**Laravel 11 - Alapok**

Laravel használata az alapoktól egy egyszerű CRUD műveletig

2024.11.

Tartalom

[1. PHP keretrendszerek 2](#_Toc181906238)

[1.1 CakePHP 2](#_Toc181906239)

[1.2. CodeIgniter 2](#_Toc181906240)

[1.3. Laravel 2](#_Toc181906241)

[1.4. Symfony 3](#_Toc181906242)

[1.5. Zend Framework (ma Laminas) 3](#_Toc181906243)

[2. Miért Laravel? 4](#_Toc181906244)

[3. VSC extensions 4](#_Toc181906245)

[4. MVC – modell nézet vezérlő 5](#_Toc181906246)

[5. A Laravel alapjai 6](#_Toc181906247)

[2.1. Mi az az artisan? 7](#_Toc181906248)

[1.2. Model – Az Adatok és a Logika Kezelése 7](#_Toc181906249)

[1.3. View – Az Adatok Megjelenítése 8](#_Toc181906250)

[1.4. Controller – A Közvetítő Réteg 8](#_Toc181906251)

[1.5. Routing – útvonal kezelés 9](#_Toc181906252)

[7. Migration 12](#_Toc181906253)

[7.1. Schema 15](#_Toc181906254)

[8. Public mappa 17](#_Toc181906255)

[9. Blade template 17](#_Toc181906256)

[9.1. Változók kiíratása 17](#_Toc181906257)

[9.2. Feltételek használata 18](#_Toc181906258)

[9.3. Ciklusok használata 18](#_Toc181906259)

[9.4. Blade Öröklődés és Kiterjesztés 18](#_Toc181906260)

[9.5. Nézet kiterjesztése egy másik sablonban 20](#_Toc181906261)

# PHP keretrendszerek

pl: CakePhp, Codelgniter, Laravel, Symfony, Zend Framework

## CakePHP

* **Előnyök**:
  + **Gyors fejlesztés**: Beépített CRUD (Create, Read, Update, Delete) generátor, amely gyorsítja az alapvető alkalmazáskomponensek létrehozását.
  + **Egyszerű konfiguráció**: Nincs szükség sok előzetes konfigurációra.
  + **Beépített biztonsági eszközök**: Alapértelmezetten támogatott az SQL injection és XSS támadások elleni védelem.
* **Hátrányok**:
  + **Kevésbé rugalmas**: A szoftverstruktúra és a mappák elrendezése szigorúan szabályozott, ami nehezítheti az egyedi megoldások beépítését.
  + **Dokumentáció és közösség mérete**: Bár aktív közössége van, kisebb és kevesebb erőforrással rendelkezik, mint a Laravel vagy Symfony. Mivel kijött az újabb verziója, és nem találtam megoldást a régi feladataim tovább fejlesztésére én el is engedtem végleg, de egy 3-as verziót még tudok mutatni, utoljára két éve csináltunk végig egy feladatot vele.
* **Elterjedtség**: Mérsékelt népszerűséggel bír, leginkább kis- és középvállalati projektekhez használják.
* **Népszerűség**: Csökkent az utóbbi években, de továbbra is használt.

## CodeIgniter

* **Előnyök**:
  + **Könnyű és gyors**: Nagyon gyors és könnyű, ideális egyszerű, nagy teljesítményű alkalmazásokhoz.
  + **Kisebb tanulási görbe**: Könnyen megtanulható és használható, minimális függőségekkel.
  + **Rugalmas struktúra**: Nincsenek szigorú szabályok az alkalmazás felépítésére, így többféleképpen is kialakítható.
* **Hátrányok**:
  + **Hiányos beépített funkciók**: Kevesebb beépített eszközzel rendelkezik, mint más keretrendszerek, például ORM és automatikus validáció hiányzik.
  + **Korlátozott közösségi támogatás**: A közössége kisebb, mint a Laravel vagy Symfony esetében.
* **Elterjedtség**: Használt, különösen olyan fejlesztők körében, akik egyszerű és gyors megoldást keresnek.
* **Népszerűség**: Az elmúlt években csökkent, de még mindig népszerű az egyszerűbb projektekhez.

## Laravel

* **Előnyök**:
  + **Gazdag funkcionalitás**: Számos beépített funkcióval rendelkezik, például Eloquent ORM, automatikus validáció, kényelmes routing, valamint Blade sablonmotor.
  + **Erős közösség és dokumentáció**: Laravel rendelkezik az egyik legnagyobb és legaktívabb közösséggel, rengeteg dokumentáció és oktatási anyag áll rendelkezésre.
  + **Könnyű integráció modern technológiákkal**: Laravel könnyen használható Vue.js-sel, Reacttal és modern front-end eszközökkel.
  + **Erős támogatás a RESTful és API fejlesztéshez**.
* **Hátrányok**:
  + **Teljesítményigény**: Laravel nagyobb szerver-erőforrásokat igényel, mint más könnyebb keretrendszerek, mint a CodeIgniter.
  + **Nagyobb tanulási görbe kezdőknek**: Az újonnan tanulók számára eleinte bonyolult lehet, mivel rengeteg funkcióval rendelkezik.
* **Elterjedtség**: A legnépszerűbb PHP keretrendszer, mind a kis, mind a nagy projektekhez, valamint az egyéni fejlesztők és nagyobb fejlesztői csapatok körében is elterjedt.
* **Népszerűség**: Jelenleg a legnépszerűbb PHP keretrendszer a piacon, ezért ezt választottam, és Nektek is ezt fogom megmutatni.

