internetet) a nagy felhőhasználat növelheti a torlódás esélyét, és csökken az internet teljesítménye. Ez a fennakadás általában a kisvállalkozásokat érinti, mivel kevesebbet ruháznak be a sávszélesség és az internet sebességének növelésébe. Egy jó szolgáltató együttműködik az ügyfeleivel és segít abban, hogy ütemezésen és automatizáláson keresztül elkerülhessék ezeket a problémákat.

Nem lenne teljes ez a hátrányokat ismertető felsorolás, ha nem szerepelne benne a bizalmatlanság témaköre. A vállalatok vezetői a legtöbbször nem szeretnék a vállalat érzékeny (pénzügyi, működési stb.) adatait kiadni a kezei közül, ha publikus felhőszolgáltatást használna. Ilyenkor érdemes javasolni, hogy alkalmazható magán- vagy hibrid felhőszolgáltatás is: úgy, hogy az érzékeny adatokat megtartja magának a cég, azokat az adatokat, alkalmazásokat, amelyeket pedig hatékonyabban lehet használni és működtetni a publikus felhőben, akkor azt tegyék meg ott.

14.3. Felköltözés a felhőbe

A fenti előnyöket és hátrányokat tehát jól át kell gondolni, mielőtt arra gondolnánk, hogy felköltöztetjük (migráljuk) a vállalatunk rendszereit a felhőbe. A legnépszerűbb publikus felhőszolgáltatók általában lefedik a teljes palettát, ami egy ilyen vállalati, komplex rendszer együttes üzemeltetéséhez kellhet: virtuális gépek, tárhely szolgáltatások, dokumentumkezelők, adatbáziskezelő rendszerek, hálózatbiztonsági megoldások stb. A különbség a részletekben van, amelyek a számunkra esetleg létfontosságú, fontos vagy nem annyira fontos rendszerek és szolgáltatások árazásában lehet.

A 2020-as évek elején – a Covid-19 járványnak is *„köszönhetően"* – rettentően felpörgött a vállalati rendszerek felhőbe való felköltöztetése, ezáltal a digitális átalakulási folyamatok a legkülönbözőbb iparágakban kaptak lendületet, hiszen fontos volt, hogy az alkalmazottak a home office nyújtotta lehetőségekkel is hatékonyan tudjanak élni (tudják végezni a munkájukat otthonról is). Így a felköltözés

már nem csak egy lehetséges opció volt a vállalatok esetében, hanem gyakran az egyetlen működő lehetőség is.

A felköltöztetéskor, és majd utána is, nem feltétlenül szoftveres (programozókra) vagy hardveres (üzemeltetőkre) van szüksége a vállalatnak, hanem úgynevezett *DevOps (Development + Operations)* szakemberekre, amely munkakör főleg a felhők elterjedésének köszönheti létrejöttét.

Költözéskor az nagy előny tud lenni, ha már korábban használt a vállalat virtualizációs megoldásokat (pl.: VMWare eszközeit), mert ezeknek az integrálása sokkal zökkenőmentesebben tud működni, mint a *„nem virtualizált"* rendszereké. Ha virtuális rendszereink vannak, akkor a bonyodalmakat el tudjuk kerülni, mert ugyanúgy fognak működni a felhőben is, és élvezhetjük a skálázhatóságból adódó előnyöket is.

Tekintsük át, hogy milyen lépésekből állhat az a folyamat, amely a felhőbe való felköltözést jelenti:

1. Igényfelmérés, tervezés: a folyamat megkezdése előtt mindenképpen alakítsunk ki egy tiszta helyzetképet azzal kapcsolatban, hogy mennyire is állnak készen a rendszereink a migrációs folyamatra. Például, mennyire vannak virtualizálva a szervereink. Érdemes *„megnyerni"* a folyamat támogatásához a vállalat kulcsfontosságú szereplőit, alkalmazottait, mert az ő ellenállásukon sokat bukhat a cég, emiatt egy sikertelen átállási folyamat nemhogy előnyt, de inkább hátrányt jelenthet majd a jövőben. Ha meg szeretnénk úszni ezt a felmérési, tervezési folyamatot, akkor az a későbbiekben sokkal fájdalmasabb lehet, több időveszteséggel, ezáltal költséggel járhat a megvalósítás során, úgyhogy ne akarjuk kikerülni ezt a kulcsfontosságú lépést.
2. A mindenre kiterjedő felmérés lezárultával már el is lehet készíteni az üzemeltetési költségtervet. Erre a publikus felhőszolgáltatók saját felületeiken is lehetőséget biztosítanak. Sőt érdemes már most ismereteket szerezni arról, hogy hogyan lehet költséghatékonyan üzemeltetni a felhőszolgáltatásokat.
3. A saját rendszereink felmérése után próbáljuk meg eldönteni, hogy melyik publikus felhőszolgáltatáshoz passzolnak leginkább a rendszereink. Ehhez fel kell mérnie minden cégnek a saját prioritásait, elvárásait, mielőtt belekezdene a költöztetésbe.
4. A megcélzott felhőszolgáltató kiválasztása után következhet a költöztetés, amely azonos gyártójú rendszereknél minimális problémával járhat: például, ha az adatbáziskezelő rendszereket nézzük: egy Microsoft SQL Server-t viszonylag könnyű felköltöztetni az Azure felhőjébe, mert a rendszer alapja ugyanaz lokálisan és a felhőben is.
5. Ne akarjunk egyszerre mindent elvégezni! Válasszunk olyan részrendszereket, alkalmazásokat, amelyeknek a függőségei száma a többi rendszertől/alkalmazástól minimálisak. Ismerkedjünk a felhővel, próbáljuk ki, merjünk hibákat elkövetni azért, hogy utána már, amikor éles helyzet adódik, gyakorlottabbak legyünk a megoldásnál.
6. Azoknál a rendszereknél, amelyeknél nem ennyire könnyű a költözési lépés megtétele, ott szükség lehet alkalmazás és/vagy a rendszer újra szervezésére, tervezésére. Az alkalmazásokat nem kell üzleti logikai szinten újra elkészítenünk, sem megváltoztatnunk, viszont amire figyelnünk kell, azok az alkalmazás kapcsolódási felületei (interfaces). Ezek működését esetleg újra kell gondolnunk és tesztelnünk a helyes működés eléréshez. Rendszerek újratervezésénél inkább az

erőforráshasználatra, a skálázásra kell figyelnünk amiatt, hogy optimális működést érjünk el mind válaszidő, mind használati idő (és pénz) területén.

7. Opcionálisan igény lehet arra, hogy bizonyos régi alkalmazásoktól vagy rendszerektől megszabaduljunk a jövőre nézve. Ez akkor egy ideális pont lehet arra, hogy új, már az alapoktól kezdve felhős technológiákat nyújtó szolgáltatásokat használjunk fel a fejlesztéshez, építéshez.

Mindegyik lépésnek megvannak a buktatói, ezzel együtt az előnyei és hátrányai is, úgyhogy érdemes a felköltöztetési folyamathoz szakemberek segítségét igénybe venni, ezáltal elkerülhetünk olyan tipikus hibákat, amelyekbe esetleg saját magunk belefutnánk. Arra figyeljünk, hogy egy sikeres migrációs folyamattal nem fejeződik be a munka. Sőt! Onnan kezdődik igazán! Hiszen a céges rendszereket, alkalmazásokat működtetni kell a felhőben is, de már sokkal inkább tudunk a lényegi részekre koncentrálni, mintsem arra, hogy például elromlott-e a zár a szerverteremben, vagy megfelelő-e a hőmérséklet, vagy miért nem kapcsolódik az alkalmazás az adatbázisához. A lényeg ezekután már a hatékonyság lesz, ami a felhőbeli működtetés által egy jó nagy lökést tud jelenteni a vállalat folyamatainak.

14.3.1. Adatbázis kiszolgáló felköltöztetése az Azure felhőbe

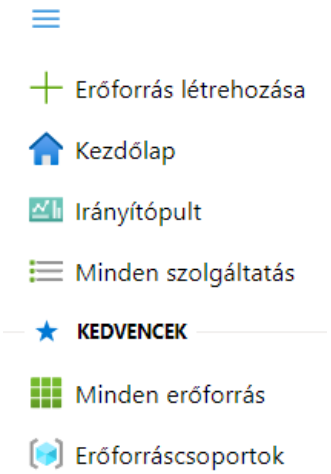
Mivel a Microsoft Azure felhője megfelelő a Laravel-es projektünk futtatásához a felhőben, ezért ennek segítségével mutatom be a felköltöztetési folyamatot. Ebben az alfejezetben a MySQL adatbázisszerverre és a webes alkalmazás adatbázisára koncentrálunk. Mindehhez szükségünk van egy Microsoft-os e-mail címre, Azure hozzáférésre, amelyet egy regisztráció után mindenki meg tud szerezni (https://portal.azure.com/). A regisztráció során szükség van a bankkártya adatainkra is, de megijedni ettől nem kell, mert számos népszerű szolgáltatás 12 hónapig ingyenes, 55 egyéb szolgáltatás mindig ingyenes marad, illetve kapunk $200 értékű (2024-ben ez nagyjából 180 EUR) kreditet, amit az első hónapban szabadon felhasználhatunk. Az általunk igénybe vett szolgáltatások 12 hónapig ingyenesek (MySQL flexibilis szerver), illetve a mindig ingyenes (App Service) kategóriába esnek majd.

Az Azure-t többféle módon is elérhetjük, akár weben keresztül, akár ún. CLI-n (parancssoros felületen) keresztül, de a későbbiekben majd SSH[30]-n keresztül is becsatlakozunk rá, Shell script-eket is írhatunk, de ami a Microsoft termékeinél mindig hangsúlyos, az a már sokak számára ismert, logikusan működő (webes) felhasználói felület. A webes felületet használhatjuk magyarul és angolul is, kinek melyik a kényelmesebb (én igyekszem mindig a megfelelő fordítást adni az egyes elemekhez).

14.3.1.1. Erőforráscsoport (Resource Group)

Először egy Erőforráscsoportra lesz szükségünk. Ebbe *„pakolhatjuk bele”* azokat a szolgáltatásokat, amiket majd használni szeretnénk. Az Erőforráscsoportot hozzuk létre weben először, így többféle módszerrel ismerkedünk meg használat közben. Az Erőforráscsoportokat nagyon könnyen elérhetjük, használhatjuk például a weboldal tetején középen lévő keresőt is, de kinyithatjuk a bal felső sarokban lévő navigációs menüt (*„hamburger menü ikont”*), ahol az első menüpontok között ott van az Erőforráscsoportok menüpont.

[30] Secure Shell



14–19. ábra: Menü részlet, benne Erőforráscsoportok menüponttal

A menüpont megnyitása után kapunk egy listát az Erőforráscsoportokról. Mivel még nem volt ilyenünk, ezért kattintsunk a *„Létrehozás" („Create")* gombra. Ki kell választani az *„Előfizetés" („Subscription")* lenyíló lista elemet, az Erőforráscsoport nevét, én a GA24_RG[31] (monogramomból, évszámból és a *„Resource Group"* elnevezésből adódó rövidítést, mozaikszót) nevet adtam neki, és a *„Régió" („Region")* lenyíló lista elemet, ahol helyileg minél közelebbit érdemes választani, hogy a későbbiekben a válaszidőkkel ne legyen gond, talán a (Europe) Germany West Central van a legközelebb hozzánk, így ez lehet a logikus választás. *Megjegyzés:* előfordulhat, hogy bizonyos régiókban éppen nincsen szabad erőforrás ilyen ingyenes szolgáltatás létrehozására, ilyenkor másik régiót kell választanunk. *„Felülvizsgálat + létrehozás"* gomb megnyomásával a rendszer ellenőrzi, hogy minden megfelelő-e, érvényes-e az előfizetésünk, van-e szabad kapacitás a régióban stb. Ha minden sikeres volt, akkor létre is hozza az Erőforráscsoportot, amely eseményről jobb felül a harang ikonnál (Értesítések menü) megjelenő értesítés is tájékoztat minket.

14–20. ábra: Felhasználói menüpontok az Azure portálon (1 értesítéssel és kiemelve bal oldalon az Azure Cloud Shell ikont)

14.3.1.2. Flexibilis MySQL Adatbáziskezelő szerver

Amire még szükségünk van kezdetben, az az Azure Cloud Shell. Ezt külön nem kell telepítenünk, hiszen a webes Azure portálon elérhető. Indítsuk el a futását az ikonra való kattintással. Egy inicializációs folyamat lefutása után megjelenik a prompt, ahova utasításokat tudunk beírni, és ki tudjuk adni őket.

14.3.1.2.1. Létrehozás

Adjuk ki a következő parancsot, de előtte olvassuk el az utána lévő leírást, mivel a parancsot érdemes a saját meglátásunk szerint paraméterezni (sárga hátterű paraméter érték):

[31] Sárga kiemeléssel jelölöm azokat az értékeket, amelyeket módosítani kell vagy érdemes, amikor a saját szolgáltatásait hozza létre, módosítja, állítja be az Olvasó.

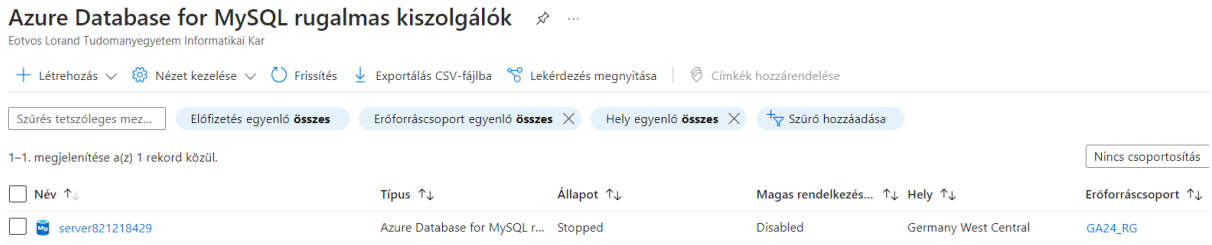
az mysql flexible-server create --resource-group GA24_RG

Ezekről a parancsokról elmondható, hogy az-vel kezdődnek, ez utal az Azure-ra, utána magát a szolgáltatást kell azonosítani, majd kapcsolók segítségével kell beállítani a szolgáltatásokat (a kapcsolók -- jelekkel kezdődnek), utánuk pedig ezen kapcsolók értékeinek beállítása következik mindig. Ezek az értékek az én saját rendszeremre illeszkednek, de mindenkinek a saját rendszerét kell beállítani, például az itt létrehozott szerver neve biztosan más *„azonosító számot”* kapott a nevébe, mint másoknál.

Mindig figyeljünk, és próbáljuk értelmezni azokat a válaszokat, amelyek JSON formátumban, kulcs-érték párként érkeznek vissza. Érdemes megnyitni egy szövegszerkesztőt is, és oda kimásolni, elmenteni a válaszok lényegi elmeit. Ezen válasz elemek többsége nem feltétlenül érthető, de például a **username, name, password** stb. értékeket érthetjük és elmenthetjük őket egy külön dokumentumba (csak addig tároljuk itt, amíg végre nem hajtjuk a fejezetben foglaltakat, utána a jelszót mindenképpen érdemes áthelyezni egy jelszókezelő alkalmazásba). Ha nem cselekednénk így, akkor sincsen probléma, csak utána mindig elő kell keresni a beállítási felületet, esetleg meg kell változtatni a jelszót is. Szóval hatékonyabb, ha mindent elmentünk a beállításainkról, és akkor egy helyen megtalálhatóak lesznek itt a kezdésnél.

A rendszer itt rögtön ellenőrzi, hogy a GA24_RG nevű Erőforráscsoport létezik-e, és érzékeli, hogy igen. Ezután megkérdezi, hogy szeretnék-e hozzáférést adni annak az eszköznek és IP címének, ahonnan éppen be vagyunk jelentkezve és menedzseljük az Azure-t. Erre természetesen igennel (**y**-nal) kell válaszolni. Megjegyzés: próbáltam úgy is, hogy már az utasításban hozzáadtam a --public-access paraméternek a korábban lekérdezett publikus IP címemet, de úgy nem működött, mert más IP címet érzékelt a Shell, úgyhogy így megoldottam azt a problémát, hogy engedtem neki, hogy rákérdezzen az IP címem engedélyezésére.

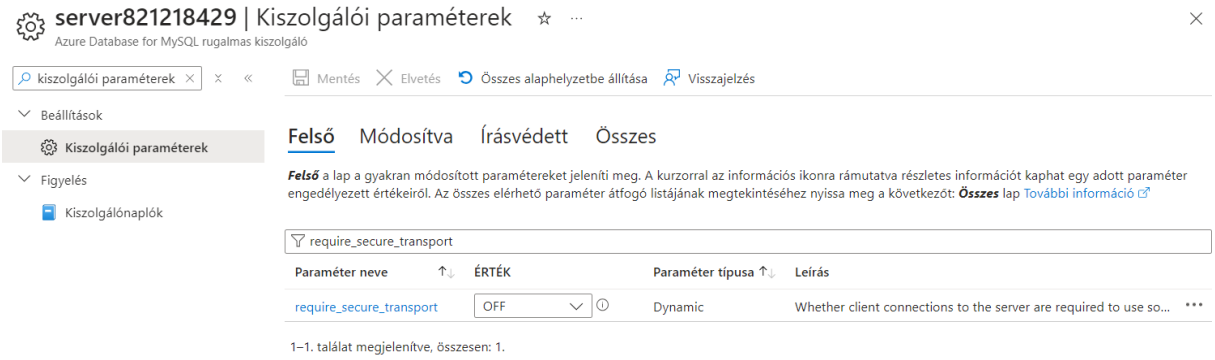
A parancs kiadása után, azért hosszabban tevékenykedett a rendszer, viszont végül sikeresen lefutott a flexibilis MySQL adatbáziskezelő szerver létrehozása. Ezt akár az Azure Portál segítségével is tudjuk ellenőrizni, ha rákeresünk az *„Azure Database for MySQL flexible servers”* (vagy magyarul a *„Azure Database for MySQL rugalmas kiszolgálók”)* szövegre fent, de ezen kívül a portál főoldalán is megjelenik már az erőforrások felsorolása között.



14–21. ábra: Létrehozott rugalmas MySQL adatbáziskezelő szerver szolgáltatás

Az Azure portál segítségével tudjuk menedzselni ezt a létrejövő szerverünket, de továbbra is fogunk kiadni terminal-beli utasításokat, és a portált többnyire a beállítások megváltozására vagy ellenőrzésre használjuk.

Válasszuk ki a szerverünket, majd a bal oldali menüben a *„Kiszolgálói paraméterek" („Server parameters")* menüpontot. Majd a jobb oldali keresőben a *"require_secure_transport"* paramétert **OFF**-ra kell állítani, és utána felül el kell menteni. Ha nem látjuk a listában a beállítás nevét, akkor használjuk a táblázat felett lévő keresőmezőt.



14–22. ábra: Flexibilis MySQL szerver kiszolgálói paraméterének módosítása

Amíg ez a beállítás be volt kapcsolva, addig a nem biztonságos átvitelt használó kapcsolatok tiltva voltak, így most már nem.

14.3.1.2.2. Tűzfal beállítások

Következhet a szerver tűzfalának beállítása, hogy majd engedje kapcsolódni a Laravel-es alkalmazásunkat hozzá.

```
az mysql flexible-server firewall-rule create --rule-name allanyAzureIPs --name server821218429 --resource-group GA24_RG --start-ip-address 0.0.0.0 --end-ip-address 0.0.0.0
```

Hosszúnak tűnik, de igazából mindössze egy soros az utasítás. Itt nevet adtunk a szabálynak, a `name` attribútumnak az előbb létrehozott szerverünket kell beírni, az Erőforráscsoport ugyanaz, mint korábban, a `start-ip` és `end-ip` pedig azt szabályozza, hogy engedünk mindenféle Azure szolgáltatást hozzáférni majd.

További paraméterek ezen leírás szerint állíthatók be: https://docs.microsoft.com/en-us/cli/azure/mysql/flexible-server/firewall-rule?view=azure-cli-latest

A saját IP címet is hozzá kellett adni a tűzfal szabályhoz így:

```
az mysql flexible-server firewall-rule create --resource-group GA24_RG --name server821218429 --rule-name allowip --start-ip-address 176.63.24.179
```

Látható ebből is, hogy a kapcsolók (paraméterek) sorrendje nem releváns. Könnyen lekérdezhetjük a saját IP címünket, ha valamelyik internetes keresőbe beírjuk ezeket: *„get my ip"* és rögtön adja is a választ.

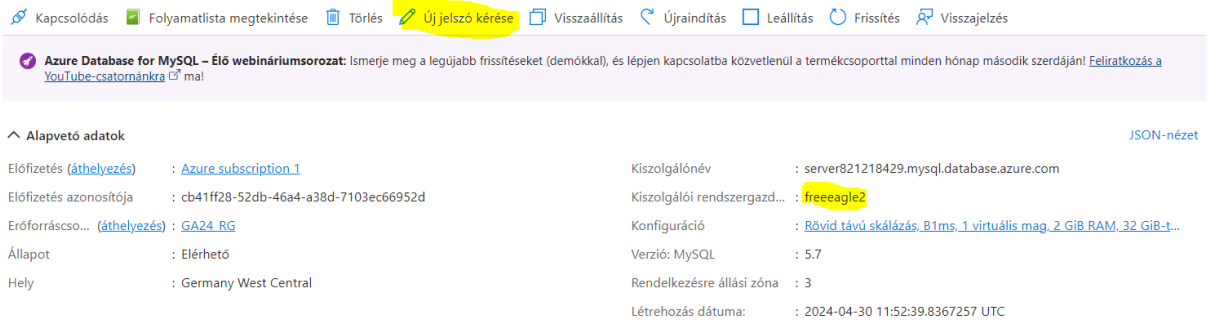
14.3.1.2.3. Csatlakozás az adatbáziskezelő szerverhez terminal-ból

Ha nincs a **mysql.exe** hozzáadva a környezeti változókhoz, akkor érdemes elnavigálni erre a helyre a parancssorban: **c:\xampp\mysql\bin** és kiadni a következő utasítást. Az `-u` után arra a felhasználónévre

lesz szükségünk, ami a flexibilis MySQL szerver létrehozásakor definiálásra került (és kimásoltuk a szövegszerkesztőbe), a -h után pedig a szerver neve van, így ezeket módosítani kell az Olvasónak az utasításban, de a többi maradhat.

```
mysql -u freeeagle2 -h server821218429.mysql.database.azure.com -P 3306 -p
```

Megjegyzés: ha nem tudnánk, mert nem mentettük ki a felhasználónevünket és jelszavunkat a parancsok lefutásának eredményéül kapott JSON-ökből, akkor használjuk a webes felületet a MySQL szerverbeállításait az Azure portálon. Itt látható alább a kép, amin a *„Kiszolgálói rendszergazda bejelentkezési neve”* látható, felül pedig van egy *„Új jelszó kérése” („Reset password”)*menüpont, amivel megváltoztathatjuk a jelszavunkat.

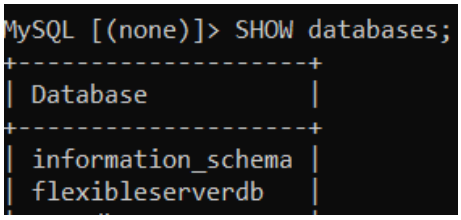


14–23. ábra: Flexibilis MySQL szerver kiszolgáló alapbeállításainak áttekintése

Ha sikeresen csatlakozunk, akkor egy SQL utasítással ellenőrizhetjük a MySQL Console segítségével, hogy ténylegesen működik-e a hozzáférésünk az Azure-on futó MySQL adatbáziskezelő szerveréhez:

```
SHOW databases;
```

Eredményül megkapjuk a szerverünkön lévő adatbázisokat, itt látható belőlük egy részlet:



14–24. ábra: Flexibilis MySQL adatbáziskezelő rendszerben lévő adatbázisok (részlet)

Hozzunk létre egy új adatbázist:

```
CREATE DATABASE l11_azure_db;
```

Ehhez az adatbázishoz hozzunk létre egy felhasználót (laravelappuser) és jelszavát (MySQLAzure2024), majd az utána következő utasításban rendeljük hozzá az adatbázishoz ezt az új felhasználót:

```
CREATE USER 'laravelappuser' IDENTIFIED BY 'MySQLAzure2024';
GRANT ALL PRIVILEGES ON l11_azure_db.* TO 'laravelappuser';
```

Ezután kiléphetünk a MySQL Console-ból egy quit utasítással.

14.3.1.2.4. Csatlakozás az adatbáziskezelő szerverhez Laravel alkalmazásból

Hozzunk létre egy új Laravel 11-es verziójú projektet:

```
laravel new l11-azure
```

Mivel ez alapértelmezetten SQLite adatbázishoz csatlakozik, ezért is majd meg kell változtatnunk az adatbázis kapcsolódás paramétereit, de tegyük mindezt egy új környezeti fájlban. Adjuk ki a terminal-ban a következő utasítást:

```
cp .env .env.production
```

Ezután hozzuk létre az **APP_KEY** attribútum értékét az új fájlban:

```
php artisan key:generate --env=production --force
```

A --force kapcsoló nélkül megkérdezi a rendszer, hogy biztosan szeretnénk-e a kiadott utasítást érvényre juttatni, válaszolhatunk igennel (*„yes”*).

Végezzük el a következő módosításokat az új **.env.production** fájlban:

```
APP_ENV=production
...

DB_CONNECTION=mysql
DB_HOST=server821218429.mysql.database.azure.com
DB_PORT=3306
DB_DATABASE=l11_azure_db
DB_USERNAME=laravelappuser
DB_PASSWORD=MySQLAzure2024
```

14–4. kódrészlet: Az .env.production környezeti fájl új beállításai

Hozzunk létre egy **ssl** nevű mappát a projekt mappánk gyökerében! Majd töltsük le ezt a fájlt, és mentsük el az **ssl** mappánkba: https://dl.cacerts.digicert.com/DigiCertGlobalRootCA.crt.pem

Nyissuk meg a **config / database.php** fájlt, és adjuk hozzá a **'mysql'** részhez két új kulcs-érték párost (**sslmode** és **options** kulcsokkal, a meglévő **options**-t írjuk felül ezzel az újjal) az alábbiak szerint:

```
'sslmode' => env('DB_SSLMODE', 'prefer'),
'options' => (env('MYSQL_SSL') && extension_loaded('pdo_mysql')) ? [
  PDO::MYSQL_ATTR_SSL_KEY    => '/ssl/DigiCertGlobalRootCA.crt.pem',
] : []
```

14–5. kódrészlet: A config / database.php connections tömb mysql szekciójának egy új és egy módosított beállítása

Ellenőrizhetjük az adatkapcsolatot a távoli kiszolgáló és a Laravel alkalmazásunk között:

```
php artisan db:show --env=production
```

Ha nem kapunk hibát, hanem a felhőbeli adatbázis kiszolgáló szerver paraméter értékeit kapjuk vissza, akkor a működés megfelelő.

```
PS C:\xampp\htdocs\l11-azure> php artisan db:show --env=production

  MySQL ..... 5.7.44-log
  Database ..... l11_azure_db
  Host ..... server821218429.mysql.database.azure.com
  Port ..... 3306
  Username ..... laravelappuser
  URL .....
  Open Connections ..... 6
  Tables ..... 0
```

14–25. ábra: Működő, élő felhőbeli adatkapcsolat

Migráljuk a felhős adatbázisba az adattábláinkat:

```
php artisan migrate --env=production --force
```

Töltsük is fel a **users** adattáblát egy tesztadattal (amihez kapcsolódó adatgyár szerepel a **database / seeders / DatabaseSeeder** osztály **run()** metódusában):

```
php artisan db:seed --env=production --force
```

Futtassuk az alkalmazásunkat helyben, de a felhőbeli adatbázist használva:

```
php artisan serve --env=production
```

Látszólag helyesen működik az alkalmazás, de a helyes adatlekérés bizonyosságáért regisztráljunk egy útvonalat a **routes / web.php** fájlba:

```php
Route::get('/users', function () {
  return User::all();
});
```

14–6. kódrészlet: Felhőbeli adatbázisban lévő felhasználók lekérése egy útvonal segítségével

A fájl elején ne felejtsük el importálni a **User** Model osztályt. Majd utána kérjük is le a böngészőben a http://127.0.0.1:8000/users linket, és ellenőrizzük, hogy visszaadja-e a Test User-t a lekérésünk egy JSON fájlban. Ha igen, akkor jó munkát végeztünk!

