

LARAVEL

PREDAVANJA

SADRŽAJ

Ovdje je obuhvaćen dio o osnovama PHP-a, uvod u baze podataka, SQL jezik, napredni MySQL, napredno PHP programiranje, spajanje PHP-a s SQL bazama i testiranje. To je ono što se naziva Vanilla PHP.

Administrator

[Course title]

| | |
|---------------------------------------------------------------------------------------------------------|----|
| Laravel | 9 |
| Instalacija u Windowsima pomoću Composer-a..... | 9 |
| Pokretanje Artisan-a..... | 9 |
| Pokretanje Laravel servera..... | 15 |
| Ekstenzije za Visual Studio Code | 17 |
| Struktura direktorija u Laravel projektu..... | 17 |
| Datoteke u root-u Laravel-u projekta | 18 |
| Instaliranje ovisnosti s Composerom u Lavarelu..... | 20 |
| Artisan inspire – korištenje inspirativnih citata..... | 21 |
| Uklanjanje direktorija i datoteka koje nisu potrebne u projektu..... | 21 |
| Kreiranje prvog Laravel repozitorija..... | 23 |
| Više imenskih prostora (engl. namespace) unutar jedne datoteke | 24 |
| Kako pokupiti Laravel project sa udaljenog repozitorija..... | 25 |
| Kloniranje repozitorija..... | 25 |
| Ulazak u direktorij | 25 |
| Instaliranje ovisnosti | 25 |
| Kopiranje i podešavanje <code>.env</code> datoteke | 26 |
| Generiranje ključa | 26 |
| Brisanje postojeće baze podataka..... | 26 |
| Pokretanje migracija | 26 |
| Pokretanje seeder-a | 26 |
| Pokretanje Artisan servera..... | 26 |
| Ažuriranje (osvježavanje) lokalne kopije Laravel projekta sa novim kodom s udaljenog repozitorija | 26 |
| Ulazak u direktorij projekta..... | 26 |
| Povlačenje koda sa udaljenog repozitorija radi ažuriranja | 26 |
| Instaliranje ovisnosti | 27 |
| Code First i Database First pristup relacionim bazama podataka..... | 27 |
| Code First..... | 27 |
| Database First..... | 27 |
| Monolitne aplikacije i API..... | 28 |
| Monolit..... | 28 |
| API (Aplikacijski programski interface)..... | 28 |
| Monolit aplikacija za rad s postovima..... | 29 |
| Fasade | 29 |
| MySQL migracije..... | 30 |

| | |
|---------------------------------------------------------------------------------------------------|-----|
| Migracije..... | 31 |
| Kako funkcioniraju migracije? | 32 |
| Modifikatori..... | 34 |
| Indeks modifikatori | 34 |
| 0001_01_01_000000_create_users_table.php | 35 |
| Datoteka 0001_01_01_000000_create_users_table.php je Laravel migracija koja kreira nekoliko | 35 |
| Timestamp-ovi – vremenske oznake..... | 36 |
| Soft deleted..... | 36 |
| .env datoteka pregled i podešavanje | 38 |
| Pokretanje Arisan servera i migracije..... | 42 |
| Kreiranje tablice rolâ u Laravelu..... | 46 |
| Modeli | 53 |
| Default model User | 55 |
| dd i dump | 60 |
| Roles i Post modeli | 60 |
| Ispravak postojećih tablica bez rollback-a | 68 |
| Relacije | 73 |
| Seeder-i | 90 |
| Ugrađeni seeder UserFactory.php | 91 |
| Seeder Post | 94 |
| Seeder RoleSeeder | 97 |
| Definiranje kontrolera..... | 100 |
| Rute | 104 |
| Pogledi..... | 106 |
| Kreiranje linkova u Blade..... | 107 |
| Layout opcija | 108 |
| Login forma (obrazac) | 109 |
| Validacija i autentifikacija korisnika | 113 |
| Servisi | 119 |
| Middleware | 132 |
| Osnovna struktura middleware-a | 133 |
| Vlastiti middleware | 137 |
| Ispravljanje pogrešne rute | 142 |
| Odlogiranje s aplikacije dodavanjem Logout dugmeta..... | 142 |
| Bacanje iznimke (engl. query exception) u aplikaciji | 146 |

| | |
|-----------------------------------------------------------------|-----|
| Blade direktive..... | 147 |
| dashboard stranica..... | 147 |
| nav.blade.php stranica..... | 148 |
| CRUD u Laravelu..... | 152 |
| Kontroler resursa (engl. resource controller) | 152 |
| Radnje kojima upravljaju kontroleri resursa | 153 |
| Grupiranje ruta..... | 154 |
| Ponovo uključivanje Posts rute | 155 |
| Ispravak apsolutnih ruta u relativne | 157 |
| Podešavanje CRUD | 164 |
| Kreiranje novog posta | 166 |
| Dugme za editovanje postova | 189 |
| Dugme za brisanje postova | 196 |
| Sortiranje podataka..... | 198 |
| Korištenje paginatora..... | 198 |
| Ograničavanje mijenjanja i pogleda na postove s rolama | 199 |
| Višejezičnost..... | 210 |
| API (Aplikacijski programski interface) za rad s postovima..... | 212 |
| Što je API?..... | 212 |
| Što je RESTful API?..... | 213 |
| Ostali API u Laravelu..... | 217 |
| API backend u Laravelu | 220 |
| Laravel Sanctum | 220 |
| JWT | 220 |
| Kreiranje Laravel projekta | 223 |
| Izgradnja API..... | 228 |
| Autorizacija korisnika | 234 |
| Migracije i modeli..... | 244 |
| Podešavanja | 244 |
| Logout() metoda | 294 |
| Event-i u projektu..... | 296 |
| Event..... | 304 |
| Kako rade event-i u Laravelu? | 304 |
| Struktura event-a u Laravelu..... | 304 |
| Kreiranje event-a..... | 304 |

| | |
|-----------------------------------------------------------------------------|-----|
| Kreiranje listener-a (slušatelja) | 305 |
| Registracija event-a i listener-a..... | 306 |
| Emitiranje event-a | 306 |
| Prednosti korištenja event-a | 306 |
| Laravel Events i Queues | 306 |
| Livewire | 307 |
| Kako funkcioniра Livewire? | 307 |
| Bundling asset | 307 |
| Node.js | 307 |
| Zašto je Node.js potreban? | 308 |
| Kako instalirati Node.js?..... | 308 |
| Što je Vite? | 309 |
| Kako Vite radi u Laravelu?..... | 309 |
| Što radi vite.config.js? | 310 |
| Uključivanje <code>.css</code> dатоке у <code>layout.blade.php</code> | 310 |
| Povezivanje s Vite..... | 310 |
| Ručno povezivanje <code>.css</code> датоеке..... | 310 |
| Laravel Dusk | 311 |
| Glavna svojstva Laravel Duska | 311 |
| Kako koristiti Laravel Dusk?..... | 311 |
| Kada koristiti Laravel Dusk? | 312 |
| Ostalo | 313 |
| Frontend Developer | 313 |
| Kriptografija..... | 315 |
| Simetrična i asimetrična kriptografija | 315 |
| WYSIWYG editori..... | 317 |
| TinyMCE..... | 317 |
| Konkurenčija..... | 317 |
| CKEditor | 317 |
| Ostali | 318 |
| DataTables..... | 318 |
| Ključne karakteristike DataTables-a | 319 |
| Kako radi?..... | 319 |
| Povezivanje DataTables i Laravel-a | 319 |
| Korištenje CSS-a u Laravel-u..... | 322 |

| | |
|---------------------------------------------------------------------|-----|
| Što je Webpack?..... | 322 |
| Što je Sass i a što SCSS? | 322 |
| Kako koristiti Bootstrap s Laravelom, a kako Sass?..... | 322 |
| Što je grid?..... | 324 |
| Za što se koristi Bootstrap, a za što Sass? | 324 |
| Postman | 324 |
| Postman i Laravel: Osnovne stvari | 325 |
| Kako koristiti Postman s Laravelom | 325 |
| 1. Pokrenite vaš Laravel server | 325 |
| 2. Postavljanje Postman zahtjeva..... | 325 |
| 3. Postavljanje zaglavlja (headers) | 325 |
| 4. Korištenje API-a s autentifikacijom | 326 |
| 5. Testiranje validacije podataka..... | 326 |
| 6. Pohrana zahtjeva i kreiranje kolekcija..... | 326 |
| Povezivanje Laravel API-ja i Postmana..... | 326 |
| Cloudflare | 327 |
| Glavne usluge koje Cloudflare nudi..... | 327 |
| CDN (Content Delivery Network) | 327 |
| DDoS zaštita | 327 |
| SSL certifikati i HTTPS | 327 |
| WAF (Web Application Firewall) | 327 |
| Cache i optimizacija sadržaja..... | 327 |
| Boot Management | 328 |
| DNS usluge | 328 |
| Argo Smart Routing | 328 |
| Jenkins | 328 |
| Automatizacija izgradnje (build)..... | 328 |
| Automatizirano testiranje | 329 |
| Automatizacija migracija i seedanja baze | 329 |
| Postavljanje aplikacije (Deployment)..... | 329 |
| Integracija s Dockerom..... | 329 |
| Konfiguriranje Jenkins Pipelines za Laravel..... | 329 |
| Osnovni koraci za konfiguraciju Jenkins pipeline-a za Laravel | 329 |
| Firebase | 330 |
| Ključne funkcionalnosti Firebase-a..... | 330 |

| | |
|-----------------------------------------------------------------------------------------------|-----|
| Firebase Authentication | 330 |
| Firebase Realtime Database..... | 330 |
| Cloud Firestore | 330 |
| Firebase Hosting..... | 330 |
| Firebase Cloud Messaging (FCM) | 330 |
| Firebase Analytics..... | 330 |
| Firebase Cloud Functions | 331 |
| Firebase i Laravel..... | 331 |
| 1. Firebase Authentication s Laravelom..... | 331 |
| 2. Firebase Realtime Database ili Firestore kao zamjena za tradicionalne baze podataka: | 331 |
| 3. Firebase Cloud Messaging (FCM) za obavijesti | 331 |
| 4. Firebase Hosting za frontend aplikacije: | 331 |
| 5. Firebase Analytics i Laravel: | 331 |
| 6. Serverless funkcije s Cloud Functions: | 331 |
| Digital Ocean usluge..... | 332 |
| 1. Droplets (Virtualni serveri) | 332 |
| 2. One-Click Laravel Deployment | 332 |
| 3. Managed Databases..... | 332 |
| 4. App Platform | 332 |
| 5. Block Storage i Spaces | 333 |
| 6. Automatski Backupi i Snapshoti | 333 |
| 7. Monitoring i Alati za Sigurnost | 333 |
| 8. Dokumentacija i Podrška..... | 333 |
| WebAssembly (Wasm) | 334 |
| Ključne karakteristike WebAssembly-ja | 334 |
| Kako se koristi | 334 |
| Prednosti WebAssembly-ja | 334 |
| Programiranje WebAssembly-ja..... | 335 |
| Rust..... | 335 |
| AssemblyScript | 335 |
| C / C++ | 336 |
| Laravel Forge | 336 |
| Ključne funkcionalnosti Laravel Forge-a | 336 |
| Kako Laravel Forge funkcioniра | 337 |
| Prednosti Laravel Forge-a..... | 337 |

| | |
|----------------------------------------------------------------------|-----|
| Podman i Docker | 337 |
| Arhitektura | 337 |
| Sigurnost..... | 338 |
| Kompatibilnost | 338 |
| Upravljanje kontejnerima i image-ima..... | 338 |
| Orkestracija | 338 |
| Systemd integracija | 338 |
| Rootless vs Root | 339 |
| Ekosistem | 339 |
| Pojam kontejnerizacije | 339 |
| Pojam orkestracije..... | 339 |
| Agilno upravljanje projektima – najbolje prakse i metodologije | 340 |
| Faze upravljanja projektom..... | 340 |
| Tradicionalna metodologija upravljanja projektima | 340 |
| Što je agilno upravljanje projektima | 341 |
| Agilni pristup i proces..... | 342 |
| Agilni koraci razvoja softvera | 342 |
| Agile najbolje prakse | 343 |
| Agilni okviri i metodologije upravljanja projektima | 343 |
| Scrum..... | 343 |
| Kanban..... | 346 |
| Hybrid..... | 347 |
| Lean | 347 |
| Bimodal | 348 |
| Extreme Programming (XP)..... | 348 |
| Crystal..... | 349 |
| Automatizacija i Continuous Integration | 349 |
| Agilno podešavanje funkcionalnosti | 349 |
| CI/CD | 350 |
| CI (Continuous Integration) kod Laravela..... | 350 |
| Automatizacija testiranja | 350 |
| Integracija novih promjena | 350 |
| CD (Continuous Delivery/Deployment) u Laravela | 350 |
| Kontinuirana isporuka (Continuous Delivery)..... | 350 |
| Kontinuirani deployment (Continuous Deployment)..... | 350 |

| | |
|----------------------------------------------|-----|
| Alati za CI/CD kod Laravela..... | 351 |
| GitHub Actions | 351 |
| GitLab CI | 351 |
| Jenkins | 351 |
| Laravel Forge i Envoyer | 351 |
| Prednosti CI/CD u Laravelu | 351 |
| Mapiranje fabule (engl. Story Mapping) | 351 |
| Aimeos..... | 353 |
| Glavna svojstva Aimeos..... | 353 |
| Kako se koristi Aimeos?..... | 353 |

Laravel

Instalacija u Windowsima pomoću Composer-a

Laravel diže svoj interni server s kojim ćemo raditi. Projekt koji otvaramo može biti bilo gdje. Ja sam ga otvorio u `I:\Laravel`. Na njemu ćemo otvoriti GitBash s opcijom Open Git Bash here (na vrhu).

Upisat ćemo:

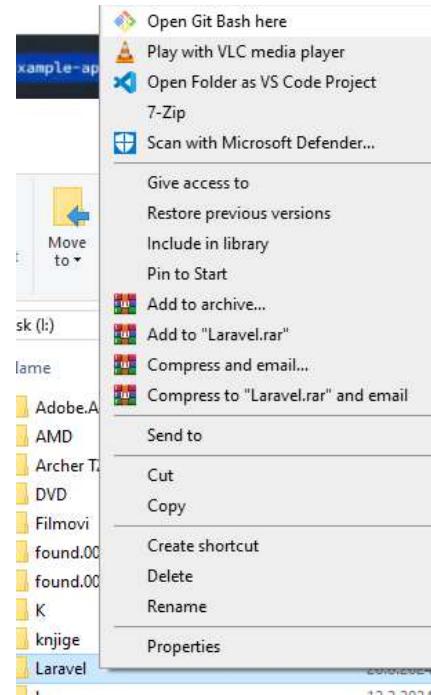
```
composer create-project laravel/laravel AlgebraBlog
```

gdje je `AlgebraBlog` ime našeg projekta.

```
Administrator@DESKTOP-4J5DF41 MINGW64 /i/Laravel
$ composer create-project laravel/laravel AlgebraBlog|
```

Dowloadovat će instalaciju Laravel-a i krenuti s instalacijom i downloadom ostalih komponenti.

To traje par minuta, pa treba pričekati. Instalira 107 komponenti, nisu sve Laravelove.



Unutar `AlgebraBlog` vidjet ćemo strukturu:

Pokretanje Artisan-a

Prvo trebamo ući u `AlgebraBlog` direktorij:

Sada trebamo pokrenuti njegov interni server:

```
php artisan
```

Artisan je bash skripta koja nam praktično da popis svih mogućih naredbi.

Laravel Framework 11.21.0

Upotreba:

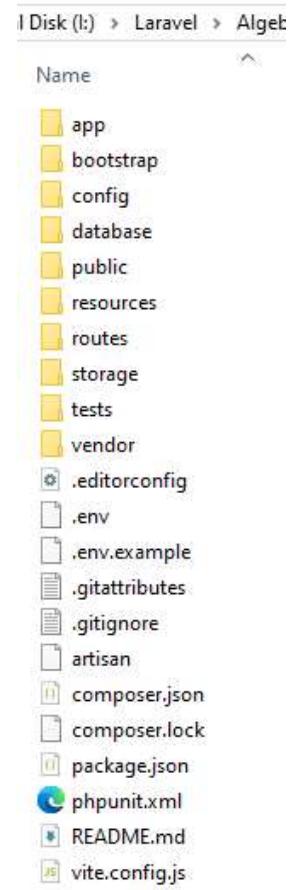
naredba [opcije] [argumenti]

Mogućnosti:

`-h, --help` Prikaži pomoć za danu naredbu. Kada nije dana naredba, prikaži pomoć za izlistane naredbe

`-q, --quiet` Ne izlaze nikakve poruke

`-V, --verzija` Prikaži ovu verziju aplikacije



--ansi | --no-ansi Forsiraj (ili onemogući --no-ansi) ANSI izlaz

-n, --no-interaction Ne postavljam nikakva interaktivna pitanja

--env [=ENV] Okolina u kojoj se naredba treba izvoditi

-v | vv | vvv, --verbose Povećaj opširnost poruka: 1 za normalan izlaz, 2 za detaljniji izlaz i 3 za ispravljanje pogrešaka