## Symfony

* **Előnyök**:
  + **Modularitás és újrahasználhatóság**: A keretrendszer rendkívül moduláris, számos különálló csomagból áll, amelyeket külön-külön is lehet használni.
  + **Erős közösség és nagyvállalati támogatás**: Nagyvállalatok által is használt, valamint az egyik legszélesebb körben támogatott PHP keretrendszer.
  + **Szabványok követése**: Symfony szigorúan betartja az ipari szabványokat, ami könnyebbé teszi a kód újrahasznosítását más projektekben.
* **Hátrányok**:
  + **Komplexitás**: Nagyobb tanulási görbe és bonyolultabb beállítási folyamat, különösen kezdők számára.
  + **Magasabb erőforrásigény**: Nagyobb erőforrásigénnyel bír, ami nagyobb terhelés mellett csökkentheti a teljesítményt.
* **Elterjedtség**: Különösen nagyvállalati alkalmazásokban és olyan projektekben használják, ahol fontos a szabványok betartása.
* **Népszerűség**: Nagyon elterjedt a komplexebb, hosszabb távú projekteknél.

## Zend Framework (ma Laminas)

* **Előnyök**:
  + **Stabil és robusztus**: Nagyvállalati szintű alkalmazások fejlesztésére készült, különösen erős a teljesítmény és biztonság terén.
  + **Moduláris kialakítás**: Minden komponens külön csomagként érhető el, így szükség szerint használhatók.
  + **Nagyvállalati támogatás**: Laminasként folytatódott a fejlesztése, és vállalati környezetben népszerű.
* **Hátrányok**:
  + **Magas tanulási görbe**: Az összetettség miatt lassabban tanulható meg, és sok beállítást igényel.
  + **Kisebb közösség**: A Zend Framework neve alatt már nem folytatódik a fejlesztés, helyette a Laminas projekt létezik, így a régi Zend közösség szétszóródott.
* **Elterjedtség**: Több nagyvállalat is használja, főként olyan helyeken, ahol stabil és skálázható megoldásra van szükség.
* **Népszerűség**: A Laminasra való áttérés óta a népszerűsége csökkent, de továbbra is elismert nagyvállalati keretrendszer.

**Általános Elterjedtség és Népszerűség**

A jelenlegi piaci trendek alapján **Laravel** a legnépszerűbb és legelterjedtebb PHP keretrendszer, köszönhetően a könnyű tanulhatóságának, az erős közösségi támogatásnak, valamint a széles körű funkcióknak. A **Symfony** szintén elterjedt, különösen nagyvállalati környezetben és komplexebb projektekben. A **CodeIgniter** és **CakePHP** kisebb projektekhez és egyszerűbb alkalmazásokhoz használatosak, míg a **Laminas** a nagyvállalati megoldások között található meg, de népszerűsége mérséklődött.

**Feladat : Sorolj fel néhány PHP keretrendszert, mik az előnyeik?**

1. Miért Laravel?

A Laravel-t egy „progresszív” keretrendszernek hívják. Ez azt jelenti, hogy a Laravel együtt fejlődik veled. Ha éppen csak most teszed meg az első lépéseket a webfejlesztés világában, a Laravel átfogó dokumentációja, útmutatói és videós anyagai segítenek megtanulni az alapokat, anélkül hogy túlterhelnének. <https://laravel.com/docs/11.x>

Haladóknak a Laravel számos robusztus eszközt kínál a függőségkezeléshez, az egységteszteléshez, a sorok kezeléséhez, valós idejű eseményekhez, és még sok máshoz. A Laravel finomhangolt, professzionális webalkalmazások építésére alkalmas, és készen áll nagyvállalati terhelések kezelésére is.

A Laravel keretrendszer skálázhatósága valóban kiemelkedő, ami miatt nagy terhelésű alkalmazásokhoz is ideális. A PHP eleve jól skálázható, és a Laravel további eszközöket és integrációkat kínál a horizontális skálázás támogatására, ami azt jelenti, hogy a szerverek számát lehet bővíteni, így az alkalmazás könnyedén kezelhet nagy terhelést is.

A Laravel egyesíti a PHP ökoszisztéma legjobb csomagjait, hogy a lehető leghatékonyabb és fejlesztőbarátabb keretrendszert kínálja. Ezen felül a világ minden tájáról érkező tehetséges fejlesztők ezrei járultak hozzá a keretrendszer fejlesztéséhez. Ki tudja, talán egyszer te is a Laravel közösség hozzájáruló tagjává válsz.

# VSC extensions

Érdemes pár extensiont hozzá adni a VSC-hez; Laravel Blade Snipets (Winnie Lin); Laravel Blade Spacer (Austen Cameron); Laravel Blade (Christian Howe).

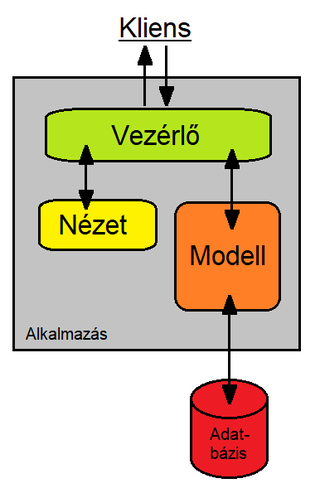
Laragon : <https://laragon.org/> - otthon kipróbálhatod, nagyon jó kis fejlesztői környezet, nincs hozzá szükség XAMPP-ra csak a portok ütközésére vigyázz, ne indítsd el őket egyszerre.

Új fejlesztői környezet: <https://herd.laravel.com/windows>

Ebben a videóban 14 percben bemutatják szuperül: TechAtival-csatornája

<https://www.youtube.com/watch?v=uvJedYUQdLw>

# MVC – modell nézet vezérlő

****[**https://hu.wikipedia.org/wiki/Modell-n%C3%A9zet-vez%C3%A9rl%C5%91**](https://hu.wikipedia.org/wiki/Modell-n%C3%A9zet-vez%C3%A9rl%C5%91)

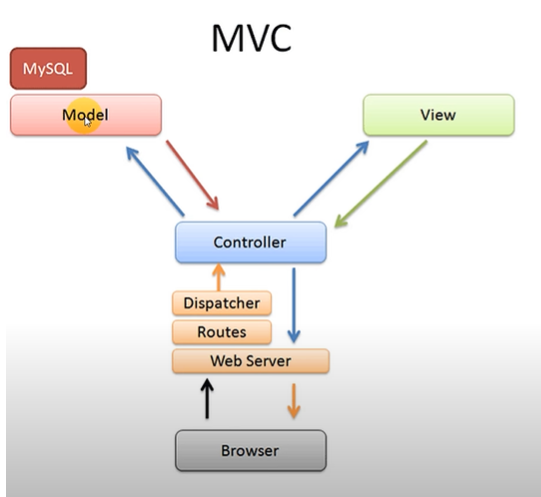
Az információt az adatot jelenti, illetve az adatkezelés szabályait tartalmazza

felhasználói felület

a controller össze kapcsolja a view-t a nézetet és a modellt, ami a felhasználóval is tartja a kapcsolatot, vagyis a usertől érkező akciók, billentyű leütések, illetve egér kattintások, ezeket feldolgozza és lekérdezéseket továbbít, a modellhez, és amit a modelltől vissza kap, azt továbbítja a viewnek, azaz a megjelenítésnek.