14.3.2. Webalkalmazás felköltöztetése az Azure felhőbe

Az adatbázis után magát a webalkalmazást is telepítsük, publikáljuk a felhőbe. A folyamat korántsem egyszerű! Időről-időre változnak is a körülmények, de igyekszem olyan tanácsokkal ellátni az olvasót, amelyekkel utána hatékonyan képes alkalmazkodni a változások követéséhez, így az esetlegesen felmerülő problémák megoldásához. A körülmények, amelyek változtak az elmúlt időszakban az Azure-ban a webes alkalmazások publikálásánál: PHP verziókövetés (a 8.2. a legfrissebb verzió és ezt is várja el a rendszer, amikor egy új alkalmazást publikálunk), korábban alapértelmezetten az Apache webszerver látta el a webalkalmazás kiszolgálását, most már az Nginx webszerver. Emellett adminisztrációs oldalról az Azure webes felülete is többször változtatásokon esett át, például egy-egy beállítás megadása átkerült egy másik menedzselési lapra. Ami viszont nem változott az elmúlt években, hogy a publikálásra kerülő webes alkalmazás (*„App Service”*) mindig ingyenes marad. Az ingyenességnek persze „ára” van, mivel, ha nincsen folyamatosan látogatva az alkalmazásunk címe, akkor az Azure elaltatja a virtualizált környezetet, amelyben fut, így, ha egy ideig nem látogatjuk meg, majd utána újra igen, akkor elég lassú a betöltése, esetleg időtúllépést is jelezhet a böngésző. Az örökké ingyenességnek az is a feltétele, hogy

például ne használjunk hozzá MySQL szervert, mert az 1 év után fizetőssé válik. Ez a Laravel 10 esetén alapértelmezetten problémát okozhatna, de láttuk az 5.1.2. alfejezetben, hogy milyen könnyű változtatni az adatbázis kiszolgálót MySQL-ről SQLite-ra. Továbbá a Laravel 11 esetén már amúgy is alapértelmezett az SQLite használata. Mi most először a már felhőben lévő MySQL adatbázishoz fogunk kapcsolódni a felköltöztetett Laravel projektünkből, aztán megnézzük, hogy milyen módon lehet majd megváltoztatni MySQL-ről SQLite-ra az adatbázis kiszolgálót már a felhőben.

14.3.2.1. Alkalmazás szolgáltatási terv („App Service Plan") létrehozása

Most több utasítást is végrehajtunk az Azure Shell segítségével, az előző alfejezetben láthattuk, hogy hogyan kell megnyitni az Azure Portal-on a Cloud Shell terminal-t. A korábban definiált szabályt most is betartom az útmutatás során, vagyis az, ami sárga háttérszínt kap, az módosítható vagy módosítani is kell a saját utasításokban, kódokban.

```
az appservice plan create --name myAppServicePlan --resource-group GA24_RG --sku F1 --is-linux
```

Az Alkalmazás szolgáltatási terv neve myAppServicePlan lett, de ez szabadon változtatható, arra viszont figyeljünk, hogy utána ugyanazt használjuk, amit itt definiáltunk. Én továbbra is a GA24_RG Erőforráscsoportomat használom, amibe elhelyezem a szolgáltatásokat. Az --sku az árazásra vonatkozik, az F1 pedig itt azt mutatja, hogy ingyenes lesz a használata. Az --is-linux kapcsolóval tudjuk Linux alapú konténerben működtetni majd az alkalmazásunkat. Később ez kisebb problémával fog járni, hogy nehezebben tudunk fejlesztői szolgáltatásokat igénybe venni, nem csak néhány kattintással, de az ingyenességnek *ez is az ára*. A Cloud Shell terminal továbbra is mindig vissza fog jelezni nekünk, remélhetőleg a pozitív eredményről adott JSON válasszal, vagy pedig valamilyen beszédes hibaüzenettel, amelyet majd meg kell oldanunk. Ha hibát kapunk, akkor további információ az *„App Service Plan"* létrehozásának mikéntjéről, parancshoz tartozó paramétereiről itt olvasható: https://docs.microsoft.com/en-us/cli/azure/appservice/plan#az_appservice_plan_create

14.3.2.2. Telepítési felhasználó („Deployment User") létrehozása

Itt majd egy helyi Git repo-t akarunk feltölteni az Azure App Service-be, ehhez egy telepítési felhasználót kell létrehoznunk az Azure Portal-on belüli Cloud Shell terminal-t használjuk:

```
az webapp deployment user set --user-name gludovatza --password ABCdef1234
```

Itt a --user-name és a --password kapcsolók utáni érték beállítások a saját egyedi értékeim ismét. Ha hibát kapunk (*„Conflict"*), akkor a Password kulcshoz nem lesz értékünk. Lehetséges hibakódok és okok:

- 409-es hibakódot is kaphatunk, ekkor próbáljuk meg módosítani a felhasználónevet a fenti parancsban.
- 400-as hibakódot is adhat a rendszer, ekkor próbáljunk meg erősebb jelszót megadni: legalább 8 karakteres és legyen benne kisbetű, nagybetű, szám is, mindegyikből legalább 2.
- Egyéb hibák esetén egyedi megoldások keresését javaslom.

További információ erről elérhető itt: https://learn.microsoft.com/en-us/azure/app-service/deploy-configure-credentials?tabs=cli

14.3.2.3. Webes alkalmazás („Web App") helyének létrehozása

Itt is az Azure Cloud Shell-t fogjuk használni: most hozzuk létre konkrétan a webes alkalmazásunkat az Azure-ban ezzel az utasítással:

```
az webapp create --resource-group GA24_RG --plan myAppServicePlan --name l11-azure --runtime "PHP|8.2" --deployment-local-git
```

A szokásos Erőforráscsoportot (GA24_RG) adtam meg itt is. A korábban létrehozott Alkalmazás szolgáltatási tervre (*„App Service Plan"*) itt lesz szükségünk: adjuk meg neki a myAppServicePlan nevet. A --name kapcsoló után következik a webes alkalmazásunk neve. Ennél nagyon fontos, hogy valamilyen egyedi, eddig senki más által nem használt név legyen, mivel ebből fog majd generálódni az alkalmazásunk URL címe. A későbbiekben ezzel a névvel azonosítjuk az alkalmazásunkat. A --runtime (futtatókörnyezet) a PHP 8.2-es verziója (a legfrissebb 2024. májusában), mert ezt használjuk lokálisan is, és tudjuk, hogy működik, illetve ezzel működik a Laravel 10-es és 11-es verziója is. Az utolsó kapcsoló a --deployment-local-git is fontos, mert majd Git-et fogunk használni a lokális projektünk felhőbe való telepítésére.

A parancs kiadása után itt még fontosabb, hogy figyeljünk a JSON válaszra, ami az Azure-től érkezik. Annak is konkrétan a **deploymentLocalGitUrl** kulcshoz tartozó értékét mentsük el egy szöveges fájlba. Ennek az értéknek a formája a következő minta szerint épül fel:

https://<username>@<app-name>.scm.azurewebsites.net/<app-name>.git

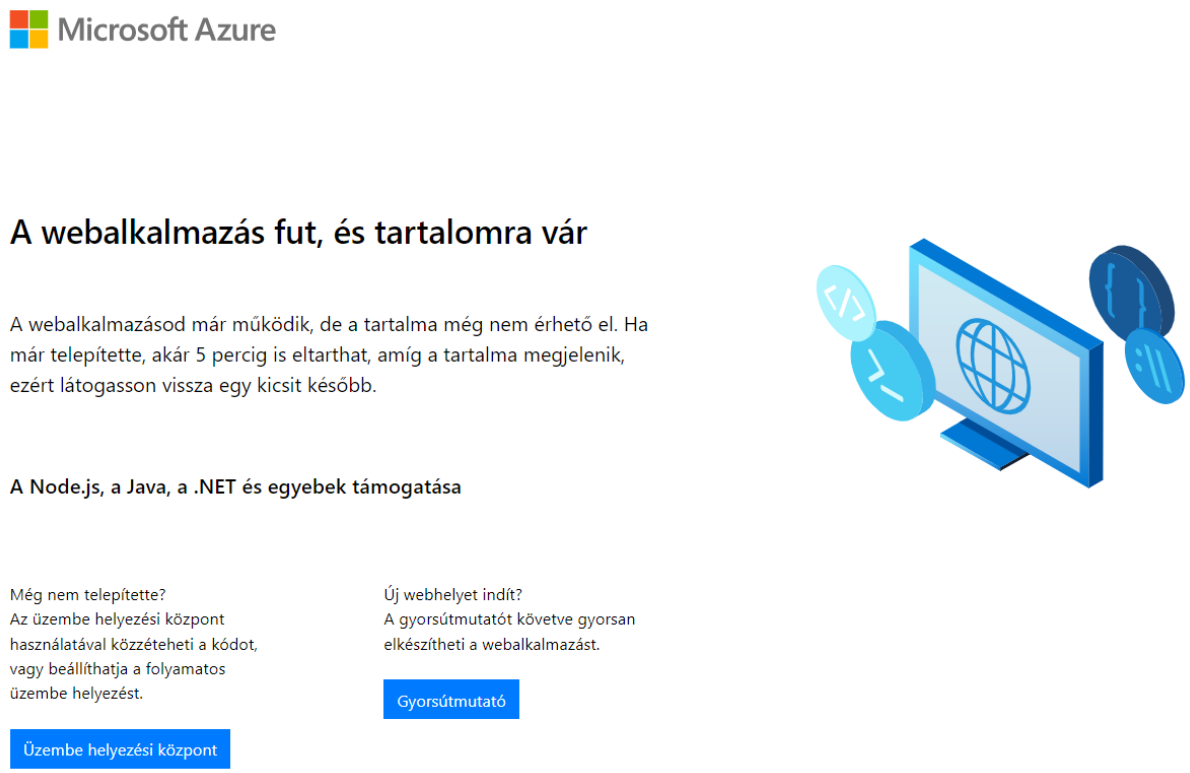
Sárgával kiemeltem a **<username>**-et, ami a korábban létrehozott telepítési felhasználónk lesz, az **<app-name>** pedig az itt létrehozott alkalmazás neve. Így nálam ez az érték így néz ki:

https://gludovatza@l11-azure.scm.azurewebsites.net/l11-azure.git

Az *„enabledHostNames"* szekcióban pedig további két fontos URL található meg:

1. l11-azure.azurewebsites.net – ezen a címen lesz elérhető a webalkalmazásunk, amikor már mindent beállítottunk. Egyelőre az látható itt, amit a 14–26. ábra mutat.
2. l11-azure.scm.azurewebsites.net – ezen a címen érünk el fejlesztő eszközöket az alkalmazáshoz (14–27. ábra).
    a. *Alternatíva:* ugyanez a link kibővítetten egy új felülethez (KuduLite), de nagyjából ugyanazokhoz a szolgáltatásokhoz vezet első látásra, de ez azért mégiscsak egy kicsit többet nyújt a számunkra, elég, ha csak a *„File manager"* részt megnézzük, és láthatjuk, hogy új fájlt tudunk feltölteni, a meglévő fájlokat tudjuk szerkeszteni, törölni stb. () Ezekután viszont rajtunk múlik, hogy melyik használatát tekintjük kényelmesebbnek: l11-azure.scm.azurewebsites.net/**newui**

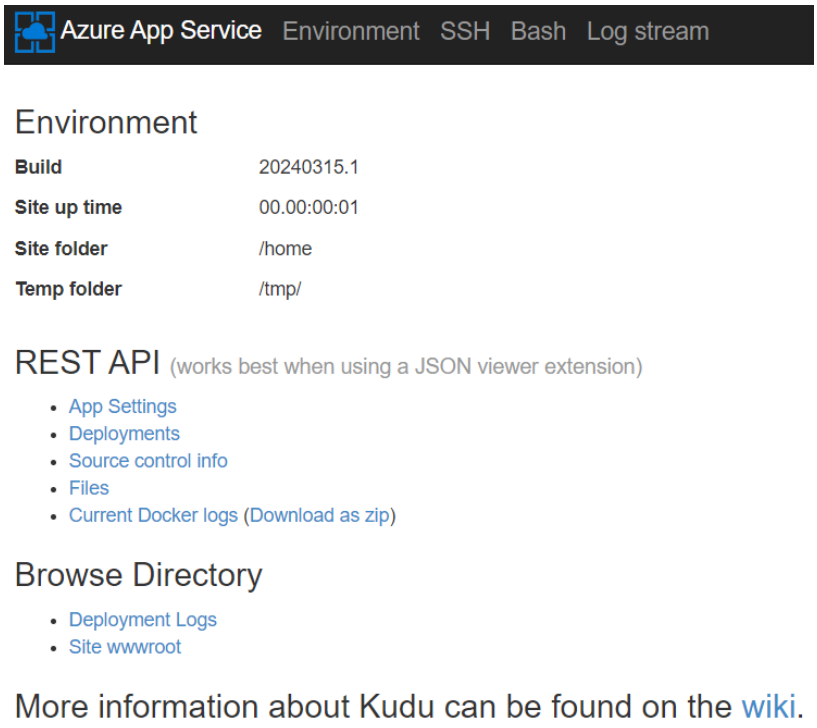
Először látogassuk meg az első linket! A kapott eredmény egy egyszerű HTML fájl, amely azt jelzi, hogy az alkalmazás fut és tartalomra vár.



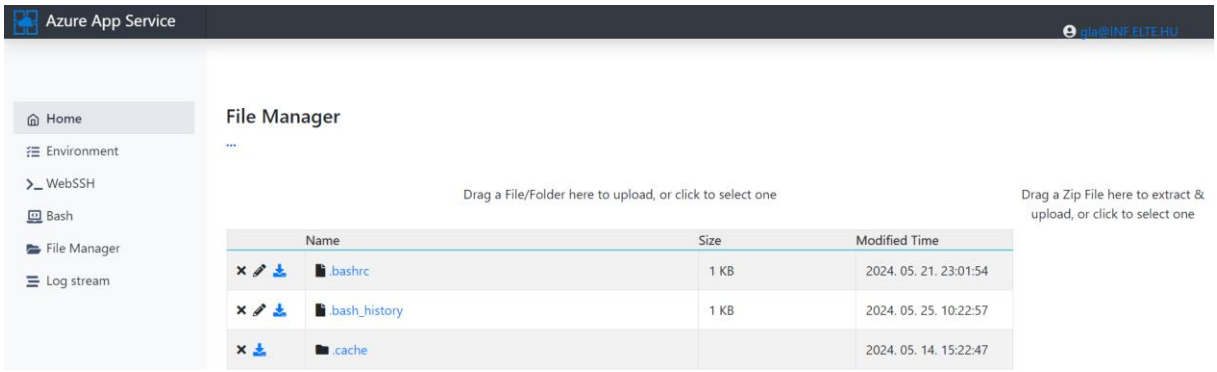
14–26. ábra: Felhőbeli azure-11 webalkalmazásunk kezdőlapja (még lényegi tartalom nélkül)

Az Azure Portal-ról való kényelmesebb elérés miatt, én maradok a *„régimódi"* kinézetnél és funkcióknál. A *„Fejlesztői eszközök"* Azure Portal menü szekcióban a *„Speciális eszközök"*-et tartalmazó oldalt is látogassuk meg és kattintsunk ott az *„Ugrás →"* vagy *„Go →"* linkre! Ekkor be kell jönnie a speciális fejlesztői eszközeinknek.

Ha itt (14–27. ábra) megtekintjük a *„Deployment Logs"* link utáni tartalmat, akkor láthatjuk, hogy még nem helyeztünk át semmit a felhőbe. A *„Site wwwroot"*-ban pedig mindössze egyetlen **hostingstart.html** fájl látható, aminek az eredményét, tartalmát mutatta a 14–26. ábra.

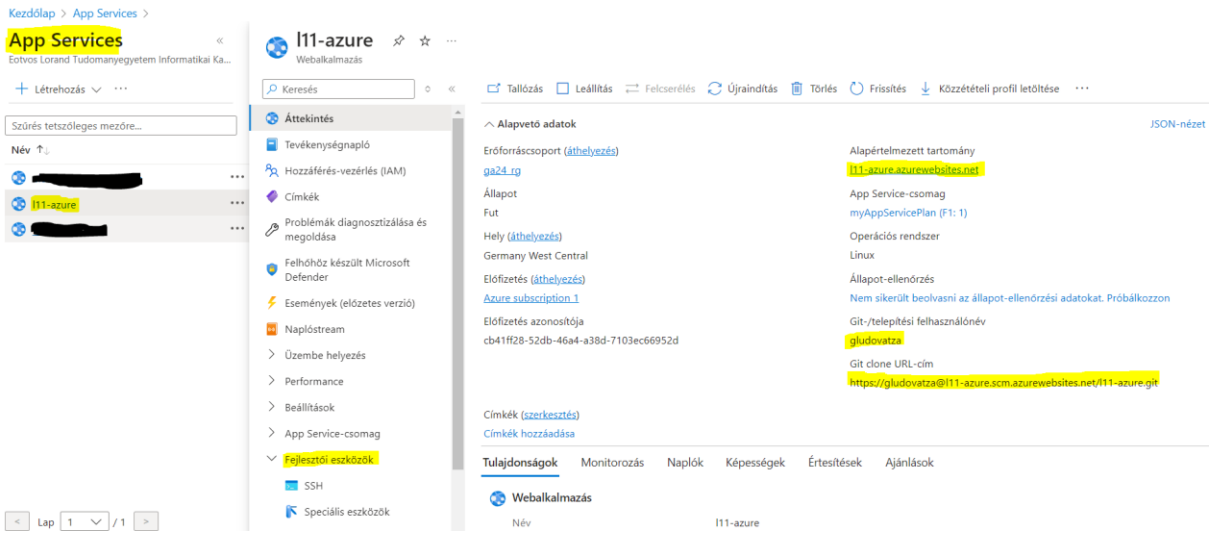


14–27. ábra: Felhőbeli Azure-11 webalkalmazásunk speciális fejlesztői eszközei



14–28. ábra: Felhőbeli webes alkalmazásunk speciális fejlesztői eszközei új felületen: „KuduLite" (File Manager részlet)

Az új webalkalmazásunkat már láthatjuk is az Azure Portal-on, ha az *„App Services"* szolgáltatásra rámegyünk, és a listában kiválasztjuk az újonnan létrejött **I11-azure** webalkalmazást.



14–29. ábra: Webalkalmazásunk felülete az Azure portálon

14.3.2.4. Adatbázis-elérés paramétereinek beállítása a webalkalmazásban

Az Azure-ban lévő webes alkalmazásnál is be kell állítanunk azokat a paramétereket, amelyeket a Laravel projektünkben az **.env** fájlban tettünk meg. Ehhez a következő utasítást adjuk meg, természetesen figyelve arra, hogy a korábban eltérő nevű paraméterek itt is térjenek el (sárga hátterű értékek):

```
az webapp config appsettings set --name l11-azure --resource-group GA24_RG --settings
DB_CONNECTION="mysql" DB_HOST="server821218429.mysql.database.azure.com"
DB_DATABASE="l11_azure_db" DB_USERNAME="laravelappuser"
DB_PASSWORD="MySQLAzure2024" MYSQL_SSL="true"
```

A fenti utasításban a webes alkalmazás nevét az imént állítottuk be, azt kell itt is alkalmazni, a többi beállítás adja magát. Figyeljünk arra, hogy a kulcs-érték páros érték oldalán (egyenlőségjeltől jobbra) idézőjelek közé kerüljenek az értékek. A sárga kiemeléssel jelzett részeket mindig le kell cserélni a saját alkalmazásunkban használtra, ezt sose feledjük!

A következő utasításhoz szükségünk van az alkalmazásunk **APP_KEY** értékére. Adjuk ki a VSCode terminal-jában, tehát a saját helyi projektünknél a következő utasítást:

```
php artisan key:generate --show
```

Ez egy *„base64”* szövegrésszel kezdődik, amit másoljunk ki a vágólapunkra, és az alábbi utasításba illesszük be a megfelelő helyre:

```
az webapp config appsettings set --name l11-azure --resource-group GA24_RG --settings
APP_KEY="imént-kimásolt-érték-beillesztése-ide" APP_DEBUG="true"
SCM_DISABLE_BUSTER_KUDU="true"
```

Az APP_KEY utáni idézőjelek közé kell beilleszteni az imént kimásolt értéket. Az APP_DEBUG attribútum true-ra állítása kezdetben segíthet nekünk, ha problémák adódnak, akkor emiatt a beállítás

miatt látni fogjuk a konkrét hibákat a böngészőben (illetve 404-es hibakódot kapunk, amely nem arra utal, hogy valamilyen útvonal vagy erőforrás nem létezik, hanem bármilyen más problémát is 404-es hibakóddal jelez nekünk a rendszer). Látni fogjuk tehát, hogy ez a felhőbeli éles (production) környezetben nem fog érvényesülni, nem fogjuk látni a hibákat, azokat majd a naplózó fájlból kell kinyernünk (**storage / logs / laravel.log** fájl végén tartalmazza mindig a legfrissebb hibákat). Ha majd hibamentesen fog menni az alkalmazásunk, akkor ezt a paramétert átállíthatjuk **false**-ra. Az SCM_DISABLE_BUSTER_KUDU attribútum amiatt kell, hogy fel tudjuk majd tölteni Git segítségével a projektünket a távoli, felhőbeli repository-ba.

14.3.2.5. Laravel projekt: Git repository, Azure repository

Kezdésként inicializáljuk a lokális Laravel projektünk Git repository-ját a VSCode-ban:

```
git init
```

Most van szükség a korábban elmentésre javasolt **deploymentLocalGitUrl** paraméterre:

```
git remote add azure https://gludovatza@l11-azure.scm.azurewebsites.net/l11-azure.git
```

Adjuk hozzá a könyvtárakat és fájlokat a *„Staging directory”*-hoz, majd a lokális repository-hoz:

```
git add .
```

```
git commit -m "First commit"
```

A Laravel projektünk függőségei közül csak a szerver oldaliak (**composer.json** -> **vendor**) települnek alapértelmezetten. A kliens oldali függőségek telepítéséről (**package.json** -> **node_modules**) magunknak kell majd gondoskodnunk, kezdetben manuálisan, aztán majd automatikusan is.

Most következhet a projekt feltöltése:

```
git push azure master
```

Ha nem tudnánk, hogy melyik branch-en (ágon) vagyunk, akkor a VSCode bal alul jelzi számunkra (**master** vagy **main**), de a `git status` paranccsal lekérhetjük konkrétan is, hogy melyik ágon vagyunk.

A *„push”* parancs kiadása után felugrik egy kis ablak, ami kéri tőlünk a telepítési felhasználónk nevét és jelszavát, amit 14.3.2.2. alfejezetben beállítottunk.

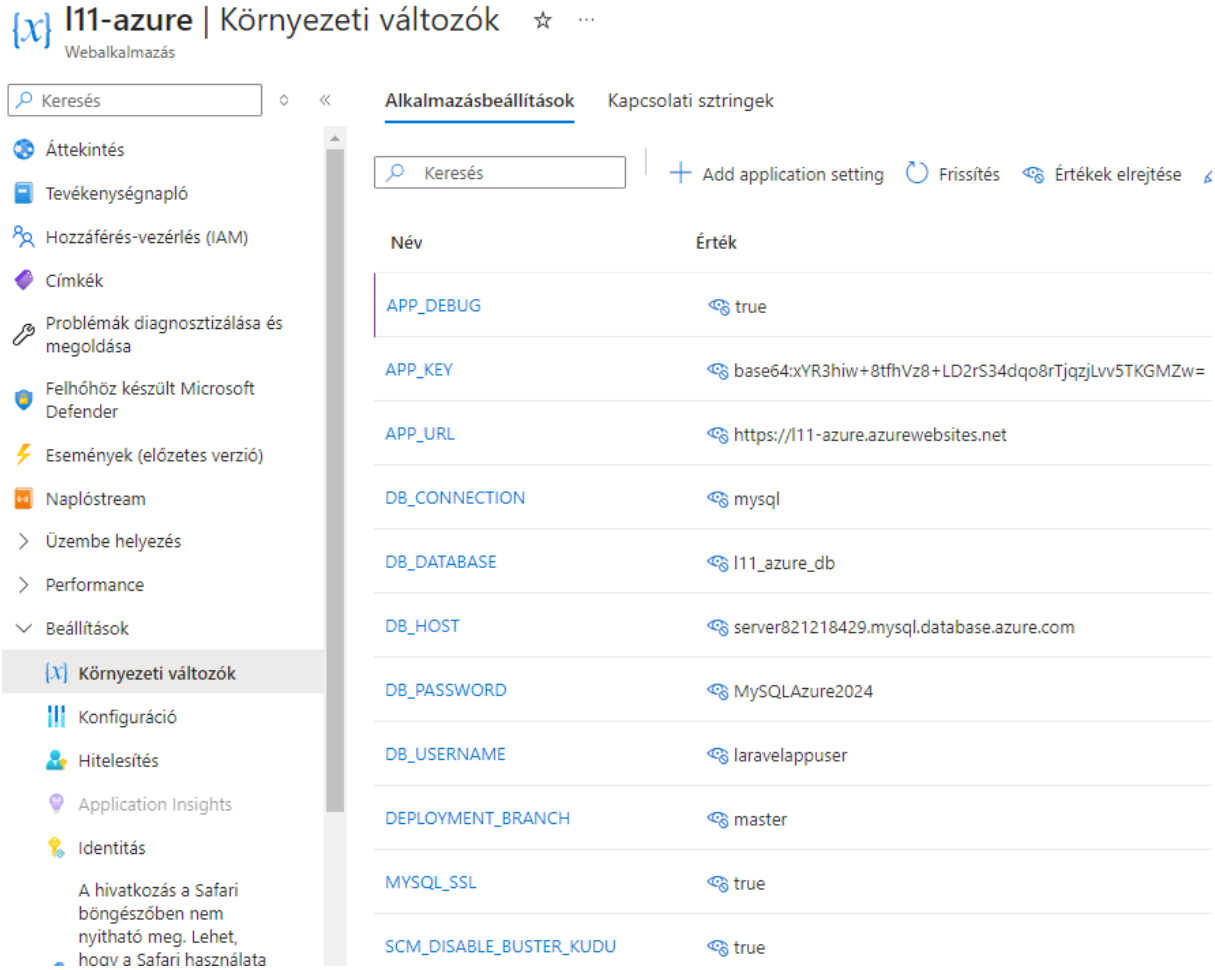
Megvárhatjuk a *„push”* folyamat végét, majd utána térjünk vissza az Azure Portal-on a Cloud Shell-hez, mert két környezeti változót még hozzá kell adnunk:

```
az webapp config appsettings set --name l11-azure --resource-group GA24_RG --settings
DEPLOYMENT_BRANCH="master" APP_URL="https://l11-azure.azurewebsites.net"
```

A DEPLOYMENT_BRANCH az az ág, amit az imént a `git status` parancs kiadásánál is láthattunk, előfordulhat, hogy ezt itt annak megfelelően **main**-re kell állítani. Az APP_URL-nek pedig az eleje az, ahova a webes alkalmazásunk nevét kell beírni. *Megjegyzés:* ezért is említettem korábban azt, hogy ennek

egyedinek kell lennie, más nem fogja tudni így hívni (**l11-azure**) az alkalmazását, amíg nálam elérhető egy ilyen nevű és elérhetőségű webes projekt.