Dostupne naredbe:

| Naredba | Opis |
|-------------------------------|-----------------------------------------------------------------------------------|
| about | Prikaži osnovne informacije o vašoj aplikaciji |
| clear-compiled | Ukloni datoteku kompajlirane klase |
| completion | Izbaci skriptu dovršetka ljudske |
| db | Pokreni novu CLI sesiju baze podataka |
| docs | Pristupi Laravel dokumentaciji |
| down | Stavi aplikaciju u način rada za održavanje/demo |
| env | Prikaži trenutnu okolinu radnog okruženja (engl. framework environment) |
| help | Prikaži help za naredbu |
| inspire | Pokažite inspirativan citat |
| list | Popis naredbi |
| migrate | Pokreni migracije baze podataka |
| optimize | Keširaj framework bootstrap, konfiguraciju i metapodatke za povećanje performansi |
| serve | Servisiraj aplikaciju na PHP razvojnom serveru |
| test | Pokreni testove aplikacije |
| tinker | Interakcija sa svojom aplikacijom |
| up | Izbacite aplikaciju iz moda održavanja |
| auth | |
| auth:clear-resets | Isprazni (engl. flush) istekle tokene za resetiranje lozinke |
| cache | |
| cache:clear | Isprazni keš aplikacije |
| cache:forget | Ukloni stavku iz keša |
| cache:prune-stale-tags | Ukloni zastarjele keš tagove iz keša (samo za Redis) |
| channel | |
| channel:list | Izlistaj sve registrirane privatne broadcast kanale |
| config | |
| config:cache | Kreiraj keš datoteku za brže učitavanje konfiguracije |
| config:clear | Ukloni konfiguracijsku keš datoteku |

| | |
|-----------------------------------|--------------------------------------------------------------------------------------|
| <code>config:publish</code> | Objavi konfiguracijsku datoteku za moju aplikaciju |
| <code>config:show</code> | Prikaži sve vrijednosti za danu konfiguracijsku datoteku ili ključ |
| <code>db</code> | |
| <code>db:monitor</code> | Pratiti broj konekcija na navedenoj bazi podataka |
| <code>db:seed</code> | Zasij bazu podataka slogovima |
| <code>db:show</code> | Prikaži informacije o danoj bazi podataka |
| <code>db:table</code> | Prikaži informacije o danoj tablici baze podataka |
| <code>db:wipe</code> | odbaci sve tablice, prikaze (engl. views) i tipove |
| <code>env</code> | |
| <code>env:decrypt</code> | Dešifriraj datoteku okoline (engl. environment file) |
| <code>env:encrypt</code> | Šifriraj datoteku okoline (engl. environment file) |
| <code>event</code> | |
| <code>event:cache</code> | Otkrij i keširaj događaje i slušatelje (engl. events and listeners) aplikacije |
| <code>event:clear</code> | Obriši sve spremljene događaje i slušatelje (engl. events and listeners) |
| <code>event:list</code> | Izlistaj događaje i slušatelje (engl. events and listeners) aplikacije |
| <code>install</code> | |
| <code>install:api</code> | Kreiraj datoteku API ruta i instaliraj Laravel Sanctum ili Laravel Passport |
| <code>install:broadcasting</code> | Kreiraj datoteku s rutama kanala emitiranja (engl. broadcasting channel routes file) |
| <code>key</code> | |
| <code>key:generate</code> | Postavi ključ aplikacije |
| <code>lang</code> | |
| <code>lang:publish</code> | Objavi sve jezične datoteke koje su dostupne za prilagođavanje |
| <code>make</code> | |
| <code>make:cache-table</code> | [cache:table] Kreiraj migraciju za keš tablicu baze podataka |
| <code>make:cast</code> | Kreiraj novu prilagođenu klasu Eloquent |
| <code>make:channel</code> | Kreiraj novu klasu kanala |
| <code>make:class</code> | Kreiraj novu klasu |
| <code>make:command</code> | Kreiraj novu naredbu Artisan |
| <code>make:component</code> | Kreiraj novi pogled (engl. view) klasu komponente |
| <code>make:controller</code> | Napravi novu klasu kontrolera |
| <code>make:enum</code> | Napravi novi enum |

| | |
|---------------------------------------|---------------------------------------------------------------------------------------------------|
| <code>make:event</code> | Kreiraj novu klasu događaja |
| <code>make:exception</code> | Kreiraj novu prilagođenu klasu izuzetaka |
| <code>make:factory</code> | Kreiraj novi model factory |
| <code>make:interface</code> | Napravi novi interface |
| <code>make:job</code> | Kreiraj novu job klasu |
| <code>make:listener</code> | Kreiraj novu klasu slušatelja događaja |
| <code>make:mail</code> | Izradi novu klasu e-pošte |
| <code>make:middleware</code> | Kreiraj novu middleware klasu |
| <code>make:migration</code> | Kreiraj novu datoteku za migraciju |
| <code>make:model</code> | Kreiraj novu klasu modela Eloquent |
| <code>make:notification</code> | Napravi novu klasu obavijesti |
| <code>make:notifications-table</code> | [notifications:table] Napravi migraciju za tablicu obavijesti |
| <code>make:observer</code> | Kreiraj novu klasu promatrača |
| <code>make:policy</code> | Kreiraj novu klasu pravila |
| <code>make:provider</code> | Kreiraj novu klasu pružatelja usluga |
| <code>make:queue-batches-table</code> | [queue:batches-table] Kreiraj migraciju za tablicu baze podataka serija |
| <code>make:queue-failed-table</code> | [queue:failed-table] Kreiraj migraciju za tablicu baze podataka neuspjelih poslova u redu čekanja |
| <code>make:queue-table</code> | [queue:table] Kreiraj migraciju za tablicu baze podataka poslova u redu čekanja |
| <code>make:request</code> | Kreiraj novu klasu zahtjeva (engl. request) za formu |
| <code>make:resource</code> | Kreiraj novi resurs |
| <code>make:rule</code> | Kreiraj novo validacijsko pravilo |
| <code>make:scope</code> | Kreiraj novu klasu opsega |
| <code>make:seeder</code> | Kreiraj novu seeder klasu |
| <code>make:session-table</code> | [session:table] Kreiraj migraciju za tablicu baze podataka sesije |
| <code>make:test</code> | Kreiraj novu test klasu |
| <code>make:trait</code> | Kreiraj novu osobinu (engl. trait) |
| <code>make:view</code> | Kreiraj novi pogled (engl. view) |
| <code>migrate</code> | |
| <code>migrate:fresh</code> | Ispusti sve tablice i ponovno pokreni sve migracije |
| <code>migrate:install</code> | Kreiraj spremište za migraciju |
| <code>migrate:refresh</code> | Resetiraj i ponovno pokreni sve migracije |
| <code>migrate:reset</code> | Vrati sve migracije baze podataka |
| <code>migrate:rollback</code> | Vrati zadnju migraciju baze podataka |

| | |
|-----------------------------------|-----------------------------------------------------------------------------------------|
| <code>migrate:status</code> | Prikaži status svake migracije |
| <code>model</code> | |
| <code>model:prune</code> | Potkreši modele koji više nisu potrebni |
| <code>model:show</code> | Prikaži informacije o Eloquent modelu |
| <code>optimize</code> | |
| <code>optimize:clear</code> | Ukloni keširane bootstrap datoteke |
| <code>package</code> | |
| <code>package:discover</code> | Ponovno izgradi keširani manifest paketa |
| <code>queue</code> | |
| <code>queue:clear</code> | Izbriši sve poslove (engl. jobs) iz navedenog reda čekanja (engl. queue) |
| <code>queue:failed</code> | Navedi sve neuspješne poslove čekanja (engl. queue jobs) |
| <code>queue:flush</code> | Isprati sve neuspjele poslove čekanja |
| <code>queue:forget</code> | Obriši neuspjeli posao na čekanju (engl. queue jobs) |
| <code>queue:listen</code> | Osluškuj zadani red čekanja |
| <code>queue:monitor</code> | Prati veličinu navedenih redova čekanja |
| <code>queue:prune-batches</code> | Ukloni zastarjele unose iz batches baze podataka |
| <code>queue:prune-failed</code> | Ukloni zastarjele unose iz tablice neuspjelih poslova |
| <code>queue:restart</code> | Ponovno pokreni deamone (rade u pozadini) radnika u redu nakon njihovog trenutnog posla |
| <code>queue:retry</code> | Ponovi neuspjeli posao na čekanju |
| <code>queue:retry-batch</code> | Ponovno pokušaj s neuspjelim poslovima za batch (grupu) |
| <code>queue:work</code> | Počni s obradom poslova u redu čekanja kao deamon |
| <code>route</code> | |
| <code>route:cache</code> | Kreiraj datoteku keš rute za bržu registraciju rute |
| <code>route:clear</code> | Ukloni datoteku keš rute |
| <code>route:list</code> | Izlistaj sve registrirane rute |
| <code>sail</code> | |
| <code>sail:add</code> | Dodaj uslugu postojećoj Sail instalaciji |
| <code>sail:install</code> | Instaliraj zadalu datoteku Docker Compose za Laravel Sail |
| <code>sail:publish</code> | Objavi datoteke Laravel Sail Docker |
| <code>schedule</code> | |
| <code>schedule:clear-cache</code> | Izbriši keširane mutex datoteke koje je izradio upravljački program (engl. scheduler) |
| <code>schedule:interrupt</code> | Prekin trenutnog izvođenje upravljačkog programa (engl. scheduler) |
| <code>schedule:list</code> | Navedi sve planirane zadatke |

| | |
|-----------------------------------|-------------------------------------------------------------------------------------------|
| <code>schedule:run</code> | Pokreni planirane naredbe |
| <code>schedule:test</code> | Pokreni planiranu naredbu |
| <code>schedule:work</code> | Pokreni planirane radnje |
| <code>schema</code> | |
| <code>schema:dump</code> | Ispiši zadane sheme baze podataka |
| <code>storage</code> | |
| <code>storage:link</code> | Kreiraj simboličke veze konfigurirane za aplikaciju |
| <code>storage:unlink</code> | Izbriši postojeće simboličke veze konfigurirane za aplikaciju |
| <code>stub</code> | |
| <code>stub:publish</code> | Objavi sve dopune koje možete prilagoditi |
| <code>vendor</code> | |
| <code>vendor:publish</code> | Objavi sva sredstva koja se mogu objaviti iz paketa isporučitelja (engl. vendor packages) |
| <code>view</code> | |
| <code>view:cache</code> | Sastavi sve Blade predloške aplikacije |
| <code>view:clear</code> | Obrisi sve kompajlirane datoteke pogleda (engl. view files) |
| <code>queue:retry</code> | Ponovi neuspjeli zadatak na čekanju (engl. queue job) |
| <code>queue:retry-batch</code> | Ponovno pokušaj s neuspjelim poslovima za batch |
| <code>queue:work</code> | Počni s obradom poslova u redu čekanja kao demon |
| <code>route</code> | |
| <code>route:cache</code> | Kreiraj datoteku keš rute za bržu registraciju rute |
| <code>route:clear</code> | Ukloni datoteku keš rute |
| <code>route:list</code> | Izlistaj sve registrirane rute |
| <code>sail</code> | |
| <code>sail:add</code> | Dodaj uslugu postojećoj Sail instalaciji |
| <code>sail:install</code> | Instaliraj zadanu datoteku Docker Compose za Laravel Sail |
| <code>sail:publish</code> | Objavi datoteke Laravel Sail Docker |
| <code>schedule</code> | |
| <code>schedule:clear-cache</code> | Izbriši keširane mutex datoteke koje je izradio upravljački program (engl. scheduler) |
| <code>schedule:interrupt</code> | Prekini trenutno izvođenje upravljačkog programa (engl. scheduler) |
| <code>schedule:list</code> | Izlistaj sve zadatke upravljačkog programa (engl. scheduler) |
| <code>schedule:run</code> | Pokreni planirane naredbe |
| <code>schedule:test</code> | Pokreni planiranu naredbu |
| <code>schedule:work</code> | Pokreni planirane radnje |
| <code>schema</code> | |

| | |
|-----------------------------|-----------------------------------------------------------------|
| <code>schema:dump</code> | Ispiši zadane sheme baze podataka |
| <code>storage</code> | |
| <code>storage:link</code> | Kreiraj simboličke veze konfiguirirane za aplikaciju |
| <code>storage:unlink</code> | Izbriši postojeće simboličke veze konfiguirirane za aplikaciju |
| <code>stub</code> | |
| <code>stub:publish</code> | Objavi sve stub-ove koje je moguće prilagoditi |
| <code>vendor</code> | |
| <code>vendor:publish</code> | Objavi sve assete koji se mogu objaviti iz paketa isporučitelja |
| <code>view</code> | |
| <code>view:cache</code> | Sastavi sve Blade predloške aplikacije |
| <code>view:clear</code> | Obriši sve sastavljene datoteke pogleda (engl. view files) |

Pokretanje Laravel servera

Ovako pokrećemo Laravel:

```
php artisan serve
```

Ako želimo pokrenuti Laravel na određenom portu, npr. 8001:

```
php artisan serve --port=8001
```

Ovo ne radi ako nismo u projektu. Dakle treba obavzeno prethodno napraviti:

```
cd AlgebraBlog/
```

pristupiti možemo sa `localhost:8000`. To je lokalna verzija i vidimo početnu stranicu:

```
$ php artisan serve
INFO Server running on [http://127.0.0.1:8000].
Press Ctrl+C to stop the server
2024-08-30 17:05:49 / ..... ~ 1s
2024-08-30 17:05:50 /favicon.ico ..... ~ 0.51ms
```

localhost:8001

Laravel

Documentation

Laravel has wonderful documentation covering every aspect of the framework. Whether you are a newcomer or have prior experience with Laravel, we recommend reading our documentation from beginning to end.

Laracasts

Laracasts offers thousands of video tutorials on Laravel, PHP, and JavaScript development. Check them out; see for yourself, and massively level up your development skills in the process.

Laravel News

Laravel News is a community driven portal and newsletter aggregating all of the latest and most important news in the Laravel ecosystem, including new package releases and tutorials.

Vibrant Ecosystem

Laravel's robust library of first-party tools and libraries, such as [Forge](#), [Vapor](#), [Nova](#), [Envoyer](#), and [Herd](#) help you take your projects to the next level. Pair them with powerful open source libraries like [Cashier](#), [Dusk](#), [Echo](#), [Horizon](#), [Sanctum](#), [Telescope](#), and more.

Laravel v11.21.0 (PHP v8.2.12)

laravelversions.com/en

Laravel Versions

Release dates and timelines for security and bug fixes for all versions of Laravel.

To learn more about Laravel's versioning strategy, check out the [Laravel News "Laravel Releases" page](#).

Status

- ALL Bug & security fixes
- SEC Security fixes only
- EOL End of Life
- FUT Future release

| VERSION | RELEASE DATE | BUG FIXES UNTIL | SECURITY FIXES UNTIL | PHP VERSIONS | STATUS |
|---------|-------------------|-----------------|----------------------|---------------|--------|
| 11 | March 12, 2024 | August 5, 2025 | February 3, 2026 | 8.2, 8.3 | ALL |
| 10 | February 14, 2023 | August 7, 2024 | February 7, 2025 | 8.1, 8.2, 8.3 | SEC |

No longer receiving security updates!

Need help upgrading your app? Try [Laravel Shift](#) for automated upgrades or [contact Tighten](#) if you need more than just upgrades.

| VERSION | RELEASE DATE | BUG FIXES UNTIL | SECURITY FIXES UNTIL | PHP VERSIONS | STATUS |
|---------|------------------|-----------------|----------------------|---------------|--------|
| 9 | February 8, 2022 | August 8, 2023 | February 6, 2024 | 8.0, 8.1, 8.2 | EOL |

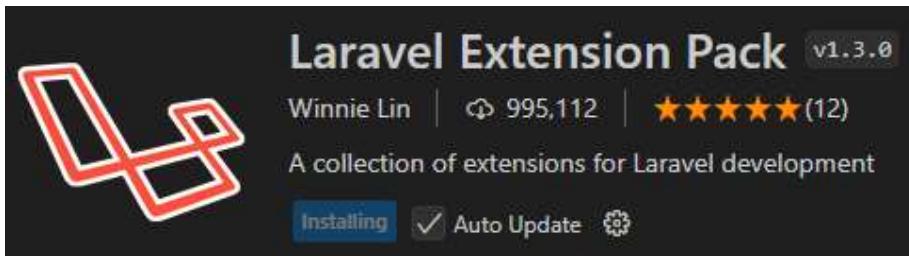
Za projekte bitno je držati ažurne verzije. Na [laravelversions.com](#) možemo vidjeti kada je objavljena zadnja verzija, kada su izašli fiksevi, kada izlaze novi itd. Vidi se i kada određena verzija ostane bez podrške.

Strana 16 od 353

Na classroom se nalazi projekt koji je nekad bio završni rad za Backend tečaj.

Ekstenzije za Visual Studio Code

Laravel Extension Pack sadrži u sebi 13 ekstenzija: EditorConfig for VS Code, DotENV, Laravel Blade Snippets, Laravel Snippets, Laravel Artisan, Laravel Extra Intellisense, Laravel goto view, Laravel Blade formatter, laravel-goto-components, Laravel Blade Wrapper, Laravel Create View, laravel-jump-controller, DevDb.



Struktura direktorija u Laravel projektu

1. **app/**
 - Sadrži: Glavnu poslovnu logiku aplikacije.
 - Poddirektoriji:
 - o [Console/](#)
 - o [Exceptions/](#)
 - o [Http/](#)
 - o [Models/](#)
 - [Providers/](#) i u njemu [AppServiceProvider.php](#), to je jedini servis provider koji Laravel bootstrap-a
2. **bootstrap/**
 - Sadrži:
 - o [app.php](#)
 - o [cache/](#)
3. **config/**
 - Sadrži konfiguracijske datoteke za različite funkcionalnosti aplikacija (baza podataka, sesije, autentifikacija, itd.)
4. **database/**
 - Sadrži:
 - o [migrations/](#)
 - o [factories/](#)
 - o [seeders/](#): Punjenje
5. **public/**
 - Sadrži:
 - o [index.php](#)
 - o Javne datoteke (slike, CSS, Java)
6. **resources/**
 - Sadrži:
 - o [views/](#)
 - o [lang/](#)
 - o [css/, js/](#)
7. **routes/**

- Sadrži definicije ruta (`web.php`, `api.php`, `console.php`, `channels.php`).
- 8. **storage/**
- Sadrži:
 - o `app/`
 - o `framework/`:
 - o `logs/`:
- 9. **tests/**
- Sadrži datoteke za testiranje (feature i unit test)
- 10. **vendor/**
- Sadrži sve vanjske biblioteke instalirane su pomoću Composer-a, uključujući i Laravel framework.

Datoteke u root-u Laravel-u projekta

1. **.env**
 - Ova datoteka sadrži konfiguracijske varijable specifične za vašu okolinu, kao što su postavke za bazu podataka, server e-pošte i druge ključne parametre. Ove se varijable koriste tokom izvođenja aplikacija. Tu ima osjetljivih podataka i ovo ne smije biti na Git-u. Bez ove datoteke opet Laravel neće raditi.
2. **.env.example**
 - Primjer `.env` datoteke ako smo skinuli neki primjer s Git-a ali bez osjetljivih podataka.
3. **.gitattributes**
 - Ova datoteka definira kako Git treba tretirati određene datoteke u projektu, kao što su razmaci, završeci linije i specifične bitne karakteristike za kolaboraciju na različitim operativnim sistemima
4. **.gitignore**
 - Sadrži listu datoteka i direktorija koje Git treba ignorirati, odnosno ne verzionirati. Ovo obuhvaća privremene datoteke, keširane podatke, lokalne konfiguracije i druge datoteke koje nisu relevantne. Neke od ovih stvari su osjetljive a neke su nepotrebne.

```
/phpunit.cache
/node_modules
/public/build
/public/hot
/public/storage
/storage/*.key
/vendor
.env
.env.backup
.env.production
.phpactor.json
.phpunit.result.cache
Homestead.json
Homestead.yaml
auth.json
npm-debug.log
yarn-error.log
```

```
/.fleet  
.idea  
.vscode
```

`.env` je osjetljiv. Dok su druge stvari nepotrebne. `/vendor` možemo dobiti iz `composer.json`, `/node_modules` možemo dobiti iz `package.json`, `/public/build` možemo dobiti iz `vite.config.js`.

5. `.editorconfig`

Ova datoteka sadrži pravila za stil kodiranja koja su univerzalno primjenjiva unutar projekta. Cilj je osigurati konzistentan stil kodiranja između različitih editora i razvojnih okolina, uključujući pravila za uvlačenje, veličinu tabova i završne linije.