A böngésző két irányú kommunikációt folytat a szerverünkkel, egyrészt küld kéréseket, másrészt kap válaszokat. Elküldi a kérést a webszervernek, az átmegy bizonyos részeken és az ún controllerek, controller osztályok azok, amelyek megkapják ezeket a kéréséket. A kontroller azt csinálja, hogy egyrészt kapcsolatba lép az adatbázissal, ezt a modell osztályon keresztül teszi meg, és a modell osztályban vannak azok a függvények, amelyek lekérdezik az adatbázisból az adatokat, de semmi mással nem foglalkozik csak adatlekérdezéssel. Lekérdezi, írja, olvassa, updateli, törli az adatokat, tehát csak az adatbázissal foglalkozik.

Lekéri az adatokat a controller az adatbázisból, majd utána beolvassa a nézetet, az adatokat elhelyezi abban az állományban, amit a nézet létrehozott neki, majd ezután küldi a webszervernek, az pedig a böngészőnek vissza küldi az elkészült dolgokat. Tehát a controller az olyan, mint egy karmester a zenekarban, ő irányítja az adatbázis, ő irányítja a megjelenítést. A program kódot nagyon könnyű ez által karban tartani, vagy akár tovább fejleszteni, ha így kerül felépítésre egy rendszer. Ugyanis ha valamit szeretnénk változtatni az adatkezelésen, nem kell az egész kódot átböngésznünk, csak a modellel kell foglalkoznunk. Ha a nézeten a view-en szeretnénk változtatni akkor csak a nézettel kell foglalkoznunk, nem kell az egész kódot újra reorganizálni, átdolgozni ahhoz, hogy megváltoztassuk az alkalmazásunkat, és ez ugyanígy igaz a controllerre is.



Az **MVC (Model-View-Controller)** egy népszerű szoftverarchitektúra, amelyet a Laravel és más keretrendszerek használnak. Az MVC célja, hogy a kód struktúráját rendezetten és jól kezelhetően tartsa, azáltal, hogy a kódot három különálló rétegbe szervezi: **Model**, **View**, és **Controller**. Lássuk, hogyan működnek ezek együtt:

**Kérdés: Hogyan Működik Együtt Az MVC?**

* A felhasználó egy URL-t (például /users) kér le az alkalmazásból.
* A Laravel route-ok alapján meghívja a megfelelő controllert (pl. UserController).
* A controller meghívja a modellt, hogy hozzáférjen a szükséges adatokhoz.
* Az adatokkal a controller betölti a megfelelő nézetet, és a view-n keresztül megjeleníti azokat a felhasználónak.

Az MVC megközelítésnek köszönhetően a különböző rétegek (adatkezelés, üzleti logika, megjelenítés) elkülönülnek, így könnyebb a kód olvasása, karbantartása és tesztelése.

# A Laravel alapjai

A Laravel egy MVC keretrendszer. Az MVC rövidítés a Model-View-Controller hármast fedi. Röviden-tömören annyit jelent, hogy három alapvető részre bontja az alkalmazásodat. Ezen felül rengeteg kisebb-nagyobb része van a keretrendszernek, ami megkönnyíti a munkádat.

## Mi az az artisan?

**Artisan** a Laravel beépített parancssori felülete, amely megkönnyíti a különféle fejlesztési feladatokat. Ezzel a CLI eszközzel gyorsan generálhatunk különböző fájlokat (mint például vezérlők, modellek, migrációk), kezelhetjük az adatbázis-migrációkat, és számos más feladatot végezhetünk el egyetlen parancs kiadásával.

**Néhány fontos Artisan parancs**

1. **php artisan list**Felsorolja az összes elérhető Artisan parancsot.
2. **php artisan make:controller ControllerNeve**  
   Létrehoz egy új controllert az app/Http/Controllers könyvtárban.
3. **php artisan make:model ModelNeve**  
   Létrehoz egy modellt az app/Models könyvtárban. Az -m opció hozzáadásával egy kapcsolódó migrációs fájlt is készít.
4. **php artisan migrate**  
   Futtatja az adatbázis-migrációkat, létrehozva az adatbázis-táblákat és azok oszlopait.
5. **php artisan serve**  
   Elindít egy helyi fejlesztői szervert, általában a http://127.0.0.1:8000 címen.
6. **php artisan route:list**  
   Megjeleníti az összes elérhető útvonalat, megmutatva azok metódusait, URL-jeit és a hozzájuk tartozó vezérlőket.

**Miért hasznos az Artisan?**

Az Artisan parancsok jelentősen felgyorsítják a fejlesztést, hiszen automatizálják az ismétlődő feladatokat, és segítenek a projekt struktúrájának gyors létrehozásában.

**Feladat: Sorolj fel néhány artisan parancsot! pl: szerver indítása, migráció, controller létrehozása**

## Model – Az Adatok és a Logika Kezelése

A Model felelős az alkalmazás adatainak kezeléséért. Ez magába foglalja az adatbázishoz való kapcsolódást, az adatok lekérdezését, mentését és frissítését. Laravelben az Eloquent ORM-et (Object-Relational Mapping) használjuk, amely egyszerűsíti a különböző adatbázisműveletek végrehajtását.