Az új beállításokat akkor láthatjuk az Azure portálon, ha a bal oldali menüben lenyitjuk a *„Beállítások” („Settings”)* szekciót, és rámegyünk a *„Környezeti változók” („Environment variables”)* menüpontra. *Megjegyzés:* ezeknek az Azure Portal-on elérhető menüknek a neve és tartalma is valamikor változhat, én legalábbis ezt tapasztaltam az elmúlt években, de a főbb dolgokat meg kell találnunk az adott helyeken.



14–30. ábra: Webalkalmazás környezeti változói (részlet)

A beállítások értéke alapértelmezetten nem látszódik, csak ha rákattintunk a szövegre (*„Hidden value. Click to show value”*), ezért láthatóvá tenni őket, hogy ellenőrizni tudjuk a saját beállításainkat is.

14.3.2.6. Problémák és megoldásuk

Kezdetben a legfontosabb és legnehezebb, hogy valamilyen hibaüzenetet csikarjunk ki a rendszerből. Először egy statikus HTML oldalt (**hostingstart.html**) jelenít meg nekünk a rendszer, ha behozzuk a böngészőben az alkalmazásunk URL-jét (lásd: 14–26. ábra), amelyben azt jelzi, hogy az alkalmazás rendben fut, csak várnunk kell, amíg a helyes tartalmat nem mutatja. A várakozás pedig eltarthatna rendkívül sokáig, eredményt nem kapnánk további intézkedések nélkül. Ezért még több műveletet és beállítást végre kell hajtanunk a rendszerben, hogy működőképessé tegyük a webes alkalmazásunkat a felhőben.

14.3.2.6.1. Kliens oldali függőségek manuális telepítése (Speciális fejlesztői eszköz: Bash)

Ahogy azt korábban említettem, a kliens oldali csomag függőségeket manuálisan kell telepítenünk, mert a *„deployment"* folyamatot végző kudu script (a `git push azure master` parancs kiadása utáni folyamat a terminal-ban) nem hajtja végre automatikusan az `npm install` parancsot, emiatt pedig nem lesz működőképes az alkalmazásunk alapértelmezetten, ha valamilyen kliens oldali csomagot is igényelne a működése. *Megjegyzés:* egy friss, nyers Laravel projekt alapból nem igényelné a **node_modules** mappa elemeit, így anélkül is képes lenne futni. Azonban bármilyen kliens oldali programozást érintő érdemi dolgot csinálnánk (például a Tailwind CSS-t használnánk a Vite-tal, lásd 4.3. alfejezet), akkor már szükség lenne a **node_modules**-ban lévő csomagokra, előtte pedig az `npm install` parancs futtatására.

Menjünk az Azure portálon keresztül az alkalmazásunk **l11-azure** App Service-ére! Ott a bal oldali menüben keressük ki a *„Fejlesztői eszközök"* (*„Development Tools"*) csoportban a Speciális eszközöket (lásd: 14–27. ábra)!

Ezeket a speciális fejlesztői eszközöket az úgynevezett Kudu szolgáltatás nyújtja nekünk. Maga a Kudu[32] egy meghajtó motor, ami az App Service-ek számára, azok beállítására ad támogatást.

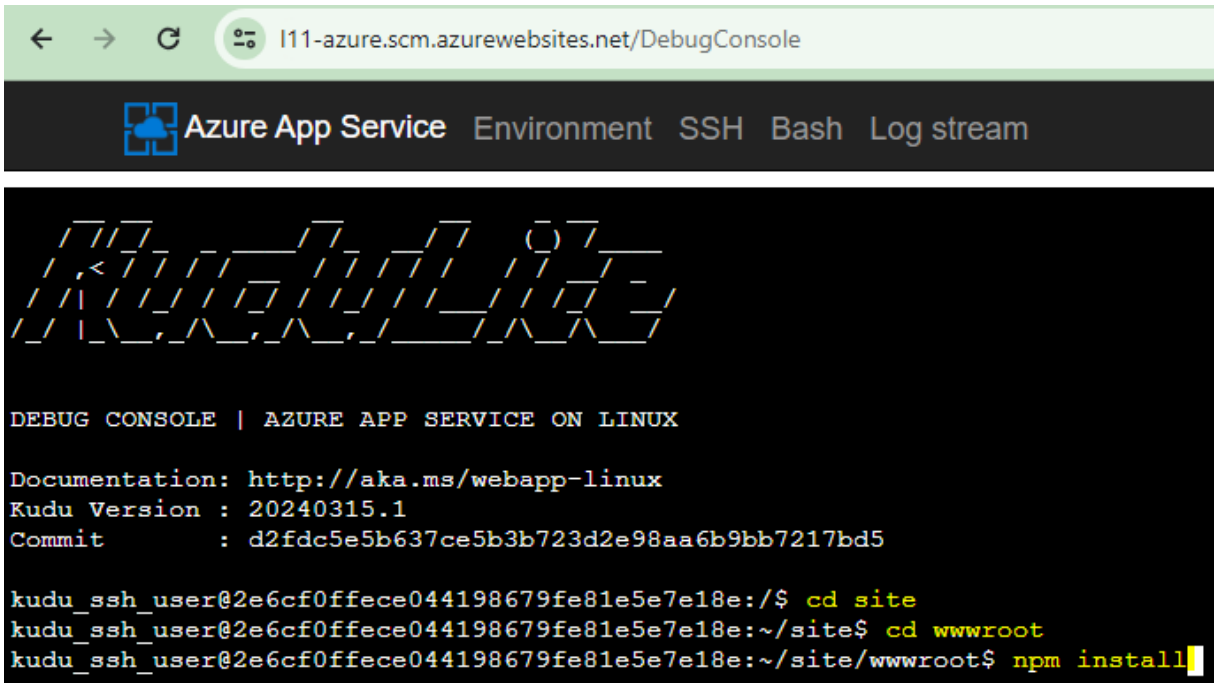
- Az *„Environment"* menüpont alatt elérhetőek a (konténerizált, virtuális) rendszer jellemzői, beállításai, környezeti változói és azok értékei. Érdekesség, hogy maga a rendszer egy Microsoft SQL Server-re csatlakoztatva fut (lásd a *„Connection string"*-et), ami nekünk persze nem okoz problémát.
- A Bash-t és az SSH-t pedig ebben az alfejezetben fogjuk is használni. Ezekkel parancsokat tudunk futtatni, illetve akár a mappákhoz, fájlokhoz, illetve azok tartalmához is hozzá tudunk férni és szerkeszteni tudjuk őket.
- A Log stream eszközzel a rendszer naplózását tudjuk nyomon követni, de alapértelmezetten ebből nem túl sok hasznos információ nyerhető ki, amely az alkalmazásunk működését, problémáit, vagy éppen a kitelepítési eljárást érintené.

Ha elértük a Kudu szolgáltatások felületét, akkor a fenti menüben válasszuk ki a *„Bash"*-t! Ott lépjünk be a projektünk mappájába, majd telepítsük manuálisan a kliens oldali függőségek csomagjait az alábbi parancsok kiadásával:

```
cd site

cd wwwroot

npm install
```

Mindez egy ábrán összefoglalva itt látható (kiemelve az utasításokat):

[32] Kudu projekt GitHub oldala: https://github.com/projectkudu/kudu



14–31. ábra: Kliens oldali függőségek csomagjainak telepítése a projekt mappájába

A futtatás eredményeként figyelmeztetéseket (warning) kaphatunk, de remélhetőleg végzetes hibát nem. Így a csomagok telepítésre kerülnek a projektünkbe a felhőben is.

Ez a manuális telepítés persze nem túl kényelmes, főleg, ha egy projektben többször változik a kliens oldali függőségi csomagösszetételünk, vagy csak időről-időre frissíteni szeretnénk a meglévőket, úgyhogy majd erre egy automatizálási lehetőséget érdemes megvizsgálni és kipróbálni a későbbiekben.

14.3.2.6.2. Felhőbeli Nginx webszerver beállítása

Linux virtuális konténert működtetve a felhőben, az Azure átállt a korábban használt Apache webszerverről Nginx-re. Ennek a beállításai között módosításokat kell végrehajtanunk azért, hogy ténylegesen működésre tudjuk bírni a webalkalmazásunkat.

A fejlesztési eszközök közül válasszuk most az SSH-t! Indulás után az Nginx beállításait másoljuk le egy **home** mappában lévő **default** fájlba így:

```
cp /etc/nginx/sites-enabled/default /home/default
```

Utána nyissuk meg szerkesztésre a lemásolt fájlt:

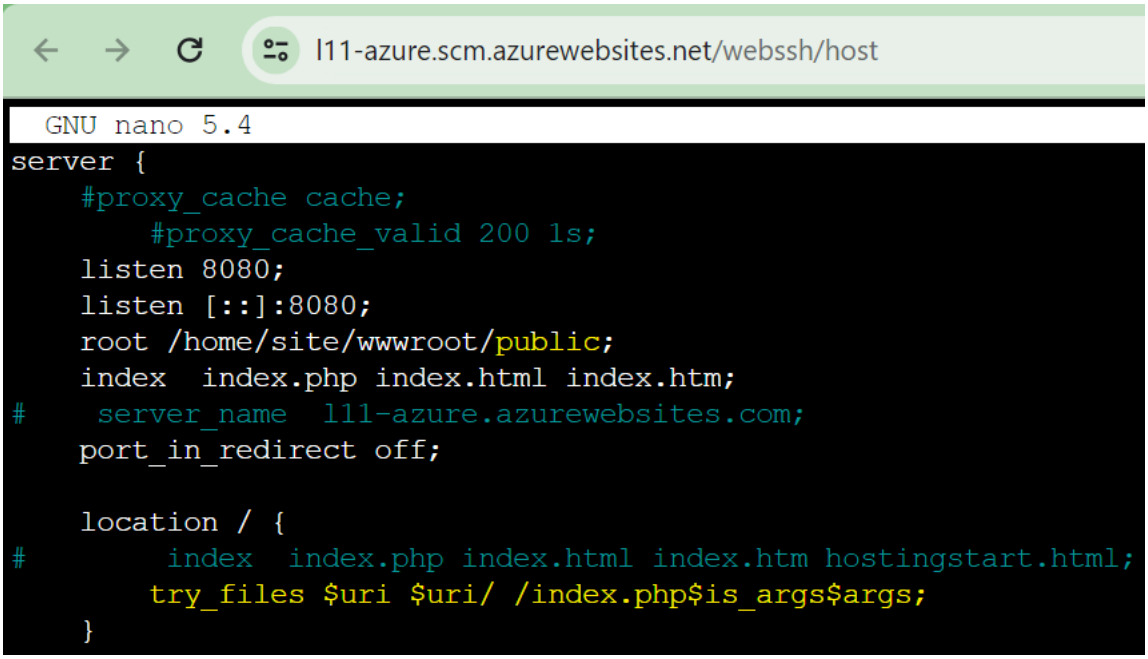
```
nano /home/default
```

Ezzel az utasítással megnyitjuk a **nano** nevű szövegszerkesztőt, és ezt a lemásolt fájlt szerkeszthetjük. A megnyitott fájlban két beállítást kell módosítanunk (lásd még az alábbi ábrát segítségül):

1. a **root** -os elérési út végére oda kell írnunk a **wwwroot** után a **/public** szöveget, hiszen a Laravel alkalmazás kiszolgálása a **/public** mappában lévő **index.php**-től indul el,
2. azért, hogy az útvonalválasztás (routing) is működjön az oldalunkon a **location / { ... }** részt is szerkesztenünk kell, adjuk hozzá ennek a magjához a következő sort:

try_files $uri $uri/ /index.php$is_args$args;

Az eredmény, amit látnunk kell, kiemelve a módosított részekkel (a *„server_name”*-et és a *„location”* magjának első sorát kikommentelhetjük), így néz ki:



14–32. ábra: Nginx beállítási fájl másolatának módosítása

Ctrl + x billentyűkombinációval ki tudunk lépni a nano szövegszerkesztőből, és ekkor megkérdezi alul, hogy mentsük-e a fájl módosításait, nyomjunk y-t, majd egy ENTER-t, hogy ugyanazzal a fájlnévvel mentse.

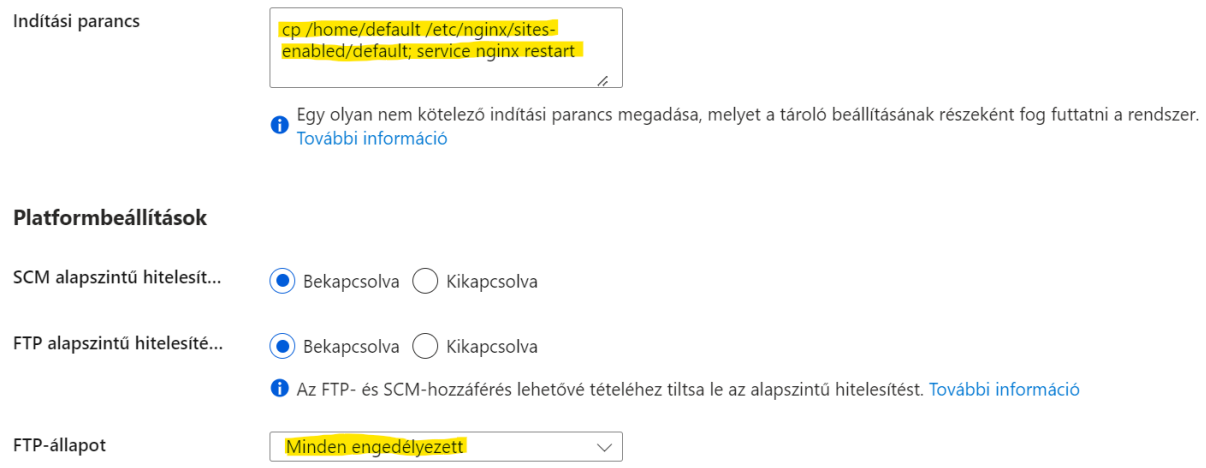
A lemásolt fájlunk így már megvan, amiben átírtuk az Nginx webszerver beállításait, azonban ennek még nem lenne hatása a mi alkalmazásunk futására. Ezért most beállítjuk azt, hogy az újonnan szerkesztett fájl mindig írja felül a webszerver indulásakor az Nginx eredeti beállítási fájlját.

Menjünk az Azure Portal-ra, vissza és az *„App Service”*-ünk menüjében keressük ki a *„Beállítások” („Settings”)* szekció *„Konfiguráció” („Configuration”)* menüpontot a beállítások közül, majd a jobb oldali tartalmi részen válasszuk az *„Általános beállítások” („General settings”)* lapfület, és ott az *„Indítási parancs” („Startup Command”)* szövegdoboz részbe írjuk be ezt az utasítást:

```
cp /home/default /etc/nginx/sites-enabled/default; service nginx restart
```

Ez az utasítás gyakorlatilag az imént szerkesztett fájlt visszamásolja az alkalmazás indulásakor mindig az Nginx beállításainak helyére, és indítja a webszervert az új beállításokkal. Ezután felül kattintsunk a *„Mentés” („Save”)* gombra, hogy érvényre jussanak a beállítások.

Megjegyzés: ha már ezen a lapon járunk, akkor az *„FTP-állapot”* lenyíló listában változtassuk meg a kiválasztott értéket erre: Minden engedélyezett. Ez később hasznos lesz, amikor esetleg hibát kapunk az alkalmazásunk megnyitásakor, és egy új naplóbejegyzés készül a felhőbeli projektünk **storage / logs / laravel.log** fájljában. Módosítás után ismét mentsünk!



14–33. ábra: Azure Webalkalmazás beállításai (Nginx webszerver, FTP)

14.3.2.6.3. Projekt megtekintése: csatlakozás az alkalmazáshoz FTP-n keresztül

Ahogy korábban említésre került, a 404 HTTP hibakód a felhőben valamilyen hibát jelez, de nem feltétlenül az erőforrás hiányát. Ennek kiderítéséhez fel kell csatlakoznunk a tárhelyre, és meg kell néznünk a **storage / logs / laravel.log** legutóbbi hibáját, majd ki kell javítani azt a hibát vagy a webes alkalmazásunkban, vagy valamelyik környezeti beállítás paraméternél az Azure-ban.

Az Azure-ban lévő webes alkalmazáshoz számos alkalmazással tudunk kapcsolódni FTP-n vagy SCP-n keresztül. Az Azure Portal-on az *„App Service”*, a projektünk felületén a bal oldali menüben válasszuk ki az *„Üzembe helyezés”* szekciót, azon belül pedig az *„Üzembe helyezési központ”* (*„Deployment Center”*) menüpontot! A tartalmi részen válasszuk ki a megjelenő vízszintes menü szekcióból a *„Helyi Git/FTPS hitelesítő adatok”* menüpontot!

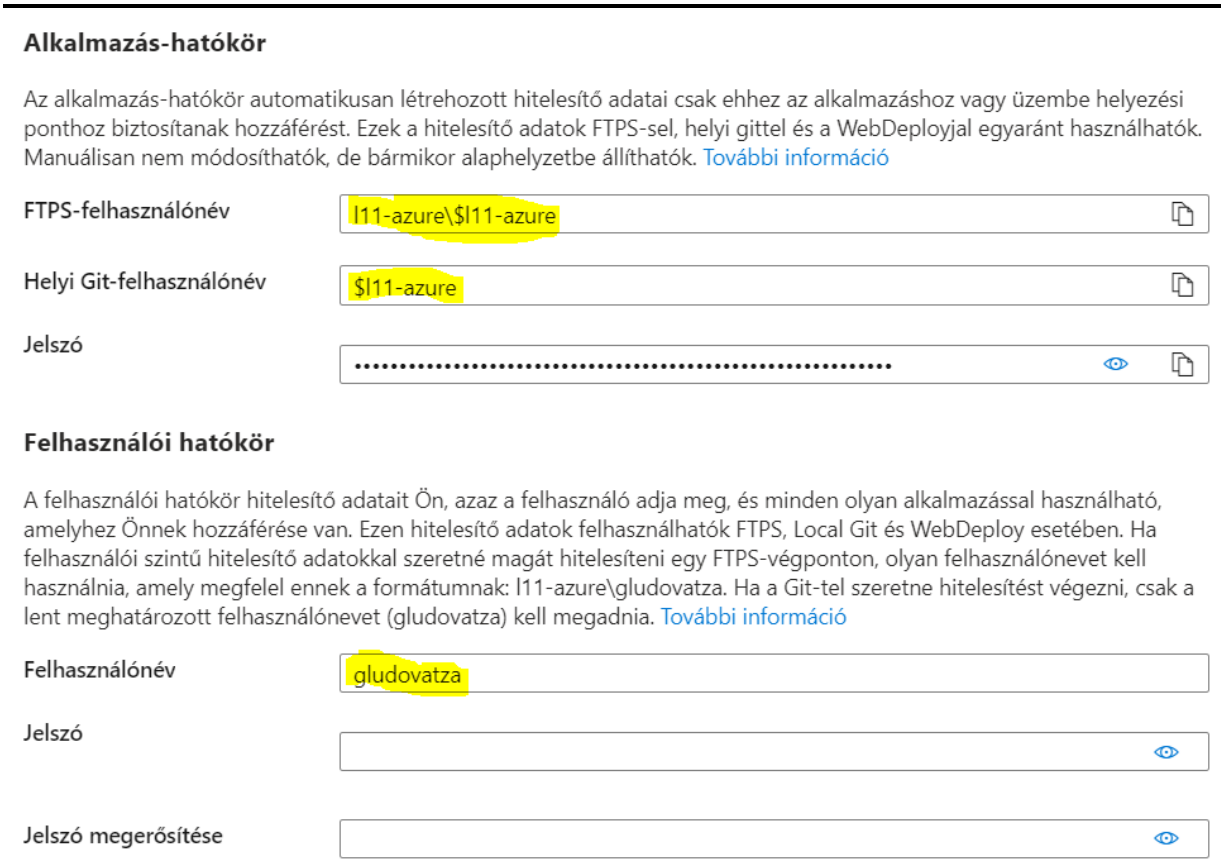
Az *„FTPS-végpont”* mezőben látható érték szükségeltetik az FTPS[33] kapcsolat létrehozásához majd. Nálam ez így néz ki:

ftps://waws-prod-fra-013.ftp.azurewebsites.windows.net/site/wwwroot

Ebből a protokoll rész (**ftps://**) nem kell. Utána a kiszolgáló gép (*„Host name”*) címének meghatározása szükségeltetik majd, illetve a távoli mappa (*„Remote Dir”*), ahova a webes projektünk tartalma (könyvtárai, fájljai kerülnek) a távoli szerver **site/wwwroot** mappájába kerülnek be.

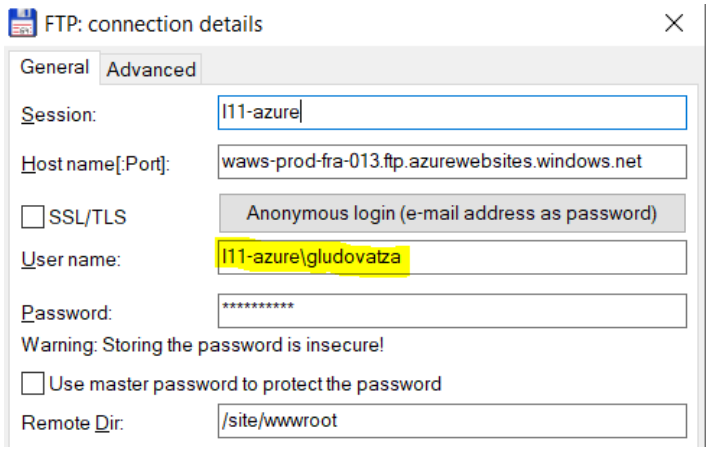
Az *„Alkalmazás-hatókör”* szekcióban látható az FTPS felhasználónevünk (a *$* jellel kezdődő *„név”* mindenkinek a saját felhasználónevét rejti, nálam ez gludovatza). A „Felhasználói hatókör” szekcióban látható *„Jelszó”* és *„Jelszó megerősítése”* szükségeltetik majd az FTP-s eléréshez. Adjunk meg ezekre a helyekre egy megfelelően erős jelszót és erősítsük meg!

[33] File Transfer Protocol Secure



14–34. ábra: Azure App Service Üzembe helyezési központ beállításai

Ezután következhet a becsatlakozás a webes alkalmazásunk tárhelyére például a Total Commander alkalmazással (a *„Session"* neve nem releváns, csak a kapcsolataink megkülönböztetéséhez szükséges):



14–35. ábra: Becsatlakozás a felhőbeli webes alkalmazásba FTP-n keresztül (Total Commander alkalmazással)

Egy példa probléma lehet az, hogy a Laravel 10 esetében nem kellett megadni a **DB_CONNECTION** környezeti változó értékét, mivel az alapértelmezetten **mysql** volt. Laravel 11 esetén viszont ez alapértelmezetten **sqlite**, így, ha Laravel 11-et szeretnénk feltelepíteni a felhőbe, akkor ezt a beállítást kötelező megadni (mi megadtuk), ha az alapértelmezett értéktől eltérőt szeretnénk használni. Alapvetően egy jó ökölszabály az, hogy minden környezeti változót állítsunk be saját értékre, ha az nem az alapértelmezett értéket veszi fel.

```
[2024-05-07 18:32:15] production.ERROR: Database file at path [l11_azure_db] does not exist. Ensure this is an absolute path to the database.
[stacktrace]
#0 /home/site/wwwroot/vendor/laravel/framework/src/Illuminate/Database/Connection.php(767): Illuminate\\Database\\Connection->runQueryCallbac
#1 /home/site/wwwroot/vendor/laravel/framework/src/Illuminate/Database/Connection.php(398): Illuminate\\Database\\Connection->run('select * f
#2 /home/site/wwwroot/vendor/laravel/framework/src/Illuminate/Database/Query/Builder.php(2993): Illuminate\\Database\\Connection->select('sel
#3 /home/site/wwwroot/vendor/laravel/framework/src/Illuminate/Database/Query/Builder.php(2978): Illuminate\\Database\\Query\\Builder->runSele
#4 /home/site/wwwroot/vendor/laravel/framework/src/Illuminate/Database/Query/Builder.php(3566): Illuminate\\Database\\Query\\Builder->Illumin
#5 /home/site/wwwroot/vendor/laravel/framework/src/Illuminate/Database/Query/Builder.php(2977): Illuminate\\Database\\Query\\Builder->onceWit
#6 /home/site/wwwroot/vendor/laravel/framework/src/Illuminate/Database/Concerns/BuildsQueries.php(335): Illuminate\\Database\\Query\\Builder-
#7 /home/site/wwwroot/vendor/laravel/framework/src/Illuminate/Database/Query/Builder.php(2900): Illuminate\\Database\\Query\\Builder->first(A
#8 /home/site/wwwroot/vendor/laravel/framework/src/Illuminate/Session/DatabaseSessionHandler.php(97): Illuminate\\Database\\Query\\Builder->f
#9 /home/site/wwwroot/vendor/laravel/framework/src/Illuminate/Session/Store.php(113): Illuminate\\Session\\DatabaseSessionHandler->read('MrEG
#10 /home/site/wwwroot/vendor/laravel/framework/src/Illuminate/Session/Store.php(101): Illuminate\\Session\\Store->readFromHandler()
#11 /home/site/wwwroot/vendor/laravel/framework/src/Illuminate/Session/Store.php(85): Illuminate\\Session\\Store->loadSession()
#12 /home/site/wwwroot/vendor/laravel/framework/src/Illuminate/Session/Middleware/StartSession.php(147): Illuminate\\Session\\Store->start()
#13 /home/site/wwwroot/vendor/laravel/framework/src/Illuminate/Support/helpers.php(363): Illuminate\\Session\\Middleware\\StartSession->Illum
#14 /home/site/wwwroot/vendor/laravel/framework/src/Illuminate/Session/Middleware/StartSession.php(144): tap(Object(Illuminate\\Session\\Stor
#15 /home/site/wwwroot/vendor/laravel/framework/src/Illuminate/Session/Middleware/StartSession.php(116): Illuminate\\Session\\Middleware\\Sta
#16 /home/site/wwwroot/vendor/laravel/framework/src/Illuminate/Session/Middleware/StartSession.php(64): Illuminate\\Session\\Middleware\\Star
#17 /home/site/wwwroot/vendor/laravel/framework/src/Illuminate/Pipeline/Pipeline.php(183): Illuminate\\Session\\Middleware\\StartSession->han
#18 /home/site/wwwroot/vendor/laravel/framework/src/Illuminate/Cookie/Middleware/AddQueuedCookiesToResponse.php(37): Illuminate\\Pipeline\\Pi
#19 /home/site/wwwroot/vendor/laravel/framework/src/Illuminate/Pipeline/Pipeline.php(183): Illuminate\\Cookie\\Middleware\\AddQueuedCookiesTo
#20 /home/site/wwwroot/vendor/laravel/framework/src/Illuminate/Cookie/Middleware/EncryptCookies.php(75): Illuminate\\Pipeline\\Pipeline->Illu
```

14–36. ábra: Naplózott adatbázis elérési hiba a felhőbeli alkalmazásnál

Az iménti hiba az FTP-s becsatlakozás után látható **storage / logs / laravel.log** szöveges fájl végén volt megtalálható. A rendszer kereste az **l11_azure_db** nevű SQLite fájlt a Laravel 11-es alapértelmezett beállításnak köszönhetően, de nem találta azt. Ehelyett a DB_CONNECTION="mysql" korábbi Azure Cloud Shell terminal-beli helyes parancs beállításnak köszönhetően a MySQL kiszolgáló szerverünk **l11_azure_db** nevű adatbázisát kell megtalálnia a megfelelő működéshez. Maga a naplófájl tartalmazza a hiba teljes kivétel továbbdobási listáját, ahogy a Laravel keretrendszer magja próbálta feldolgozni, de nekünk a hibából arra a sorra kell leginkább figyelni, amely egy időbélyeggel kezdődik, mert ez adja a leginkább érthető (és kereshető) információt a hiba okáról.