6. `artisan`

- Ovo je CLI (Command Line Interface) tj. bash skripta koja omogućuje pristup Artisan naredbama u Laravelu. Artisan je alat koji nudi različite funkcionalnosti, kao što je generiranje koda, migracije baza podataka, pokretanja testova i drugih razvojnih zadataka. Artisan je moguće nadograditi svojim naredbama.

7. `composer.json`

- Ova datoteka sadrži konfiguraciju za Composer, alat za upravljanje PHP ovisnostima. U njoj su definirane sve PHP biblioteke koje projekt koristi, osnovne informacije o projektu, skripte za automatizaciju zadatka te pravila za automatsko učitavanje klasa

8. `composer.lock`

- 9. Ova se datoteka automatski generira kada instalirate ili ažurirate ovisnosti putem Composera. Sadrži tačne verzije svih ovisnosti koje su instalirane u projektu, čime se osigurava da svi koji rade na projektu koriste istu verziju biblioteke, izbjegavajući potencijalne probleme s kompatibilnošću

10. `package.json`

- Ova datoteka sadrži metapodatke o projektu, uključujući naziv, verziju, autora i ovisnosti potrebne za rad aplikacije. Također uključuje skripte koje se mogu pokretati putem npm (Node Package Manager), kao što su naredbe za pokretanje build-a, testova i drugih zadatka vezanih za JavaScript

11. `package-lock.json`

- Ova datoteka se automatski generiše kada se instaliraju npm paketi. Sadrži tačne verzije svih ovisnosti koje su instalirane u projektu, zajedno s njihovim podzavisnostima. Ova datoteka pomaže u održavanju dosljednosti instalacije između različitih okruženja, čime se smanjuju problemi s kompatibilnošću.

12. `phpunit.xml`

- Konfiguracijska datoteka za PHPUnit, alat za automatsko testiranje u PHP-u. Ova datoteka definira pravila i opcije za izvođenje testova, uključujući lokacije testnih datoteka, potrebne ekstenzije i postavke za izvještavanje o rezultatima testiranja.

13. `server.php`

- Alternativni ulazni fajl koji se koristi na serverima koji ne podržavaju `mod_rewrite`. Ova datoteka preusmjerava sve HTTP zahtjeve na `index.php`, što omogućuje Laravel aplikaciji da ispravno obradi zahtjeve korisnika.

14. `webpack.mix.js`

- Datoteka koja sadrži konfiguraciju za Laravel Mix, alat za upravljanje front-end resursima. U ovoj se datoteci definiraju ulazni i izlazni fajlovi, kao i obrade koje će se primijeniti na CSS i JavaScript datoteke, kao što su preprocesiranje Sass-a, transpiliranje JavaScript-a i minifikacija.

15. `vite.config.js`

- Konfiguracijska datoteka za Vite, moderan alat za razvoj frontende koji se koristi kao alternativa Laravel Mix-u u novijim verzijama Laravel-a. Ovdje se definiraju opcije za razvojni poslužitelj, ulazne datoteke i dodatke koji se koriste za optimizaciju performansi i brzinu razvoja aplikacija.

16. [README.md](#)

- Markdown datoteka koja sadrži osnovne informacije o projektu, uključujući upute za instalaciju, korištenje, primjere i druge relevantne informacije. Ova je datoteka često prvi izvor informacija za druge programe koji pregledavaju projekt i trebala bi biti jasno strukturirana i informativna

Instaliranje ovisnosti s Composerom u Lavarelu

Naredba `composer install` se koristi u PHP okolini za upravljanje ovisnostima pomoću Composer alata. Kada je pokrenete, Composer će:

1. **Instalirati ovisnosti:** pretražujete `composer.json` datoteku koja opisuje ovisnosti o projektu. Zatim preuzima i instalira te ovisnosti iz `composer.lock` datoteke (ako postoji) ili prema verzijama navedenim u `composer.json`.
2. **Generirati vendor direktorij:** Ovdje će biti smještene sve instalirane biblioteke i paketi.
3. **Ažurirati automatsko učitavanje:** Kreira ili ažurira `vendor/autoload.php` datoteku koja omogućuje automatsko učitavanje klasa iz instaliranih paketa bez potrebe za ručnim uključivanjem svake klase.

Ukratko, `composer install` priprema radnu okolinu instaliranjem svih potrebnih PHP paketa za vaš projekt prema specifikacijama iz `composer.json`. Ako koristite `composer install` u već postojećem projektu, bit će točno onakvo kako je definirano u `composer.lock` ako postoji, čime se osigurava dosljednost verzije paketa u različitim okruženjima.

```
$ composer install
Installing dependencies from lock file (including require-dev)
Verifying lock file contents can be installed on current platform.
Nothing to install, update or remove
Generating optimized autoload files
> Illuminate\Foundation\ComposerScripts::postAutoloadDump
> @php artisan package:discover --ansi

  INFO: Discovering packages.

  laravel/sail ..... DONE
  laravel/tinker ..... DONE
  nesbot/carbon ..... DONE
  nunomaduro/collision ..... DONE
  nunomaduro/termwind ..... DONE

79 packages you are using are looking for funding.
Use the `composer fund` command to find out more!

Administrator@DESKTOP-4J5DF41 MINGW64 /i/Laravel/AlgebraBlog
$ |
```

Ovime Composer automatski pronađe pakete i postavi ih. Ovo je način kako servisni paket dobije pristup paketima.

Sa Artisanom možemo kreirati **Policies**:

```
php artisan make:Policy UserPolicy --model=User
```

Pojavit će se direktorij **Policies** i datoteka **UserPolicy.php** u njemu.

Artisan je odličan jer generira datoteke koje su nam korisne. Naravno, možemo i ručno raditi nove direktorije.

Artisan inspire – korištenje inspirativnih citata

Kao što smo već vidjeli na popisu naredbi, **php artisan inspire** prikazuje inspirativan citat.

```
Administrator@DESKTOP-4J5DF41 MINGW64 /i/Laravel/AlgebraBlog
$ php artisan inspire

" It is never too late to be what you might have been. "
- George Eliot
```

To je moguće zbog **console.php** u **routes** direktoriju u kome se nalazi:

```
use Illuminate\Foundation\Inspiring;
use Illuminate\Support\Facades\Artisan;

Artisan::command('inspire', function () {
    $this->comment(Inspiring::quote());
})->purpose('Display an inspiring quote')->hourly();
```

Inspiring::quote() poziva metodu iz Inspiring klase (koja se nalazi u Laravelu), a ta metoda vraća slučajno odabrani inspirativni citat. **Artisan::command()** je metoda koja definira novu Artisan komandu. Artisan je alat izgrađen unutar Laravel okvira koji omogućava izvršavanje različitih zadataka iz komandnog reda. **Artisan::command('inspire', function () {...})** definira novu Artisan naredbu s nazivom **inspire**. Kada korisnik uneše **php artisan inspire** u terminalu, pokrenut će se ova naredba. Funkcija unutar naredbe poziva metodu **Inspiring::quote()** koja generira nasumični inspirativni citat. Taj citat se zatim prikazuje u terminalu pomoću **\$this->comment()**. **->purpose('Display an inspiring quote')** određuje opis ove komande koji se prikazuje kada netko upiše **php artisan list**. Pomaže korisnicima da razumiju svrhu naredbe. **->hourly()** označava da je komanda zakazana da se pokrene svakog sata. Laravelova funkcionalnost za raspoređivanje zadataka omogućava zakazivanje komandi putem **kernel.php** ili direktno u definiciji komande, kao u ovom primjeru.

Uklanjanje direktorija i datoteka koje nisu potrebne u projektu

U **database/migrations** ostavimo **0001_01_01_000000_create_users_table.php** a obrišemo

0001_01_01_000001_create_cache_table.php i **0001_01_01_000002_create_jobs_table.php**

Evo šta svaka od navedenih migracija radi:

0001_01_01_000000_create_users_table.php: Ova migracija kreira **users** tablicu koja je ključna za korisničke autentifikacije i upravljanje korisnicima. Ako vaša aplikacija koristi autentifikaciju ili bilo

kakvo upravljanje korisnicima, ovu migraciju ne biste trebali uklanjati. Predavač je ovu migraciju ostavio.

Unutar te datoteke uklonio je kreiranje sa shemom: '**sessions**'. Oko '**password_reset_tokens**' se predavač dvoumio i na kraju ga ostavio.

```
Schema::create('password_reset_tokens', function (Blueprint $table) {
    $table->string('email')->primary();
    $table->string('token');
    $table->timestamp('created_at')->nullable();
});
```

Taj dio stvara tablicu **password_reset_tokens** koja sadrži tri stupca: email (primarni ključ i jedinstvena e-mail adresa korisnika), **token** (za resetiranje lozinke) i **created_at** (vrijeme kreiranja tokena, što omogućuje provjeru validnosti tokena (npr. može isteći nakon određenog vremena))

U dijelu koda s funkcijom **down()**:

```
/**
 * Reverse the migrations.
 */
public function down(): void
{
    Schema::dropIfExists('users');
    Schema::dropIfExists('password_reset_tokens');
    Schema::dropIfExists('sessions');
}
```

Predavač je obrisao red **Schema::dropIfExists('sessions');**

Predavač također kaže da često obriše **users** i doda svoje atribute.

```
Schema::create('users', function (Blueprint $table) {
    $table->id();
    $table->string('name');
    $table->string('email')->unique();
    $table->timestamp('email_verified_at')->nullable();
    $table->string('password');
    $table->rememberToken();
    $table->timestamps();
});
```

0001_01_000001_create_cache_table.php: Ova migracija kreira **cache** tablicu, koja se koristi ako odlučite koristiti bazu podataka za keširanje. Ako ne planirate koristiti bazu podataka za keširanje (npr. koristite file, redis, ili neki drugi driver za keširanje), ova migracija može biti uklonjena. Predavač je ovu migraciju uklonio.

0001_01_000002_create_jobs_table.php: Ova migracija kreira **jobs** tablicu koja se koristi za odlaganje poslova u redove (queue). Ako vaša aplikacija ne koristi redove poslova, ova migracija može biti uklonjena. Predavač je ovu migraciju uklonio.

Kao što smo već spomenuli, **.env datoteka** sadrži konfiguracijske varijable specifične za vašu okolinu. Kako neće završiti na GIT-u nema potrebe mijenjati ništa.

Kreiranje prvog Laravel repozitorija

Predavač je kreirao udaljeni repozitorij. Trebat će veza s tim udaljenim repozitorijem.

Predavač je napravio:

```
git init  
git remote add origin git@github.com:tkescec-algebra/laravel-algebra-blog.git  
git remote -v  
git add .  
git commit -m „App init“  
git push origin -u origin master
```

git init naredba inicijalizira novi lokalni Git repozitorij u trenutnom direktoriju. To stvara skrivenu **.git** mapu koja sadrži sve potrebne datoteke za praćenje verzija u tom direktoriju. Sa naredbom:

git remote add origin git@github.com:tkescec-algebra/laravel-algebra-blog.git dodao je udaljeni repozitorij s imenom **origin** koji pokazuje na GitHub repozitorij na adresi **git@github.com:tkescec-algebra/laravel-algebra-blog.git**. Tamo su pohranjene kopije njegovog koda na serveru.

git remote -v služi za provjeru jer prikazuje sve udaljene repozitorije povezane s njegovim lokalnim repozitorijem i njihove URL-ove. Ova provjera pomaže u potvrđi da je ispravno dodan udaljeni repozitorij.

Naredba **git add .** dodaje sve nove ili promijenjene datoteke iz tekućeg direktorija (označenog sa **.**) u staging area (područje za pripremu), tj. priprema ih za commit.

S naredbom **git commit -m "App init"** stvorio je novi commit s porukom "App init". Poruka opisuje što je promijenjeno u tom commitu, u ovom slučaju to je inicijalizacija aplikacije.

Zadnja naredba **git push origin -u origin master** šalje (push) promjene iz lokalnog repozitorija u udaljeni repozitorij na grani **master**. Opcija **-u** postavlja **origin master** kao zadalu udaljenu granu za buduće **git push** komande, tako da ne morate svaki put specificirati granu.

U stvarnoj situaciji, generirali bi granu **dev** na kojem bi razvijali aplikaciju i poslije bi to push-nuli u **master** granu. U ovom slučaju ostat ćemo u **master** grani i nećemo praviti **dev** granu.

Više imenskih prostora (engl. namespace) unutar jedne datoteke

Unutar jedne PHP datoteke moguće je deklarirati više imenskih prostora (engl. namespaces), ali postoji nekoliko važnih pravila i ograničenja. **Imenski prostor (namespace)** omogućava organizaciju koda i izbjegava sukobe imena (klasa, funkcija, konstanti) unutar većih projekata.

Jedan primarni imenski prostor po datoteci: Uobičajena praksa je da se u jednoj datoteci nalazi jedan imenski prostor. Međutim, PHP dopušta postojanje više imenskih prostora, ali oni moraju biti jasno odvojeni unutar datoteke.

Korištenje više imenskih prostora: Da biste koristili više imenskih prostora u jednoj datoteci, morate ih definirati odvojeno, i sve klase, funkcije i konstante unutar svakog imenskog prostora pripadaju isključivo tom prostoru. Ograničenja:

Kada koristite više imenskih prostora u jednoj datoteci, nije moguće kombinirati ih unutar istog bloka koda.

Morate jasno odvojiti blokove koda pomoću namespace ključne riječi.

```
<?php
// Prvi imenski prostor
namespace App\Models;

class User {
    public function getInfo() {
        return "Ovo je klasa User iz App\Models namespace-a.";
    }
}

// Drugi imenski prostor
namespace App\Helpers;

class User {
    public function getInfo() {
        return "Ovo je klasa User iz App\Helpers namespace-a.";
    }
}

// Korištenje oba imenska prostora
// Moramo koristiti puni naziv klase, jer su imenski prostori različiti.
$modelsUser = new \App\Models\User();
echo $modelsUser->getInfo(); // Ispisuje: Ovo je klasa User iz App\Models
namespace-a.

$helpersUser = new \App\Helpers\User();
echo $helpersUser->getInfo(); // Ispisuje: Ovo je klasa User iz App\Helpers
namespace-a.
```

Prvi namespace: `App/Models` sadrži klasu `User` koja ima svoju metodu `getInfo()`.

Dруги namespace: `App/Helpers` sadrži također klasu `User`, ali ta klasa nema veze s klasom u `App/Models`, jer su im uloge odvojene imenskim prostorom.

U korištenju klase iz različitih imenskih prostora potrebno je navesti puni naziv imenskog prostora koristeći npr. `/App/Models/User()`.

Kako pokupiti Laravel project sa udaljenog repozitorija

Kloniranje repozitorija

Treba GitBash u u root-u za sve Laravel projekte:

```
git clone git@github.com:tkescec-algebra/laravel-algebra-blog.git
```

```
$ git clone git@github.com:tkescec-algebra/laravel-algebra-blog.git
Cloning into 'laravel-algebra-blog'...
remote: Enumerating objects: 255, done.
remote: Counting objects: 100% (255/255), done.
remote: Compressing objects: 100% (165/165), done.
remote: Total 255 (delta 88), reused 230 (delta 63), pack-reused 0 (from 0)
Receiving objects: 100% (255/255), 98.71 KiB | 292.00 KiB/s, done.
Resolving deltas: 100% (88/88), done.
```

```
|Administrator@DESKTOP-4J5DF41 MINGW64 /i/Laravel
$ |
```

Ovo će preuzeti cijeli repozitorij na vaš lokalni kompjuter i kreirati direktorij s istim imenom kao udaljeni repozitorij (u ovom slučaju `laravel-algebra-blog`).

Ulazak u direktorij

Nakon što smo klonirali repozitorij, uđemo u taj direktorij sa:

```
cd laravel-algebra-blog
```

```
|Administrator@DESKTOP-4J5DF41 MINGW64 /i/Laravel
$ cd laravel-algebra-blog/
Administrator@DESKTOP-4J5DF41 MINGW64 /i/Laravel/laravel-algebra-blog (master)
$ |
```

Instaliranje ovisnosti

Sada je potrebno [instalirati ovisnosti](#):

```
composer install
```

Instalirale su se sve moguće ovisnosti (u ovom trenutku njih 109).

Kopiranje i podešavanje .env datoteke

Nakon ovoga potrebno je podesiti `.env` datoteku. Ako nemamo gotovu i spremnu trebamo iskopirati primjer koji dođe. Dakle:

```
cp .env.example .env
```

`.env` treba podesiti kao što je to objašnjeno u poglavlju [.env datoteka pregled i podešavanje](#).

Generiranje ključa

Idući korak je generirati `APP_KEY`, Laravelov ključ. To radimo sa:

```
php artisan key:generate
```

Za više pogledati u [.env datoteka pregled i podešavanje](#).

Brisanje postojeće baze podataka

Ako je već nismo obrisali potrebno je **obrisati postojeću bazu podataka** pomoću alata koji preferirate (npr. phpMyAdmin ,MySQL Workbench ,DBeaver , itd.). Ja sam koristio MySQL Workbench.

Pokretanje migracija

```
php artisan migrate
```

Pokretanje seeder-a

```
php artisan db:seed
```

Pokretanje Artisan servera

```
php artisan serve
```

Sada možemo raditi s lokalnom verzijom koda.

Ažuriranje (osvježavanje) lokalne kopije Laravel projekta sa novim kodom s udaljenog repozitorija

Ulazak u direktorij projekta

Potrebno je ući u projekt. U našem slučaju:

```
cd Laravel-algebra-blog
```

Povlačenje koda sa udaljenog repozitorija radi ažuriranja

Osvježavanje koda vršimo sa:

```
git pull origin master
```

`git pull` preuzima promjene iz udaljenog repozitorija i automatski ih spaja s trenutnom lokalnom granom. `origin` je naziv udaljenog repozitorija. Kada klonirate repozitorij, Git automatski postavlja "`origin`" kao ime za taj repozitorij. Može postojati više udaljenih repozitorija, ali "`origin`" je najčešće korišteno ime za glavni repozitorij. `master` je naziv grane koju želite ažurirati. U mnogim repozitorijima, "master" je glavna ili primarna grana (međutim, u novijim praksama, ova grana se često naziva "`main`"). Ova naredba će preuzeti promjene s grane "`master`" iz udaljenog repozitorija "`origin`" i spojiti ih s vašom lokalnom granom. Ako se naša grana zove drugačije npr. `master` ili neki drugi naziv), treba zamijeniti to ime s imenom naše grane.

Instaliranje ovisnosti

Sada je potrebno [instalirati ovisnosti](#):

```
composer install
```

```
composer install
```

Code First i Database First pristup relacionim bazama podataka

Code First i Database First su dva različita pristupa u razvoju aplikacija koje koriste relacione baze podataka, posebno u kontekstu ORM (Object-Relational Mapping) tehnologija kao što je Entity Framework. Evo kako svaki od ovih pristupa funkcioniра:

Code First

Code First pristup podrazumijeva da se najprije definiraju modeli podataka u kodu aplikacije, a zatim se na osnovu tih modela generira baza podataka.