* Példa: Képzeljük el, hogy van egy User model, amely tartalmazza a felhasználói adatokat (pl. name, email).
* A model lehetővé teszi, hogy a User::all() vagy User::find(1) parancsokkal könnyen lehessen adatokat lekérdezni, és Laravel automatikusan az users adatbázistáblához kapcsolódik.

## View – Az Adatok Megjelenítése

A View az, ami a felhasználó számára látható – itt jelenik meg az adat vizuálisan, legyen szó egy egyszerű HTML oldalról vagy egy összetettebb felhasználói felületről. Laravelben a View réteget általában Blade sablonfájlok formájában hozzuk létre a resources/views mappában.

* A View fogadja a controllertől kapott adatokat, majd megjeleníti azokat a felhasználónak.
* Példa: A welcome.blade.php fájl lehet egy egyszerű kezdőoldal, ami megjeleníti a felhasználók nevét.

Tehát ezek a nézetek gyakorlatilag magát a HTML kódot tartalmazzák. A Laravel a nézetekhez a **Blade sablonrendszert** használja. A Blade segítségével tudod a dinamikus adatok kezelését, beszúrását elvégezni, illetve lehetőséget ad különböző vezérlési szerkezetek használatára és más nézetek beszúrására. A nézeteket a resources/views mappában találod. Lényegében az összes html fájl itt található. A blade arra kiváló, hogy a html-t és a php-t kicsit finomabb módon tudjuk együtt használni, mint a natív php esetében. pl ne kelljen minden php kód elé kezdő zárótag, stb.

A resources mappába kerülnek a js-ek, css-ek,

A lang mappába vannak a nyelvesítsére vonatkozó fájlok. Alapértelmezetten angol van benne, ha akarunk magyart, akkor csak egy másolatot kell róla készíteni, és átnevezni hu-ra, és át kell írni benne a szövegeket magyarra. Az app.php fájlunkban, ha a nyelvet magyarra állítjuk, és talál magyar fordítást, akkor az üzeneteket magyarul fogja megjeleníteni.

## Controller – A Közvetítő Réteg

A **Controller** az összekötő kapocs a model és a view között. A controller fogadja a felhasználói kéréseket, meghatározza, milyen adatokat kell előhívni a modelből, majd továbbítja ezeket a nézet felé, hogy a megjelenítés megtörténhessen.

* Példa: A UserController nevű controller tartalmazhat olyan metódusokat, mint index() vagy show($id), amelyek a felhasználók listázását vagy egy adott felhasználó adatainak megjelenítését végzik.
* Egy tipikus folyamat:
  + A felhasználó az /users oldalra lép.
  + A UserController index() metódusa megkapja a kérést, lehívja az összes felhasználói adatot a User modelből, majd elküldi ezeket a users/index.blade.php nézetnek.

Tehát a controller a nézetek és modellek közötti kapcsolatot tartja fent. Minden lekérés egy controllerbe fut be. A controller tartalmazza a program logikáját: itt töltöd be adatbázisból az adatokat, itt számolsz velük, majd a végén átadod egy nézetnek amit megkap a felhasználó. A kontrollerek az app/Http/Controllers mappában találhatóak.

## Routing – útvonal kezelés

A Laravel routing rendszere egy olyan alapvető komponens, amely a beérkező kéréseket a megfelelő erőforrásokhoz irányítja, hogy a felhasználó mindig a kívánt tartalmat lássa.

A router alapvető szerepe az, hogy a weboldal különböző URL-jeire beérkező kéréseket a megfelelő vezérlőkhöz (Controllers) irányítsa. Például, ha a weboldaladon van egy /contact oldal, a routernek meg kell határoznia, hogy melyik vezérlő kezelje a hozzá tartozó kéréseket. Ez azt jelenti, hogy minden URL (route) mögött egy meghatározott logika vagy nézet (view) található, amit a router közvetít.

Laravelben a routeokat a routes könyvtárban található fájlokban hozhatod létre, és ezeket itt irányíthatod különböző vezérlőkhöz vagy nézetekhez. Az alapértelmezett fájl, amelyet a webes útvonalak kezelésére használunk, a web.php. Itt adhatod meg azokat az útvonalakat, amelyek a böngészőből érhetők el.

Nézzünk először néhány nagyon egyszerű beállítást ami segít megérteni az útvonalak megértésést. Ami a welcome nézetre vezet, ez egy alapértelmezett útvonal, ún. default route, ami rossz példa, mert a route-ba normál esetben sosem írunk functiont, mert ezt a route-ot a webszerverünk el casheli, és ez nem fog tudni funcitont cash-elni, csak szöveget, úgyhogy nem így fogjuk ezt csinálni, a routokat a controllerek fogják kiszolgálni, de vizsgálódásra egyenlőre megfelel.

Másoljuk le és kicsit módosítsunk:

Route::get('/Home', function () {

return "<h1>Szia Uram!<h1>";

});

itt a böngészőbe már a crud.test/Home –ot kell beírni, hogy megjelenjen a szöveg.

Ez nem túl praktikus megoldás, ezért használjuk a view-t. A views-t a resources mappán belül találjuk meg. A welcome.blade.php fájlon belül találjuk a welcome page tartalmát.

Amit a 28-ik sor után látunk @ és {} ez a blade szintaxis. A Blade a Laravel által biztosított sablonozó motor. Más népszerű PHP sablonozó motorokkal ellentétben a Blade nem korlátozza a sima PHP kód használatát a nézetekben.