14.3.2.7. Folyamatos kihelyezés, publikálás, közzététel (Build & Continuous deployment)

A célunk az, hogy minden egyes programkód vagy függőség (szerver- vagy kliensoldali) megváltozásakor egy új kódbázist építsünk fel, és helyezzünk ki éles környezetbe, az Azure-ba. Ez a *„push”* folyamatnak köszönhetően meglehetősen jól működik alapértelmezetten, de a kliens oldali függőségek automatikus frissítéséhez még módosításokat kell végrehajtanunk a projektünkben.

Megjegyzés: ha azt tapasztaljuk, hogy nem indul el a *„build”*-elési folyamat a git push azure master parancs kiadása után (ezt például abból vehetjük észre, hogy csak másolásokat jelez a deployment naplózása a terminal-ban és nem telepíti a node.js-t, a php-t, a composer-t, majd nem települnek a szerver oldali függőségek csomagjai sem), akkor az Azure Portal-on az alkalmazásunk környezeti változóinak beállításainál adjuk hozzá ezt a kulcsot: **SCM_DO_BUILD_DURING_DEPLOYMENT**, **true** értékkel. További lehetséges problémákat és megoldásaikat ezen a linken lehet megtalálni.

Az Oryx[34] build rendszer végzi el a felépítési (build) és kihelyezési (deploy) folyamatokat. Ez lefordítja a forráskódokat és futtathatóvá alakítja őket. Az Azure App Service tipikusan ezt az eszközt használja a lokális Git repository-kból való építkezéskor és közzétételkor.

Az Oryx a kódbázis tartalmának elemzése alapján egy építési script-et generál és futtat egy konténeren belül. Például, ha a **composer.json** fájlt megvizsgálja a repository-ban, akkor utána aszerint futtatja az Oryx

[34] Oryx build rendszer GitHub oldala és dokumentációja: https://github.com/microsoft/Oryx/tree/main

a composer install (vagy update) parancsot és generálja le a **vendor** mappa tartalmát. Ugyanezt kellene megtennie a **package.json** fájl megvizsgálása után.

Az Oryx egy futásidejű indító script-et is generál az alkalmazáshoz, amely tipikus indítási parancsokat tartalmaz, például npm run start-ot Node.js esetén, vagy npm run dev-et Laravel esetén. A felépített függőségi csomag könyvtárak és az indító script egy minimalista futtatási konténerbe töltődnek és futtathatók is így már. A git push azure master parancs futtatása elején láthatjuk is ennek az Oryx build-nek az indulását, paramétereit.

```
PS C:\xampp\htdocs\l11-azure> git push azure master
Enumerating objects: 5, done.
Counting objects: 100% (5/5), done.
Delta compression using up to 12 threads
Compressing objects: 100% (3/3), done.
Writing objects: 100% (3/3), 294 bytes | 294.00 KiB/s, done.
Total 3 (delta 2), reused 0 (delta 0), pack-reused 0
remote: Deploy Async
remote: Updating branch 'master'.
remote: Updating submodules.
remote: Preparing deployment for commit id 'ad3d210007'.
remote: PreDeployment: context.CleanOutputPath False
remote: PreDeployment: context.OutputPath /home/site/wwwroot
remote: Repository path is /home/site/repository
remote: Running oryx build...
remote: Operation performed by Microsoft Oryx, https://github.com/Microsoft/Oryx
remote: You can report issues at https://github.com/Microsoft/Oryx/issues
remote:
remote: Oryx Version: 0.2.20230508.1, Commit: 7fe2bf39b357dd68572b438a85ca50b5ecfb4592, ReleaseTagName: 20230508.1
remote:
remote: Build Operation ID: 6aae10f51591d6be
remote: Repository Commit : ad3d2100077114f8153a36995a277d625dc83da0
remote: OS Type           : bullseye
remote: Image Type        : githubactions
```

14–37. ábra: Oryx build rendszer futásának kezdete a VSCode terminal-ban

Az Oryx build rendszer építési szolgáltatásának *„receptjére”* a deployment folyamat terminal-beli naplózásából következtethetünk, például a futtatókörnyezet elemeinek meghatározásakor és felhasználásakor:

```
Detecting platforms...
......
Detected following platforms:
  nodejs: 16.20.2
  php: 8.2.17
Version '16.20.2' of platform 'nodejs' is not installed. Generating script to install it...
Version '8.2.17' of platform 'php' is not installed. Generating script to install it...

Using intermediate directory '/tmp/8dc7a4f0d45750c'.

Copying files to the intermediate directory...
Done in 1 sec(s).

Source directory     : /tmp/8dc7a4f0d45750c
Destination directory: /home/site/wwwroot


Downloading and extracting 'nodejs' version '16.20.2' to '/tmp/oryx/platforms/nodejs/16.20.2'...
Detected image debian flavor: bullseye.
Downloaded in 2 sec(s).
Verifying checksum...
Extracting contents...
..
performing sha512 checksum for: nodejs...
Done in 10 sec(s).


Downloading and extracting 'php' version '8.2.17' to '/tmp/oryx/platforms/php/8.2.17'...
Detected image debian flavor: bullseye.
Downloaded in 1 sec(s).
Verifying checksum...
Extracting contents...
.
performing sha512 checksum for: php...
Done in 8 sec(s).


Downloading and extracting 'php-composer' version '2.0.8' to '/tmp/oryx/platforms/php-composer/2.0.8'...
Detected image debian flavor: bullseye.
Downloaded in 0 sec(s).
Verifying checksum...
Extracting contents...
performing sha512 checksum for: php-composer...
Done in 0 sec(s).
```

14–38. ábra: Futtatókörnyezet elemeinek azonosítása, beállítása, telepítése (később működtetése)

Ahogy azt korábban már megtapasztalhattuk, maga a kliens oldali csomag telepítési folyamat nem hajtódik végre ennek a folyamatnak a hatására, csak a Node.js-t érzékeli.

A build folyamatot testre tudjuk szabni, amikor SCM_DO_BUILD_DURING_DEPLOYMENT **true**-ra van állítva. Az App Service építési folyamatának automatikus lépései így vázolhatók fel:

1. Script-eket tudunk futtatni a *„build"*-elési folyamat *előtt* az Oryx beállításainál[35] megadható PRE_BUILD_COMMAND (mindössze egyetlen utasítást lehet megadni) vagy a PRE_BUILD_SCRIPT_PATH (egy konkrét fájlban bármennyi kódsornyi script-et lehet megadni).

[35] Oryx build rendszer beállítási kulcsok, leírások, alapértelmezett értékek és példák: https://github.com/microsoft/Oryx/blob/main/doc/configuration.md

2. Build folyamat magja: futtatókörnyezeti elemek használata a telepítéshez, például a composer install parancs futtatása.
3. Script-eket tudunk futtatni a *„build"*-elési folyamat *után* az Oryx beállításainál megadható POST_BUILD_COMMAND (mindössze egyetlen utasítást lehet megadni) vagy a POST_BUILD_SCRIPT_PATH (egy konkrét fájlban bármennyi kódsornyi script-et lehet megadni).

Az Azure Portal-on a webes App Service-ünk Környezeti változóihoz adjuk hozzá a következő új név-érték párost: a beállítás neve: POST_BUILD_COMMAND értéke: npm install

Alkalmazás (mentés) és újraindulás után futtathatjuk a VSCode-ban a git push azure master parancsot (tipikusan valamit meg kell változtatni ehhez, hogy el tudjuk indítani, például a gyökérben lévő README.md fájlt bővítsük azzal a folyamatleírással, amit most megcsináltunk, majd mehet a git add, commit és push).

Előfordulhat a *post-build* parancs futása során, hogy a kliens oldali csomagok igényelnék egy frissebb verziójú Node.js vagy npm használatát, erről az alábbi példa ad jelzést:

```
Executing post-build command...
...
npm WARN EBADENGINE Unsupported engine {
npm WARN EBADENGINE   package: 'laravel-vite-plugin@1.0.2',
npm WARN EBADENGINE   required: { node: '^18.0.0 || >=20.0.0' },
npm WARN EBADENGINE   current: { node: 'v16.20.2', npm: '8.19.4' }
npm WARN EBADENGINE }
npm WARN EBADENGINE Unsupported engine {
npm WARN EBADENGINE   package: 'rollup@4.17.2',
npm WARN EBADENGINE   required: { node: '>=18.0.0', npm: '>=8.0.0' },
npm WARN EBADENGINE   current: { node: 'v16.20.2', npm: '8.19.4' }
npm WARN EBADENGINE }
npm WARN EBADENGINE Unsupported engine {
npm WARN EBADENGINE   package: 'vite@5.2.11',
npm WARN EBADENGINE   required: { node: '^18.0.0 || >=20.0.0' },
npm WARN EBADENGINE   current: { node: 'v16.20.2', npm: '8.19.4' }
npm WARN EBADENGINE }
```

14–39. ábra: Túl alacsony a node verziója bizonyos függőségi csomagok telepítéséhez

Ennek megoldásához újra az Azure Portal-on lévő webes App Service-ünk Környezeti változóinál kell módosítást végeznünk: egy újabb beállítás értékét kell hozzáadnunk: név: NODE_VERSION értékét kell beállítanunk, például a nodejs.org weboldal alapján. De sajnálatos módon a nodejs verziói az Azure App Service esetén nem mindig egyeznek azzal, amit esetleg a nodejs.org, tehát a saját weboldala javasol használatra (20.13.1).

```
2024-05-22T11:22:40    Detecting platforms...
2024-05-22T11:22:44    Error: Platform 'nodejs' version '"20.13.1"' is unsupported. Supported versions: 12.19.0, 12.20.0, 12.21.0,
12.22.0, 12.22.11, 12.22.12, 12.22.4, 12.22.6, 12.22.9, 14.15.0, 14.15.1, 14.16.0, 14.17.0, 14.17.4, 14.17.6, 14.18.3, 14.19.1,
14.20.1, 14.21.2, 14.21.3, 16.13.1, 16.13.2, 16.14.0, 16.14.2, 16.18.0, 16.19.0, 16.20.0, 16.20.1, 16.20.2, 16.5.0, 16.6.1, 16.8.0,
18.0.0, 18.1.0, 18.12.0, 18.12.1, 18.14.0, 18.15.0, 18.16.0, 18.16.1, 18.17.1, 18.19.0, 18.19.1, 18.2.0, 20.11.0, 20.11.1, 20.9.0,
12.19.0, 12.20.0, 12.21.0, 12.22.0, 12.22.11, 12.22.12, 12.22.4, 12.22.6, 12.22.9, 14.15.0, 14.15.1, 14.16.0, 14.17.0, 14.17.4,
14.17.6, 14.18.3, 14.19.1, 14.20.1, 14.21.2, 14.21.3, 16.13.1, 16.13.2, 16.14.0, 16.14.2, 16.18.0, 16.19.0, 16.20.0, 16.20.1,
16.20.2, 16.5.0, 16.6.1, 16.8.0, 18.0.0, 18.1.0, 18.12.0, 18.12.1, 18.14.0, 18.15.0, 18.16.0, 18.16.1, 18.17.1, 18.19.0, 18.19.1,
18.2.0, 20.11.0, 20.11.1, 20.9.0
```

14–40. ábra: Nem támogatott nodejs verzió a build során, de a támogatottak felsorolásával

Ha kiválasztjuk a javasoltak közül a legmagasabb verziójút, akkor még arra is érdemes figyelni, hogy amikor a NODE_VERSION változónak értéket adunk az Azure Portal-on, akkor ne tegyük a verziószámot idézőjelek közé, hiába mutatja így az Oryx rendszer saját beállítása is. A 20.11.1 verzió kiválasztása után már megfelelően fog működni az npm csomagkezelővel a telepítési folyamat, amelyet a build *után* szekcióban határoztunk meg.

Ellenőrzésként láthatjuk a folyamat naplózásánál a következő részletet:

```
Executing post-build command...
.............

added 23 packages, and audited 24 packages in 14s

5 packages are looking for funding
  run `npm fund` for details

found 0 vulnerabilities
Finished executing post-build command.
```

14–41. ábra: POST_BUILD_COMMAND környezeti változóban megadott npm install parancs sikeres lefutása a naplózásban

Az iménti ábra arról tanúskodik, hogy sikeresen lefutott a *„post-build command”,* amit mi definiáltunk az Azure Portal webes App Service Környezeti változói között (**POST_BUILD_COMMAND**).

Érdekesség: megtehetjük azt is, hogy mi felügyeljük, határozzuk meg a teljes kihelyezési folyamatot, ezzel azonban vállalnunk kell azt, hogy időről-időre alkalmazkodnunk kell az Azure webes App Service környezetének vagy alapértelmezett beállításainak módosításához. Így nem csak szoftverfejlesztőként kell helyt állnunk, hanem az üzemeltetés körülményeivel is tisztában kell lennünk, úgynevezett *DevOps mérnökökké* is válhatunk.

A teljes kihelyezési folyamatot úgy tudjuk a Laravel projektek vezérelni, ha a projekt gyökerében elhelyezünk egy **.deployment** beállítási fájlt, amelyben környezeti változók definiálása mellett egy bash script-et is futtathatunk, amelyet egy külön állományban (például egy **deploy.sh** fájlban) helyezünk el. Ekkor a bash script olyan beállításokat tartalmazhat, amely például a Node.js elérhető verzióját lekéri, telepíti, futtatja az `npm install` utasítást, telepíti a PHP segítségével a Composer-t, telepíti vele a szerver oldali függőségeket és még számos tevékenységet végre tudunk így hajtani. Erről a Laravel 9 esetén készítettem egy blogbejegyzést, amely ezt a folyamatot mutatja be:

https://attila.gludovatz.hu/posts/adatbazis-hozzaferes-4-b-resz-felkoltozes-a-felhobe-laravel-9

A kódot (**deploy.sh**) azonban karban kell tartani, ugyanis például az Azure App Service használatakor 2024-ben már nem elérhető a PHP fordító a deployment során, így ez számos problémát tud nekünk okozni, amely nekem is jó sok munkaórámba került, mire kitapasztaltam a működését…

14.3.2.8. Felhőbeli adatbázis kiszolgáló váltása: MySQL-ről SQLite-ra

Ha azt szeretnénk, hogy a kihelyezett alkalmazásunk ne csak 1 évig legyen ingyenes az Azure felhőjében, akkor a MySQL kiszolgálóról válthatunk az SQLite-ra, ez fájl alapú, ahogy megtanulhattuk az 5.1.2. alfejezetben, nem kell hozzá egy külön szerver szolgáltatást működtetni, ami fizetős lenne, hanem megoldhatjuk ezáltal úgy is az Azure-ban történő működtetést, hogy mindig ingyenes maradjon a webalkalmazásunk futtatása.

Ha megvizsgáljuk az SQLite beállításait, akkor legelőször a **config / database.php** fájllal kell kezdenünk. Ebben a **'connections'** és **'sqlite'** szekcióban azt láthatjuk, hogy az adatbázis fájl neve **database.sqlite** alapértelmezetten és a database_path() metódussal érhető el, ami a projektünk gyökerének **database** mappájában fogja keresni ezt a fájlt. Ezeket az alapértelmezett beállításokat módosíthatjuk szabadon, de első körben nekünk ez megfelelő lesz.

Az **.env** fájlban a **DB_CONNECTION** értékét kell **sqlite**-ra állítani a helyes működéshez, a többi **DB_** prefixű beállítást pedig megjegyzésbe tehetjük. Az Azure Portal webes App Service-ében ugyanezt a beállítást kell módosítani **mysql**-ről **sqlite**-ra, a helyes működéshez. Tegyük is ezt meg! Azonban, ha létezik is a **database.sqlite** fájl a **database** mappában a helyi projektünkben (ha nem létezne, hozzuk létre manuálisan egy új fájl hozzáadásával), akkor azt is láthatjuk, hogy ez a **database** mappa tartalmaz egy **.gitignore** fájlt, amelyben minden **.sqlite** kiterjesztésű fájl feltöltése a verziókezelő rendszerekbe (például Azure, GitHub) le van tiltva így.

Több lehetőségünk is van ennek feloldására, de kettőt kiemelhetünk ezek közül:

1. Ha azt szeretnénk, hogy az aktuálisan, helyben használt sqlite fájl (és a benne lévő táblák, az adataikkal együtt) felkerüljön az éles környezetbe, akkor a 14.3.2.6.3. alfejezetben látható módon, például Total Commander alkalmazással felcsatlakozunk FTP protokollon keresztül a tárhelyre és az ott látható **/ home / site / wwwroot / database /** mappába felmásoljuk a **database.sqlite** fájlunkat. Ez persze csak egy egyszeri, aktuális képét fogja mutatni az éles környezetben a lokális környezetben létező adatbázisunknak. Ez egy egyszerű másolási folyamat, ezért inkább nézzük meg a második lehetőséget!
2. Dolgozzunk egy üres adatbázissal, töltsük fel adattal, és ellenőrizzük le, hogy bekerült-e oda az adat!
    a. Készüljünk elő az éles környezetben történő tesztelésre:
        i. A **database / seeders / DatabaseSeeder.php** fájlunkban vegyük ki a **run()** metódusban a kommentelést a *„Test User"* nevű felhasználó létrehozási utasításai elől.
        ii. A **routes / web.php** fájlunkban hozzunk létre egy új útvonalat, amivel lekérjük az adatbázisban lévő felhasználókat: ez egy egyszerű **get**-es **/users** útvonal lesz, ami visszatér a **User::all()** utasítással.
        iii. Hozzunk létre egy tesztesetet (`php artisan make:test UserTest`), ami ellenőrzi, hogy bekerült-e a tesztadat az adatbázisba, és onnan sikeresen le tudjuk kérni.

iv. *Megjegyzés:* ezeket a programkódokat a felsorolás után láthatjuk. Emellett a **Unit** és **Feature** mappákban lévő **ExampleTest.php** fájlok tesztelő metódusait kikommentezhetjük.
v. Utána következhet a *„build & deploy"* folyamat a git add, commit, push parancsok kiadásával.

b. Az Azure Portal-on állítsuk be megfelelően a környezeti változónkat (**DB_CONNECTION**) a többi **DB_**-vel kezdődőt pedig töröljük ki!
c. Hozzunk létre az éles környezet **database** mappájában egy üres **database.sqlite** fájlt!
d. Utána pedig ott az éles környezetben migráljuk az adattáblákat!
e. Töltsük fel tesztelési adattal!
f. Futtassuk a tesztet, amely ellenőrzi, hogy valóban létezik-e ott az sqlite adatbázisban lévő tesztadat.

```php
// database / seeders / DatabaseSeeder.php
public function run(): void
{
  // User::factory(10)->create();

  User::factory()->create([
    'name' => 'Test User',
    'email' => 'test@example.com',
  ]);
}

// routes / web.php
Route::get('/users', function () {
  return User::all();
});

// tests / Feature / UserTest.php
public function test_the_application_returns_a_successful_response(): void
{
  $response = $this->get('/');

  $response->assertStatus(200);
}
```

14–7. kódrészlet: Seed, felhasználókat lekérő útvonal, teszt az adatbázis eléréséhez és lekérdezéséhez

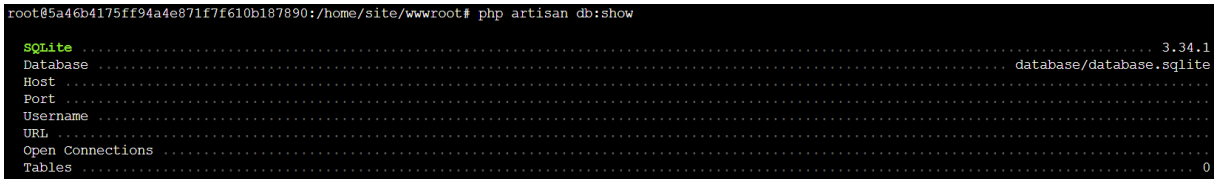
A 2. pontban leírt folyamat kódoláson és a *„build & deploy"* részen túli elemei az alábbiakban kerül ismertetésre lépésről-lépésre.

Lépjünk be az Azure Portal-on a Fejlesztői eszközök, Speciális eszközök részbe, majd az ugrás után az SSH menübe! Adjuk ki a következő utasításokat:

cd site/wwwroot

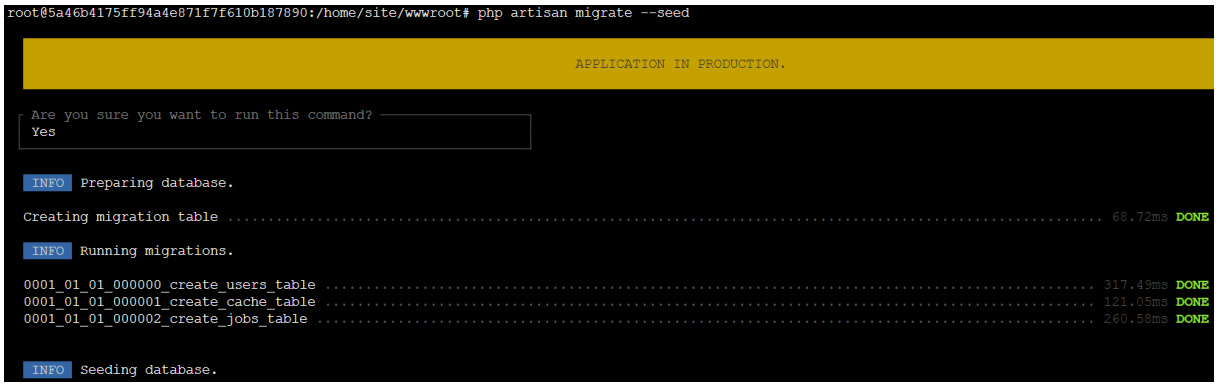
touch database/database.sqlite

php artisan db:show



14–42. ábra: SQLite adatkapcsolat sikeresen felépült a felhőben

php artisan migrate --seed



14–43. ábra: Migrálás és seed-elés sikeresen lefutott a felhőbeli SQLite adatbázisban

Bár az **.env** fájl az Azure App Service konténerében igazából felesleges, de ha futtatjuk a tesztelést, akkor figyelmeztetést ad, mivel azt érzékeli, hogy nem létezik ez a fájl, amire neki szüksége lenne alapesetben. Ezért hozzuk létre a fájlt:

cp .env.example .env

Utána már futtathatjuk a tesztesetünket:

php artisan test tests/Feature/UserTest.php

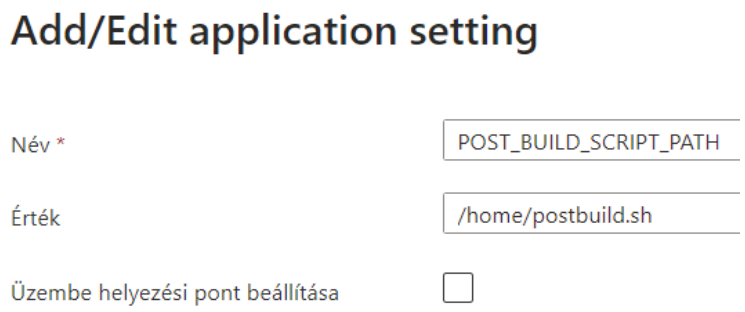
14–44. ábra: Teszteset sikeresen lefut a felhőbeli SQLite adatbázis lekérdezése után

Az adatbázist sikeresen áthelyeztük a felhőben MySQL helyett SQLite alapokra. Így, mivel az egyszerűen fájl alapú, továbbra is mindig ingyenesen használható az Azure-ban.

14.3.2.9. Folyamat elemeinek automatizálása: tesztelés, optimalizálás, közzététel

A *„build & deploy”* folyamatunk egész jól automatizálásra került, azonban jó lenne közzététel előtt az automatikus teszteket lefuttatni, ellenőrizni, hogy hibamentesen lefutnak-e, majd, ha igen, akkor optimalizálhatjuk is az alkalmazásunkat. A tesztelést és optimalizálást pedig a *„build & deploy”*

folyamatok közé kellene beilleszteni. Adódik a lehetőség, hogy a POST_BUILD_COMMAND környezeti változót használjuk megint az Azure Portal-on, de jelenleg ez már tartalmaz egy utasítást (npm install), amelyet nem lenne jó elhagyni. Lehetőségünk van egy másik környezeti változóval (POST_BUILD_SCRIPT_PATH) több utasítást is egybe foglalni, és szintén a build folyamat után, de még a deploy előtt lefuttatni. Adjuk hozzá az új környezeti változót:



14–45. ábra: Új környezeti változó hozzáadása az Azure webes App Service-hez

Ezzel egyúttal a POST_BUILD_COMMAND változó (kuka ikonnal) törölhető is a listából. Majd mentsük el a változók aktuális állapotát és ezután várjuk meg, amíg újraindul az alkalmazásunk.