Razvijate klase (modele) u kodu koje predstavljaju entitete u vašoj aplikaciji. Može automatski generirati strukturu baze podataka (tablice, stupce) na osnovu tih modela kada aplikacija prvi put pokrene. Podržava migracije koje omogućuju praćenje promjena u modelima i automatsko ažuriranje baze podataka. Omogućava brže iteracije u razvoju, jer se promjene u modelima mogu odmah reflektirati u bazi. Koristi se kada razvijate novu aplikaciju i želite imati potpunu kontrolu nad modelima i strukturom baze i kada očekujete česte promjene u modelima tokom razvoja. U radu s Laravelom koristimo Code First pristup.

Database First

Database First pristup podrazumijeva da se najprije kreira baza podataka, a zatim se koriste postojeće strukture baze za generiranje modela u aplikaciji. Počinje se s već definiranom bazom podataka, koja može biti kreirana ručno ili putem nekih drugih alata. Postoje alati koji mogu generirati klase i kontekst na osnovu strukture postojeće baze podataka. Promjene u bazi podataka mogu zahtijevati dodatne korake za ažuriranje modela, što može biti nezgodno ako se često mijenja struktura baze. Ovaj pristup korisiti.NET. On ima i code first.

Monolitne aplikacije i API

Razlika između monolita i API-ja odnosi se na način na koji se aplikacije organiziraju, implementiraju i komuniciraju s drugim aplikacijama ili uslugama.

Monolit

Monolit je tradicionalni način organiziranja aplikacija, gdje su sve funkcionalnosti integrirane u jednu cjelinu aplikacije.

Monolit nije nužno u jednoj datoteci, iako je organiziran kao jedinstvena aplikacija. Monolitna arhitektura podrazumijeva da je čitava aplikacija u jednoj kodnoj bazi (repozitorij), a svi dijelovi aplikacije (npr. korisnička sučelja, poslovna logika, pristup bazi podataka) međusobno povezani i integrirani.

Karakteristike:

- **Jedinstvena kodna baza:** Cijela aplikacija se piše u jednoj kodnoj bazi, a sve njene funkcionalnosti (UI, logika, baza podataka) nalaze se u jednom mjestu.
- **Centralizirana arhitektura:** Monolit je jedna jedinstvena aplikacija koja se obično implementira na jednom poslužitelju.
- **Blisko povezan kod:** Različiti moduli ili dijelovi aplikacija (npr. autentifikacija, upravljanje korisnicima, obrada plaćanja) međusobno su povezani i funkcionišu kao jedna cjelina.
- **Teže skaliranje:** Ako aplikacija raste i potrebno je povećati resurse, obično je potrebno skalirati cijelu aplikaciju, čak i dijelove koji ne zahtijevaju dodatne resurse.
- **Jedna točka neuspjeha:** Ako se neki dio aplikacije pokvari, može uzrokovati pad cijele aplikacije.

Primjer:

Ako imate web aplikaciju za online trgovinu, sve njene funkcionalnosti, kao što su narudžbe, korisnički računi, plaćanja i detaljni prikaz proizvoda, nalaze se unutar jedne aplikacije. Ako postoji problem s jednom funkcionalnošću, to može utjecati na cijelu aplikaciju.

API (Aplikacijski programski interface)

API je interface koji različitim aplikacijama ili komponentama omogućuje međusobno komuniciranje. Najčešće se koristi u mikroservisnoj arhitekturi, gdje različite funkcionalnosti aplikacije postoje kao odvojeni moduli ili servisi koji komuniciraju preko API-ja.

Karakteristike:

- **Decentralizirana arhitektura:** Aplikacija je podijeljena na manje servise koji su međusobno neovisni i komuniciraju putem API-ja.
- **Interoperabilnost:** API omogućuje različitim aplikacijama, napisanim u različitim tehnologijama, za međusobno komuniciranje.

- **Skalabilnost:** API omogućuje da se svaki servis ili dio aplikacije skalira neovisno o ostatku aplikacije. Na primjer, ako sistem za upravljanje narudžbama postane prezauzet, samo njega možete skalirati.
- **Modularnost:** Svaka funkcionalnost može postojati kao samostalan servis, a komunikacija među njima odvija se preko API-ja. To omogućuje veću fleksibilnost i održavanje koda.
- **Različiti protokoli:** API-ji mogu koristiti različite protokole za komunikaciju, kao što su HTTP, gRPC, SOAP ili RESTful API.

Primjer:

U aplikaciji za online trgovinu, umjesto jedne monolitne aplikacije, mogli biste imati privatne usluge za korisničke račune, upravljanje narudžbama, plaćanja i proizvode. Svaki od tih servisa komunicira putem API-ja, pa ako sistem za narudžbe treba nadogradnju ili dodatnu snagu, samo taj dio možete skalirati ili mijenjati.

Ključne razlike:

| Karakteristika | Monolit | API/Mikroservisi |
|---------------------------|-------------------------------------------------------|--------------------------------------------------|
| Arhitektura | Sve je integrirano u jednu aplikaciju | Funkcionalnosti su podijeljene u odvojene usluge |
| Kodna baza | Jedna zajednička kodna baza | Svaki servis ima svoju kodnu bazu |
| Skalabilnost | Teže skalirati (potrebno skalirati cijelu aplikaciju) | Lakše skalirati pojedine servise |
| Nezavisnost modula | Moduli su međusobno ovisni | Moduli su neovisni, komuniciraju preko API-ja |
| Održavanje | Teže zbog složenosti aplikacija | Lakše zbog modularnosti servisa |

Ukratko, monolit je cjelovita aplikacija s integriranim komponentama, dok je API alat koji omogućuje komunikaciju između odvojenih modula aplikacija, često u kontekstu mikroservisne arhitekture.

Monolit aplikacija za rad s postovima

Fasade

Fasade u Laravelu su klasične klase koje pružaju "statički" pristup uslugama unutar aplikacije. One su dizajnirane da budu lako razumljive i omogućuju jednostavno korištenje raznih komponenti bez potrebe za instanciranjem objekata.

Ključne karakteristike fasada:

- **Statistički pristup:** Fasade omogućuju pristup instancama servisa putem statičkih metoda, čime se olakšava njihovo korištenje bez brige o stvaranju instanci.
- **Pojednostavljena sintaksa:** Korisnici mogu koristiti fasade na jednostavan način, što čini kod čitljivijim i lakšim za održavanje. Na primjer:
- **Provjera i testiranje:** Fasade se mogu lako zamijeniti ili lažirati prilikom testiranja, što olakšava pisanje testova.

Fasade su unaprijed definirane klase koje statički pristup raznim uslugama unutar aplikacije. Fasade služe kao "most" prema stvarniminstancama servisa u Laravelovom kontejneru zavisnosti.

Neki od najčešće korištenih fasada u Laravelu:

App: Omogućava pristup instanci aplikacije.

Auth: Koristi se za upravljanje autentifikacijom korisnika.

Cache: Omogućava rad s keširanjem podataka.

DB: Pruža pristup bazi podataka.

Event: Omogućava emitiranje i slušanje događaja unutar aplikacije.

File: Pomaže u radu s datotekama, uključujući kreiranje, čitanje i brisanje datoteka.

Log: Koristi se za zapisivanje logova i poruka za praćenje.

Mail: Omogućava slanje e-mailova.

Queue: Koristi se za rad s redovima zadataka.

Route: Omogućava definiranje ruta u aplikaciji.

MySQL migracije

Na ispitu se pojavi pitanje da objasnimo **MySQL migracije**. To nisu Laravel migracije, o kojima će nešto kasnije biti riječi.

MySQL migracije, (nevezano za Laravel ili neku drugu radnu okolinu), odnose se na proces upravljanja promjenama u strukturi baze podataka kroz niz SQL skripti. Migracije omogućavaju praćenje promjena nad bazom podataka, kao što su kreiranje, modificiranje ili brisanje tablica, stupaca i drugih objekata u bazi.

Migracija se sastoji od niza SQL naredbi koje mijenjaju strukturu baze podataka. Svaka migracija predstavlja neku promjenu, kao što je kreiranje nove tablice, dodavanje stupca (polja), izmjene tipa podataka, itd. Evo primjera **migracijske skripte** SQL migracije:

```
CREATE TABLE users (
    id INT AUTO_INCREMENT PRIMARY KEY,
    ime VARCHAR(100),
    email VARCHAR(100) UNIQUE,
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);
```

Svaka migracija treba biti vezana uz neku verziju (**verzioniranje**) ili vremenski pečat kako bi se jasno definiralo redoslijed primjene migracija. Na primjer, datoteke koje sadrže migracije mogu biti imenovane prema datumu i vremenu kada su kreirane ([20240912_create_users_table.sql](#)).

Dobra praksa u migracijama je pisanje skripti koje omogućavaju **vraćanje (rollback) promjena**. Ovo je važno kada dođe do grešaka u migraciji ili kada je potrebno vratiti bazu podataka na ranije stanje. Evo primjera SQL skripte:

```
DROP TABLE IF EXISTS users;
```

Migracije se mogu primjenjivati ručno, koristeći MySQL alat za izvršavanje SQL naredbi MySQL Workbench. Primjer migracije:

```
mysql -u korisnik -p baza_podataka < 20240912_create_users_table.sql
```

U mnogim timovima, migracijske datoteke se verzioniraju u repozitorij koda (npr. Git), što omogućava timovima da zajednički prate promjene u strukturi baze podataka.

Migracije

Migracije u Laravelu su alat koji omogućava verzioniranje i upravljanje bazom podataka na način sličan verzioniranju koda. One omogućavaju programerima da definiraju strukturu baze podataka (kao što su tabele, indeksi, ključevi itd.) u PHP skriptama, umjesto da ručno pišu SQL naredbe. Ovo olakšava timovima da dosljedno dijele i upravljaju promjenama u strukturi baze podataka kroz različite faze razvoja i među različitim okruženjima (npr. lokalni razvoj, testiranje, produkcija).

Osnovne karakteristike migracija:

1. **Kreiranje tablica:** Pomoću migracija možete kreirati nove tablice u bazi podataka. Na primjer, možete definirati tablicu `users` sa stupcima za `ime`, `email`, `password`, itd. Migracije omogućavaju automatizirano kreiranje tablica kroz PHP umjesto pisanja direktnih SQL naredbi. Laravel koristi Schema Builder koji generira odgovarajući SQL za ciljanu bazu.

```
php artisan make:migration create_users_table
```

Ova naredba stvara novu migracijsku datoteku unutar direktorija `database/migrations`. Svaka datoteka sadrži `up()` metodu (koja primjenjuje promjene u bazi) i `down()` metodu (koja uklanja promjene u slučaju vraćanja).

2. **Izmjena tablica:** Migracije nam omogućavaju da dodate ili uklonite stupce iz postojećih tablica, promjenite tipove podataka, kreirate ili uklanjate indekse, te izvršite druge promjene u strukturi tabele.
3. **Verzioniranje promjena:** Svaka migracija se kreira s vremenskom oznakom, što omogućava praćenje redoslijeda u kojem su promjene izvršene. Laravel čuva evidenciju o tome koje migracije su već pokrenute, tako da nikada nećete slučajno ponoviti istu migraciju.
4. **Rollback migracija:** Ako se dogodi greška ili ako je potrebno vratiti promjene u bazi podataka, Laravel omogućava vraćanje izvršenih migracija. To znači da možete lako "poništiti" migracije i vratiti bazu podataka u prethodno stanje.

Kako funkcioniraju migracije?

Kreiranje migracije: Kreirate migraciju pomoću Artisan komandne linije (`php artisan make:migration kreiraj_korisničku_tablicu`). To će generirati novu PHP datoteku u direktoriju `database/migrations` koja sadrži metodu `up()` za definiranje strukture tablice i metodu `down()` za poništavanje te strukture. Tu možemo dopisati stupce koji nam trebaju za tablicu.

Pokretanje migracija: Nakon što kreirate migraciju, pokrećete je komandnom `php artisan migrate`, koja izvršava sve migracije koje još nisu pokrenute, te kreira odgovarajuće tablice i stupce (polja) u bazi podataka.

Vraćanje migracija (rollback): Ako želite poništiti migraciju, možete koristiti `php artisan migrate:rollback`, koja će izvršiti metodu `down()` u vašim migracijama i vratiti bazu podataka na prethodno stanje.

Migracije su ključan dio Laravelovog sistema za upravljanje bazama podataka, omogućavajući timovima da lako upravljaju i dijele promjene u strukturi baze podataka tokom cijelog životnog ciklusa aplikacije.

Pokretanjem migracije pomoću Artisana. Time kroz migracije možemo pratiti i verzioniziranje. Moguć je i suprotan smjer (tzv. roll-back). S roll-back je moguće lako vratiti na prethodno stanje.

Modifikatori i indeks modifikatori se odnose na način na koji definirate stupce i indekse u migracijama za baze podataka. Migracije u Laravelu omogućuju verziranje strukture baze podataka na način sličan verziranju koda, a modifikatori pomažu u prilagodbi tih struktura.

Pogledajmo ovaj primjer:

```
<?php
class Test{
    public function h(){
        return "Hello";
    }
}

$t = new Test();
echo ($t->h());
```

Instanciranje objekta radimo s `$t = new Test();`. To stvara novi objekt klase `Test`. Taj objekt se pohranjuje u varijablu `$t`. U ovom trenutku, možete koristiti `$t` da pozovete metode definirane unutar klase `Test`.

`(new Test())->h();`: se ponovo instancira novi objekat klase `Test` (bez da se pohranjuje u varijablu) i odmah se poziva metoda `h` na tom objektu. Ovo vraća string "`Hello`".

Cijeli ovaj uvod bio je da objasnimo red koji se pojavljuje na početku migracije, odmah iza use:

```
return new class extends Migration
```

Ovaj izraz se koristi za definiranje anonimne klase koja nasljeđuje (ili „proširuje“) drugu klasu, u kontekstu migracija. `new class` se koristi za kreiranje anonimne klase. Anonimne klase nemaju ime, ali mogu imati svojstva, metode i mogu se instancirati kao klasične klase. `extends Migration` označava da nova anonimna klasa nasljeđuje klasu `Migration`. U praksi, ovo bi izgledalo ovako:

```
use Illuminate\Database\Migrations\Migration;

return new class extends Migration {
    public function up() {
        // Ovdje se definiraju promjene koje će se izvršiti
        // kada se migracija pokrene
    }

    public function down() {
        // Ovdje se definiraju promjene koje će se izvršiti
        // kada se migracija vrati unazad
    }
};
```

Klasa `Schema` služi za interakciju s bazom podataka, omogućujući vam da kreirate, mijenjamo ili brišemo tablice. Metode koje su u njoj su:

`create()`: za kreiranje nove tablice.

`table()`: za izmjenu postojeće tablice.

`drop()`: za brisanje tablice.

`hasTable()`: za provjeru postojanja tablice.

Klasa `Blueprint` koristi se unutar migracija za definiranje strukture tablice. Ona predstavlja „plan“ za tablicu koju želite kreirati ili izmijeniti.

Osnovne funkcije: Omogućuje definiranje stupaca, tipova podataka, indeksa i drugih karakteristika tablice. Neki od najčešće korištenih metoda uključuju:

`increments()`: za definiranje auto-increment primarnog ključa.

`string()`: za definiranje string stupca (polja).

`integer()`: za definiranje cjelobrojnog stupca.

`foreign()`: za definiranje stranog ključa.

`timestams()`: za automatsko dodavanje `created_at` i `updated_at` stupaca.

Postoji velik broj vrsta stupaca koje možete dodati tablicama baze podataka. Sve dostupne metode moguće je vidjeti na <https://laravel.com/docs/11.x/migrations>. Tu je moguće vidjeti i kako raditi s indeksima.

Modifikatori

Modifikatori u migracijama omogućuju dodavanje specifičnih svojstava stupcima baze podataka. Oni se primjenjuju nakon definiranja tipa stupca.

Primjeri modifikatora u Laravelu:

1. `nullable()`: Označava da stupac može imati `NULL` vrijednosti.

```
$table->string('email')->nullable();
```

2. `default($value)`: Postavlja zadanu vrijednost za stupac.

```
$table->boolean('is_active')->default(true);
```

3. `unique()`: Dodaje jedinstveni indeks na stupac.

```
$table->string('email')->unique();
```

4. `index()`: Dodaje jednostavni indeks na stupac.

```
$table->integer('user_id')->index();
```

5. `unsigned()`: Označava da je broj u stupcu bez predznaka (ne može biti negativan).

```
$table->integer('votes')->unsigned();
```

6. `autoIncrement()`: Označava da stupac automatski povećava svoju vrijednost pri svakom novom unosu.

```
$table->bigIncrements('id')->autoIncrement();
```

Indeks modifikatori

Indeks modifikatori specifično se odnose na definiranje različitih tipova indeksa u bazama podataka. Laravel nudi nekoliko različitih tipova indeksa koje možete dodati u svoje migracije:

1. `primary()`: Definira stupac kao primarni ključ.

```
$table->primary('id');
```

2. `unique()`: Kreira jedinstveni indeks kako bi se osiguralo da sve vrijednosti u tom stupcu budu jedinstvene.

```
$table->string('email')->unique();
```

3. `index()`: Dodaje indeks na stupac, čime se omogućuje brže pretraživanje podataka.

```
$table->integer('user_id')->index();
```

4. `spatialIndex()`: Koristi se za prostorne podatke, omogućava bržu pretragu geografskih podataka.

```
$table->geometry('location')->spatialIndex();
```

5. `foreign()`: Dodaje strani ključ koji referencira drugu tablicu.

```
$table->foreign('user_id')->references('id')->on('users');
```

0001_01_01_000000_create_users_table.php

Datoteka `0001_01_01_000000_create_users_table.php` je Laravel migracija koja kreira nekoliko ključnih tablica u bazi podataka, a koristi se za definiranje strukture tablica potrebnih za upravljanje korisnicima, resetiranjem lozinki i sesijama.

U početnom dijelu koda:

```
use Illuminate\Database\Migrations\Migration;
use Illuminate\Database\Schema\Blueprint;
use Illuminate\Support\Facades\Schema;
```

Ovo su use izjave koje uvoze potrebne klase iz Laravelovog frameworka. `Migration` klasa omogućuje kreiranje i upravljanje migracijama, `Blueprint` se koristi za definiranje strukture tablica, a `Schema` omogućava rad s bazom podataka.

```
return new class extends Migration
{
```

Ovo je anonimna klasa koja nasljeđuje `Migration` klasu. Unutar ove klase definirane su dvije ključne metode: `up()` i `down()`.