Hozzunk létre egy új fájlt : [home.blade.php](http://home.blade.php)

Módosítsuk az oldalt, írjunk bele egy üdvözlő szöveget és adjuk meg hozzá a helyes routot

Route::get('/Home', function () {

return view('home');

});

a böngészőben a <http://crud.test/Home> hivatkozással tudjuk megnyitni

A routes view-ban megadhatunk változót is második argumentumként:

Route::get('/Home', function () {

return view('home', ['name' => "Zita"]);

});

Majd ezt követően a [home.blade.php-ben](http://home.blade.php-ben) pedig írassuk ki: <h1>Szia <?php echo $name ?></h1>

A változók nevét megadhatjuk göndör zárójelekben is, ugyanúgy fog működni:

<h1>Szia {{ $name }}</h1>

**Controller használatával már az alábbi kódot használhatod: 2024.11.08**

Először is hozzunk létre egy új controllert a következő paranccsal:

php artisan make:controller ContactController

Ez a parancs létrehoz egy új vezérlőt ContactController.php néven az app/Http/Controllers mappában.

Miután létrehoztad a vezérlőt, hozzáadhatod a szükséges metódusokat, például egy show metódust a kapcsolat oldal megjelenítésére:

public function show() { return view('contact'); }

Hozzuk létre a contact.blade.php fájlt a következő tartalommal:

<form action="/contact" method="POST">

@csrf

<label for="name">Név:</label>

<input type="text" name="name" id="name" required>

<label for="email">Email:</label>

<input type="email" name="email" id="email" required>

<label for="message">Üzenet:</label>

<textarea name="message" id="message" required></textarea>

<button type="submit">Küldés</button>

</form>

A @csrf egy Blade direktíva Laravelben, amely az űrlapok biztonságossá tételére szolgál. A **CSRF (Cross-Site Request Forgery)** védelem biztosítása érdekében Laravel minden űrlapbeküldést egyedi tokennel lát el. Ez megakadályozza, hogy más weboldalak rosszindulatú kéréseket küldjenek a szervernek a felhasználó nevében.

A laravel automatikusan létrehoz egy **token**-t minden egyes felhasználói munkamenethez. Amikor egy űrlapot küldünk be, a Laravel elvárja, hogy ez a token szerepeljen az űrlapban, és ellenőrzi, hogy a token megegyezik-e a szerver által elvárttal.

A @csrf Blade direktíva ezt a tokent automatikusan beilleszti az űrlapba egy rejtett mezőben: <input type="hidden" name="\_token" value="a\_csrf\_token\_hash">

Ezután a web.php fájlban hozzá adjuk a szükséges class-t:

use App\Http\Controllers\ContactController;

majd az útvonalat: (*de ha elég jó extensionokat letelepítettél, akkor elég az útvonal, a szükséges osztályt be tudod importálni egy kattintással*)

Route::get('/contact', [ContactController::class, 'show']);

Ez a példa azt mutatja, hogyan irányítja a Laravel a /contact URL-re érkező GET kéréseket a ContactController vezérlő show metódusához.

A laravel controllerei az app mappán belül, a http mappán belül a Controllers mappán belül található controllers.php.

Tehát szintaxis:

php artisan [utasítás] [--paraméterek]

**Objektumok létrehozása:**

php artisan make:[objektum típus] [--paraméterek]

artisan make:controller - -help a helpereket mutatja meg

php artisan make:controller [kontroller\_fájl\_neve]

**Feladat készíts a welcome.blade.php fájlhoz controller-t, és javítsd át hozzá az útvonalat!**

A Laravel támogatja a dinamikus routeokat is, ahol változókat adhatunk át az útvonalban. Például, ha egy konkrét felhasználói profilt szeretnél megjeleníteni:

Route::get('/profile/{id}', [ProfileController::class, 'show']);

Ebben az esetben a {id} egy dinamikus paraméter, amelyet a felhasználó azonosítójaként használhatsz. A Laravel automatikusan kinyeri az URL-ből az értéket, és átadja a show metódusnak.

A Laravel routing rendszere lehetővé teszi a **middleware-ek** beállítását, amelyek egyfajta szűrőként működnek, mielőtt a kérést továbbítaná a vezérlőhöz. A middleware-ek segítségével például ellenőrizheted, hogy a felhasználó be van-e jelentkezve, vagy hogy van-e megfelelő jogosultsága az adott oldal eléréséhez.

Route::get('/dashboard', [DashboardController::class, 'index'])->middleware('auth');

ebben az esetben a dashboard oldal eléréséhez a auth middleware biztosítja, hogy csak a bejelentkezett felhasználók férjenek hozzá a dashboard-hoz.

A Laravel támogatja az összes alapvető HTTP metódust, beleértve a GET, POST, PUT, DELETE és más módszereket. Ezek használatával szabályozhatod, hogy egy-egy route milyen típusú kéréseket fogadjon.

* **GET**: Az adatok lekérésére szolgál. Pl. egy bejelentkezési oldal megjelenítéséhez.
* **POST**: Adatok beküldésére használatos. Pl. egy regisztrációs űrlap adatainak mentésére.
* **PUT/PATCH**: Adatok frissítésére.
* **DELETE**: Adatok törlésére.

Route::get('/posts', [PostController::class, 'index']); // Listázás

Route::post('/posts', [PostController::class, 'store']); // Létrehozás

Route::put('/posts/{id}', [PostController::class, 'update']); // Frissítés

Route::delete('/posts/{id}', [PostController::class, 'destroy']); // Törlés

Kezdetben elegendő a web.php fájlra koncentrálni, amely a böngészőből elérhető, általános webes útvonalakat kezeli. Ebben a fájlban minden útvonal közvetlenül a felhasználók által hozzáférhető URL-ekre vonatkozik, és tartalmazhat statikus oldalakat, dinamikus tartalmakat, vagy vezérlőkhöz rendelt logikát.

Laravel routing rendszere rugalmas és könnyen használható, így segít egyszerűen felépíteni egy komplex, skálázható webes alkalmazást. A web.php segítségével az alkalmazás struktúráját alapozhatod meg, mielőtt továbblépsz az összetettebb route-konfigurációkra és API-útvonalakra.