Hozzuk létre azt a fájlt, amelyre hivatkoztunk az imént a helyi kódszerkesztőnkben, Speciális fejlesztői SSH eszközénél (alapértelmezetten a /home fog megnyílni):

```
touch postbuild.sh
```

Szerkesszük a fájlt:

```
nano postbuild.sh
```

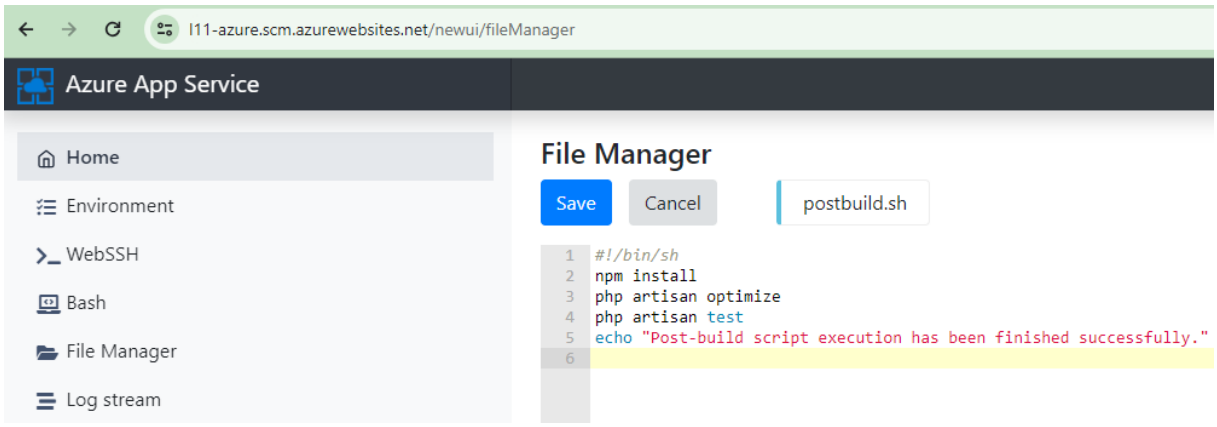
A fájl tartalma:

```
#!/bin/sh
npm install
php artisan optimize
php artisan test
echo "Post-build script execution has been finished successfully."
```

14–8. kódrészlet: Post-build script (kliens oldali függőségek telepítése, tesztelés, optimalizálás)

Mentsük el Ctrl + x megnyomása után y, majd egy Enter-rel lezárhatjuk a mentési folyamatot!

Nyissuk meg ezután a Speciális fejlesztői eszközöket tartalmazó alkalmazást a böngészőben az új kinézettel (nálam itt érhető el, de látható, hogy az alkalmazás nevét kell csak kicserélni a linkben: https://l11-azure.scm.azurewebsites.net/newui) és válasszuk a File Manager menüpontot! Utána ellenőrzésként szerkeszthetjük is ott a fájlt a ceruza ikonnal, és láthatjuk, hogy tényleg az van benne, amit mi szerettünk volna ott elhelyezni.



14–46. ábra: postbuild.sh fájl az Azure tárhelyén (/home mappában)

A **postbuild.sh** fájl tartalma egy shell script, amelyre utal az első kódsora. Utána telepítjük a kliens oldali függőségeket az npm install utasítással. Következik az optimalizálás a php artisan optimize utasítással. Végül a tesztelés a php artisan test utasítással. Az optimalizálást talán még nem ismerjük.

Ez igazából parancsok összességét jelenti, amelyekkel gyorsabbá tehető majd az alkalmazásunk működése. A sebesség javulást a gyorsítótárazással érjük el. Laravel 10 esetén ez a php artisan optimize parancs két utasítást rejtett magában:

1. php artisan route:cache
2. php artisan config:cache

Ez ugye az útvonalakat és az alkalmazás beállításait gyorsítótárazta. Azonban a Laravel 11 egy rejtett fejlesztése volt, hogy további két utasítás is bekerült abba a *„csomagba”,* ami akkor hajtódik végre, ha kiadjuk a php artisan optimize parancsot:

1. php artisan route:cache
2. php artisan config:cache
3. php artisan view:cache
4. php artisan event:cache

A csomag kiegészült a nézetek és az események gyorsítótárazásával. Ekkor persze figyeljünk arra, hogy ha ezután bármit megváltoztatunk, ami az útvonalakat, beállításokat, nézeteket, eseményeket érintené, akkor előtte hívjuk meg a **/ home / site / wwwroot** helyen a cache helyett a clear-re végződő utasításokat azért, hogy a lefordított és optimalizált kódok törlődjenek és ne azokat mutassa nekünk a böngésző, amikor az alkalmazáshoz akarunk hozzáférni.

Ha elvégeztük ezeket a változásokat, akkor következhet a *„build & deploy”* folyamat a VSCode-ban és kövessük nyomon az újonnan hozzáadott **POST_BUILD_SCRIPT_PATH** parancsfájl sorait érintő lefutások eredményeit.

```
 FAIL  Tests\Feature\UserTest
× user created successfully                                          1.63s

 FAILED  Tests\Feature\UserTest > user created successfully
Expected: <!DOCTYPE html>\n
<html lang="en">\n
    <head>\n
... (30 more lines)

To contain: Test User

at tests/Feature/UserTest.php:15
   11 |     function test_user_created_successfully() : void
   12 |     {
   13 |         $response = $this->get('/users');
   14 |
➜  15 |         $response->assertSee('Test User');
   16 |     }
   17 | }
   18 |
```

14–47. ábra: Elbukó teszteset a build és deploy között

Eredményül egy kis hibát kapunk, ugyanis a **UserTest** osztályban definiált tesztesetünk el fog bukni, mivel még nem tudja megnézni a tesztet, hogy a **/users** útvonal lekérésével látható-e az oldalon a *„Test User”*. Így érdemesebb erre egy olyan tesztesetet definiálni, ami a felhasználó **users** táblában való meglétét ellenőrzi.

```
function test_user_created_successfully(): void
{
  $this->assertDatabaseHas('users', [
    'name' => 'Test User',
  ]);
}
```

14–9. kódrészlet: Tesztfelhelhasználó meglétének ellenőrzése a users adattáblában

Egy további (korábban már létező) környezeti változót is hozzá kell még adnunk a megfelelő működéshez: **DB_DATABASE** alapértelmezett értéke **/database/database.sqlite**, de ezt a fájlt a rendszer a *„build & deploy”* során még a **tmp** ideiglenes mappájában keresi, úgyhogy a relatív útvonal helyett abszolút útvonalat adjunk meg a környezeti változónak értékül: **/home/site/wwwroot/database/database.sqlite**

Ha elvégeztük ezt a változást, akkor következhet a *„build & deploy”* folyamat a VSCode-ban és most már koncentrálhatunk csak a teszteset lefutásának helyességére, mivel a többi kódrészlet megfelelően lefutott. A folyamatnak most már hibátlanul le kell futnia:

```
Executing post-build command...
.........

added 23 packages, and audited 24 packages in 10s

5 packages are looking for funding
  run `npm fund` for details

found 0 vulnerabilities

   INFO  Caching framework bootstrap, configuration, and metadata.

  config ..................................................... 85.17ms DONE
  events ..................................................... 9.93ms DONE
  routes ..................................................... 79.24ms DONE
  views ..................................................... 107.38ms DONE


   PASS  Tests\Feature\UserTest
  ✓ user created successfully                                      0.31s

  Tests:    1 passed (1 assertions)
  Duration: 0.53s

Post-build script execution has been finished successfully.
```

14–48. ábra: Kliens oldali csomagok sikeresen települtek, optimalizálás szintén sikeres és a teszteset lefutása is

Megjegyzés: ha netán azt tapasztaljuk a frissen végrehajtott *„build & deploy"* folyamat után, hogy az alkalmazásunk 404-es HTTP hibakódot mutat a böngészőben, akkor lépjünk be az SSH eszközzel a site/wwwroot mappába és adjuk ki az utasítást: php artisan config:clear (további információ erről a Laravel dokumentációjában olvasható itt a felkiáltójeles figyelmeztetés részben: https://laravel.com/docs/11.x/configuration#configuration-caching)

14.3.3. Folyamatos fejlesztés, integráció, leszállítás, közzététel

Ha az alkalmazásunkat folyamatosan fejleszteni szeretnénk, de emellett folyamatosan működő megoldást kell nyújtani a felhasználóinknak, akkor a CI/CD[36] folyamat követése lehet a segítségünkre.

A CI és CD a folyamatos integráció és a folyamatos szállítás/folyamatos telepítés szavak rövidítése. Nagyon leegyszerűsítve, a CI egy modern szoftverfejlesztési gyakorlat, amelyben a kódot gyakran és megbízhatóan inkrementálisan módosítják. A CI által kiváltott automatizált építési (build) és tesztelési lépések biztosítják, hogy a tárolóba (repository) integrált kódváltozások megbízhatóak legyenek. A kód ezután gyorsan és zökkenőmentesen kerül átadásra a CD-folyamat részeként. A szoftverek világában a CI/CD csővezeték az automatizálást jelenti, amely lehetővé teszi, hogy a fejlesztők asztaláról a növekményes kódmódosítások gyorsan és megbízhatóan eljussanak az éles rendszer (production) környezetbe.

A CI/CD lehetővé teszi a szervezetek számára a szoftverek gyors és hatékony leszállítását, publikálását. A CI/CD hatékony folyamatot tesz lehetővé a szoftvertermékek minden eddiginél gyorsabb piacra

[36] Continuous Integration and Continuous Delivery/Continuous Deployment

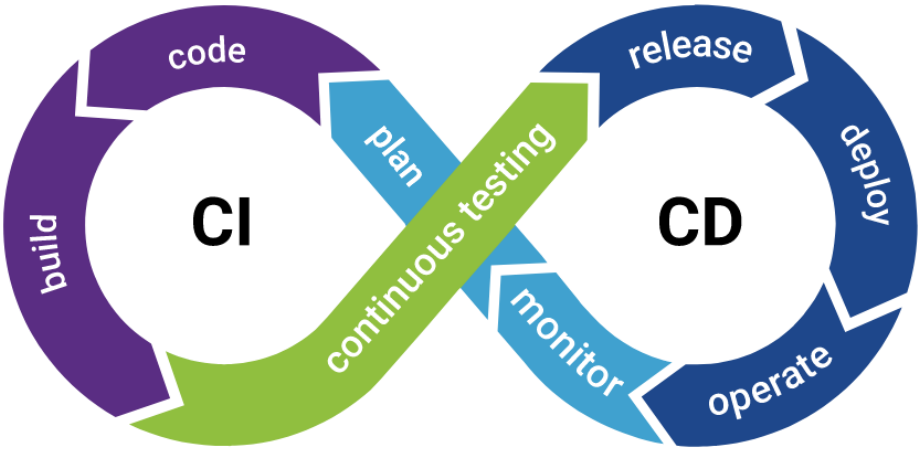
juttatásához, a kód folyamatos szállítására az éles rendszerbe, valamint az új funkciók és hibajavítások folyamatos áramlására.

A folyamatos integráció (CI) egy olyan gyakorlat, amely során a fejlesztők apró változtatásokat és ellenőrzéseket végeznek a kódjukon. A követelmények nagyságrendje és a lépések száma miatt ezt a folyamatot automatizálják, hogy a csapatok megbízható és megismételhető módon tudják építeni és tesztelni alkalmazásaikat. A CI segít a kódmódosítások racionalizálásában, ezáltal növelve a fejlesztők számára a változtatásokra rendelkezésre álló időt, így jobb szoftvert kapnak a felhasználók.

A folyamatos szállítás (Continuous Delivery – CD) az elkészült kód automatikus szállítása olyan környezetekbe (environment), mint a tesztelés (test) és a fejlesztés (dev). A CD automatizált és konzisztens módot biztosít a kódnak az említett környezetekbe való eljuttatására.

A folyamatos telepítés (Continuous Deployment – szintén CD a rövidítése) a folyamatos szállítás következő lépése. Minden olyan változtatás, amely átmegy az automatizált teszteken, automatikusan az éles rendszer környezetbe kerül, ami számos production oldali telepítést eredményez. A folyamatos telepítés a legtöbb olyan vállalat célja kell, hogy legyen, amelyet nem korlátoznak szabályozási vagy egyéb követelmények.

Röviden, a CI olyan gyakorlatok összessége, amelyeket a fejlesztők a kód írása közben végeznek, a CD pedig olyan gyakorlatok összessége, amelyeket a kód elkészülte után végeznek [8].



14–49. ábra: CI/CD folyamat és elemei (Forrás: [8])

A CI/CD folyamatot hívják CI/CD csővezetéknek (pipeline) is a folyamatos áramlás és működés miatt.

A *„végtelenített”* folyamat részei:

1. A rendszer és a környezet megtervezése (plan).
2. Implementálás, kódolás (code), kódminőség és hibák javítása (optimalizálás).
3. Alkalmazás felépítése (build) a tervezési fázisban dokumentált követelményeknek megfelelően.
4. A szoftvertermék tesztelése (continuous testing) a tesztelő csoport által, szintén a követelményeknek (például a funkciókon túl a biztonsági kritériumoknak való megfelelés) a cél.
5. A production környezetbe helyezhető, megfelelően letesztelt szoftververzió elkészülése a publikálás előtt (release).

6. A program publikus közzététele (deploy).
7. Folyamatos használat, működtetés (operate).
8. A működés felügyelete és az esetleges hibák nyomon követése (monitor).

Majd a folyamat indul újra az elejéről (plan) a tervezéssel és jár ugyanígy körbe-körbe.

A folyamat bizonyos részeit (a tervezést, kódolást, *„build"*-elést, tesztelést, release létrehozását, közzétételt) már mi is kipróbáltuk önmagukban vagy egy-két lépést összeillesztve az Azure segítségével. De ennél sokkal hatékonyabb módon is meg lehet valósítani ezt a folyamatot az Azure Pipeline vagy akár a GitHub Actions segítségével létre tudjuk hozni ezt a CI/CD folyamatot, a lépéseket össze tudjuk integrálni az automatizmusokkal együtt a végrehajtást tudjuk támogatni.

Ha érdekel a téma további folytatása, akkor kövesd a blogomon az #azure címkével ellátott bejegyzéseket.

Természetesen más felhőszolgáltatásokat is ki lehet próbálni Laravel szempontból. Korábban a Heroku-val voltak tapasztalataink, de ez 2024-re már nem nyújtott ingyenes szolgáltatást. Ha valaki mégis ezt választaná, akkor a blogomon készültek hozzá leírások:

- Laravel alkalmazás telepítése a Heroku felhőrendszerébe
- Heroku és a ClearDB MySQL

Mindenesetre érdemes megtekinteni, az Amazon Web Services, a Google Cloud vagy hallgatóim javaslatára a Digital Ocean szolgáltatásait is.

14.4. Összegzés

A fejezet során megismerhettük a Laravel webes alkalmazásaink publikálásának, közzétételének lehetőségeit, folyamatait.

Először egy nyilvános tárhely és domain szolgáltató által tettük publikusan elérhetővé a Laravel alkalmazásunkat. SQLite és MySQL adatbázis kiszolgálóval is működőképes megoldásokat követhettünk végig.

Ezután egy rövid ismertetést olvashattunk a felhőszolgáltatásokról, fajtáikról, felépítésükről. Majd gyakorlati szempontból kiválasztottunk egy nyilvános felhőszolgáltatót, a Microsoft Azure-t és elkezdtük felköltöztetni a webes alkalmazásunkat a felhőbe. Kezdetben az adatbáziskiszolgáló részére koncentráltunk és egy MySQL adatbáziskezelő szerver szolgáltatást használtunk erre a célra.

Végül a Laravel projektünket is publikáltuk a felhőben lévő környezetbe és tettük publikusan elérhetővé. Ez a folyamat már tartalmazott nehézségeket, de minden problémát megoldottunk és közben olyan fejlesztési eszközök használatával ismerkedtünk meg, amelyek a fejlesztői mellett, üzemeltetői tapasztalatokat is szolgáltatott számunkra.

A fejezet legvégén az általunk megvalósított felhőbe publikáló *„build & deploy"* folyamat kiterjesztéseként a CI/CD folyamatok hátterét is áttekintettük.

15. Kiegészítő útravaló a jövőre (Additional guidance for future work)

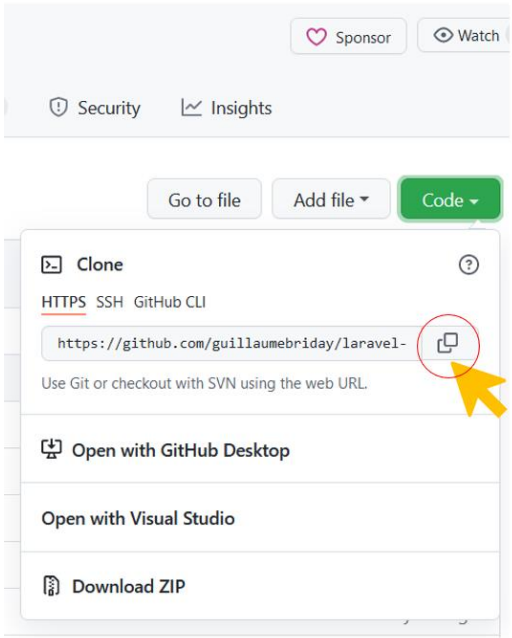
A fejezet során olyan hasznos technikákkal ismerkedünk meg, amelyek használata sokszor elengedhetetlen a Laravel keretrendszerrel való fejlesztés során.

15.1. Projektek (repository-k) clone-ozása

Mivel gyakran előfordul, hogy egy már GitHub-on meglévő Laravel projektet szeretnénk letölteni és használni, ezért érdemes végignézni ennek a folyamatát, hogy hogyan tudjuk életre kelteni a már elkészített projekteket. Klónozni fogunk! Utána pedig elvégezzük a szükséges beállításokat a működtetéshez.

A GitHub-on rengeteg hasznos, kész webes alkalmazás projekt érhető el. Számos Laravel projekt is megtalálható ott, amiket, ha mi magunk is ki szeretnénk próbálni, akkor képesek vagyunk „lemásolni", klónozni azokat. Ennek a folyamatát tekintem át ebben az alfejezetben. Ehhez a gyakorlás szempontjából egy olyan projektet érdemes választani, aminek nem csak backend-je, hanem fontend-je és adatbázis háttere is van. A projekt, amit fel fogunk használni a teszteléshez: https://github.com/guillaumebriday/laravel-blog *(2023. 03. 30-án még a Laravel 9-es verziója volt az alapja, de ez a mi szempontunkból nem okoz problémát a clone-ozás bemutatása kapcsán).*

A GitHub projekt megnyitása után másoljuk ki a HTTPS linket (előtte a Code gombbal nyitható le ez a kis ablak).



15–1. ábra: Klónozáshoz a repository címének másolása

Nyissunk meg egy parancssort (vagy terminal-t), és lépjünk be abba a mappába, ahol szeretnénk létrehozni a projektet, például a C:\Users\<felhasználónév> mappába a cd parancs segítségével, vagy már közvetlenül abban a mappában nyissuk meg a parancssort. Ezután következhet a projekt klónozása:

```
git clone https://github.com/guillaumebriday/laravel-blog.git
```

Az utasítás hatására létrejött a laravel-blog nevű mappa. Nyissuk meg a mappát a VSCode-ban. Szemfülesebbek észrevehetik, hogy a mappaszerkezet *„karcsúbb”,* mint a már korábban meglévő projektjeinknél. A mappa jelenlegi mérete nálam 4,74 MB. Ez a karcsúság abból adódik, hogy itt nincsenek még telepítve a Laravel projekt függőségei, úgyhogy nem is tudnánk még futtatni ezt a projektet és a böngészőben tallózni.

15.1.1. Függőségek telepítése

A Laravel projekteknek mindig vannak szerver- és kliensoldali függőségei, amelyeket kezdetben telepíteni kell, utána pedig időről-időre frissíteni is szükséges.

15.1.1.1. Szerver oldali függőségek

A megnyitott VSCode ablakban nyissunk egy új terminal-t azért, hogy először a composer segítségével telepítsük a szerver oldali Laravel és egyéb függőségi csomagok legfrissebb verzióit:

```
composer install
```

Az utasítás kiadása után remélhetőleg települnek azok a csomagok a projektbe, amelyek a **composer.json** fáljban, kvázi, mint egy tartalomjegyzék a csomagokról) definiálva voltak. (Előfordulhat, hogy hibát kapunk az utasítás kiadásakor: Your requirements could not be resolved to an installable set of packages. Ekkor a következő parancsot futtassuk, ami megoldja a problémánkat:

```
composer install --ignore-platform-reqs
```

Az utasítás kiadásának hatására jó sok csomag töltődik le és települ a projektünkbe. Ezek a **vendor** mappába fognak eltárolódni. A **vendor** angol szó magyarul beszállítót jelent, ide kerülnek tehát az úgynevezett 3rd party, vagyis a 3. féltől származó csomagok, amelyekhez közvetlenül sohase nyúljunk hozzá, ne oldjunk meg úgy problémákat, hogy ezeket szerkesszük, hiszen ezek egy következő parancs hatására úgyis felülíródnak, amikor frissítjük például a csomagok verzióját.

```
composer update
```

Mindeközben a feltöltött **vendor** mappával 60,7 MB-os lett már a fő mappánk, tehát több mint tizenkétszeres méretűre hízott. Ez a nagy méretbeli különbség is az oka, hogy a GitHub-on csak a függőségek leírása (**composer.json** és **composer.lock**) fájlok vannak fenn, hiszen a *„recept”* alapján mindenki maga tudja telepíteni a csomagokat, amikor már klónozta a projektet. A **.gitignore** fájl emiatt tartalmazza a felsorolásában a **vendor** mappát is, hogy azt senki se akarja feltölteni a git-es repository-jába, mivel felesleges helyfoglalás lenne ott.

15.1.1.2. Kliens oldali függőségek

Következhet a további (főleg kliens oldali) csomagok frissítése az npm-mel:

```
npm install
```

Ha valakinek hibát adna (a *„WARN”,* vagyis warning, figyelmeztetés üzenetek nem hibák), akkor ezzel javítható:

```
npm install --legacy-peer-deps
```

Ennek hatására a **package.json** (és a **package-lock.json**) fájlban lévő csomagok és azok további függőségei fognak települni a **node_modules** mappába. Az így kapott projekt mappaméret: 152 MB, így jó nagyra megnőtt a kezdeti 4,79 MB-os méretről. Ez azért is van, mert ebben a laravel-blog-ban számos olyan funkcionalitás van előre telepítve (vagy telepítésre kijelölve, ha a klónozás szempontjából nézzük), amely hasznos lehet egy blog oldal menedzseléséhez. Ezekről a szolgáltatásokról a projekt GitHub oldalán olvashatunk a mappa- és fájlszerkezet alatti leírásban.

15.1.2. Az .env fájl

A következő fontos lépés, hogy beállítsuk az **.env** fájl tartalmát, azon belül is főleg majd az adatbázis elérést. Mivel a clone-ozás után még nem létezik a fájl, ezért adjuk ki a parancsot:

```
cp .env.example .env
```

Ezzel lemásoljuk a projekt mappájában lévő **.env.example** fájl tartalmát az új **.env** fájlba. Nyissuk meg az **.env** fájlt! A Laravel-nek szüksége van egy kódoló kulcsra (**APP_KEY**), aminek jelenleg még nincs értéke. Ez egy véletlenszerűen generált karaktersorozat lesz, amit sok mindenhez felhasznál a Laravel a működése során, például a felhasználók jelszavainak kódolásához (hash) is. Hozzuk létre a kulcsot a következő utasítással:

```
php artisan key:generate
```

Vegyük észre, hogy oda is került a kódoló kulcs az **APP_KEY** attribútumhoz értékként.

15.1.3. Adatbázis: kapcsolódás, migrálás, feltöltés

Megjegyzés: előfordulhat, hogy az általatok klónozott projekt nem tartalmaz adatbázis specifikus dolgokat, ekkor a folyamatból ez a lépés (alfejezet) kihagyható, de ez eléggé ritka eset lehet. Az adatbázis kapcsolatokról és a hozzáférésekről, valamint a migrálásról az 1. fejezetben esett szó részletesebben.

Ezután tekintsük át a **DB_** kezdetű attribútumokat. Itt fontos megjegyezni, hogy ez a **laravel-blog** projekt, amelyet klónoztunk, úgy lett kialakítva, hogy képes legyen futni egy virtuális környezetben is, azonban ehhez több minden mást is telepíteni kellene, amelyet most nem fogunk mi megtenni, hanem maradunk helyben, a fizikai környezetünkben (akit mégis érdekelne a virtuális környezet létrehozása, az nézze meg a projekt GitHub oldalán előkövetelményként feltüntetett *VirtualBox* és *Vagrant* alkalmazásokat). Maradva tehát helyben, szükségünk lesz egy adatbázisra, ahova majd bekerülnek a blog adatai. Maradhatunk a **mysql** driver-nél, mint **DB_CONNECTION**, hiszen ezt már korábban úgyis telepítettük a gépünkre. A MySQL

adatbázis-kezelőnket a MySQL Workbench alkalmazással menedzseljük. A MySQL-hez való kapcsolódás után a Workbench-ben létrehozok egy új adatbázist, a következő paranccsal:

```
CREATE DATABASE l10_laravel_blog_db;
```

Ezzel létre is jött az üres adatbázisunk, amelybe tudunk dolgozni. Állítsuk be az adatkapcsolatot az **.env** fájlban eszerint:

```
DB_CONNECTION=mysql
DB_HOST=127.0.0.1
DB_DATABASE=l10_laravel_blog_db
DB_USERNAME=root
DB_PASSWORD=
```

15–1. kódrészlet: Adatkapcsolat a laravel-blog projekt adatbázisához

Következhet a migrálás, amivel az adatbázis tábláinak szerkezetét létrehozzuk.

```
php artisan migrate
```

Megjegyzés: a **2022_04_30_011124_add_generated_conversions_to_media_table.php** migrációs fájlban 2023. 03. 30-án egyetlen hiba van:

```
//'generated_conversions' => DB::raw('custom_properties-
>"$.generated_conversions"'),
'generated_conversions' => DB::raw("JSON_EXTRACT(custom_properties,
'$.generated_conversions')"),
```

15–2. kódrészlet: Hibajavítás a migrációs fájlban

Az iménti kódrészletben kommentben látszódik a hibás sor, alatta pedig a megfelelő sor, de ezt lehetséges, hogy javítják, mert jeleztem a fejlesztők felé, így remélhetőleg hiba nélkül lefut majd a migrálás. Ha mégis valamilyen hiba adódna, akkor a kijavítása után futtassuk a következő parancsot:

```
php artisan migrate:fresh
```

Létrejönnek az adattábláink, de még üresek. Ez a projekt tesztelési adatokat is biztosít ahhoz, hogy bemutassa az alkalmazás használatát. A teszt adatokat így tudjuk elhelyezni az adattáblákban.

```
php artisan db:seed
```

Ezzel létrejönnek a blogban szerepkörök, egy felhasználó, egy blogbejegyzés stb.

15.1.4. Kliens oldali fájlok fordítása és optimalizálása

Egy dolog van hátra, mielőtt elindítanánk a weboldalunk kiszolgálását: a kliens oldali elemeket (nézeteket, CSS és JS fájlokat) még le kell fordítanunk és optimalizálnunk kell.

Érdekesség: a Laravel 9-es verziójától kezdve erről már automatikusan a Vite gondoskodik (4.3. alfejezet), de a 9-es verzió előtt a Vite helyett a Laravel Mix gondoskodott. Erről javaslom átolvasásra ezt a blogbejegyzésemet: a Laravel Mix használatáról itt írtam a blogomon.

Tehát a laravel-blog projekt **public** mappa részei (css, js stb. mappák) is lokálisan generálódnak (hiszen a **.gitignore** fájl is tartalmazza őket). A **public** mappa elemei a **resources** mappából származnak a Laravel Mix alapján. Az ott lévő SCSS és JS fájlok, valamint függőségeik kerülnek lefordításra és optimalizálásra, és így kerülnek át a **public** mappába. Az összerendelés alapja a projekt gyökerében található **webpack.mix.js** fájl, ebből egy részlet:

15–2. ábra: Laravel Mix működése (miből mit és hova generál)

Az tehát megvan, hogy mi az összerendelés alapja. Most már csak el kell végezni a fordítást és optimalizálást.