Metoda `up()` se pokreće kada izvršavate migraciju (tj. kada se tablice kreiraju u bazi podataka).

```
public function up(): void
{
    // Kreiranje tablice 'users'
    Schema::create('users', function (Blueprint $table) {
        $table->id(); // Primarni ključ, autoinkrementalno polje
        $table->string('name'); // Ime korisnika
        $table->string('email')->unique(); // Email korisnika, jedinstveno polje
        $table->timestamp('email_verified_at')->nullable(); // Vrijeme verifikacije emaila, može biti NULL
        $table->string('password'); // Lozinka korisnika
        $table->rememberToken(); // Token za "remember me" funkcionalnost
        $table->timestamps(); // Polja 'created_at' i 'updated_at'
    });

    // Kreiranje tablice 'password_reset_tokens'
    Schema::create('password_reset_tokens', function (Blueprint $table) {
        $table->string('email')->primary(); // Primarni ključ je email
        $table->string('token'); // Token za reset lozinke
        $table->timestamp('created_at')->nullable(); // Vrijeme kreiranja zapisa, može biti NULL
    });

    // Kreiranje tablice 'sessions'
    Schema::create('sessions', function (Blueprint $table) {
        $table->string('id')->primary(); // Primarni ključ, identifikator sesije
    });
}
```

```

        $table->foreignId('user_id')->nullable()->index(); // Strani ključ prema
'users' tablici, može biti NULL
        $table->string('ip_address', 45)->nullable(); // IP adresa korisnika, može
biti NULL
        $table->text('user_agent')->nullable(); // Informacije o pretraživaču
korisnika, može biti NULL
        $table->longText('payload'); // Podaci o sesiji
        $table->integer('last_activity')->index(); // Vrijeme posljednje aktivnosti
korisnika
    );
}

```

Tablica `users`: Pohranjuje osnovne podatke o korisnicima, kao što su ime, email, lozinka itd. Također, koristi `timestamps()` koji automatski dodaje polja `created_at` i `updated_at` u tablicu.

Tablica `password_reset_tokens`: Pohranjuje token za resetiranje lozinke povezan s emailom korisnika. Primarni ključ ove tablice je `email`.

Tablica `sessions`: Pohranjuje informacije o sesijama korisnika, uključujući IP adresu, `user_agent` (informacije o pretraživaču), i podatke o sesiji. `user_id` je strani ključ prema tablici users.

Metoda `down()` se pokreće kada se migracija poništava (tj. kada se tablice brišu iz baze podataka).

```

public function down(): void
{
    Schema::dropIfExists('users');
    Schema::dropIfExists('password_reset_tokens');
    Schema::dropIfExists('sessions');
}

```

Timestamp-ovi – vremenske označke

`$table->timestamps();`:

Dodaje stupce `created_at` i `updated_at`, koje automatski bilježe vrijeme kada je zapis kreiran i kada je posljednji put ažuriran. Timestamp-ovi omogućuju jednostavno praćenje kada su podaci zadnji put mijenjani. To je korisno za analizu podataka, kontrolu verzija ili povijest promjena. Timestamp-ovi olakšavaju filtriranje i sortiranje podataka. Na primjer, možete lako izvući sve zapise koji su kreirani ili ažurirani unutar određenog vremenskog perioda.

Soft deleted

Soft delete mehanizam u Laravelu omogućava "brisanje" zapisa iz baze podataka na način da se zapravo ne uklone trajno, već se jednostavno označe kao obrisani. Ovo je korisno kada želite zadržati podatke za potencijalnu buduću upotrebu, ali ih ne želite prikazivati u aplikaciji.

Kada se koristi soft delete, umjesto da zapis bude trajno izbrisani iz baze podataka, samo se stupac (polje) `deleted_at` popuni s vremenskom oznakom. Laravel tada automatski ignorira te zapise prilikom dohvaćanja podataka, osim ako eksplicitno ne zatražite i "obrisane" zapise.

Predavač je u `up()` metodu sa `'users'` definicijom dodao red:

```
$table->softDeletes(); // Dodaje u tablici stupac 'deleted_at'
```

On predlaže da bude upaljen u svim tablicama (osim na pivot tablicama koje imaju many to many relaciju).

Svaki puta će Laravel gledati da li je ovaj stupac `NULL`.

Nakon što postavite soft delete u modelu i migraciji, možete koristiti sljedeće funkcionalnosti:

Brisanje zapisa (soft delete):

Kad obrišete zapis koristeći metodu `delete()`, Laravel će ažurirati kolonu `deleted_at` umjesto trajnog brisanja.

```
$post = Post::find(1);  
$post->delete(); // Samo ažurira 'deleted_at', ne briše stvarno iz baze
```

Dohvaćanje zapisa bez "obrisanih":

Laravel automatski ignorira zapise koji imaju vrijednost `deleted_at`. Na primjer, sljedeći upit vraća samo zapise koji nisu "obrisani":

```
$posts = Post::all(); // Vraća samo zapise gdje je 'deleted_at' NULL
```

Dohvaćanje "obrisanih" i aktivnih zapisa:

Ako želite dohvatiti i zapise koji su soft-deleted, koristite metodu `withTrashed()`:

```
$posts = Post::withTrashed()->get(); // Vraća sve zapise, uključujući i obrisane
```

Dohvaćanje samo "obrisanih" zapisa:

Ako želite dobiti samo zapise koji su soft-deleted, možete koristiti metodu `onlyTrashed()`:

```
$posts = Post::onlyTrashed()->get(); // Vraća samo obrisane zapise
```

Vraćanje "obrisanih" zapisa (restore):

Zapis koji su soft-deleted mogu se "vratiti" pomoću metode `restore()`:

```
$post = Post::withTrashed()->find(1);  
$post->restore(); // Postavlja 'deleted_at' na NULL i vraća zapis
```

Trajno brisanje (hard delete):

Ako želite trajno izbrisati zapis iz baze, koristite metodu `forceDelete()`:

```
$post = Post::withTrashed()->find(1);  
$post->forceDelete(); // Trajno briše zapis iz baze
```

Predavač preporuča da ovo ne koristimo nikada.

.env datoteka pregled i podešavanje

`.env` datoteka u Laravelu sadrži konfiguracijske postavke koje su ključne za rad aplikacije. Ove postavke omogućuju jednostavno prilagođavanje aplikacije bez potrebe za promjenama u kodu. Svaka postavka je predstavljena kao ključ-vrijednost par, gdje ključ definira konfiguracijski parametar, a vrijednost određuje postavku tog parametra.

Opće postavke aplikacije

```
APP_NAME=Laravel
```

Ime aplikacije koje se može koristiti unutar aplikacije, npr. u naslovima ili e-mailovima.

```
APP_ENV=local
```

Definira okruženje u kojem aplikacija radi (npr. local, production, staging). Ovo može utjecati na ponašanje aplikacije, kao što su logiranje grešaka ili način rada baza podataka.

```
APP_KEY=base64:NasumičnoGeneriraniKljuč
```

Tajni ključ koji se koristi za enkripciju podataka. Ovaj ključ je ključan za sigurnost aplikacije i mora biti jedinstven za svaku instancu aplikacije. Generira se sa `php artisan key:generate`. Ako naredba uspješno završi, dobit ćete poruku s novim ključem, a ključ će biti upisan u `.env` datoteku. `APP_KEY` koristi se za enkripciju podataka, sesije i kolačiće i zaštitu od manipulacija.

```
APP_DEBUG=true
```

Kada je postavljeno na `true`, prikazuje detaljne greške i informacije o pogreškama, što je korisno za razvoj. U produkciji, ovo bi trebalo biti postavljeno na `false`.

```
APP_TIMEZONE=UTC
```

Postavka vremenske zone koju aplikacija koristi. Ova postavka određuje vrijeme koje će se koristiti u aplikaciji. Moguće je npr. Koristiti `Zagreb\Europe`.

```
APP_URL=http://localhost
```

URL aplikacije. Ovo se koristi kao osnovna URL adresa za generiranje linkova unutar aplikacije.

Postavke jezika

`APP_LOCALE=en`

Zadani jezik (lokalizacija) aplikacije.

`APP_FALLBACK_LOCALE=en`

Jezik na koji će se aplikacija prebaciti ako zadani jezik nije dostupan.

`APP_FAKE_LOCALE=en_US`

Jezik koji se koristi za generiranje lažnih podataka (npr. imena, adrese) koristeći biblioteku Faker.

Postavke za održavanje

`APP_MAINTENANCE_DRIVER=file`

Definira gdje će se čuvati informacije o održavanju aplikacije. Moguće opcije su `file` ili `database`.

Sigurnosne postavke

`BCRYPT_ROUNDS=12`

Broj rundi koje će se koristiti za BCrypt algoritam prilikom hashiranja lozinki. Veći broj rundi povećava sigurnost, ali i zahtjeva više procesorske snage.

Logiranje

`LOG_CHANNEL=stack`

Definira koji će se kanal koristiti za logiranje. Laravel podržava različite kanale kao što su `single`, `daily`, `slack`, itd. **Mi koristimo daily, jer se onda logovi slažu po danima.**

`LOG_STACK=single`

Ako se koristi stack kanal, definira koje kanale logiranje koristi. U ovom slučaju, single znači da se logovi spremaju u jednu datoteku.

`LOG_LEVEL=debug`

Nivoa logiranja, koja određuje koje poruke će se zapisivati u logove. Opcije uključuju `debug`, `info`, `notice`, `warning`, `error`, `critical`, `alert`, i `emergency`.

Baza podataka

```
DB_CONNECTION=mysql
```

Definira koji driver za bazu podataka se koristi (npr. `mysql`, `pgsql`, `sqlite`). Ovdje je postavljen na `mysql`.

Ostale postavke (`DB_HOST`, `DB_PORT`, `DB_DATABASE`, `DB_USERNAME`, `DB_PASSWORD`) su u originalu **s komentarima koje treba ukloniti**. To nije potrebno kada se koristi `sqlite`, ali su potrebne kada se koristi npr. MySQL.

```
DB_HOST=127.0.0.1  
DB_PORT=3306  
DB_DATABASE=ime-projekta  
DB_USERNAME=root  
DB_PASSWORD=
```

`DB_CONNECTION=sqlite` treba promijeniti u `DB_CONNECTION=mysql` iako koristimo MariaDB.

`DB_DATABASE` treba izmjeniti u `Taravel-blog`, jer se tako zove ime našeg projekta.

Kada projekt dižemo na testni server (pogotovo na produkcijski server) trebamo unijeti `DB_PASSWORD`, to lokalno nije bitno. Treba pripaziti da `ime-projekta` bude isto kao i ime direktorija gdje se nalazi projekt.

Sesije

```
SESSION_DRIVER=file
```

Određuje način na koji će se pohranjivati sesije. Opcije uključuju file, `database`, `redis`, `memcached`, itd.

Mi koristimo `file`. Također i `CACHE_STORE=file`

```
SESSION_LIFETIME=120
```

Definira trajanje sesije u minutama.

Postavke za pohranu i redove zadataka

```
FILESYSTEM_DISK=local
```

Definira koji se disk koristi za pohranu datoteka. local znači da se koristi lokalni sistem datoteka.

```
QUEUE_CONNECTION=database
```

Definira koji se sistem koristi za upravljanje redovima zadataka. Ovdje je postavljeno na database, što znači da se koristi baza podataka za pohranu zadataka.

Redis

Redis u kombinaciji s Laravelom služi kao keš koji poboljšava performanse aplikacije.

`REDIS_CLIENT=phpredis`

Definira koji Redis klijent će se koristiti. Opcije uključuju phpredis i predis.

`REDIS_HOST=127.0.0.1`

Adresa Redis servera.

`REDIS_PORT=6379`

Port na kojem Redis sluša.

E-mail

`MAIL_MAILER=log`

Definira način na koji će se slati e-mailovi. log znači da će se e-mailovi zapisivati u log umjesto slati.

`MAIL_FROM_ADDRESS="hello@example.com"`

E-mail adresa s koje se šalju e-mailovi.

`MAIL_FROM_NAME="${APP_NAME}"`

Ime koje će se prikazivati kao pošiljatelj e-maila. Ovdje koristi naziv aplikacije.

AWS

`AWS_ACCESS_KEY_ID, AWS_SECRET_ACCESS_KEY, AWS_DEFAULT_REGION, AWS_BUCKET`

Postavke za povezivanje na Amazon Web Services (AWS) i korištenje S3 pohrane ili drugih AWS usluga.

Vite

`VITE_APP_NAME="${APP_NAME}"`

Postavka za frontend alat Vite, koji se koristi za razvoj JavaScript i CSS modula. Koristi naziv aplikacije definiran u `APP_NAME`.

`.env` datoteka omogućava jednostavno upravljanje konfiguracijskim postavkama u Laravel aplikaciji. Ove postavke mogu se lako mijenjati u različitim okruženjima (lokalno, produkcija), bez potrebe za izmjenama koda, što čini aplikaciju fleksibilnijom i jednostavnijom za održavanje.

Pokretanje Arisan servera i migracije

Znamo da pokretanjem migracije `php artisan migrate`, ide u `database.sqlite` ali mi to ne želimo. Mi želimo da to ide u MySQL. Zato moramo podesiti `.env` datoteku. U `web.php` datoteci izmjenit ćemo kod tako da umjesto pogleda `welcome`, napišemo `welcome.bla` (koji ne postoji).

```
<?php

use Illuminate\Support\Facades\Route;

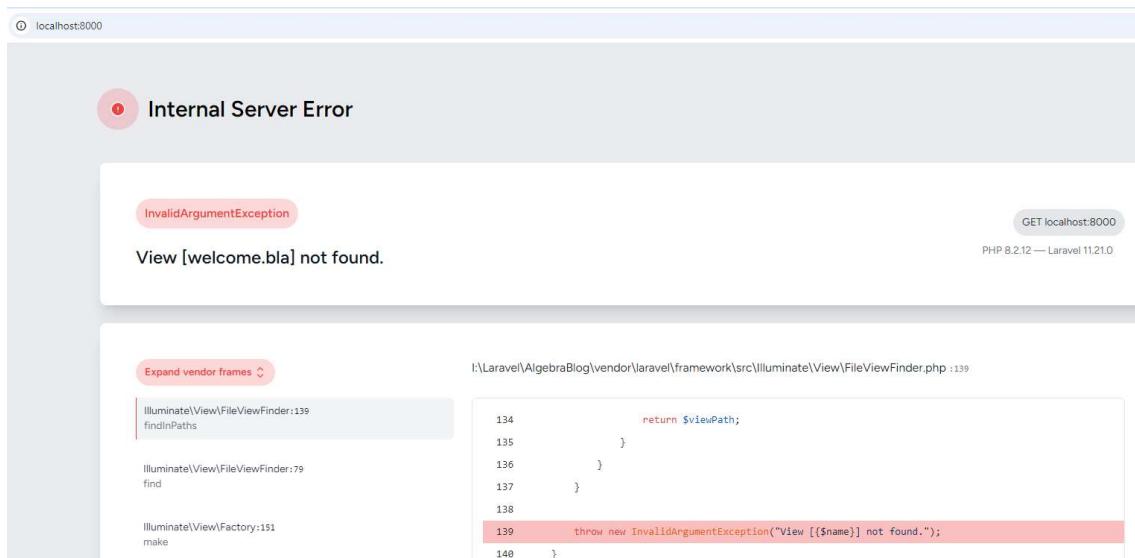
Route::get('/', function () {
    return view('welcome.bla');
});
```

Prvo ćemo pokrenuti devlopement server u drugom prozoru:

```
php artisan serve
```

```
php artisan serve
INFO Server running on [http://127.0.0.1:8000].
Press Ctrl+C to stop the server
```

Dobit ćemo grešku jer pogled `welcome.bla` ne postoji.



U produkciji `APP_DEBUG` ne smije ostati na `true`, treba ga prebaciti na `false`.

Ako sada pogledamo `localhost:8000` dobit ćemo sljedeće:



Dakle vrlo jednostavna greška **500 Server Error**, bez objašnjenja.

Vratit ćemo u `web.php` datoteci kod u pogledu `welcome` da bude kako je bilo, na `welcome` (koji postoji).

Ako sada pokušamo pokrenuti migraciju imat ćemo problem ako MySQL nije upaljen.

Ako je sve u redu, `php artisan migrate` kreirao je bazu podataka i migracijsku tablicu kako bi pratio kada je koja migracija pokrenuta. Ova migracija je batch 1 tj. kreirana je `0001_01_01_000000_create_users_table.php`.

```
$ php artisan migrate
[WARN] The database 'laravel-blog' does not exist on the 'mysql' connection.
Would you like to create it? (yes/no) [yes]
> yes
[INFO] Preparing database.
Creating migration table ..... 54.49ms DONE
[INFO] Running migrations.
0001_01_01_000000_create_users_table ..... 72.72ms DONE
Administrator@DESKTOP-4J5DF41 MINGW64 /i/Laravel/AlgebraBlog
$ |
```

Zato je uvijek bolje pokretati migraciju po migraciju a ne odjednom više migracija.

Ako pokrenemo MySQL WorkBench vidjet ćemo da li su kreirane tablice:

Navigator

SCHEMAS

Filter objects

- ▶ adventureworkshop
- ▶ algebra_crm
- ▶ fcc
- ▶ firma
- ▼ laravel-blog
 - ▼ Tables
 - ▶ migrations
 - ▶ password_reset_tokens
 - ▶ sessions
 - ▶ users
 - ▶ Views
 - ▶ Stored Procedures
 - ▶ Functions

Vidimo `users` i `password_reset_tokens`. Tablica `migrations` se kreira automatski kod prvog pokretanja migracija zato što u toj tablici se nalaze `id`, `migration` i `batch`. tj. broj i ime migracije tako da znate koju ste migraciju pokrenuli.

| <code>id</code> | <code>migration</code> | <code>batch</code> |
|-----------------|--------------------------------------------------|--------------------|
| 1 | <code>0001_01_01_00000_create_users_table</code> | 1 |
| NULL | NULL | NULL |

Recimo da sada u tablici `users` želimo imati umjesto samo `name`, `first_name` i `last_name`.

Ako ponovo pokrenemo migraciju sa `php artisan migrate` javit će nam:

```
$ php artisan migrate
INFO Nothing to migrate.
```

Dakle iako smo napravili izmjenu migracija ne rolazi. Moramo napraviti rollback.

Kako smo u metodi `down()` stavili dropanje tablica, to će se i desiti kada rollback pokrenemo:

```
/**
 * Reverse the migrations.
 */
public function down(): void
{
    Schema::dropIfExists('users');
    Schema::dropIfExists('password_reset_tokens');
}
```

Pokrenimo rollback:

```
php artisan migrate:rollback
```

```
$ php artisan migrate:rollback
INFO Rolling back migrations.

0001_01_01_00000_create_users_table ..... 31.21ms DONE

Administrator@DESKTOP-4J5DF41 MINGW64 /i/Laravel/AlgebraBlog
$ |
```

`-step=` se koristi ako želimo napraviti više koraka unazad. To nam ovdje ne treba.