# Migration

Az adatbázisok kezelése, míg egyedül dolgozol egy projekten, elég egyértelmű. Ha viszont egy komplex projekten csapatnyi fejlesztő dolgozik, problémás lehet a szerkezeti módosítások követése, egymás között frissen tartása. Az adatbázis migrációk a Laravel query builderére építve tartalmazzák az adatbázisok tábláinak, kapcsolatok beállításainak minden egyes lépését, amik oda-vissza futtathatóak, azaz visszaállíthatóak lépésenként, minden egyes műveletet úgy visszaállítva, mintha meg sem történtek volna. Ha minden egyes változtatáshoz új migrációt veszel fel, a társadnak csak annyi dolga van, hogy frissíti az adatbázist a php artisan migrate paranccsal. E mellett azért is nagyon hasznos a migrációk használata, mert így szintén ennek a parancsnak a kiadásával létre tudod hozni a teljes adatbázist az alkalmazásod telepítésekor. Ezen a migrációk a database/migrations/ mappába kerülnek.

Tehát migration-okat használunk ahhoz, hogy az adatbázisod sémáját, kapcsolatokat, kezdő rekordokat felvedd. Új migrationt az artisan segítségével tudsz létrehozni, a projekt gyökérkönyvtárában a következő parancs kiadásával:

***php artisan make:migration create\_names\_table***

Az elnevezést mindig ilyen módon válaszd, tehát létrehozásnál create\_, a tábla neve az adatokra utaló és többes számban van, majd az, amit létre akarsz hozni, esetünkben table. Ha módosítani akarsz a szerkezeten, **create\_ helyett válaszd az alter\_ kifejezést** vagy pl. új mező beszúrásához az add\_mezo\_to\_tabla\_table formát.

Ha kiadtad a parancsot, létrejött a database/migrations/ alatt egy új migration a kért névvel.

A fájlon belül két function jön létre:

**up() function**

Ez a migration futásakor fog lezajlani, ide kell megírnod azt, amit maga a migration csinál: táblák létrehozását, mezők létrehozását, stb. Tudsz puskázni az alapból létrehozott táblákból, pl a user táblából.

Az Artisan okos és ha a fenti névkonvencióval hoztad létre a migrationt, egyből készített is neked egy megfelelő keretet tábla létrehozásra: a nevét is megadta, adott hozzá egy AUTO\_INCREMENT-es elsődleges kulcsot és két speciális mezőt a timestamps segítségével (created\_at és updated\_at). Ezek a létrehozási és módosítási időt tartalmazzák, amiket az Eloquent automatikusan kitölt az adott rekord létrehozásánál és szerkesztésénél.

**down() function**

Ez akkor fut le, ha szeretnéd visszaállítani az adott migrationt, azaz visszaállni az adatbázisban egy régebbi verzióra. Tábla létrehozás esetén ez egyértelműen DROP-ot jelent. Ha pl. csak egy új mezőt hozol létre az up()-ban, csak ennek a mezőnek az eltávolítását kell megírni. Ez egyébként nem kötelező, de javallott, hisz így oda-vissza tudsz lépegetni az adatbázisod sémájának állapotai között.

A php artisan make:migration parancsot használva hozhatunk létre egy új migrációt:

php artisan make:migration create\_names\_table

Keretrendszerek használata esetén az ajánlott formátum, hogy a tábla neveket többes számban hozd létre.

Ez a parancs létrehoz egy új fájlt a database/migrations mappában, amely tartalmazza a create\_names\_table nevű migrációt.

A migrációs fájl két metódust tartalmaz:

* **up()**: Ez a metódus tartalmazza az adatbázis módosításainak logikáját. Ebben a példában egy names táblát hozunk létre, benne néhány alapvető mezővel.

Alapértelmezetten az id-t és az időbélyeget adja hozzá, de itt kiegészítheted:

public function up() {

Schema::create('names', function (Blueprint $table) {

$table->id(); // Elsődleges kulcs (AUTO\_INCREMENT)

$table->string('first\_name');

$table->string('last\_name');

$table->timestamps(); // created\_at és updated\_at mezők

}); }

* **down()**: Ez a metódus visszaállítja az adatbázist az eredeti állapotba, azaz törli a names táblát.

A migráció futtatásához használd a következő parancsot:

php artisan migrate

Ha valamilyen oszlopot elfelejtettél hozzá adni vissza kell vonni a migrációt:

php artisan migrate:rollback

és hozzá adhatsz új oszlopot a táblához:

php artisan make:migration add\_birth\_date\_to\_names\_table

Egészítsd ki az új migrációs fájlod up metódusát:

public function up(){

Schema::table('names', function (Blueprint $table) {

$table->date('birth\_date')->nullable();

});}

Ezt követően lefuttathatod a migrációt és ha megnézed az adatbázisban a táblád oszlopait, láthatod, hogy új oszloppal került kiegészítsére.

## Schema

A Laravel Schema Builder egy eszköz az adatbázis-táblák és -mezők egyszerű és rugalmas létrehozásához, amely különféle adatbázis-kezelőkkel kompatibilis. Ezzel a megoldással könnyen kezelhetők az adatbázis-sémák a Laravelben, így nem kell aggódnod az adatbázis-kezelők közötti eltérések miatt.

**Miért Használjunk Schema Builder-t?**

A Laravel támogat több adatbázis-kezelőt, például MySQL-t, PostgreSQL-t és MSSQL-t. Ezek között különbségek lehetnek a mezők típusainál és karakterlánc-kezelésnél. A Schema Builder használatával Laravel automatikusan kezeli ezeket a különbségeket, így az alkalmazásod ugyanúgy működik bármely támogatott adatbázis-kezelőn.

**Hogyan Működik a Schema Builder?**

A Schema Builder a Schema osztályra épül, amely az adatbázis sémájának kezelését szolgálja. A táblák és mezők létrehozásánál megadhatod:

Az első paraméterként a táblanevet.

A második paraméterként egy ún. Blueprint objektumot, amely tartalmazza a mezők és azok típusainak leírását.

Például a Schema::create('books', function (Blueprint $table) { ... }) létrehozza a books táblát, és az abban lévő mezőket a Blueprint osztályon keresztül definiálhatod.