A folyamat egy utasítás kiadásából áll:

```
npm run dev
```

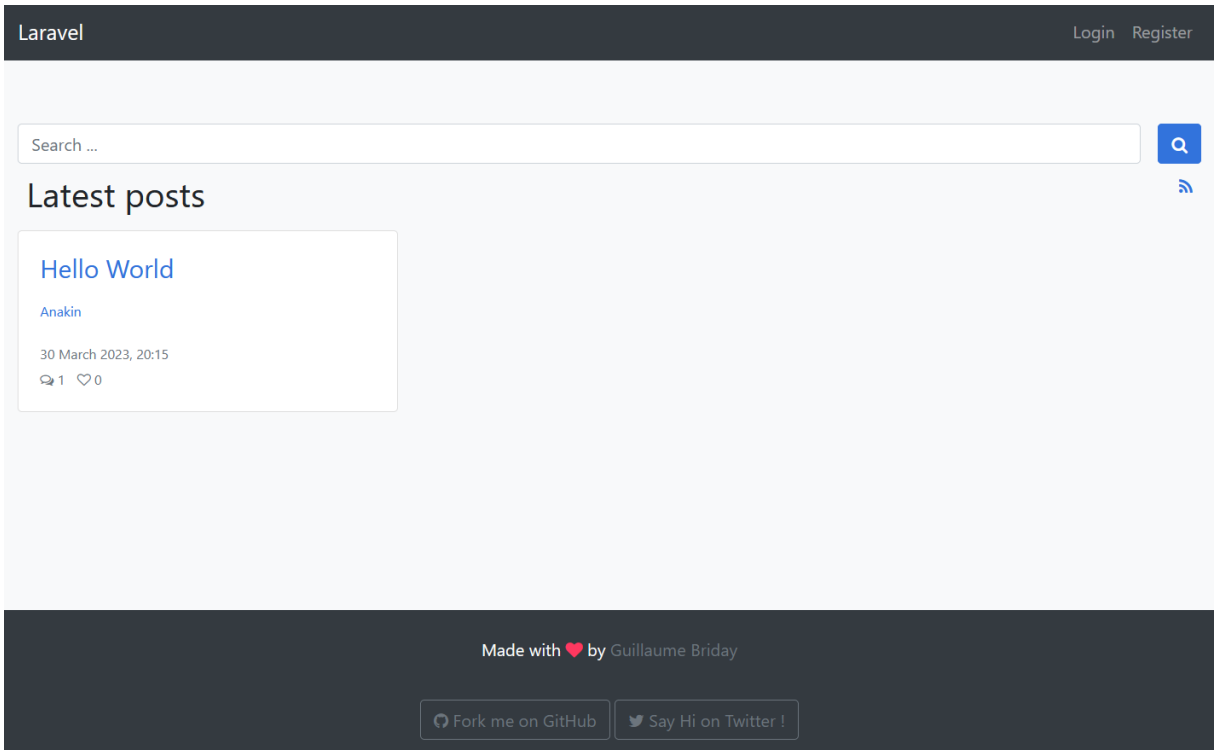
15.1.5. Kiszolgálás és futtatás

Most már menni fog a blog alkalmazás, amelynek van egy látogatók számára készült felülete és a blog menedzserei, adminisztrátorai, szerkesztői számára pedig egy másik felülete. Indítsuk el a webalkalmazás kiszolgálását:

```
php artisan serve
```

Látogatók és sima felhasználók számára látható blog oldal elérhető itt: http://127.0.0.1:8000/

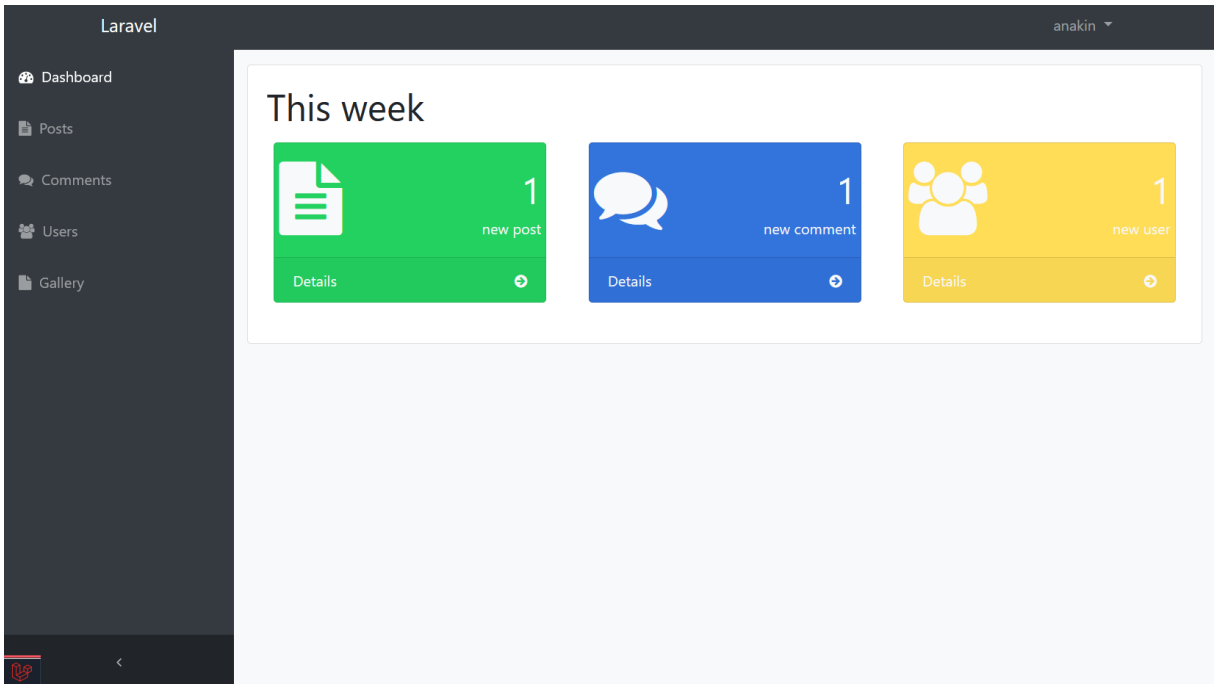
Ezzel készen vagyunk! Ugyanezeket a lépéseket követve képesek vagyunk más GitHub-os Laravel projekteket is klónozni, de emellett, ha valaki végig követte a lépéseimet, megkapott egy teljes értékű blog alkalmazást is adatbázissal, adminisztrációs felülettel stb.



15–3. ábra: Klónozott blog oldalunk látogatói/felhasználói felülete

Az adminisztrátori felület elérhető a sikeres bejelentkezés (login) után, ha legeneráltuk a db:seed utasítással a teszt adatokat: e-mail: darthvader@deathstar.ds és jelszó: 4nak1n

Bejelentkezés utáni adminisztrációs felület itt látható: http://127.0.0.1:8000/admin/dashboard



15–4. ábra: Klónozott blog oldalunk adminisztrátori/szerkesztői felülete

15.2. Függőségek kezelése, frissítése

Előfordulhat, hogy hosszabb ideig nem kerülünk napi szintű kapcsolatba korábban fejlesztett projektjeinkkel. Emiatt, amikor újra hozzájuk kell nyúlnunk, mindenképpen érdemes felfrissíteni ezeket a Laravel projektjeinket, sőt, előfordulhat, hogy a fejlesztőkörnyezet elemeit (PHP verzió, adatbáziskezelőrendszer verziója, Laravel telepítő verziója, esetleg, ha a XAMPP szervercsomagot használjuk, akkor annak a verzióját is).

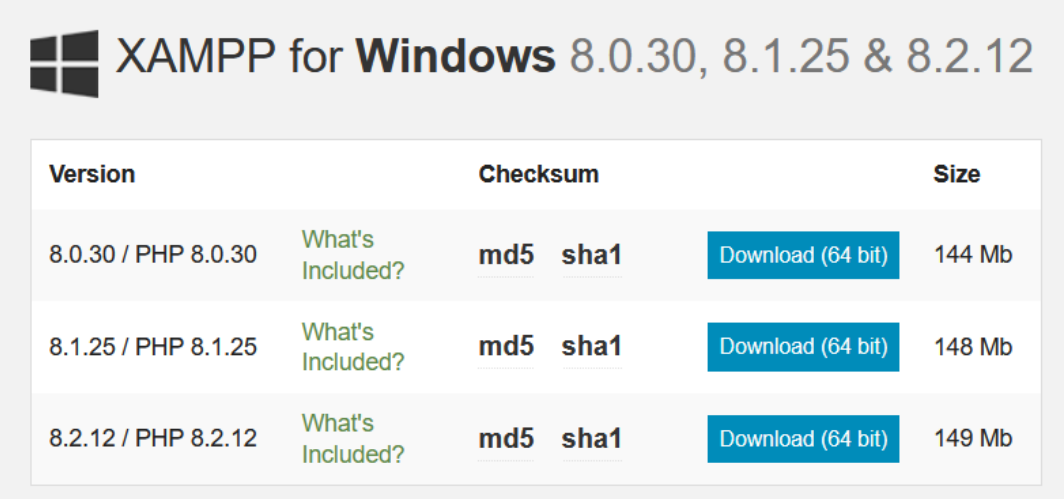
Azt már tudjuk, hogy a webes világban szerencsére nem kell mindig *"újra és újra feltalálni a kereket"*, ezért Laravel-es projektjeinkben dolgozhatunk úgynevezett *harmadik féltől származó osztálykönyvtárakkal, csomagokkal* (3rd party libraries). A weben mindig (legalább) két oldallal kell foglalkoznunk: a kliens és a szerver oldali kódjainkkal.

15.2.1. Környezet frissítése

A XAMPP szervercsomag elérhető Windows, Linux és MAC OS X operációs rendszerekre is. A csomagon belül az Apache webszerver képes kiszolgálni a kéréseket, ennek frissítése a XAMPP újratelepítésével lehetséges egy letöltés után. A verziószámok a PHP fordító verziójára utalnak, mivel ez a leginkább kardinális kérdés a telepítés megkezdésekor. Az Apache webszerver verziószáma annyira nem releváns, úgyhogy ha csak a PHP verziót szeretnénk frissíteni, akkor érdemes a 15.2.1.2. alfejezetet áttekinteni.

15.2.1.1. XAMPP csomag frissítése

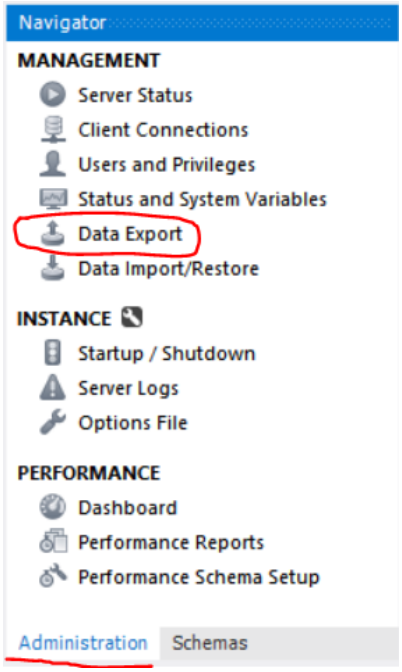
A XAMPP csomag letöltését innen (https://www.apachefriends.org/download.html) tudjuk megtenni.



15–5. ábra: XAMPP letöltése Windows operációs rendszerhez (PHP verziókkal)

A XAMPP csomag letöltését innen (https://www.apachefriends.org/download.html) tudjuk megtenni. Mielőtt azonban belevágnánk az újra telepítésébe, *biztonsági okokból készítsünk előtte mentéseket!*

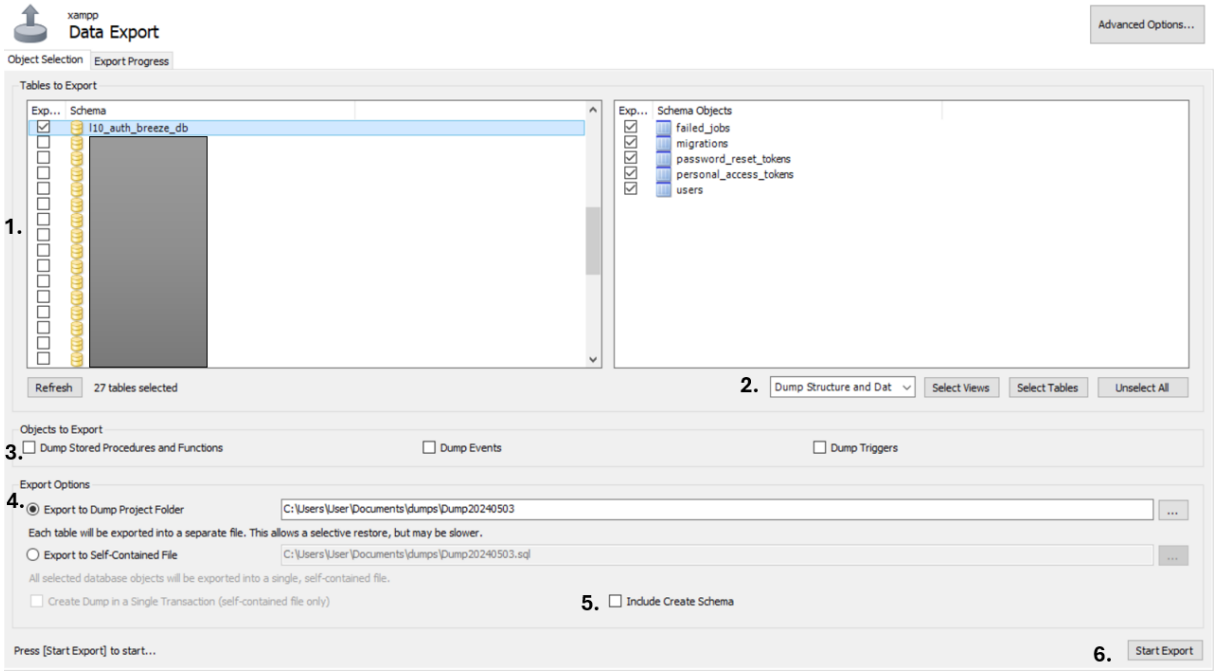
1. C:\xampp\htdocs mappa tartalmáról, amiben a projektjeink vannak,
2. Az adatbázisaink tartalmát is exportáljuk séma információkkal és adatokkal együtt! Az adatbázis exportokat elvégezhetjük MySQL Workbench alkalmazásban a bal oldali *„Navigator"* nevű panel *„Administration"* lapfülén lévő *„Data Export"* gomb segítségével.



15–6. ábra: Adatbázis sémák és adataik exportálása a MySQL Workbench-ben

A következő ablakban beállíthatjuk a következőket (lásd mellé a felsorolás utáni ábrát):

1. Melyik adatbázisokat szeretnénk exportálni (ki tudjuk pipálni).
2. A 2. lenyíló listában válasszuk ki, hogy a sémákat és az adatokat is mentse ki.
3. A 3. jelnél lévő checkbox-okat is érdemes bejelölni, attól függően, hogy használtunk-e már ilyeneket.
4. A 4. pontban a mappa nevét tudjuk meghatározni, hogy hova mentse ki a rendszer az SQL fájlokat.
5. Az 5. pontban a séma létrehozó utasításokat is ki tudjuk menteni.
6. A 6. *„Start Export"* gomb megnyomásával pedig elindul az exportálási folyamat.



15–7. ábra: Adatstruktúra és adathalmaz exportálása a MySQLWorkbench-ben

Ha ezeket az adatmentéseket elvégeztük, utána már nagy baj nem érhet minket. Érdemes amúgy is időről időre backup-ot készítenünk, ha másról nem, akkor az adatbázisban lévő adatokról, mivel a projektjeinket (és az adattábláink struktúráját a migrációs fájlokkal) GitHub segítségével is tudjuk *„verziókövetni"*. Ezután következhet az uninstall folyamat, amely során a telepítő megkérdezi, hogy törölje-e a **htdocs** mappa tartalmát, erre válaszoljunk úgy, hogy a mappa tartalma, és így a korábbi projektjeink megmaradjanak. Majd, ha az sikeresen lefutott, telepítsük fel az újabb verziójú XAMPP-ot, és utána egy `php -v` parancs kiadásával tudjuk ellenőrizni, hogy már a legújabb (vagy a nekünk éppen megfelelő) verziójú PHP-t használjuk-e. A MySQL Workbench segítségével pedig képesek vagyunk importálni a korábban exportált adatbázisokat ott, ahol a *„Data Export"* menüpont is elérhető volt.

15.2.1.2. PHP verzió frissítése a XAMPP-on belül

Ha nem szeretnénk az egész XAMPP szervercsomagot újratelepíteni, csak a hozzá tartozó PHP fordítót, akkor a következő folyamatot kell elvégeznünk:

1. Állítsuk le a XAMPP-os szolgáltatásokat a vezérlőpultján keresztül (Apache, MySQL szolgáltatások mellett a Stop gombok megnyomásával).
2. Töltsük majd le a nekünk megfelelő XAMPP verziót innen:
    a. https://sourceforge.net/projects/xampp/files/XAMPP%20Windows/ (2024. 05. 03-án a legfrissebb verzió a 8.2.12 PHP verziójú).
3. Kattintsunk rá a verziószám mappájára!
4. Töltsük le a Windows x64 típusú **zip** fájlt: **xampp-windows-x64-8.2.12-0-VS16.zip**
5. Nyissuk meg a zip fájlt!
6. Közben menjünk a **C:\xampp** mappába, és nevezzük át biztonsági okokból az ottani **apache** és **php** mappákat **apache_old**-ra és **php_old**-ra, hogy ha bármi baj történne, ne veszítsük el őket, hanem itt meglegyenek.

7. A megnyitott zip fájlból az apache és a php mappákat másoljuk át a C:\xampp mappába!
8. Indítsunk újra el a XAMPP vezérlőpultján az Apache és MySQL szolgáltatásokat!
9. Indítsunk egy parancssort vagy terminal-t, és ellenőrizzük le a PHP verzióját: `php -v`

Így a nekünk megfelelő (legfrissebb) PHP verzió telepítésre került.

15.2.1.3. Adatbáziskezelő rendszer frissítése

A XAMPP újratelepítésével gyakorlatilag a MySQL adatbáziskezelő szerver verzióját is jobb esetben frissítettük. Ha másfajta adatbázisokat használunk, például SQLite-ot, akkor könnyebb dolgunk van, mivel annál csak egy fájlra van szükség az adatok eltárolásához, viszont érdemes figyelni a PHP sqlite kiterjesztési csomagjára, hogy az megfelelően tud-e csatlakozni az adatbázis (például .sqlite vagy .db kiterjesztésű) fájlunkhoz.

További adatbáziskezelő rendszereknél (mint például Microsoft SQL Server vagy PostgreSQL esetén) mindig figyeljünk arra, hogy az újra telepítése előtt végezzük el az adatbázisok, beállítások biztonsági mentését, hogy a későbbiekben ne okozhasson gondot valamilyen adathiány. Illetve, ha az újabb verziókkal nem működne az alkalmazásunk, akkor így a verzió visszalépés sem okozhat problémát.

15.2.2. Szerver oldali függőségek

Az alfejezetben áttekintjük a szerver oldali csomagkezelő és a csomagok frissítését.

15.2.2.1. Csomagkezelő frissítése

A composer szerver oldali csomagok telepítéséért és frissítéséért felelős alkalmazás a https://getcomposer.org/download/ weboldalon keresztül tölthető le és telepíthető. A frissítése viszont már működik a terminal-ból is. Hajtsuk végre a következő parancsot:

```
composer self-update
```

15.2.2.2. Csomagok frissítése

A szerver oldali csomagjaink a projekt gyökerének vendor mappájában vannak, ezeket ugye manuálisan jobb nem módosítani, ezért van nekünk a composer csomagkezelőnk, amely a frissítés megkezdésekor a composer.json fájlt veszi alapul akkor, amikor futtatjuk a `composer install` vagy `composer update` parancsot a terminal-ban. Ez azonban még csak a kiindulási pontja a műveletnek: először megnézi, hogy milyen harmadik féltől származó csomagokat használunk, majd megpróbálja letölteni az összes olyan csomagverziót, amiből talál frissebbet (persze csak akkor, ha a verziószám előtt ott van a ^ karakter, ami azt mondja meg a composer-nek, hogy *"legalább ilyen verziószámú csomagot telepíts nekem"*), a letöltés (downloading) után pedig következik a csomagok tényleges felfrissítése (upgrading). A fő (legelső) verziószámot akkor sem frissíti, ha előtte van a ^ karakter. A composer csomagkezelő is fejlődik, belekerült egy olyan funkcionalitás is, amellyel az elavult csomagjainkat tudjuk kilistázni. A lista pedig rögtön mutatja, hogy az adott elavult csomagból melyik a legfrissebb verzió, ami elérhető a számunkra (fő verzió ugrásokkal együtt):

```
composer outdated
```

```
PS C:\xampp\htdocs\l10-components> composer outdated
Color legend:
- patch or minor release available - update recommended
- major release available - update possible

Direct dependencies required in composer.json:
barryvdh/laravel-debugbar          v3.12.2  v3.13.5 PHP Debugbar integration for Laravel
doctrine/dbal                      3.8.3    4.0.2   Powerful PHP database abstraction layer (DBAL) with many features for database...
laravel/dusk                       v7.13.0  v8.2.0  Laravel Dusk provides simple end-to-end testing and browser automation.
laravel/folio                      v1.1.6   v1.1.7  Page based routing for Laravel.
laravel/framework                  v10.48.4 v11.6.0 The Laravel Framework.
laravel/pint                       v1.14.0  v1.15.3 An opinionated code formatter for PHP.
laravel/sanctum                    v3.3.3   v4.0.2  Laravel Sanctum provides a featherweight authentication system for SPAs and si...
nunomaduro/collision               v7.10.0  v8.1.1  Cli error handling for console/command-line PHP applications.
phpunit/phpunit                    10.5.15  11.1.3  The PHP Unit Testing framework.
spatie/laravel-ignition            2.4.2    2.6.2   A beautiful error page for Laravel applications.
```

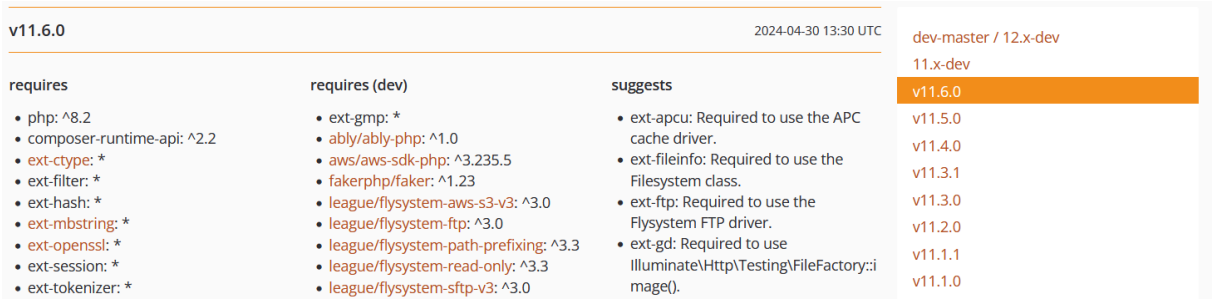
15–8. ábra: Elavult direkt módon csatolt csomagjaink

A lista tartalmazza a direkt módon csatolt, és az indirekt módon érintett csomagok neveit, aktuális és legfrissebb verziószámait is.

Ha már belekezdünk egy régebbi projektünk frissítésébe, akkor a következőkkel kell számolnunk:

- Szerencsés esetben nem volt még túl régi a Laravel projektünk, és nem kell nagyobb (több más függőséggel rendelkező) csomagokat frissítenünk, vagy túl nagy verziószámot ugrani előre.
- De minél nagyobb és szerteágazóbb a projekt és a függőségi struktúrája, annál nagyobb a hibalehetőség (*megjegyzés:* valami hasonlóról szól a SOLID elvek közül a D-re utaló, függőség megfordítási elv vagy függőséginverzió alapelve: Dependency Inversion Principle. Ezt azért is érdemes alkalmazni, hogy ne történhessen meg, hogy egy olyan programcsomagra építjük a saját alkalmazásunkat, amit aztán később a készítője eltávolít, használatát letiltja stb.). Előfordulhat az is, hogy csak egy csomagot kell frissíteni, de az például 20-50-100 másik csomagtól is függ, amelyeket a projektünkben szintén frissíteni kell... talán érzékelhető az a probléma, hogy minél több ilyen egymásra épülés van, annál nagyobb az esélye annak, hogy valaki, valahol hibázott és emiatt nem tudjuk mi frissíteni a csomagokat.
- Ha mondjuk a fő keretrendszerünk, jelen esetben a Laravel fő verzióját is frissíteni szeretnénk, akkor az még több galibához vezethet. Például, amikor a 10-esről a 11-esre frissítettünk korábban, akkor a PHP verzióját is frissíteni kellett, ami még további problémákat generálhat a számunkra. Azért a fő verziószám váltáskor a Laravel dokumentációja is szokott segíteni, hogy miket kell megváltoztatni a rendszerben, például itt van egy részletes útmutató az említett fő verziószám váltásról: https://laravel.com/docs/11.x/upgrade
- Előfordulhat olyan probléma is, hogy adott csomag függőség miatt nem upgrade-et, hanem verziószám csökkentést (downgrade) kell alkalmaznunk, mert egy adott létfontosságú csomag verzióját még nem frissítették a legújabb körülmények kívánalmaihoz. Erre persze annál kisebb az esély, minél népszerűbb a csomag használata, mert egy sokak által használt csomagot általában mindig nagyon gyorsan frissítenek a körülményekhez.
- Nevezhetjük ezeket a problémás eseteket is *"függőségi pokol"*-nak, főleg, ha ide-oda kell ugrálni a verziószámok között, amíg megtaláljuk az arany középutat, ami már mindennek és mindenkinek megfelelő lesz... Szerencsére van egy Packagist nevű weboldal (https://packagist.org/explore/), ami segít könnyedén keresni a csomagok között, verziószámokat jelöl nekünk, és leírja, hogy milyen más csomagoktól is függ az az adott csomag és annak verziója.

Itt van például a laravel/framework csomag, amely a keretrendszer magját adja.



15–9. ábra: Részlet a Packagist weboldal laravel/framework csomag bemutatójából

Látható, hogy a legfrissebb stabil (!) verzió az a 11.06.0-s (2024. 05. 01-én). Az oszlopokban bal oldalon látható, hogy milyen futtatókörnyezetet (PHP fordító verziót) és csomagokat igényel mindenképpen éles környezetben, az utána lévő oszlopban pedig a fejlesztői környezetes igényelt csomagjait is láthatjuk. A jobb oldali oszlopban, ha esetleg *„el kell indulnunk visszafelé”* (downgrading) a csomag verziószámaiban, akkor kattinthatunk, válthatunk, és akkor szintén mutatja nekünk, hogy épp az a kiválasztott verzió milyen más csomagokat és verziókat igényel. Ez a Packagist weboldal tehát rettentő hasznos tud lenni a munka során.

15.2.3. Kliens oldali függőségek

Az alfejezetben áttekintjük a kliens oldali csomagkezelő és a csomagok frissítését.