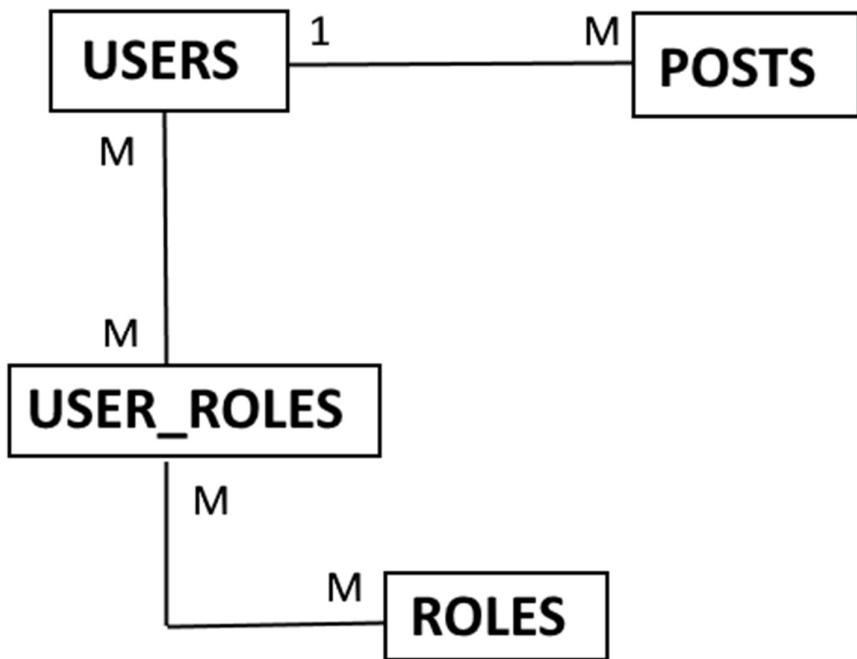
Ovime smo pokrenuli metodu `down()`. Sada su sve tablice obrisane, osim tablice `migrations` koja je prazna.

| <code>id</code> | <code>migration</code> | <code>batch</code> |
|-----------------|------------------------|--------------------|
| NULL | NULL | NULL |

Tek sada možemo pokrenuti migraciju i dobiti izmijenjeno stanje.

Ovdje je potencijalni problem ako smo upisali podatke jer rollback radi drop i podaci se izgube. Dakle, rollback je dobar u početnoj fazi.

Pogledajmo kako napraviti prvu migraciju. Razradit ćemo shemu:



1 user može imati više različitih uloga (**Roles**, može biti administrator a može biti i korisnik). O ovome u stvarnom slučaju treba razmisliti. U rolama (ulogama, Roles) moći ćemo dodijeliti nekakve dozvole (engl. permission). Dakle moći ćemo gledati user ima određenu rolu ili user ima određeni permission. Permission nećemo vezati na user-a, nego ćemo permission vezati na rolu i na taj način raditi provjeru. Permission znači da li korisnik može kreirati novi posti ili editirati postojeći post. Trebaju nam 3 tablice **roles** i pivot tablica **User_roles** za many-to-many relaciju. To ćemo u jednoj migraciji stvoriti a **post** će biti migracija za sebe.

Many-to-many tablice u bazi podataka modeliraju odnos između dva entiteta kada jedan entitet može biti povezan s više instanci drugog entiteta, i obratno. Ova vrsta relacije se ne može jednostavno predstaviti s dvije tablice, pa se koristi treća tablica, poznata kao pivot tablica, koja povezuje ta dva entiteta.

Pivot tablice u Laravelu se koriste za definiranje relacija mnogi prema mnogima (many-to-many) između dvije tablice u bazi podataka. U našem primjeru, **user_Roles** je pivot tablica koja povezuje tablice **users** i **roles**. Na ovaj način, korisnik može imati više uloga, a svaka uloga može biti dodijeljena različitim korisnicima.

Kreiranje tablice rola u Laravelu

U Laravelu, koncept rola obično se odnosi na mehanizam kontrole pristupa koji omogućuje administratorima da definiraju različite razine pristupa ili dozvola za korisnike unutar aplikacije. Radeći s rolama, programeri mogu implementirati složenije sisteme autorizacije koji osiguravaju da korisnici imaju samo onoliko pristupa koliko im je potrebno za obavljanje njihovih zadataka.

Rola (uloga) je skup dozvola koje se dodjeljuju korisnicima. Na primjer, možete imati uloge kao što su administrator, editor, korisnik, i sl. Rola može imati različite privilegije, kao što su čitanje, pisanje, ažuriranje ili brisanje resursa.

Dozvole (Permissions) su specifične akcije koje se mogu dodijeliti korisnicima ili ulogama. Na primjer, dozvola može biti create-post, edit-post, delete-post, itd. Korisnik može imati dozvolu bez obzira na to kojoj se ulozi pridružuje.

Dodjeljivanje uloga i dozvola

Korisnicima se dodjeljuju uloge, a zatim se tim ulogama dodjeljuju dozvole. To omogućuje fleksibilno upravljanje pristupom.

Laravel koristi paket kao što je Spatie Laravel Permission za implementaciju sistema uloga i dozvola, što olakšava rad s njima.

Sljedeća naredba u Laravelu služi za kreiranje nove migracije koja će se koristiti za izradu baze podataka, specifično za tablicu Roles (uloge). Ova naredba je dio Laravelovog alata za upravljanje migracijama i omogućuje jednostavno upravljanje strukturu baze podataka.

```
php artisan make:migration create_roles_table
```

Artisan je komandna linija (CLI) alata koji dolazi s Laravelom. Omogućuje programerima da izvode razne zadatke vezane uz razvoj aplikacija, kao što su migracije, testiranje, izrada kontrolera, itd.

`make:migration` koristi se za generiranje nove migracijske datoteke. Migracije su specijalizirane klase koje definiraju kako se tablica u bazi podataka kreira, ažurira ili briše.

Kada pokrenete ovu naredbu, Laravel će automatski kreirati novu migracijsku datoteku unutar direktorija `database/migrations`.

`create_roles_table` je argument imenuje migraciju. U ovom slučaju, `create_roles_table` označava da će migracija biti korištena za stvaranje tablice `roles` i Laravel to razumije. Preporučljivo je koristiti naziv koji jasno opisuje što migracija radi, tako da je kasnije lakše pretraživati i održavati kod.

Da smo umjesto `create_roles_table` naveli npr.

```
php artisan make:migration table_roles
```

Kreirat će se migracija u kojoj su samo dvije metode: `up()` i `down()`.koje su potpuno prazne. On ne zna da s fasadom `Schema::create` treba kreirati tablicu `roles` i da joj automatski dodjeli `id()` i `timestamp()`. A to ne zna zato što to nismo rekli kroz ime. Zato je bitna naming konvencija u Laravelu. Ako je poštujete bit će nam lakše:

```
php artisan make:migration create_roles_table
```

```
$ php artisan make:migration create_roles_table
INFO Migration [I:\Laravel\AlgebraBlog\database\migrations\2024_09_16_093138_create_roles_table.php] created successfully.
```

Kada pokrenete ovu naredbu, Laravel će generirati novu PHP datoteku unutar `database/migrations` direktorija. Ime datoteke će uključivati vremensku oznaku (engl. timestamp) kada je datoteka stvorena (samo početna datoteka došla s Laravelom ima oznaku `0001_01_01_000000`) koja osigurava jedinstvenost, na primjer:

```
2024_08_30_000000_create_roles_table.php
```

Datoteka će imati sljedeći osnovni format:

```
<?php

use Illuminate\Database\Migrations\Migration;
use Illuminate\Database\Schema\Blueprint;
use Illuminate\Support\Facades\Schema;

class CreateRolesTable extends Migration
{
    /**
     * Run the migrations.
     *
     * @return void
     */
    public function up()
    {
        Schema::create('roles', function (Blueprint $table) {
            $table->id();
            $table->string('name');
            $table->timestamps();
        });
    }

    /**
     * Reverse the migrations.
     *
     * @return void
     */
    public function down()
    {
        Schema::dropIfExists('roles');
    }
}
```

Vidimo da u `up()` metodi imao generiranje tablice `roles` i u `down()` imamo brisanje `roles`.

Evo detaljnije objašnjenje generirane migracije:

Namespace i korištene klase:

`use Illuminate\Database\Migrations\Migration;` – Osnovna klasa za migracije.

`use Illuminate\Database\Schema\Blueprint;` – Klasa koja omogućuje definiranje strukture tablica.

`use Illuminate\Support\Facades\Schema;` – Facade za rad s migracijama i tablicama.

Metoda `up()`:

Ova metoda sadrži logiku koja se izvršava kada se migracija primjenjuje (npr. kada se kreira nova tablica).

`Schema::create('roles', function (Blueprint $table) { ... })` – Definira novu tablicu `roles` i opisuje njenu strukturu.

U ovom primjeru, tablica `roles` će imati `id` i naziv uloge (name). `timestamps()` je kao podrazumijevana vrijednost (default) upaljen u [Migration klasi u početnoj migraciji](#). Laravel inače kroz Artisan naredbe nudi da stvorimo migracije, modele itd.

Metoda `down()`:

Ova metoda se koristi za "vraćanje" migracije. Na primjer, kada se pozove `php artisan migrate:rollback`, ova metoda će se izvršiti da bi se obrisala tablica.

`Schema::dropIfExists('roles');` – Briše tablicu `roles` ako ona postoji.

Predavač je još dodao u `roles`:

```
$table->string('name', 20)->unique();
```

Dakle polje name sa dužinom 20 koje mora biti jedinstveno (ne mogu biti dvije iste role). Ako nema zadan parametar dužine, dugačko je 255.

```
$table->json('permissions')->nullable()->default([]);
```

`json()` radi različito na MySQL i na MariaDB. U MariaDB `json()` je alias za LONGTEXT tip podataka i drugačije ga definira. To znači da je drugačiji default value. Dakle taj red neće proći ako koristimo MariaDB. Predavač kaže da inače inzistira na JSON formatu a ne običnom tekstu zato što JSON nudi mogućnost pretrage unutar JSON formata. Dakle moguće je potražiti da li je npr. neki permission = true što u bazama podataka MariaDB nema. Sličan problem se poljavljuje na SQL Lite.

Ovdje nećemo stavljati default value nego ćemo samo reći `nullable()` što znači da je permission neobavezno polje dok je `name` mandatory polje.

Obavezno polje (engl. mandatory) je ono koje mora imati neku vrijednost. U bazi podataka, obavezno polje se implementira postavljanjem kolone kao `NOT NULL`, što znači da ta kolona ne smije biti prazna ili imati vrijednost `NULL`. Neobavezno polje (engl. optional) je ono koje može, ali ne mora imati vrijednost. U bazi podataka, neobavezno polje je definirano s mogućnošću da bude `NULL`. Ako korisnik ne unese vrijednost za takvo polje, ono će automatski dobiti vrijednost `NULL`. Dakle ono koje je `nullable()` je neobavezno polje

```
$table->json('permissions')->nullable();
```

Cijela `up()` metoda izgleda ovako:

```
public function up(): void
{
    Schema::create('roles', function (Blueprint $table) {
        $table->id();
        $table->string('name', 20)->unique();
        $table->json('permissions')->nullable();
        $table->softDeletes();
        $table->timestamps();
    });
}
```

Dodat ćemo još i

```
$table->softDeletes();
```

Za [soft deleted](#) brisanje. Ovo treba stavljati na sve tablice osim pivot tablica.

Dodao je još i `user_roles` tablicu u `up()` metodu:

```
Schema::create('user_roles', function (Blueprint $table) {
    $table->foreignId('role_id')->constrained()->cascadeOnDelete();
    $table->foreignId('user_id')->constrained()->cascadeOnDelete();
    $table->primary(['role_id', 'user_id']);
    $table->timestamps();

});
```

Objasnimo `user_roles` tablicu.

Ova tablica služi kao spojna tablica (engl. pivot table) između tablica `roles` i `users`, omogućujući vam da modelirate višestruke uloge za svakog korisnika i da svaki korisnik može imati više uloga.

Predavač kaže da često u tablicama nema `$table->id();` nego se primarni ključ kreira kao **kompozitni ključ**. Kompozitni ključ je ključ koji se sastoji od dva ili više stupaca (polja) u tablici, zajedno tvoreći jedinstvenu kombinaciju vrijednosti. U pivot tablicama koje povezuju relacije "mnogi prema mnogima" (many-to-many), kompozitni ključ se često koristi kako bi se osiguralo da isti zapis ne može postojati više puta.

U našem slučaju, pivot tablica `user_roles` povezuje korisnike (users) i uloge (roles). Ova pivot tablica može imati kompozitni ključ koji se sastoji od dva polja: `user_id` i `role_id`. Kombinacija ova dva polja mora biti jedinstvena, što znači da jedan korisnik ne može imati istu ulogu više puta. Kombinacija `user_id` i `role_id` bit će jedinstveni identifikator.

Definiranje `role_id`:

```
foreignId('role_id')->constrained()->cascadeOnDelete();
```

Ovaj red stvara stranu ključa `role_id`, koja se odnosi na primarni ključ u tablici `roles`.

`constrained()` automatski dodaje ograničenja (constraints) na ovaj stupac, što znači da se очekuje da `role_id` postoji u tablici `roles`.

`cascadeOnDelete()` osigurava da ako se neka uloga (role) izbriše iz tablice `roles`, svi redovi u tablici `user_roles` koji se odnose na tu ulogu također budu izbrisani.

Definiranje `user_id`:

```
foreignId('user_id')->constrained()->cascadeOnDelete();
```

Slično kao i `role_id`, ovaj red kreira stranu ključa `user_id`, koja se odnosi na primarni ključ u tablici `users`.

Ovaj red također osigurava da se svi redovi koji se odnose na određenog korisnika izbrišu kada se korisnik obriše iz tablice `users`.

Primarni ključ:

```
$table->primary(['role_id', 'user_id']);
```

Ovaj red postavlja kombinaciju `role_id` i `user_id` kao primarni ključ za tablicu `user_roles`. To osigurava da svaki korisnik može imati samo jednu instancu svake uloge, tj. da ne možete imati dva identična reda za istog korisnika i istu ulogu.

Dodan je i `timestamps` o kojem smo već govorili.

`softDeletes()` ne treba na `user_roles` zato kada brišemo neku rolu, logično je da će se obrisati i relacija prema useru, `cascadeOnDelete()`. Obrišemo li, isto tako usera, onda nema što postojati relacija prema ulozi (roli). Zato pivot tablica nikada nema `softDeletes()`.

Višestruke uloge: Ovaj kod omogućuje korisnicima da imaju više uloga. Na primjer, korisnik može biti i "administrator" i "urednik".

Referencijalni integritet: Korištenjem `foreignId` i `cascadeOnDelete`, osigurava se da su odnosi između korisnika i njihovih uloga uvijek konzistentni. Kada se korisnik ili uloga izbrišu, njihovi podaci u tablici `user_roles` se također automatski brišu, što sprječava "siroče" zapise (engl. orphan records)¹.

Jednostavnost upravljanja: Kroz ovu tabelu, lakše je upravljati ulogama i pristupima korisnika u aplikaciji. Možete brzo provjeriti koje uloge ima određeni korisnik ili koje korisnike ima određena uloga.

Kako se koristi?

Nakon što je migracijska datoteka generirana i prilagođena prema potrebama, možete primijeniti migraciju pomoću komande:

```
php artisan migrate
```

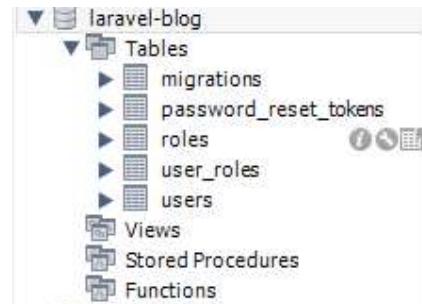
```
$ php artisan migrate
INFO  Running migrations.
, 2024_09_16_093138_create_roles_table ..... 129.63ms DONE
```

Dobro je pokrenuti ovu naredbu jer možemo imati više nepokrenutih migracija, kada pokrenemo ovu naredbu pokrenu se sve migracije. Ista priča je i s rollback-om, napravit će ih sve u nazad. Moguće je ući ručno u tablicu `migration` i u polju `batch` poredati ih sve redom.

Ako je sve u redu trebamo vidjeti te dvije tablice:

| id | migration | batch |
|----|--------------------------------------|-------|
| 1 | 0001_01_01_000000_create_users_table | 1 |
| 2 | 2024_09_16_093138_create_roles_table | 2 |

| role_id | user_id | created_at | updated_at |
|---------|---------|------------|------------|
| NULL | NULL | NULL | NULL |



Kreiranje tablice post

Ostala nam je još jedna migracija user post, 1 to many. Jedan user može imati više postova ali post može imati samo jednog usera.

```
php artisan make:migration create_posts_table
```

¹ Siroče zapisi su zapisi u bazi podataka koji nemaju odgovarajući (ili povezani) zapis u drugoj tablici. Ovi zapisi nastaju kada se referencijalni integritet ne održava, odnosno kada se podaci u jednoj tablici brišu bez odgovarajućeg brisanja ili ažuriranja povezanih podataka u drugoj tablici.