**Mezők és Típusok Meghatározása**

A Laravel különféle függvényeket kínál a mezőtípusok beállításához, például:

Szám típusú mező: $table->id(); (automatikusan elsődleges kulcs lesz)

Szöveg típusú mező: $table->string('column\_name');

**Példák a Típusok és Paraméterek Beállítására**

Alapvető szöveges mező létrehozása:

$table->string('name');

Ez egy szöveges (VARCHAR) mezőt hoz létre name névvel.

Nullázható mező:

$table->string('description')->nullable(); //Ez a mező üres (null) értéket is felvehet.

Mező hossza és alapértelmezett érték beállítása:

$table->string('status', 100)->default('active');

Ez egy VARCHAR(100) típusú mezőt hoz létre status névvel és active alapértelmezett értékkel.

**Speciális Típusok**

Néhány függvény kifejezetten gyakran használt mezőkre vonatkozik, például:

timestamps(): Két mezőt hoz létre: created\_at és updated\_at, amelyeket a Laravel automatikusan kezel (rekord létrehozásának és módosításának időpontjai).

$table->timestamps();

**1.) Feladat: Könyvtár Adatbázis Migráció Létrehozása -> books tábla létrehozása**

* id: Elsődleges kulcs (automatikusan létrejön a $table->id()-val).
* title: A könyv címe, string típus.
* author: A szerző neve, string típus.
* published\_year: A könyv megjelenésének éve, integer típus.
* genre: A könyv műfaja, string típus.
* created\_at és updated\_at mezők automatikusan (töltsd ki a $table->timestamps() segítségével).

**2.) Feladat: Új mező hozzáadása a books táblához**

* Hozz létre egy új migrációt add\_isbn\_to\_books\_table névvel
* Az up metódusban adj hozzá egy isbn mezőt, amely string típusú, és lehet nullable.

# Public mappa

**Kizárólag ezt az egy mappát látja a webszerver**!!! Minden statikus dolgodat (CSS, JS, képek, ikonok, stb.) a public mappába helyezd el. Itt található a Laravel index fájlja is, tehát ha a későbbiekben éles környezetbe akarod helyezni az alkalmazásod, ez kell legyen a gyökérkönyvtára a webszervernek. Érdemes mindent különböző mappákba helyezni ezen belül (/css, /js, /images, vagy ehhez hasonlóak). Ha hivatkozni akarsz pl. egy képre a /public/images/kep.jpg útvonalon, így kell tenned:<img src="/images/kep.jpg">

# Blade template

A Laravel Blade egy hatékony sablonkezelő rendszer, amely megkönnyíti a dinamikus tartalmak kezelését és a HTML oldalak szerkezetének újrahasznosítását a Laravel alkalmazásokban. A Blade lehetővé teszi, hogy különböző HTML-részeket sablonokká és komponensekké bonts, így egyszerűsíti az ismétlődő részek kezelését és az oldalak gyors felépítését.

**Miért használjunk Blade-et?**

A Blade egyszerű szintaxist biztosít a dinamikus tartalmak megjelenítéséhez, és mivel PHP-kódra fordítódik le, a szerver teljesítménye jobb lesz, mint ha közvetlenül használnánk a HTML-t és a PHP-t. A Blade nézetfájlok .blade.php kiterjesztést használnak, és a resources/views könyvtárban találhatók.

**Blade Alapok és Szintaxis**

## Változók kiíratása

A Blade változókat {{ }} szintaxissal írhatjuk ki, ami automatikusan biztonságos HTML-kimenetet ad. Példa:

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

<p>{{ $description }}</p>

A {{ }} kifejezést konvencionálisan "Blade Echo" vagy "Echo Statement" néven említjük. Laravelben a {{ }} szintaxist arra használjuk, hogy változókat vagy kifejezéseket adjunk ki (echo) biztonságos HTML-kimenetként a nézetekben. A Blade automatikusan védi a {{ }}-en belül megadott értékeket az XSS támadásoktól, így a kimenet HTML biztonságossá válik.

{{ $user->name }}

Ez a kód kiírja a $user->name változó értékét a nézetben, miközben biztonságossá teszi azt.

## Feltételek használata

A Blade segítségével feltételes logikát adhatsz hozzá, mint például @if, @elseif, @else, és @endif:

@if ($user->isAdmin())

<p>Üdv, Admin!</p>

@else

<p>Üdv, User!</p>

@endif

## Ciklusok használata

A Blade ciklusokhoz is nyújt direktívákat, például @foreach, @for, és @while:

<ul>

@foreach ($items as $item)

<li>{{ $item->name }}</li>

@endforeach

</ul>

Ez a ciklus végigmegy az $items tömbön, és minden elem name tulajdonságát megjeleníti egy listában.

## Blade Öröklődés és Kiterjesztés

A Blade lehetővé teszi a sablonöröklést, ahol egy alapvető layoutot használhatsz minden oldalon, majd az egyes oldalakhoz csak az eltérő részeket kell meghatároznod.

1. **Alapértelmezett Layout (alapsablon) létrehozása**  
   Hozz létre egy layoutot, például layouts/main.blade.php néven:

<!DOCTYPE html>

<html lang="hu">

<head>

<meta charset="UTF-8">

<title>@yield('title')</title>

</head>

<body>

@include('partials.header')

<main>@yield('content')</main>

@include('partials.footer')

</body>

</html>

* + **@yield('title')**: Ez a hely az oldal egyedi címének.
  + **@yield('content')**: A fő tartalom rész, amit az egyes aloldalak kiegészíthetnek.

1. **Szervezett Fájlrendszer**

* A layouts mappa tipikusan a nagyobb, teljes oldalelrendezéseket (main, app, stb.) tartalmazza.
* A kisebb, ismétlődő elemek (mint a fejléc vagy lábléc) általában partials vagy shared mappában helyezkednek el, így a layouts mappa áttekinthetőbb marad.

2. **Újrahasználható Részek Különválasztása**

* A partials mappában található kisebb nézetfájlok (pl. header, footer, stb.) könnyen újrahasználhatók a különböző layoutokban és aloldalakban is.
* Ha több layouttal dolgozol, a partials mappa tartalma ugyanúgy elérhető lesz minden layout számára, anélkül hogy ezek a fájlok közvetlenül a layoutok között keverednének.