15.2.3.1. Csomagkezelő frissítése

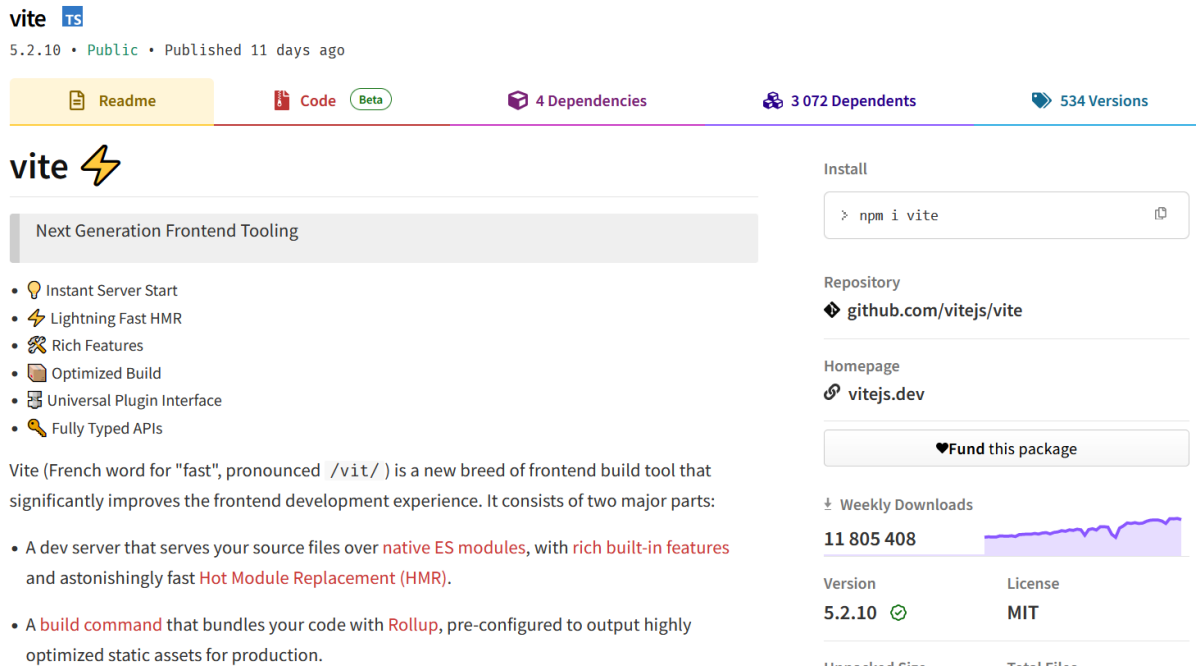
A kliens oldali csomagok frissítését az npm csomagkezelővel tudjuk végrehajtani. Ennek a frissítése működik a terminal-ban is:

```
npm install -g npm@latest
```

15.2.3.2. Csomagok frissítése

A kliens oldali függőségek csomagjai a projekt gyökerében lévő **node_modules** mappában találhatók meg. Itt az **npm** csomagkezelő lesz a segítségünkre, aminek a kiindulási pontja a **package.json** fájlunk lesz. `npm install` vagy `npm update` utasítás kiadásával szépen települnek, vagy felfrissülnek a kliens oldalt támogató csomagjaink... ha minden szép és jó az életben. De azért ez elég ritka, úgyhogy bőven előfordulhat, hogy belefutunk olyan verziókülönbségekből adódó problémákba, amelyek megnehezíthetik a dolgunkat. Ilyenkor a **package.json** fájlban kezdünk el praktikusan keresgélni, és a verziókat próbáljuk meg megfelelően beállítani azért, hogy hiba nélkül lefuthasson a csomagok frissítése. De milyen verziókat is állítsunk be azért, hogy menjen minden, mint a karikacsapás? Ebben nagy segítségünkre van egy weboldal, amivel könnyedén tudunk keresni csomagokra, és a kiválasztás után látszódik, hogy milyen verziószámmal dolgoznak aktuálisan, emellett saját maguknak milyen egyéb függőségeik vannak. A kliens oldali függőségek rengetegében is van egy segítő weboldalunk: https://www.npmjs.com/

Itt a kliens oldalon is kiemelhetünk egy példát, a korábban megismert Vite csomagot (https://www.npmjs.com/package/vite).



15–10. ábra: Részlet az npmjs weboldal Vite csomag bemutatójából

Az oldal röviden bemutatja nekünk a csomagot, megmondja, hogy hol található a repo-ja, weboldala, mi a legfrissebb verziószáma. Amire érdemes még figyelni, az a felső tab-os navigációban látható: jelenleg 4 csomagtól függ a Vite, ellenben tőle 3 072 egyéb másik csomag függ, amelyeket mind böngészhetünk... ez egy jó nagy szám!

Megjegyzés: általában nekem az volt mindig a benyomásom, hogy a kliens oldali függőségek sokkal mélyebbek, szerteágazóbbak, mint a szerver oldaliak, de lehet ez csak egy amolyan megérzés a részemről.

15.3. Többnyelvűsítés (Localization)

Van egy olyan témakör, amiről azt gondolom, hogy mindenképpen érintenünk kell, ha – akár csak sablonok segítségével is – egy teljesértékű felhasználói felületet akarunk nyújtani a webes alkalmazásainkhoz, ez pedig a többnyelvűsítés.

A többnyelvűsítés is egy olyan feladat, amit webfejlesztőként az esetek túlnyomó többségében el kell végeznünk egy weboldal vagy egy webalkalmazás elkészítése során. A Laravel keretrendszer viszont ennek könnyű megvalósításához is nagy segítséget nyújt. Ha nagyon egyszerűen szeretném megfogalmazni, akkor arról van szó, hogy az alkalmazásban megjelenő szövegeket valamilyen adott nyelvnek megfelelően szeretnénk megjeleníteni. Egy alapértelmezett nyelvre mindig szükség van, ami szerint mindenképpen meg tudnak jelenni a szövegek és ehhez képest tudunk majd további plusz nyelveket is definiálni, amik között a felhasználó tud majd választani, miközben használja az alkalmazásunkat vagy böngészi a weboldalunkat.

Az egyszerűség kedvéért, térjünk vissza a **my-first-site** nevű projektünkhöz, amelyben a **pastel** sablon található.

15.3.1. Nyelvi fájlok használatának lehetőségei

Egy új Laravel projekt alapból nem tartalmazza a többnyelvűsítés lehetőségét, mivel rábízza a fejlesztőre, hogy ő el akarja-e látni az alkalmazását ezzel a funkcionalitással. Ha szeretnénk többnyelvű alkalmazást készíteni, akkor ezzel az utasítással kezdjük:

```
php artisan lang:publish
```

Az utasítás végrehajtása miatt létre fog jönni egy **lang** mappa a projektünk fő könyvtárai között és benne az **en** mappa plusz a tartalmában néhány olyan PHP fájl, ami már tartalmaz kulcs-érték párokat angolul: egy adott kulcsszóhoz egy szó/mondat szövegezés tartozik. (*Megjegyzés:* a **lang / en** mappa és tartalma a **vendor / laravel / framework / src / Illuminate / Translation / lang / en** mappa másolata. Ha most törölnénk a fájlokat a **lang / en** mappából, a bennük lévő szótár akkor is működne, csak így nem definiálnánk felül a Laravel keretrendszer által nyújtott szövegezéseket.)

Arra pedig kétféle lehetőségünk van, hogy hogyan tudjuk megszervezni a **lang** mappában a további nyelvi mappákat és fájlokat. A két módszer egyenértékű, csak rajtunk múlik, hogy melyiket tartjuk hasznosabbnak. Az alkalmazásunk mérete (olyan szempontból, hogy mennyi szöveg és azok fordítása) fog elhelyezkedni benne egy jó irányjelző lehet amiatt, hogy melyik módszert válasszuk.

Az **első módszer** szerint, ha nagyon sok ilyen szövegünk lesz, amit majd le kell fordítanunk (mert sajnos, a Laravel a fordítást nem csinálja meg helyettünk…), akkor a következő szerkezetet érdemes követni a **lang** mappán belül a könyvtárak és fájlok kialakításánál:

- Első nyelv mappája, például *en* (mint English, angol),
    - az alkalmazás 1. menüpontjának (rész funkcionalitásának) a *fájlja*, például: **home.php** (a főoldalon található szöveges elemek *angol* nyelvű szövegei),
    - az alkalmazás 2. menüpontjának (rész funkcionalitásának) a *fájlja*, például: **contact.php** (a kapcsolati oldalon található szöveges elemek *angol* nyelvű szövegei),
    - és így tovább a weboldal további menüpontjai és funkcionalitásai fájlnév szerint.
- Második nyelv mappája, például *hu* (mint Hungarian, magyar)
    - az alkalmazás 1. menüpontjának (rész funkcionalitásának) a *fájlja*, például: **home.php** (a főoldalon található szöveges elemek *magyar* nyelvű szövegei)
    - az alkalmazás 2. menüpontjának (rész funkcionalitásának) a *fájlja*, például: **contact.php** (a kapcsolati oldalon található szöveges elemek *magyar* nyelvű szövegei)
    - és így tovább a weboldal további menüpontjai és funkcionalitásai fájlnév szerint

Így tehát kialakulhat összefoglalóan egy hasonló szerkezet a **lang** mappán belül, ha két nyelvű alkalmazást/weboldalt szeretnénk:

- /en
    - /home.php
    - /contact.php
- /hu
    - /home.php
    - /contact.php

További nyelvek esetén mindegyik nyelvnek kell egy mappa, majd azokon belül ugyanaz a fájlstruktúrának meg kell lennie.

A **második módszert** akkor érdemes alkalmazni, ha az alkalmazásunkban/weboldalunkon nincsen annyira sok szöveg, így akár nyelvek szerint egy-egy fájlba is elég szerveznünk a szövegeket (feliratokat). Persze ebben az esetben is bármennyi szöveget elhelyezhetünk az alkalmazásunkban, ezáltal a „szótár fájljainkban", csak a szervezésből kimarad az az absztrakciós szint, amikor menüpontok/funkciók szerint csoportokba és ezáltal külön fájlokba szerveztük volna a szövegeinket.

A **lang** mappában lévő fájlok ekkor json fájlok lesznek és a fájlok nevei a nyelvek (szabvány szerinti, két betűs) rövidítését kapják meg. Például en.json, hu.json fájlok. Ekkor a **lang** mappán belüli fájlszerkezet a következő lesz:

- en.json
- hu.json

15.3.2. Alapértelmezett és fallback nyelv beállítása

A webalkalmazásunk alapértelmezett nyelvét a **config / app.php** fájlban találjuk meg. Mivel most először kerül találkozunk a **config** mappa elemeivel, ezért itt röviden annyit összefoglalnék a mappáról és a fájljainak a tartalmáról, hogy ezek felelősek a Laravel projekt beállításaiért. A benne lévő fájlok neve meglehetősen beszédes: **app** (az alkalmazás fő beállításai találhatók meg itt), **auth** (felhasználói hitelesítéssel kapcsolatos beállítások), **database** (adatbázis-kezeléssel kapcsolatos beállítások) stb. Bármelyik fájlba belenézhetünk, hiszen a Laravel nyílt forráskódú, és az összes paraméterezési lehetőségnél találunk a megjegyzésekben segédletet, hogy melyik beállítás miért is felelős. Maguk a fájlok egyszerű PHP fájlok és mindössze egy-egy asszociatív tömböt tartalmaznak visszatérésként. Az asszociatív tömbben pedig kulcs-érték párok segítségével tudjuk paraméterekkel ellátni, beállítani a Laravel alkalmazásunk működését.

De visszatérve a többnyelvűsítésre, vagy lokalizációra és a **config / app.php** fájlra. Ha elkezdjük tallózni, akkor láthatjuk, hogy itt lehet beállítani az alkalmazásnak a nevét, hogy milyen környezetben (fejlesztői vagy éles) szeretnénk futtatni az alkalmazást, de azt is, hogy milyen időzóna szerint generálja majd nekünk a Laravel az időbélyegeket. A nyelvi beállítások **locale** és **fallback_locale** kulcsszavakhoz vannak rendelve. A **locale** beállításnál tudjuk megadni az alkalmazásunk alapértelmezett nyelvét. Tehát, hogy ha a felhasználóink először meglátogatják az oldalunkat vagy a webes alkalmazásunkat, akkor milyen nyelven jelenjenek meg nekik a szövegek.

A **fallback_locale** ezzel szemben azért hasznos és azért kell, mert remélhetőleg lesz egy olyan fő nyelv, amelyen el fog készülni az alkalmazás összes szövegezése és ha véletlenül a **locale** beállítás szerint olyan szövegrészt kellene megjeleníteni, amelyhez nem tartozik az adott nyelven lefordított szöveg (hiányzik), akkor ezen a **fallback_locale** beállításban megadott nyelven fog megjelenni a szöveg, ezért itt olyan nyelvet állítsunk be, amihez biztosan megvan az összes szövegezés.

Új környezeti változók: az **.env** fájlban lévő **APP_LOCALE** és **APP_FALLBACK_LOCALE**

A Laravel 11-ben már az **.env** fájlban tudjuk definiálni az alkalmazásunkra vonatkozó lokalizáció, vagyis helyi beállításokat az **APP_LOCALE** és **APP_FALLBACK_LOCALE** beállítások segítségével.

15–1. újdonság: Lokalizációs, többnyelvűsítési újdonság a beállításoknál

15.3.3. Többnyelvűsítés példa: használat és a kiíratás

Elérhető számos hasznos metódus, amely a lokalizáció témájához kapcsolódik, itt most csak a három legfontosabbat gyűjtöttem ki:

1. **App::currentLocale()** – visszaadja az aktuálisan beállított nyelvet.
2. **App:isLocale('en')** – ellenőrzi, hogy az aktuálisan beállított nyelv például angol-e. Logikai értékkel tér vissza.
3. **App::setLocale('hu')** – beállítja az aktuális nyelvet példaként magyarra.

Ezeket bárhol meg tudjuk hívni és használni tudjuk őket.

Ahogy említettem, maguk a szótár fájlok, amelyek a **lang** mappában helyezkednek el, kulcs-érték pár alapon működnek. A Laravel (vagy majd a későbbiekben: a felhasználó) aktuálisan kiválasztott nyelve szerint a nézetekben meg fogja jeleníteni a kulcshoz tartozó értéket. Hozzunk létre most egy **hu** nevű mappát a **lang** mappában. Példaként vegyünk egy új **example.php** fájlt, amit az **en** és **hu** mappában is létrehozunk, a tartalma pedig a következő (az **en** mappában):

```
<?php

return [
  'home' => 'Home',
];
```

15–3. kódrészlet: example.php nyelvi fájl (angol – en mappában lévő) tartalma

A **hu** mappában lévő **example.php** fájl tartalma ugyanez legyen, kivéve a jobb oldali *'Home'* érték, amit írjunk át *'Kezdőoldal'*-ra.

A 'home' kulcshoz tartozó 'Home' értéket akkor fogja megjeleníteni a nézet, ha a Blade sablon motor által kapott kiíratást használjuk, benne pedig a **__()** metódust. Tegyük ezt meg **pastel-layout.blade.php** sablonunkban a menüstruktúra első elemében a 'Kezdőoldal' helyett írjuk be ezt:

```
{{ __('example.home') }}
```

15–4. kódrészlet: Többnyelvű menüpont elhelyezése a sablon fájlban

A **__()** metódus szöveges paraméterében tehát a fájl neve után (.php kiterjesztés nélkül) majd ponttal elválasztva következik a fájlban található kulcs. A nézetben pedig a hozzá kapcsolódó érték fog megjelenni, kiíródni. Viszont ez még nem elég ahhoz, hogy magyarul is jelenjen meg a szöveg, de ha a **config / app.php**-ban a **locale** beállítást megváltoztatjuk **en**-ről **hu**-ra, majd elmentjük, és a böngészőben frissítjük az oldalt, akkor ezekután már magyarul fog megjelenni a nézetünkben a menüpont szövege. Ha

a locale változót de-re írjuk át, majd mentjük és frissítjük az oldalt a böngészőben, akkor ismét angolul fog megjelenni a szöveg, mivel német fordítást, vagyis a **de** könyvtárat és benne az **example.php**-t nem találja hozzá a Laravel a konvenciók szerint, ezért érvénybe lép a **fallback_locale** szerinti nyelvbeállítás, ami jelenleg is még **en**-re van állítva.

A **__()** metódus tehát a meglévő kulcsok között keresi a paraméteréül kapott értéket és a megtalált kulcsnak megfelelő értéket (szöveget) adja vissza az adott nyelven. A kulcsokra nincsen különösebb megszorítás érvényben, tehát tartalmazhat szóközt, írásjeleket stb. Ennek ellenére azonban azt tanácsolom, hogy használjunk jól olvasható és értelmezhető, az angol ABC betűit felhasználó kulcsokat a későbbi problémák elkerülése miatt.

Ha olyan kulcsot használunk, amelyet a **__()** segédmetódus nem talál meg a szótárakban, akkor simán szövegként meg fogja jeleníteni a kulcsot az adott helyen a nézetben.

Mi szükséges ahhoz, hogy valós időben tudjanak a felhasználók nyelvet váltani?

- Nyelvválasztó felület (linkek – esetleg lenyíló – listája);
- Új paraméteres útvonal (vagy Middleware, amely minden felhasználói kéréshez hozzáfűzi a nyelvspecifikus részt), amely beállítja a felhasználónak a nyelvet (**App::setLocale()**);
- Meglévő és majdani útvonalak átstrukturálása lokalizációs specifikusan: például az útvonalban szerepeljen a kiválasztott nyelv kódja.

15.3.4. Paraméterezés

Előfordulhat, hogy a statikus szótárbeli szövegeket dinamikusabbá szeretnénk tenni, például úgy, hogy a felhasználónak az adott nyelven köszönünk és meg is szólítjuk őt a nevén. Ehhez paraméterrel kell ellátni a köszöntő szöveget. A paramétert úgy jelöljük a szótárak szövegében, hogy elé teszünk egy kettőspontot. Kiíratáskor pedig a **__()** segédmetódus második paraméterében lévő tömbbe helyezzük el a paramétert és a hozzá tartozó értéket.

Adjuk hozzá az **en** mappában lévő **example.php**-ben lévő tömbhöz ezt:

```
'greeting' => 'Hi, :name',
```

15–5. kódrészlet: Angol paraméteres köszöntés

Adjuk hozzá a **hu** mappában lévő **example.php**-ben lévő tömbhöz ezt:

```
'greeting' => 'Szia, :name',
```

15–6. kódrészlet: Magyar paraméteres köszöntés

A **welcome** nézet **content** szekcióját módosítsuk erre:

```
{{ __('example.greeting', ['name' => $name])  }}
```

15–7. kódrészlet: Paraméteres köszöntés kiírása a welcome nézetben (a paraméter értéke az útvonaltól érkezik)

Ha most a **config/app.php**-ben változtatjuk a nyelvet a *locale* paraméternél és frissítjük az oldalt a mentés után, akkor láthatjuk, hogy a köszöntés is idomul a kiválasztott nyelvhez a kezdőoldalon.

15.3.5. Többesszám

A többesszám egy kicsit komplexebb probléma, mivel ahány nyelv, annyiféleképpen kezeli a többesszámot. A szótárak szövegezésénél a | karakterrel választjuk el az egyes számú szövegtől a többesszámút.

Példánk alapja a post nézetben lévő karakterszám legyen. Vegyük számba a lehetséges eseteket: mivel elvárás, hogy legalább egy karaktert tartalmazzon egy blogbejegyzés, ezért a 0 karakter hosszú teszteset ki is esik a lehetőségek közül.

1. Az első blogbejegyzés egy karaktert tartalmaz, így a blogbejegyzés hosszúságának megállapításánál egyesszámot kell alkalmaznunk. Teszteléshez: http://127.0.0.1:8000/posts/a
2. Az első blogbejegyzés több karaktert tartalmaz, így a blogbejegyzés hosszúságának megállapításánál többesszámot kell alkalmaznunk. Teszteléshez: http://127.0.0.1:8000/posts/elso-bejegyzes

A szótárak bővítése, előbb az **en** majd a **hu**:

```
'characters' => 'The post contains one character.|The post contains :count
characters.',
```

```
'characters' => 'A bejegyzés egy karakter hosszú.|A bejegyzés :count
karakter hosszú.',
```

15–8. kódrészlet: Angol (előbb) és magyar (utóbb) szótár bejegyzések a blogbejegyzés hosszúságának megállapításához

A szótárakban a **:count** beépített helyőrző határozza meg a konkrét számosságot (többesszám esetén).

A kiíratáshoz a **trans_choice()** segédfüggvényt alkalmazzuk, amely az első paraméterébe ugyanúgy, ahogy a **__()** segédfüggvény, a szótár kulcsát várja el, míg a második paraméterébe azt az értéket, ami meghatározza, hogy egyes- vagy többesszámot kell majd alkalmaznia a kiíratásnál.

A **post.blade.php** nézetben egy újabb bekezdésben (<p>) a tartalmi szekción belül kiírathatjuk a teszteléshez felhasznált linkek szerinti blogbejegyzés karakterhosszúságait:

```
{{ trans_choice('example.characters', $length)}}
```

15–9. kódrészlet: Egyes- vagy többesszám kiíratása a nézetben (számosságfüggően)

A nyelvváltáshoz ismét használjuk a **config / app.php**-ban lévő **locale** paraméter manuális módosításait, majd teszteljük a fenti weblinkek segítségével az alkalmazást.

Lehetőségünk van még teljesen testreszabni, csoportokat kialakítani ahhoz, hogy mikor mit írjunk ki a felhasználóknak. Alakítsuk át a fenti két angol majd a magyar szótárbejegyzéseket az alábbiak szerint:

```
'characters' => '{1} The post contains one character.|[2,10] The post
contains a few characters.|[11,*] The post contains many characters.',
```

```
'characters' => '{1} A bejegyzés egy karakter hosszú.|[2,10] A bejegyzés
néhány karaktert tartalmaz.|[11,*] A bejegyzés sok karaktert tartalmaz.',
```

15–10. kódrészlet: További csoportok kialakítása a "többesszámon" belül

Itt az első esetben (csoportnál) egy konkrét számot határoztunk meg a számosságra, amit kapcsos zárójelek közé kell elhelyezni: {1}. A második csoportnál a számosságot 2 és 10 közé tesszük szögletes zárójelbe, vesszővel elválasztva a minimális és maximális értékeket (a határokat beleértve). A harmadik csoport csak annyiban különbözik a másodiktól jelölési formában, hogy a * karakter jelöli a maximális értéket, ami bármennyi lehet. Különböző weblinkek megnyitásával ezt is tudjuk tesztelni a böngészőben.

Az alfejezetben végrehajtott programkód-módosítások ebben a GitHub commit-ben érhetők el.

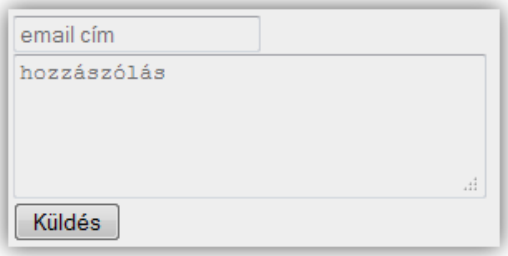
Ha érdekel a téma további folytatása, akkor kövesd a blogomon az #Többnyelvűsítés (Localization) címkével ellátott bejegyzéseket.

15.4. Kiberbiztonság (Cyber Security): CSRF támadások példákkal és a védekezés

A Laravel támogatást nyújt a CSRF (Cross-Site Request Forgery) támadástípus ellen, de talán még nem egészen tudjuk, hogy mi ez, úgyhogy ez az alfejezet ennek megértését fogja segíteni. A gyakorlás során megismerünk néhány példát, aminek segítségével CSRF támadásokat hajtunk végre és megnézzük, hogy a Laravel segítségével hogyan tudunk ellenük védekezni.

Korábban az űrlapok kezelésénél többször használtuk már a **@csrf** Blade direktívát, amivel a támadás ellen tudtunk védekezni, de most nézzünk be a *kulisszák mögé* és vizsgáljuk meg, hogy hogyan is működik ez a támadás típus és hogyan lehet ellene védekezni.

Amennyiben a felhasználótól olyan tartalmat fogadunk be, amelyet más felhasználók is láthatnak, az így befogadott és megjelenített tartalmat nagyon gondosan kell kezelni. Legegyszerűbb példa erre a fórumok, üzenőfalak, vendégkönyvek.



15–11. ábra: Megtámadható űrlap nézete

15.4.1. Cross-Site Scripting (XSS) támadás

Ha egy felhasználói bejegyzés tartalmát nem ellenőrizzük, és ellenőrzés nélkül megjelenítjük más felhasználónál, akkor vajon mi fog történni az alábbi esetben?

15–12. ábra: Támadást szimuláló űrlap kitöltés

Ha a bejegyzést a többi felhasználónak változtatás nélkül megmutatjuk, akkor mindannyiuknál egy ablak fog felugrani a fenti szöveggel *(„ENYÉM VAGY!”).* És ha csak ezt a szöveget látják, még szerencsésen megúszták… Egy támadónak ebben az esetben lehetősége van sütiket (cookie) vagy munkafolyamat azonosítót (session ID-t) lopni, illetve egyéb módokon is „megviccelheti” a többi felhasználót (vagy épp kárt okozhat neki).

XSS esetén tehát egy olyan kód fut le a felhasználó böngészőjében, amelyről se a meglátogatott weboldal, se a felhasználó nem tud. Egy **alert()** persze felhívja magára a figyelmet, de a támadó bolond lenne ilyen módon felfedni magát. A megszerzett session ID vagy sütik felhasználhatók számára, mivel így egy az egyben hozzáfér a felhasználói bejelentkezéshez, tehát ellophatja a felhasználó aktuális munkamenetét (session), amivel az adott weboldalon utána ugyanúgy tud majd viselkedni, mint ha a *„kirabolt”* felhasználó lenne. Itt csak az adott weboldalt érintő sütik és session ID-k vannak veszélyben, de különböző trükkökkel, ennél is több adathoz férhet hozzá a támadó. Léteznek például *„HttpOnly”* sütik, amikhez a script (JavaScript) nem fér hozzá, ezeket érdemes használni.