Ovime će Laravel generirati novu migracijsku datoteku koja će se nalaziti u direktoriju `database/migrations`. Ova datoteka će sadržavati strukturu tablice koju želimo definirati, u ovom slučaju tablice `posts`.

```
$ php artisan make:migration create_posts_table
INFO Migration [I:\Laravel\AlgebraBlog\database\migrations\2024_09_16_110536_create_posts_table.php] created successfully.
```

Možemo primjetiti da je vrijeme pomaknuto za 2 sata:

| |
|------------------------------------------|
| 0001_01_01_000000_create_users_table.php |
| 2024_09_16_093138_create_roles_table.php |
| 2024_09_16_110536_create_posts_table.php |

Razlog tome je što je naše lokalno ljetno vrijeme UTC+2 a zimsko je UTC+1

Ako sada pogledamo ovu migraciju, vidjet ćemo da nam `up()` metoda ne odgovara u potpunosti:

```
public function up(): void
{
    Schema::create('posts', function (Blueprint $table) {
        $table->id();
        $table->timestamps();
    });
}
```

Dodat ćemo neke redove:

```
public function up(): void
{
    Schema::create('posts', function (Blueprint $table) {
        $table->id();
        $table->string('title');
        $table->text('content');
        $table->string('slug')->unique();
        $table->string('image')->nullable();
        $table->foreignId('user_id')->constrained()->cascadeOnDelete();
        $table->timestamps();
    });
}
```

Ubacili smo i `slug` i `image`. Kada radimo stranice želimo ih optimizirati za tražilice i verziju naslova ili sličnog teksta. To je obično URL-friendly niz koji zamjenjuje razmake s crtama (npr. `moja-nova-objava`)

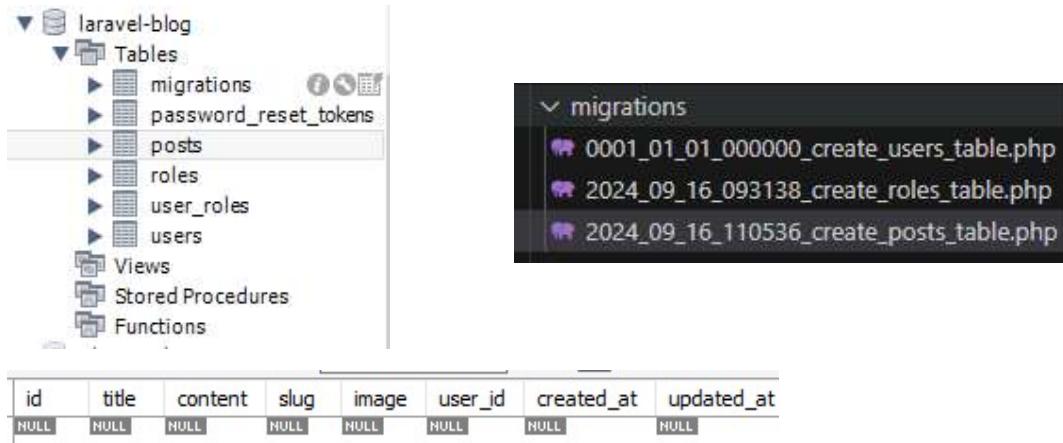
umjesto [Moja nova objava](#), dakle self friendly URL). On se generira iz naslova posta. Atribut `image` označava naziv slike povezane s postom. Tip je string, što znači da bi taj stupac mogao pohranjivati samo naziv ili putanju slike u obliku tekstualnog niza, a ne samu sliku. Opcija `nullable()` znači da nije obavezno imati sliku za svaki post, tj. ovaj stupac može ostati prazan.

Polimorfne relacije u Eloquentu (Laravelovom ORM-u) omogućuju jedan model da bude povezan s više drugih modela putem jedne relacije. Kada se radi o slikama (`image`), polimorfne relacije omogućuju da isti model slike može biti povezan s različitim entitetima, kao što su postovi, korisnici, proizvodi itd. Ova fleksibilnost znači da jedan model može "posjedovati" slike, bez obzira na to koji tip modela je u pitanju.

Možemo uploadovati fotografije i vezati za bilo koga. Za to su **zgodne polimorfne relacije (engl. Polymorphic Relationships)** koje mogu biti dijeljene s postom, user-om itd (jedan model može imati različite tipove povezanih modela). Mi ćemo za primjer ići s jednom slikom.

Pokrenut ćemo migraciju još jednom:

```
$ php artisan migrate
INFO: Running migrations.
2024_09_16_110536_create_posts_table ..... 91.93ms DONE
```



Migracijama ćemo se još vratiti. Ostatak treba pogledati u dokumentaciji.

Modeli

Nakon što smo uspostavili bazu, trebamo namapirati modele u tablice. To ćemo raditi ručno. Ako pogledamo u dokumentaciji [Databases->Getting Started](#) možemo raditi s različitim bazama (MariaDB, MySQL, PostgreSQL, SQLite, SQL Server).

Fluent Query Builder omogućava pisanje sirovih SQL upita na jednostavan, fluentan način koristeći PHP sintaksu. Primjenjuje se kada želite imati potpunu kontrolu nad SQL upitim ili kada ti treba nešto što Eloquent ne podržava direktno. Omogućuje direktni pristup bazi podataka koristeći SQL, što znači da možete pisati visoko optimizirane i prilagođene upite.

Fluent Query Builder ne koristi modele niti objektnu mapu, pa je često manje memorijski intenzivan od Eloquenta. Kada trebate pisati jako složene SQL upite s mnogim pridruživanjem (`JOIN`), podupitim, agregacijama itd., Query Builder ti daje veću fleksibilnost.

Evo primjera:

```
$users = DB::table('users')
    ->where('status', 'active')
    ->orderBy('created_at', 'desc')
    ->get();
```

Ovdje Query Builder generira SQL upit za dobivanje aktivnih korisnika, bez potrebe za modelom. Možete direktno kontrolirati kolone, `JOIN` uslove, `WHERE` uslove itd.

Eloquent ORM (Object-Relational Mapping) omogućava rad s bazom podataka na način da modeli predstavljaju tablice. Svrha Eloquenta je da olakša rad s podacima kao da su objekti, omogućujući intuitivniji i jednostavniji način interakcije s bazom podataka. Eloquent koristi modele za rad s bazama podataka. Svaki red iz baze podataka mapira se na objekt u PHP-u, što omogućava rad s podacima kao da su objekti. Eloquent pojednostavljava rad s relacijama između tablica, kao što su `hasOne`, `hasMany`, `belongsTo`, `morphTo`, itd. Automatski upravlja upitimima za povezane podatke (npr. `Post::with('comments')->get()`). Eloquent automatizira mnoge uobičajene SQL operacije, kao što su `SELECT`, `INSERT`, `UPDATE` i `DELETE`, bez potrebe za pisanjem eksplicitnih upita.

Evo primjera:

```
$users = User::where('status', 'active')->orderBy('created_at', 'desc')->get();
```

Postoji više načina kako Laravel validira podatke. Jedan od načina je direktno u kontroleru `AuthController.php`. Predavač kaže da to nikada ne koristi. On koristi form request-ove.

Ovdje Eloquent koristi model User i automatski upravlja SQL upitimima, a podaci se vraćaju kao kolekcija User objekata. Koristeći Eloquent, možete brzo i jednostavno kreirati upite jer koristi objektno orijentirani pristup. Jedna od najjačih karakteristika Eloquenta je upravljanje relacijama između različitih modela (npr. `hasMany`, `belongsTo`). Kod s Eloquentom je često čišći i intuitivniji, posebno kod jednostavnijih upita.

| Fluent Query Builder | Eloquent ORM |
|----------------------------------------------------------------|------------------------------------------------------------------------|
| Radi s SQL upitimima direktno. | Radi s modelima koji mapiraju tablice. |
| Bolji za složene upite ili prilagođene SQL upite. | Pogodniji za jednostavne upite i rad s relacijama između tablica. |
| Fleksibilniji i učinkovitiji u memoriji jer ne koristi modele. | Može biti sporiji kod kompleksnih upita zbog korištenja objekata. |
| Sirova kontrola nad SQL-om i njegovom optimizacijom. | Automatski upravlja odnosima između modela i podataka. |
| Moraš pisati upite ručno. | Upiti se generiraju automatski temeljem definiranih modela i relacija. |

Fluent Query Builder je bolji kada ti treba visok nivo kontrole nad SQL-om ili radimo s vrlo složenim upitim. **Eloquent ORM** je bolji kada radimo s modelima i relacijama između podataka te želimo iskoristiti jednostavniju sintaksu i objektno-orientirani pristup.

Predavač preporuča korištenje Eloquent ORM. On mapira podatke iz baze na model. Ti mapperi su u programiranju danas uobičajena stvar.

Default model User

Jedan model već imamo napravljen i on se zove User, on je došao sa Laravelom. Mi ćemo krenuti s [Modelima](#). To su Eloquent modeli koje kasnije ispravno mapiramo. Modeli imaju svoje generičke metode koje se moraju proširiti iz [Illuminate\Database\Eloquent\Model](#) klase. Uzmemo li naš model koji trenutno imamo u direktoriju [App\Models\User.php](#), vidjet ćemo da ime modela mora biti jednina (engl. singular). Ime modela [User.php](#) koji je tamo je jednina. Na taj način Laravelov Eloquent mapira sa tablicom u bazi. Ne moramo navoditi tom modelu na koju tablicu se mora povezati. Laravel to napravi automatski, ako se držimo naming konvencije. Tablica plural a model singular. Ne moramo se naravno toga držati ali smo tada dužni u modelu definirati tablicu na koju se vežemo. Pogledajmo prvo što ima u [User.php](#).

U [User.php](#) možemo vidjeti `class User extends Authenticatable` što znači da naša klasa [User](#) ne nasljeđuje [Model](#). Kao što smo već rekli (a i piše u dokumentaciji) nasljeđuje [Illuminate\Database\Eloquent\Model](#) klasu. To je zato što nasljeđuje alias.

```
use Illuminate\Foundation\Auth\User as Authenticatable;
```

[User](#) je klasa koja ekstndi [Authenticatable](#) model. Taj [User](#) je model izgrađen u Laravelu kako bi imali kompletan mehanizam za autentifikaciju i autorizaciju korisnika. S time je jednostavno doći do korisnika koji je napravio zahtjev (engl. request) i na taj način vidjeti ima li on dozvolu da to napravi. Ne bi smjeli mijenjati to, tj. Naš [user](#) mora uvijek naslijediti [Authenticatable](#). Zato koristimo alias da se imenski prostori ne bi sudarili.

[HasFactory](#) je trait koji služi ako koristimo tzv. seedere koji omogućuju da napunimo bazu sa hrpom podataka. Nad modelom možemo koristiti factories koji nam to omogućava. Da bi to mogli moramo imati trait [HasFactory](#). Tako [seed-er](#) može kreirati podatak. Seed-er se mora pisati ručno i zato se koristi [Factory](#), o čemu ćemo kasnije govoriti.

Klasične klase omogućavaju da funkcije (metode) iz jedne klase koristimo u drugoj klasi tako da nasljeđujemo sa [extends](#). To može stvoriti probleme. Trait je mehanizam (skup funkcionalnosti) u PHP-u koji omogućava višestruko nasljeđivanje metoda u klasi (ima svojstva, ima metode) i umjesto [class](#) piše [trait](#). Koristi se kada želite podijeliti metode između različitih klasa, bez potrebe za nasljeđivanjem iz jedne osnovne klase. [trait](#) nam omogućava ponovno korištenje koda u više klasa bez ograničenja nasljeđivanja samo jedne roditeljske klase. [trait](#) možemo uključiti u bilo koju klasu. PHP ne podržava višestruko nasljeđivanje klasa, ali pomoću trait-a možete "dodati" metode iz više različitih trait-ova u jednu klasu. Gdje nam je potreban [trait](#), unutar klase kažemo [use](#).

[notifiable](#) je trait koji omogućava korisnicima da primaju različite vrste notifikacija (npr. e-mail notifikacije).

```
# Generiraj model i FlightFactory klasu...
php artisan make:model Flight --factory
php artisan make:model Flight -f

# Generiraj model i FlightSeeder klasu...
php artisan make:model Flight --seed
php artisan make:model Flight -s

# Generiraj model i FlightController klasu...
php artisan make:model Flight --controller
php artisan make:model Flight -c

# Generiraj model, klasu resursa FlightControlleri klase zahtjeva forme...
php artisan make:model Flight --controller --resource --requests
php artisan make:model Flight -crR

# Generiraj model i FlightPolicy klasu...
php artisan make:model Flight --policy

# Generiraj model i migraciju, factory (tvornicu), seeder kontroler...
php artisan make:model Flight -mfsc

# Prečica za generiranje modela, migracije, seeder-a, politike (policy), kontrolera
# i zahtjeva forme...
php artisan make:model Flight --all
php artisan make:model Flight -a

# Generiraj pivot model...
php artisan make:model Member --pivot
php artisan make:model Member -p
```

Mi ćemo svaki od ovih posebno obraditi.

```
// Definiranje polja koja se MOGU mass assignat-i
protected $fillable = [
    'name',
    'email',
    'password',
];
```

`$fillable`: Ova varijabla definira koje se stupci iz baze podataka mogu masovno dodjeljivati (engl. mass assignment)², tj. koje vrijednosti možemo popuniti kada stvaramo ili ažuriramo korisnički model. U ovom primjeru to su `first_name`, `last_name` i `email`. To je bijela lista. Ta polja mogu biti `is_admin`, `$fillable`.

Ovo je sigurnosna mјera koja sprječava mass assignment vulnerability, gdje bi maliciozni korisnik mogao pokušati unijeti vrijednosti u neželjene stupce u bazi podataka.

`$guarded` je suprotna lista koja se koristi za zaštićene ključne atribute. Njome definiramo polja koja se NE MOGU mass assignat-i. To je crna lista.

```
protected $hidden = [
    'password',
    'remember_token',
];
```

`$hidden`: Ova varijabla definira koje će kolone biti sakrivene kada se korisnički model serijalizira (npr. kad se vraća kao JSON odgovor iz API-ja). U ovom slučaju, `password` i `remember_token` (za funkcionalnost "zapamti me") neće biti uključeni u API odgovore ili druge forme serijalizacije.

```
protected function casts(): array
{
    return [
        'email_verified_at' => 'datetime',
        'password' => 'hashed',
    ];
}
```

`$casts`: Ova metoda vraća asocijativni niz koji specificira kako će se određeni atributi preoblikovati (cast) prilikom dohvata ili pohrane u bazu podataka.

- `email_verified_at => datetime`: Polje `email_verified_at` će se automatski konvertirati u PHP `DateTime` objekt kada se pristupi tom atributu.
- `password => hashed`: Polje `password` automatski će se **hashirati** kad god se nova vrijednost pohrani u bazu podataka. Ovo je korisno jer se time osigurava da lozinke budu uvijek hashirane, a ne pohranjene u otvorenom tekstu.

² Pretpostavljamo da imamo formu koji šalje korisničke podatke (ime, prezime, email itd.). Mass assignment u Laravelu je mogućnost dodjeljivanja više atributa modelu odjednom kroz jednu matricu. Međutim, zbog sigurnosnih razloga, Laravel zahtijeva eksplicitno definiranje koja polja mogu biti mass assignable. Masovna dodjela omogućava da svi ovi podaci budu proslijeđeni modelu ovako: `User::create($request->all());` Korisnik tada može dobiti pristup administrativnim funkcijama.

Ako odgovarajuća tablica baze podataka vašeg modela ne odgovara ovoj konvenciji, možete ručno navesti naziv tablice modela definiranjem `table` svojstva na modelu:

```
<?php

namespace App\Models;

use Illuminate\Database\Eloquent\Model;

class Flight extends Model
{
    /**
     * Tablica povezana sa modelom.
     *
     * @var string
     */
    protected $table = 'my_flights';
}
```

Eloquent će također prepostaviti da odgovarajuća tablica baze podataka svakog modela ima stupac primarnog ključa pod nazivom `id`. Ako je potrebno, možete definirati zaštićeno `$primaryKey` svojstvo na svom modelu kako biste naveli drugi stupac koji služi kao primarni ključ vašeg modela:

```
<?php

namespace App\Models;

use Illuminate\Database\Eloquent\Model;

class Flight extends Model
{
    /**
     * Primarni ključ povezan s tablicom.
     *
     * @var string
     */
    protected $primaryKey = 'flight_id';
}
```

Osim toga, Eloquent prepostavlja da je primarni ključ inkrementirajuća cijelobrojna vrijednost, što znači da će Eloquent automatski pretvoriti primarni ključ u cijeli broj. Ako želite koristiti neinkrementirajući ili nenumerički primarni ključ, morate definirati javno `$incrementing` svojstvo na vašem modelu koje je postavljeno na `false`:

```
<?php

class Flight extends Model
{
    /**
     * Označava povećava li se ID modela automatski.
     *
     * @var bool
     */
    public $incrementing = false;
}
```

Ako primarni ključ vašeg modela nije cijeli broj, trebali biste definirati zaštićeno `$keyType` svojstvo na svom modelu. Ovo svojstvo treba imati vrijednost `string`:

```
<?php

class Flight extends Model
{
    /**
     * Tip podatka ID-a primarnog ključa.
     *
     * @var string
     */
    protected $keyType = 'string';
}
```

Ako primarni ključ vašeg modela nije cijeli broj, trebali biste definirati zaštićeno `$keyType` svojstvo na svom modelu. Ovo svojstvo treba imati vrijednost `string`:

```
<?php

class Flight extends Model
{
    /**
     * The data type of the primary key ID.
     *
     * @var string
     */
    protected $keyType = 'string';
}
```

Eloquent zahtijeva da svaki model ima barem jedan jedinstveni identifikacijski "ID" koji može poslužiti kao primarni ključ. "Kompozitne" primarne ključeve ne podržavaju modeli Eloquent. Međutim,

slobodni ste dodati dodatne jedinstvene indekse s više stupaca tablicama baze podataka uz jedinstveni identifikacijski primarni ključ tablice.

Promijenit ćemo još model `User.php`. Prije toga kreirat ćemo model roll i post, dva modela koja ćemo vezati s bazom i koja će znati mapirati kada bude trebao.

Ako preko modela želimo dobiti podatke, promijenit ćemo u `web.php`, da umjesto `return view('welcome');` bude `dd(User::all());`.

```
Illuminate\Database\Eloquent\Collection {#991 ▼ // routes\web.php:8
    #items: []
    #escapeWhenCastingToString: false
}
```

`dd i dump`

`dd i dump` služe za ispis podataka radi debugginga. `dd()` je Laravelova metoda (kratica za "dump and die") koja prvo ispisuje varijablu na ekran koristeći `dump()`, a zatim prekida daljnje izvršavanje skripte. Nijedna linija koda nakon `dd()` neće biti izvršena. `dump()` je PHP funkcija koja ispisuje varijablu ili više varijabli na ekran u čitljivom formatu, često za potrebe pregleda vrijednosti tijekom razvoja.

Roles i Post modeli

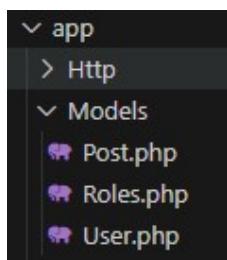
Idemo kreirati modele za `Roles` i `Post`.

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

```
$ php artisan make:model Roles
[INFO] Model [I:\Laravel\AlgebraBlog\app\Models\Roles.php] created successfully.

Administrator@DESKTOP-4J5DF41 MINGW64 /i/Laravel/AlgebraBlog
$ php artisan make:model Post
[INFO] Model [I:\Laravel\AlgebraBlog\app\Models\Post.php] created successfully.
```

Evo rezultata;



Post.php:

```
<?php

namespace App\Models;

use Illuminate\Database\Eloquent\Factories\HasFactory;
use Illuminate\Database\Eloquent\Model;

class Post extends Model
{
    use HasFactory;
}
```

Roles.php:

```
<?php

namespace App\Models;

use Illuminate\Database\Eloquent\Factories\HasFactory;
use Illuminate\Database\Eloquent\Model;

class Roles extends Model
{
    use HasFactory;
}
```

Iz `Users.php` možemo i ne moramo maknut ćemo trait koji nudi notifikacije. Dakle umjesto `use HasFactory, Notifiable` možemo ostaviti `use HasFactory`. Ipak predavač je ostavio kako je bilo.