3. **Olvashatóság és Karbantarthatóság**

* A jól szervezett nézetstruktúra könnyebben átlátható és karbantartható, különösen egy nagyobb projektnél. A partials vagy shared mappa használata segít gyorsabban megtalálni a nézetek elemeit és könnyebben módosítani őket.

A következőképpen hozd létre a fájlokat a teszt projektedben:

resources

└── views

├── layouts

│ └── main.blade.php # Fő layout

├── partials

│ ├── header.blade.php # Fejléc

│ └── footer.blade.php # Lábléc

└── home.blade.php # Kezdőlap

Pl:

<!-- resources/views/partials/header.blade.php -->

<header>

<h2>Weboldal Fejléc</h2>

<nav>

<ul>

<li><a href="/">Kezdőlap</a></li>

<li><a href="/blog">Bejegyzések</a></li>

<li><a href="/contact">Kapcsolat</a></li>

</ul>

</nav>

</header>

és

<!-- resources/views/partials/footer.blade.php -->

<footer>

<p>&copy; 2024 Minden jog fenntartva.</p>

</footer>

* 1. Nézet kiterjesztése egy másik sablonban  
     Ha egy aloldalt készítesz, például egy kezdőlapot, akkor kiterjesztheted a fő sablont az @extends segítségével és meghatározhatod a @section blokkokat:

<!-- resources/views/home.blade.php -->

@extends('layouts.main')

@section('title', 'Kezdőlap')

@section('content')

<h1>Üdvözlünk a kezdőlapon!</h1>

<p>Ez egy példa tartalom.</p>

@endsection

**Feladat: Hozzuk létre a blogot ami a navigációban szerepel!**

Hozz létre egy új nézetfájlt resources/views/blog/index.blade.php néven. Ez a fájl a blogbejegyzések listáját jeleníti meg. A bejegyzéseket egy egyszerű tömbben tárolhatod a teszteléshez.

**index.blade.php:**

@extends('layouts.blog')

@section('title', 'Blog - Bejegyzések')

@section('content')

<h2>Bejegyzések</h2>

<div class="post-list">

@foreach ($posts as $post)

<x-post-card :title="$post['title']" :excerpt="$post['excerpt']" />

@endforeach

</div>

@endsection

**Kártya Komponens Létrehozása**

Készíts egy új komponenst a blogbejegyzések kártyájának megjelenítéséhez. A kártya tartalmazza a bejegyzés címét és egy rövid leírást.

1. Hozz létre egy post-card.blade.php fájlt a resources/views/components mappában:

**post-card.blade.php:**

<div class="post-card">

<h3>{{ $title }}</h3>

<p>{{ $excerpt }}</p>

<a href="#">Olvass tovább</a>

</div>

1. Az x-post-card komponenst a blog/index.blade.php fájlban így tudod használni, paramétereket átadva neki.

**Teszt Adatok Meghatározása a Kontrollerben**

Hozz létre egy egyszerű kontrollert, amely biztosítja a szükséges adatokat a nézethez. Például egy BlogController segítségével.

**BlogController:**

<?php

namespace App\Http\Controllers;

class BlogController extends Controller

{

public function index()

{

$posts = [

['title' => 'Első bejegyzés', 'excerpt' => 'Ez az első bejegyzés rövid leírása.'],

['title' => 'Második bejegyzés', 'excerpt' => 'Ez a második bejegyzés rövid leírása.'],

['title' => 'Harmadik bejegyzés', 'excerpt' => 'Ez a harmadik bejegyzés rövid leírása.'],

];

return view('blog.index', compact('posts'));

}

}

Ne felejtsd el hozzáadni a megfelelő class-t és route-ot a routes/web.php-hoz:

use App\Http\Controllers\BlogController;

Route::get('/blog', [BlogController::class, 'index']);

**6. Lépés: Stílusok és Tesztelés**

Stílusold meg a layoutot, a fejlécet, a láblécet és a kártyákat a **public/css/style.css** fájlban (opcionális). A kód futtatásával a /blog útvonalon kell megjelennie a blogbejegyzések listájának, mindegyik bejegyzés saját kártyaként.

Adjunk hozzá stílust:

/\* Alapértelmezett stílusok \*/

body {

font-family: Arial, sans-serif;

background-color: #f4f4f4;

color: #333;

margin: 0;

padding: 0;

}

header {

background-color: #333;

color: #fff;

padding: 15px;

text-align: center;

}

header h1 {

margin: 0;

font-size: 24px;

}

header nav ul {

list-style: none;

padding: 0;

}

header nav ul li {

display: inline;

margin-right: 15px;

}

header nav ul li a {

color: #fff;

text-decoration: none;

}

header nav ul li a:hover {

text-decoration: underline;

}

footer {

background-color: #333;

color: #fff;

text-align: center;

padding: 10px;

position: fixed;

bottom: 0;

width: 100%;

}

.main-content {

max-width: 800px;

margin: 20px auto;

padding: 20px;

background-color: #fff;

box-shadow: 0 0 10px rgba(0, 0, 0, 0.1);

}

.post-list {

display: flex;

flex-direction: column;

gap: 20px;

}

.post-card {

background-color: #e2e2e2;

padding: 15px;

border-radius: 5px;

box-shadow: 0 2px 5px rgba(0, 0, 0, 0.1);

}

.post-card h3 {

margin-top: 0;

color: #333;

}

.post-card p {

color: #555;

}

.post-card a {

display: inline-block;

margin-top: 10px;

color: #0066cc;

text-decoration: none;

}

.post-card a:hover {

text-decoration: underline;

}

A stílusfájl betöltéséhez adj hozzá egy linket a layouts/main.blade.php fájlban a <head> részbe írd be:

<title>@yield('title', 'Blog')</title>

<!-- CSS stílus betöltése -->

<link rel="stylesheet" href="{{ asset('css/style.css') }}">