15.4.2. Cross-Site Scripting (XSS) elleni védekezés

A legegyszerűbb védekezési mód, ha minden felhasználói bemenetet (input-ot) alaposan vizsgálunk és csak azt engedélyezzük, amiről tudjuk, hogy megbízható. A PHP **htmlspecialchars()** függvénye nagyon hasznos védekezés, mivel kihúzza a támadás méregfogát: a beillesztett kódot egyszerű szöveggé alakítja, amely nem hajtódik végre. Amikor a Laravel-ben a Blade nézet fájlokban a **{{ $valtozo }}** kiíratás megtörténik, akkor gyakorlatilag a háttérben egy **htmlspecialchars()** metódushívás hajtódik végre a **$valtozo** paraméterrel. Bonyolultabb eset, amikor a felhasználó bizonyos HTML tartalmat is beszúrhat, amellyel a saját bejegyzését színesítheti. Ebben az esetben fontos, hogy csak olyan HTML tageket engedélyezzünk, amelyeket felsorolunk (pl.: , és hasonló szövegformázások). A helyes HTML tag szűrő mögöttes elgondolása így néz ki: *„Csak azt szabad, amit megengedek”.* Egy rossz szűrő pedig ilyen: *„Mindent szabad, kivéve ami tilos”.*

15.4.3. Cross-Site Request Forgery (XSRF, Laravel-ben CSRF) támadás

Nézzük meg az alábbi kódot:

```
<img src="https://bank.example.com/atutalas?mennyit=100000&kinek=Bob"
width="0" height="0" border="0">
```

15–11. kódrészlet: Rosszindulatú kódrészlet „láthatatlan” képként

Ha egy rosszindulatú felhasználónak sikerül elérnie, hogy egy másik felhasználó betöltsön egy URL-t, akkor máris lehetséges az CSRF támadás. *Mi történik, ha a felhasználó egy másik ablakban be van jelentkezve az e-bankjába és megnyílik a fent jelölt kép?*

Ha a bank oldala nem készült ilyen támadások lekezelésére, akkor a felhasználó a kép megtekintését követően egy „élménnyel gazdagabb" lesz, és néhány ezer dollárral szegényebb. Ebben az esetben tehát nincs feltétlenül szükség egy **<script>** tagre, az automatikusan betöltődő **<img>** tag is épp elegendő.

Bárki a támadás áldozatává válhat, nem csak egy hibás kódot tartalmazó webszerver.

15.4.4. Cross-Site Request Forgery elleni védekezés általában

Mivel ez a támadás típus elsősorban a felhasználó más weboldalakon megnyitott munkamenetei ellen irányul, ezért fontos a védekezés a weboldalunk minden pontján. Mindig ellenőrizzük az alábbiakat:

- Egy fontos műveletet csak POST metóduson keresztül lehessen igénybe venni.
- Ellenőrizzük a kérés (request) HTTP fejlécében **Referer**-t*, ha elérhető (hogy a felhasználó milyen címről jött).
- Sajnos, azonban ezek nem nyújtanak elegendő védelmet (illetve bizonyos tűzfalak szűrik a *„Referer"*-t).

A támadástípus ellen úgy védekezhetünk, ha minden session-höz egy külön azonosítót rendelünk, amelyet bekérünk fontos műveletek végrehajtásakor. Így a támadó kódja nem működhet enélkül.

Távoli eljárások meghívásánál ne csak a sütiket ellenőrizzük, hanem egyéb fejléceket is, például ezeknél legyen szükséges egy külön *„Authorize"* fejléc: ami nem kompatibilis az alap HTTP azonosítással, a kettő nem működőképes együtt, de az alap HTTP azonosítást úgyis csak nagyon egyszerű alkalmazásoknál szokás használni.

Megjegyzés: Referer – helyesen *„referrer",* de a hibás írásmód került be a szabványba, így mindenki a hibás írásmódot követi, amikor a HTTP *„referrer"*-re gondol.

Elvileg tehát még egy GET-re épülő URL is biztonságos lehet abban az esetben, ha egyedi azonosító használatát követeljük meg, például így:

http://bank.example.com/atutalas?mennyit=100000&kinek=Bob&azonosito=EGYEDIKOD

Az „EGYEDIKOD" egy olyan kód legyen, amely minden munkamenet (session) alkalmával más. Így hiába helyez el a támadó egy végrehajtódó URL-t bármely weboldalon, az azonosító hiányában azok nem hajtódhatnak végre.

Megjegyzés: Az **<img>** tagnél az URL-be az egyértelműség kedvéért írhatunk *„&"*-t a *„&"* helyett, de HTML5 esetén a kódunk így is szabályos. Ha a felhasználónak lehetősége van az oldalunkra kódot beszúrni, követeljük meg a tökéletesen helyes kód használatát, máskülönben nem tudhatjuk, hogy a böngésző azt hogyan fogja értelmezni.

Tegyük fel, hogy a weboldalunkról a **logout.php**-val tudunk kijelentkezni, aminek a teljes URL-je:

http://localhost/logout.php

Képzeljük el, hogy egy másik weboldal, mondjuk a http://funny.example.com tartalmaz egy képet:

<img src="http://localhost/logout.php">

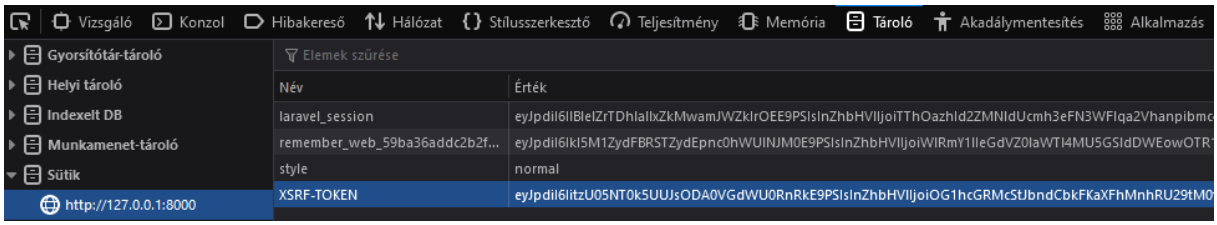
Ha a http://funny.example.com-ot betöltjük, akkor a böngészőnk szolgálatkészen betölti a képeket is. Az egyik kép azonban a logout.php-nkre mutat. A böngészőnek teljesen mindegy, hogy a kép az milyen URL forrással *„jelenik meg”,* így szépen meg fogja hívni, és ha be voltunk jelentkezve a localhost-on, akkor a logout.php most szépen kiléptetett minket. Pedig a http://funny.example.com-nak semmi köze a localhost-hoz, a http://funny.example.com most mégis megviccelt minket. Örüljünk, hogy a bankunk weboldala azért ennél felkészültebb… *Vajon tényleg felkészültebb? Érdemes utánanézni, mert most kezünkben lehet a gyors meggazdagodás kulcsa. :-)*

15.4.5. Cross-Site Request Forgery elleni védekezés a Laravel-ben

Szerencsére már többször belefutottunk abba, hogy mi van, ha az űrlapok elküldésénél, vagy egy weblinkre kattintásnál, vagy tegyük fel, hogy a fenti példában látható img betöltésénél nem történik védekezés. Ezekben az esetekben a Laravel-ben 419-es HTTP státuszkódot (hibakódot) kapunk.

Ha meg szeretnénk vizsgálni, hogy miért is kapjuk ezt a hibakódot, akkor érdemes rákeresni a vendor / symfony / http-foundation/ mappában lévő Response.php-re, mert abban vannak a HTTP státuszkódok felsorolva. Viszont a 419-es nincsen benne, szóval ezt valahol máshol kell keresnünk… A megoldás útja az app / Http / Kernel.php-hoz vezet, azon belül pedig a $middlewareGroups tömbön és a ‘web’-en belül a VerifyCsrfToken middleware-hez. A Middleware-ek ismét feltűnnek (lásd még a 10.1. alfejezetet és a Laravel 11 változásait a Middleware-ek kapcsán), akárcsak a teljeskörű jogosultsági rendszer létrehozásakor. Itt most csak annyit érdemes tudnunk róla, hogy az itt ($middlewareGroups-ban) felsorolt osztályok különböző dolgokért felelősek minden egyes oldalhoz intézett webes kérésnél vagy betöltődésnél. Ha megvizsgáljuk a VerifyCsrfToken.php-t, akkor abban – a korábbi Laravel verziókhoz képest már csak egy egyszerűsített tartalmat találunk, amiben megadhatjuk, hogy melyik útvonal lekérésnél tegyen kivételt a rendszer a CSRF token ellenőrzésénél. Viszont ha megnézzük az osztály ősét (vendor / laravel / framework / src / Illuminate / Foundation / http / Middleware / VerifyCsrfToken.php), akkor ott találunk a handle() metóduson belül egy tokensMatch() metódushívást: ez fogja összeegyeztetni a weboldal lekéréshez (request) tartozó token-t a munkamenet-ben (session-ben) eltárolt token-nel. Ha pedig nem egyeznek, akkor nem tud betöltődni az oldal (dob egy TokenMismatchException-t a rendszer) és hibát kapunk, az említett 419-es HTTP hibakóddal. Ez nekünk tökéletes, hiszen a fejezet korábbi részében megfogalmazott védekezésekkel nem kell foglalkoznunk, a Laravel megteszi ezt helyettünk.

Ha megvizsgáljuk valamelyik adatküldő create vagy edit nézet fájlunkat, akkor láthatjuk, hogy az űrlapunkban benne van a @csrf direktívának köszönhetően a rejtett _token nevű mező. Ezt meg is vizsgálhatjuk a böngészőben a form-ra kattintva jobb egérgombbal → Vizsgálat (Inspect) → utána ott is találjuk a hidden input _token elemet, aminek az értéke egy hosszú véletlen számokból és betűkből álló karaktersorozat. Ez megegyezik azzal a karaktersorozattal, amit a rendszer eltárolt már a munkamenetben (session-ben) is. Amikor elküldjük az űrlapot, akkor ez a _token mező is elküldésre kerül a szerver oldalnak, így ott megtörténhet a token-ek összeegyeztetése és csak akkor folytatódik az új oldal betöltődése, ha a token-ek megegyeztek.



15–13. ábra: Sütik megtekintése a böngészőben (Firefox)

A böngészőben a vizsgálat során a Tároló lapfülön tekinthetők meg a Sütik, az azon belüli XSRF-TOKEN a CSRF token kódolt változata.

Ezért, ha (POST, PUT, PATCH vagy DELETE metódussal történő elküldést végrehajtó) form-okkal dolgozzunk, akkor mindig automatikusan tegyük bele a **@csrf** direktívát, különben 419-es hibát fogunk kapni az elküldésnél.

15.4.6. OWASP Top 10

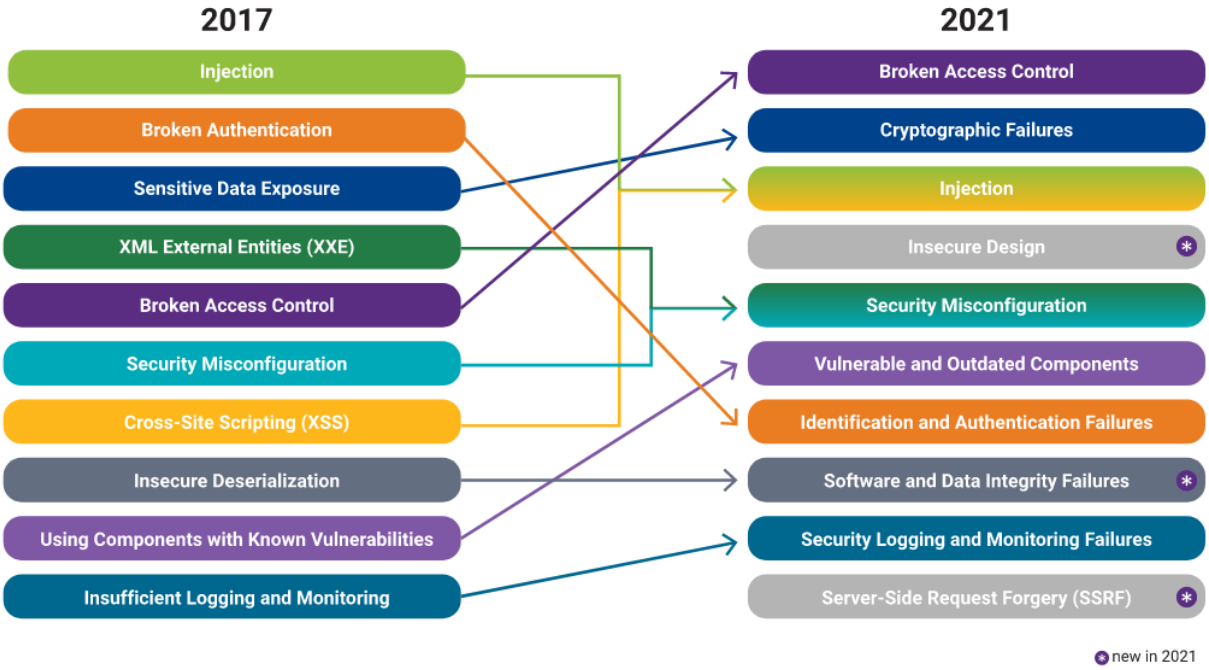
Az Open Worldwide Application Security Project (OWASP) egy nonprofit alapítvány, amely a szoftverek javítását tűzte ki céljául a kiberbiztonság szempontjából. A szervezet *„nyílt közösségi"* modell szerint működik, ami azt jelenti, hogy bárki részt vehet benne, és hozzájárulhat az OWASP-vel kapcsolatos online beszélgetésekhez, projektekhez és egyebekhez. Az OWASP az online eszközöktől és videóktól kezdve a fórumokig és eseményekig mindenben biztosítja, hogy a kínálata ingyenes és könnyen hozzáférhető maradjon a weboldalán keresztül: https://owasp.org/

A szervezet időről-időre (2003 óta) elkészíti a jelentését a legkritikusabb biztonsági kockázatot jelentő támadásokról és a kockázatokat minimalizáló védekező mechanizmusokról. A jelentés az OWASP nyílt közösségi közreműködőinek széleskörű tudását és tapasztalatát felhasználva a világ minden tájáról érkező biztonsági szakértők konszenzusán alapul. A kockázatokat a felfedezett biztonsági hibák gyakorisága, a feltárt sebezhetőségek súlyossága és a lehetséges hatások nagyságrendje alapján rangsorolják. A jelentés célja, hogy a fejlesztők és a webes alkalmazások biztonságával foglalkozó szakemberek betekintést nyerjenek a legelterjedtebb biztonsági kockázatokba, hogy a jelentés megállapításait és ajánlásait beépíthessék saját biztonsági gyakorlatukba, és ezáltal minimalizálhassák az ismert kockázatok jelenlétét alkalmazásaikban.

Az OWASP kezel egy TOP 10-es listát készít, kezel és frissít 2-4 évente 2003 óta, ahogy a piaci elvárások, kihívások ezt igénylik. A TOP 10-es lista legutóbbi két verziója 2017-ben és 2021-ben készült el (15–14. ábra). 2021-ben a lista három új elemet tartalmazott, négy kategória elnevezése és terjedelme változott, illetve az egyes listaelemeknél további finomhangolásra is sor került: azoknak részletes leírásával, és a velük szemben történő védekezési technikákkal kapcsolatban.

A lista jelentősége abban rejlik, hogy a világ legnagyobb szervezetei számára ellenőrzőlistaként és belső webalkalmazás-fejlesztési szabványként szolgál. Az ellenőrök gyakran úgy tekintenek arra, hogy ha egy szervezet nem foglalkozik az OWASP Top 10-es listájával, az arra utal, hogy más megfelelőségi szabványok tekintetében is elmaradhat. Ezzel szemben a Top 10 lista beépítése a szoftverfejlesztési

életciklusba (SDLC[37]) azt mutatja, hogy a szervezet általánosan elkötelezett a biztonságos fejlesztés iparági legjobb gyakorlatai iránt.



15–14. ábra: OWASP Top 10 lista 2017-ben és 2021-ben és a listaelemek átrendeződése (Forrás: [9])

[37] Software Development Lifecycle Management, szoftverfejlesztési életciklus menedzsment

16. Összefoglalás és továbblépés (Summary and further steps)

Ezen a ponton visszatekinthetünk a *„1.3. Miről fogunk tanulni konkrétan?"* alfejezetre, mivel az ott meghatározott tématerületeket megismertük elméleti és gyakorlati szempontból egyaránt. Most már képesek vagyunk magabiztosan és hatékonyan használni a Laravel keretrendszert!

Az MVC tervezési mintában, és ezáltal a Laravel keretrendszerben lévő elemeket, a felhasználói kérések kiszolgálásának folyamata mentén egyre jobban megismertük.

Bár a Laravel főleg backend oldalon működik, azért a frontend oldali működését is elsajátítottuk. Keretes szerkezetek, komponensek és Tailwind CSS keretrendszer segítségével sablonokat is integráltunk a Laravel működésébe.

Adatbáziskezelés szempontjából a Laravel 10-nél alapértelmezett MySQL adatbáziskezelőt, Laravel 11-nél az SQLite adatbázis kezelését, sőt, ezeken kívül még a Microsoft SQL Server-hez való kapcsolódást is megismertük és kipróbáltuk. Az adatdefiníciós, lekérdezési és manipulációs munkákat is elvégeztük a Laravel keretrendszer lehetőségeit, eszközeit felhasználva.

A felhasználóktól érkező adatokat érvényesítettük a kliens és szerver oldalon egyaránt.

Többféle felhasználói hitelesítési csomagot telepítettünk, ezeket kipróbáltuk, a funkcionalitásaikat megismertük, bővítettük és teszteltük is. Ezután a felhasználók engedélyeit is ezekhez a csomagokhoz kötöttük hozzá, így több teljes értékű jogosultsági rendszert is kialakíthattunk.

Webes alkalmazásainkat már képesek vagyunk publikusan elérhetővé tenni akár egy magyar szolgáltató által, akár egy nemzetközi IT cég felhőszolgáltatása által.

Mindeközben folyamatosan hangsúlyt fektettünk a tesztelésre, annak elméleti hátterére és a gyakorlati megvalósítás során alkalmazható eszközökre, valamint a konkrét használatukra is.

A Laravel keretrendszert nem csak önmagában tudjuk ezek után már használni, hanem backend-ként is képesek leszünk működtetni, hiszen az API útvonalakat, funkcionalitásokat, kérések kiszolgálásának folyamatát és tesztelését már ismerjük. Ezeket kiegészítettük a felhasználói hitelesítés és az engedélyezés technikáinak alkalmazásával is.

Végül útmutatást kaphattunk a továbblépési irányok felé, amikor csak lehetett az adott fejezet feldolgozása során. Továbbá *„Laravel-es szemüvegen át"* betekinthettünk még a lokalizáció és a kiberbiztonság tématerületére is.

Projektjeinket végig verziókövetéssel láttuk el, így az elvégzett programkód módosítások mindig elérhetővé váltak az alfejezetek végén.

A dokumentum az elkészülése során folyamatosan ellenőrzés alatt volt több mint 80 hallgató által, illetve a lektor is tüzetesen megvizsgálta a benne lévő tartalmakat.

A Laravel 11 a dokumentum írása során vált elérhetővé, így a különbségeket folyamatosan jeleztem, vagy adott esetben a legújabb verziójú keretrendszerben történt meg a funkcionalitások megvalósítása és bemutatása.

~

Minden sajnos nem fért bele ebbe az elektronikus dokumentumba, amit le szerettem volna írni a Laravel keretrendszer gyakorlati használatáról. Ezért mindenképpen javaslom az Olvasónak, hogy a blogomon (https://attila.gludovatz.hu/blog) kövesse a további újdonságok megosztását, projektek bemutatását.

Köszönöm, hogy elolvastad a könyvemet! Remélem, hogy sokat tanultál belőle, és ha kell, akkor időről-időre elő tudod venni, és utána tudsz benne nézni folyamatoknak, eszközök működésének!

A könyvem elkészítése és publikálása után leginkább Facebook-on leszek aktív a közösségi térben, úgyhogy ott (https://www.facebook.com/gludovatza/) tudod követni a további tevékenységeimet!

Ha értékesnek találod a munkámat, és támogatni szeretnél, akkor ezeken a felületeken biztonságosan és egyszerűen teheted ezt meg:

- Wise: https://wise.com/pay/me/attilad406
- Revolut tag-em: @gludovatzattila
- BuyMeACoffee: https://buymeacoffee.com/gludovatza

17. Hasznos hivatkozások gyűjteménye (Useful links)

Hivatkozás	Weboldal leírása
https://laravel.com/	A Laravel hivatalos weboldala. Elérhető a verziók dokumentációja és különböző, előre beállított környezetek, amelyek még jobban megkönnyítik a fejlesztést és bizonyos célok elérésére jöttek létre úgy, hogy a céloknak megfelelő szolgáltatások előre be vannak állítva bennük. Például a Spark egy számlázó webalkalmazás, amely tartalmazza az ehhez elengedhetetlen funkcionalitásokat, de ott van még például a Laravel Nova, amellyel egy teljes értékű adminisztrációs panelt tudunk használni és így ezek fejlesztésére már nem kell időt fektetnünk. Viszont ezek az előre beállított környezetek többnyire fizetősek, kivétel viszont van, ez a Homestead, vagy a Mac felhasználóknak a Valet.
https://laracasts.com/	A Laracasts oktató videósorozatai nagyban segítették ennek a dokumentumnak az elkészültét. Számos Laravel-hez kapcsolódó ingyenes tanfolyam elérhető, de az előfizetés is megéri, mert a további témákban is el tudunk így mélyedni. Több száz tanuló lecke és sorozat elérhető videó formátumban, kiegészítve a közösség hozzászólásaival.
https://laraveldaily.com/ https://www.youtube.com/c/LaravelDaily	Számos kurzus, hasznos tutorial videó sorozat elérhető főleg Povilas Korop bemutatásában. Egy népszerű Youtube-csatornát is vezet, ahol naponta jelennek meg friss hírek, információk, bemutatók, tanácsadások a Laravel fejlesztéssel kapcsolatban. A videók ingyenesek és néhány kurzus is, de a fizetős tartalmakért is érdemes előfizetővé válni. Ha lemaradnánk valamilyen napi tartalmáról, akkor a heti hírlevelében összegzi az elmúlt hét eseményeit, tartalmait.
https://laravel-news.com/	A Laravel-lel kapcsolatos legfrissebb információkat osztják itt meg blog formájában. Bemutatnak itt olyan külső féltől származó kiterjesztéseket, amik egy specifikus problémára egy előre beállított és felkészített megoldást nyújtanak (mint például: webshop, beépített Google Fordító, e-mail és hírlevél kezelő stb. De elérhetők itt tutorial-ok, valamint podcast-ok is.

https://larajobs.com/	Ha Laravel fejlesztőként keresünk munkát, akkor érdemes ezt a specifikus weboldalt meglátogatni és itt releváns találatokat kaphatunk a lehetséges állásokra.
https://forge.laravel.com/	Ha Laravel alapú webalkalmazást szeretnénk futtatni, de még nem tudjuk, hogy melyik hosting szolgáltatót válasszuk, akkor ezzel szerver nélkül, a felhőben képesek vagyunk ilyeneket létrehozni. A felhőszolgáltatásoknak számos előnye van, ezeknek érdemes utánanézni, mielőtt belevágnánk éles weboldalak futtatásába.
https://envoyer.io/	Szintén egy olyan szolgáltatás, aminek a segítségével Laravel alkalmazásokat tudunk futtatni élesben, bárki által elérhetően.
https://laravel.io/	Laravel-specifikus problémák megoldására, megbeszélésére létrehozott oldal. Céljaik között szerepel a tudás megosztása és a közösség építése is.
https://github.com/alexeymezenin/laravel-best-practices#contents	Alkalmazásfejlesztési és programtervezési tanácsok, legjobb technikák, névkonvenciók gyűjteménye, amelyekben a Laravel fejlesztői közössége megegyezett.
https://dbeaver.io/	Univerzális adatbázis-kezelő rendszert menedzselő alkalmazás, amellyel számos adatbáziskiszolgálóhoz tudunk kapcsolódni és kezelni az adatbázisaikat. Ingyenes, nyílt forráskódú verziója elérhető.
https://github.com/LaravelDaily/Best-Laravel-Packages https://laraveldaily.com/packages	A *„legjobb”* Laravel csomagok, amelyek használata megkönnyítheti a mindennapi munkánkat.
https://developer.mozilla.org/en-US/docs/Web/HTTP/Status https://www.awh.hu/kb/webtarhely/http-hiba-es-allapotkodok-es-azok-jelentesei	HTTP státuszkódok, állapotkódok listája és magyarázatuk angolul és magyarul.

18. Irodalomjegyzék (References)

[1] Stack Overflow Trends – Backend web development MVC frameworks, https://insights.stackoverflow.com/trends?tags=laravel%2Casp.net-mvc%2Cspring-boot%2Cnode.js%2Cdjango (Utolsó megtekintés dátuma: 2023. 02. 20.)

[2] Stack Overflow Trends – PHP-based backend web development MVC frameworks, https://insights.stackoverflow.com/trends?tags=laravel%2Csymfony%2Ccodeigniter%2Ccakephp%2Cyii%2Czend-framework (Utolsó megtekintés dátuma: 2023. 02. 20.)

[3] Google Trends – PHP-based backend web development MVC frameworks, https://trends.google.com/trends/explore?date=today%205-y&q=Laravel,symfony,zend%20framework,CodeIgniter,CakePHP (Utolsó megtekintés dátuma: 2023. 02. 21.)

[4] Kusper G., Kovács L., Ficsor L., Krizsán Z.: Szoftvertesztelés, https://aries.ektf.hu/~gkusper/SzoftverTeszteles.pdf (Utolsó megtekintés dátuma: 2023. 03. 03.), illetve itt is elérhető: https://gyires.inf.unideb.hu/KMITT/c12/index.html (Utolsó megtekintés dátuma: 2024. 06. 30.)

[5] Kósa M., Pánovics J.: Fejezetek az adatbázisrendszerek elméletéből: A válogatott fejezetek Ramez Elmasri és Shamkant B. Navathe Fundamentals of Database Systems című könyve alapján készültek, 2011, https://people.inf.elte.hu/kiss/DB/0046_fejezetek_az_adatbazisrendszerek_elmeletebol.pdf (Utolsó megtekintés dátuma: 2023. 07. 04.)

[6] Postman: API verziókezelés, https://www.postman.com/api-platform/api-versioning/ (Utolsó megtekintés dátuma: 2024. 03. 02.)

[7] HTTP Authentication, https://developer.mozilla.org/en-US/docs/Web/HTTP/Authentication (Utolsó megtekintés dátuma: 2024. 04. 21.)

[8] What is CI/CD?, https://www.synopsys.com/glossary/what-is-cicd.html (Utolsó megtekintés dátuma: 2024. 05. 24.)

[9] What is OASP Top 10?, https://www.synopsys.com/glossary/what-is-owasp-top-10.html (Utolsó megtekintés dátuma: 2024. 05. 10.)

19. Ábrajegyzék (List of figures)

20. Táblázatjegyzék (List of tables)

21. Kódrészletjegyzék (List of code lines)

22. Utasításjegyzék (List of commands)

A Tinker-ben végrehajtott utasítások összesítése található meg itt.

23. Újdonságjegyzék (List of new features)

Egy összesítés arról, hogy a Laravel 10-hez képest milyen újítások kerültek bele a 11-es verzióba.

Kedves Olvasó!

Fedezd fel a webfejlesztés új dimenzióját a Laravel keretrendszerrel! A Laravel nem csupán egy szimpla eszköz a webes alkalmazások létrehozásához, hanem egy teljes körű megoldás, amely ötvözi az egyszerűséget és az erőteljes funkcionalitást.

Ismerd meg a modern programozás varázslatos világát, ahol a kreativitás és a technológia összefonódik, hogy lenyűgöző online élményeket hozzon létre. Engedd, hogy a Laravel inspiráljon és új utakra tereljen a webfejlesztés izgalmas világában! Fedezd fel a tiszta kód ízét, és merülj el a kifinomult megoldások világában, amelyek segítségével könnyedén kifejezheted ötleteidet és publikálhatod azokat a világhálóra. Vágj bele az élménytervezés, a hatékony adatkezelés és a dinamikus tartalom létrehozásának izgalmas világába, ahol a határokat csak a képzeleted szabják meg... a Laravel nem!