Čim smo stvorili modele oni su funkcionalni. Pokušamo li opet u `web.php` napisati:

```
dd(User::all());
```



```
Illuminate\Database\Eloquent\Collection {#991 ▼ // routes\web.php:8
  #items: []
  #escapeWhenCastingToString: false
}
```

Vidimo da funkcionira.

Laravel nema distinkciju između monolitnih aplikacija i složenih projekata.

Ako bi smo radili rollback i pogledali batch-ove, možemo vidjeti u bazi. Ako bi radili rollback da bi smo došli do user-a i ponovo pokrenuli migracije, pogledat ćemo s MySQL našu bazu i tablicu migrations. Tamo možemo vidjeti naša 3 zapisa. Vidimo da je u 1. batch pokrenuta migracija users, u 2. batch roll i u 3. post. Ako napravimo jedan rollback onda će on otići za jedan. Međutim ako želimo nešto promijeniti na useru mi se moramo vratiti za 3 koraka. Problem je što obrišemo oba koraka. To možemo napraviti na ovaj način:

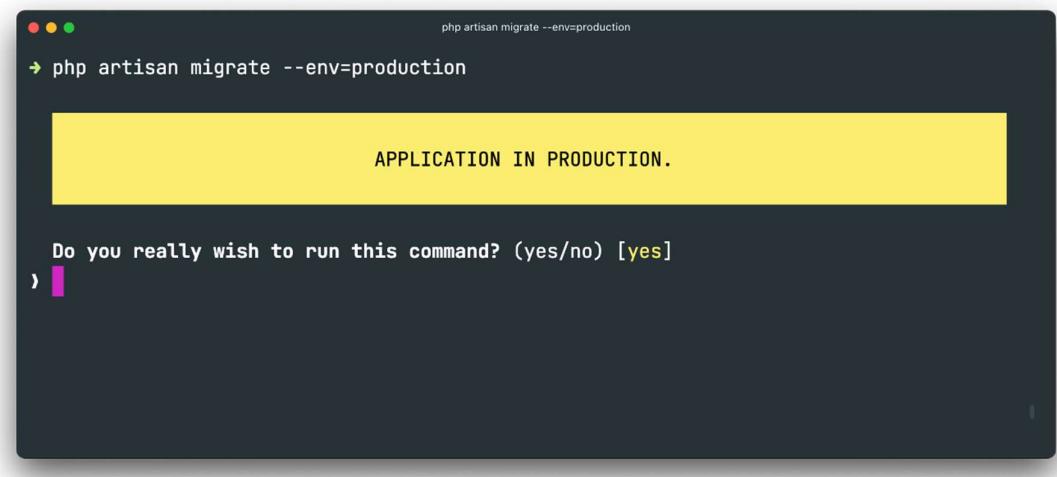
```
php artisan migrate:refresh --step=3
```

Problem je kada obrišemo role jer smo napunili tablicu s nekim ulogama. Ako to napravimo, moramo ponovo puniti tablicu. To nije problem ako imamo seeder-e, o kojima će kasnije biti riječi.

Predavač kaže da on samo dodaje novu migraciju. Ima flow, prati što se i kada mijenjalo na samoj tablici. Rollbackove preporuča koristiti samo u početku a kasnije samo nova migracija, nova migracija itd.

Zamislimo da u jednom stupcu (polju) imamo `name` i to trebamo razdvojiti u dva stupca `last_name` i `first_name`. To je problem jer kasnije ne znamo što je ime a što prezime pa ne možemo ni razdvojiti kasnije to kako treba. Moguće je bez problema dodati novi stupac ali to treba raditi tako da napravimo novu migraciju a ne na postojećoj migraciji. Taj primjer ćemo i pokazati. Čestim rollback-om će se pojaviti problemi.

Kada pokrenemo novu migraciju rollback je odličan za ispravljanje tipfelera ili krivog tipa podataka. Migracija treba biti funkcionalna i ne smije voditi gubitku podataka. Ovo pogotovo vrijedi kada smo u produkciji – migracija može napraviti itekako veliku štetu. (U environment-u mora biti da smo u produkciji).



```
php artisan migrate --env=production
→ php artisan migrate --env=production

APPLICATION IN PRODUCTION.

Do you really wish to run this command? (yes/no) [yes]
>
```

Migracije su code first pristup gdje ne moramo ništa prčkati po SQL-u. Kada pišemo migraciju, definiramo `up()` metodu, dakle s kodom. `down()` ako ne želimo da slučajno pokrenemo, zakomentiramo njen sadržaj i tada ne možemo raditi rollback. To je način kako sam sebe iskontrolirati da ne smrdamo bazu.

Krenuli smo graditi modele koji se vežu na tablicu na bazi. Tablica mora biti u pluralu (množini) a model u singularu (jednini). Ta naming konvencija omogućava da se da model automatski veže sa tablicom u bazi. Kreirali smo i novi model Post. Napunit ćemo njega i model Roll. Idemo popraviti `User.php` model. Laravel handl-a način na koji punite podatke u naš model, jedan po jedan.

Možemo reći u `web.php`:

```
<?php

use Illuminate\Support\Facades\Route;
use App\Models\User;

Route::get('/', function () {
    $user = new User();
    $user->name = 'John Doe';
    $user->email = 'bla@bla';
    $user->password = 'test';
    $user->save();
});
```

Distanciramo li taj model, možemo puniti svojstvo ručno. Međutim Laravel ima funkciju koja se zove **Mass Assignment**.

Vidjeli smo da kod kreiranja modela `php artisan make:model Ime` da će Eloquent ORM automatski kreirati klasu i naslijediti klasu Model iz Laravelovog core-a

Moguće je i puniti manualno. Moguće je reći da napunimo neko svojstvo i nakon toga `save()`. `save()` na modelu pošalje podatke koje smo napunili u bazu. Zamislimo da imamo desetke stupaca u bazi i iza svakog moramo napraviti `save()`. Ručno punjenje toga je nezgodno. Zato je Mass Assignment zgodno svojstvo.

Mi njemu predate neku matricu s podacima (kolekcija se obično naziva u Laravelu). Predate metodi iz Eloquenta, koja se zove `create` u Eloquentu i `create` će u samome modelu i praktično model iz te kolekcije će pokupiti podatke i automatski napuniti property-je u tom modelu koji su definirani u svojstvu `fillable`. Ako pogledamo `users fillable`, vidimo `name`, `email` i `password`. Ono što ovaj model omogućava je da automatski u jednom koraku kreiramo zapis u bazi i pošaljemo `name`, `email` i `password` kao nekakvu kolekciju. To radi pomoću metode `create`.

Međutim, prije korištenja metode `create`, morat ćete navesti svojstvo koje se može ispuniti ili čuvati na vašoj klasi modela. Ova su svojstva potrebna jer su svi Eloquent modeli prema zadanim postavkama

zaštićeni od ranjivosti masovne dodjele. Postoje određeni podatci koji su osjetljivi i ne bi ih korisnik trebao mijenjati (npr. ID). To se zove bindanje podataka na modelu.

Kako nam je tablica `users` potpuno prazna ubacit će ručno jednog korisnika.

| id | first_name | last_name | email | email_verified_at | password | remember_token | created_at | updated_at |
|----|------------|-----------|--------|-------------------|----------|----------------|------------|------------|
| 1 | Pero | Perić | per... | NULL | 123456 | NULL | NULL | NULL |

Apply **Revert**

Nakon toga obavezno pritisnuti dugme Apply

U Gitbash treba se pozicionirati u mapi projekta kako bi imali pristup Artisan naredbi (koji je dio Laravela). Pokrenut ćemo lokalni server ako nije pokrenut:

```
php artisan serve
```

`web.php` napraviti ćemo da izgleda ovako:

```
<?php

use Illuminate\Support\Facades\Route;
use App\Models\User;

Route::get('/', function () {
    //return view('welcome');
    dd(User::find(1));
});
```

Ako odemo na stranicu servera (`localhost:8000`) trebamo vidjeti samo taj model user-a s podacima iz baze podataka. Vidjet ćemo puno podataka koje možemo prebaciti u matricu.

Dakl umjesto `dd(User::find(1));` napisat ćemo `dd(User::find(1)->toArray());`

Sada ćemo dobiti samo matricu s podacima, bez modela:

```
array:7 [▼ // routes\web.php:8
  "id" => 1
  "first_name" => "Pero"
  "last_name" => "Perić"
  "email" => "pero@pero.com"
  "email_verified_at" => null
  "created_at" => null
  "updated_at" => null
]
```

```
App\Models\User {#990 ▼ // routes\web.php:8
  #connection: "mysql"
  #table: "users"
  #primaryKey: "id"
  #keyType: "int"
  +incrementing: true
  #with: []
  #withCount: []
  +preventsLazyLoading: false
  #perPage: 15
  +exists: true
  +wasRecentlyCreated: false
  #escapeWhenCastingToString: false
  #attributes: array:9 [▼
    "id" => 1
    "first_name" => "Pero"
    "last_name" => "Perić"
    "email" => "pero@pero.com"
    "email_verified_at" => null
    "password" => "123456"
    "remember_token" => null
    "created_at" => null
    "updated_at" => null
  ]
  #original: array:9 [▼
    "id" => 1
    "first_name" => "Pero"
    "last_name" => "Perić"
    "email" => "pero@pero.com"
    "email_verified_at" => null
    "password" => "123456"
    "remember_token" => null
    "created_at" => null
    "updated_at" => null
  ]
  #changes: []
  #casts: array:2 [▼
    "email_verified_at" => "datetime"
    "password" => "hashed"
  ]
  #classCastCache: []
  #attributeCastCache: []
  #dateFormat: null
  #appends: []
  #dispatchesEvents: []
  #observables: []
  #relations: []
  #touches: []
  +timestamps: true
  +usesUniqueIds: false
  #hidden: array:2 [▼
    0 => "password"
    1 => "remember_token"
  ]
  #visible: []
  #fillable: array:3 [▼
    0 => "name"
    1 => "email"
    2 => "password"
  ]
  #guarded: array:1 [▼
    0 => "*"
  ]
  #authPasswordName: "password"
  #rememberTokenName: "remember_token"
}
```

Ovo je čest slučaj kada frontend dobije podatke za user-a s određenim ID-om. On ako dobije te podatke, ovo iznad je njegov model. Kada update-a ove podatke, on binda model na svoju formu (obrazac) i na toj formi nešto promjeni (neki podatak). Međutim svi ovi podaci iznad se šalju backend-u tj. na bazu, tj. na naš model. On će nas pomoći Mass Assignment zaštiti da ne promjeni `created_at`, `updated_at` i `deleted_at` jer to ne želimo.

`fillable` je dozvoljeno napuniti a `guarded` nije dozvoljeno napuniti. `hidden` koristimo ako neki podatak želimo sakriti sa baze kada hvatamo podatke.

```
Protect hidden = [
    'password',
    'remember_token',
];
```

`casts` metoda omogućuje automatsku transformaciju podatka iz jednog tipa u drugi (npr. kod datuma).

Ako imamo vrijednost `boolean` u bazi to pohranjujemo kao `integer`. Kada povučemo podatke dobit ćemo `0` ili `1`. Ali mi želimo da dobijemo `true` ili `false`. Zato se koristi `cast` metoda.

Ako želimo koristi Mass Assignment moramo imati definiran `fillable` i njemu suprotan `guarded`. Rijetko kada ćemo ručno unositi podatke.

Moguće je provjeriti atribute sa `isdirty()` da bi smo provjerili da li je model promijenjen, `isclean()` se koristi da provjerimo da li je uopće rađen update. Zamislite formu gdje korisnik dobije mogućnost da promjeni podatke u svom profilu. Korisnik stisne Save i ništa ne promjeni. Tada nema potrebe išta slati bazi podataka jer se nije promijenio podatak na modelu. Upravo zato služe `isdirty()` i `isclean()`.

U `User.php` promjenili smo `$fillable` svojstvo tako da smo izbacili `name` a ubacili `first_name` i `last_name`:

```
/**
 * The attributes that are mass assignable.
 *
 * @var array<int, string>
 */
protected $fillable = [
    'first_name',
    'last_name',
    'email',
    'password',
];
```

U `Role.php` ubacit ćemo također `$fillable` tako da pogledamo `xxx_create_roles_table.php` u `up()` metodu gdje je definirana `roles` tablica:

```
<?php

namespace App\Models;

use Illuminate\Database\Eloquent\Factories\HasFactory;
use Illuminate\Database\Eloquent\Model;

class Roles extends Model
{
    use HasFactory;

    /**
     * The attributes that are mass assignable.
     *
     * @var array<int, string>
     */
    protected $fillable = [
        'name',
        'permissions',
    ];
}
```

U `Post.php` ubacit ćemo također `$fillable` tako da pogledamo `xxx_create_posts_table.php` u `up()` metodu gdje je definirana `posts` tablica. Ovdje je pitanje `user_id` kako ćemo ga zapisati u metodi `Post.php`.

Kada se kreira `posts`, pitanje je kako ćemo ga kreirati. Više je načina na koji je to moguće napraviti. Ako pogledamo User će imati relaciju prema Post-u, zato što User može imati više postova. Shodno tome ta relacija može poslužiti da preko user modela kreirate post. U tom slučaju nismo dužni slati `user_id` jer je logično ako preko user modela kreiramo post onda znamo i tko je user. Ako to ne radimo, nego želimo poslati `user_id`, to bi bio case tj. zapis koji se treba dogoditi u bazu, u ovom slučaju će to biti post. Ne želimo ga vezati za user-a koji je taj zapis kreirao. Ako želimo onda moramo kreirati `user_id`. Poslovna je logika da li će `user_id` biti štićen podatak ili ne.

Jedan način je da definiramo `title`, `content`, `slug`. Pitanje je i kako se `slug` definira. Da li je to nešto što će se poslati kroz request ili ne. Dakle želimo da se slug automatski kreira. Imat ćemo još i `content`.

```
<?php

namespace App\Models;

use Illuminate\Database\Eloquent\Factories\HasFactory;
use Illuminate\Database\Eloquent\Model;
```

```

class Post extends Model
{
    use HasFactory;

    /**
     * The attributes that are mass assignable.
     *
     * @var array<int, string>
     */
    protected $fillable = [
        'title',
        'content',
        'image',
    ];
}

```

Ispravak postojećih tablica bez rollback-a

Želimo dodatno definirati na `xxx_create_posts_table.php` metodu `softdelete()` i zato možemo napraviti rollback. Drugi način je da kreiramo novu migraciju. Osim post, modeli također nemaju uključenu opciju `softDeletes()` a moramo je naknadno upaliti. Ako pogledamo `xxx_create_roles_table.php` vidjet ćemo `softDeletes()` za `Roles.php`. Ja sam zaboravio `softDeletes()` i za `users` tablicu.

Međutim, predavač je htio prikazati situaciju kada ne možemo napraviti rollback (imamo neke podatke npr.) i napraviti promjenu sa pojedinom migracijom. Dakle, kreirat ćemo novu migraciju:

```
php artisan make:migration change_posts_table --table=posts
```

```
$ php artisan make:migration change_posts_table --table=posts
[INFO] Migration [I:\Laravel\AlgebraBlog\database\migrations\2024_09_17_144817_change_posts_table.php] created successfully.
```

Ovo će kreirati novu migraciju. Pripremiti će migraciju ne za kreiranje nove tablice nego za update postojeće tablice.

Vidjet ćemo sada razliku između ove migracije `xxx_change_yyy_table.php` i prethodnih `xxx_create_yyy_table.php`:

```

<?php

use Illuminate\Database\Migrations\Migration;
use Illuminate\Database\Schema\Blueprint;
use Illuminate\Support\Facades\Schema;

return new class extends Migration
{
    ...
}

```

```
{
    /**
     * Run the migrations.
     */
    public function up(): void
    {
        Schema::table('posts', function (Blueprint $table) {
            //
        });
    }

    /**
     * Reverse the migrations.
     */
    public function down(): void
    {
        Schema::table('posts', function (Blueprint $table) {
            //
        });
    }
};
```

Sada možemo raditi update tablice. Možemo promijeniti neki stupac, ime ili format, indeks, itd.

Nakon ubacivanja `softDeletes()` u `up()` metodu i `dropSoftDeletes()` u `down()` datoteka `xxx_change_posts_table.php` izgleda ovako:

```
<?php

use Illuminate\Database\Migrations\Migration;
use Illuminate\Database\Schema\Blueprint;
use Illuminate\Support\Facades\Schema;

return new class extends Migration
{
    /**
     * Run the migrations.
     */
    public function up(): void
    {
        Schema::table('posts', function (Blueprint $table) {
            $table->softDeletes();
        });
    }
};
```

```

/**
 * Reverse the migrations.
 */
public function down(): void
{
    Schema::table('posts', function (Blueprint $table) {
        $table->dropSoftDeletes();
    });
}

```

```
php artisan make:migration change_users_table --table=users
```

```
$ php artisan make:migration change_users_table --table=users
[INFO] Migration [I:\Laravel\AlgebraBlog\database\migrations\2024_09_17_150235_change_users_table.php] created successfully.
```

Iste promjene ćemo napraviti i `users` tablici, dodati u `up()` metodu `softDeletes()`, a u `down()` dodati `dropSoftDeletes()`.

Nakon toga `php artisan migrate`:

```
$ php artisan migrate
[INFO] Running migrations.

2024_09_17_144817_change_posts_table ..... 25.72ms DONE
2024_09_17_150235_change_users_table ..... 5.71ms DONE
```

Vidimo da je Artisan otkrio obje promjene i implementirao ih. Na `posts` vidimo `deleted_at`:

| <code>id</code> | <code>title</code> | <code>content</code> | <code>slug</code> | <code>image</code> | <code>user_id</code> | <code>created_at</code> | <code>updated_at</code> | <code>deleted_at</code> |
|-----------------|--------------------|----------------------|-------------------|--------------------|----------------------|-------------------------|-------------------------|-------------------------|
| NULL | NULL | NULL | NULL | NULL | NULL | NULL | NULL | NULL |

Na `users` vidimo također `deleted_at` bez gubljenja podataka:

| <code>id</code> | <code>first_name</code> | <code>last_name</code> | <code>email</code> | <code>email_verified_at</code> | <code>password</code> | <code>remember_token</code> | <code>created_at</code> | <code>updated_at</code> | <code>deleted_at</code> |
|-----------------|-------------------------|------------------------|--------------------|--------------------------------|-----------------------|-----------------------------|-------------------------|-------------------------|-------------------------|
| 1 | Pero | Perić | pero@pero.com | NULL | 123456 | NULL | NULL | NULL | NULL |

Ako napravimo rollback tih stupaca nema:

```
php artisan migrate:rollback
```

```
$ php artisan migrate:rollback
INFO Rolling back migrations.

2024_09_17_150235_change_users_table ..... 18.04ms DONE
2024_09_17_144817_change_posts_table ..... 13.32ms DONE
```

Vraćamo opet sa:

```
php artisan migrate
```

Zanimljivo je pogledati tablicu `migrations` i vrijednost polja `batch`:

| id | migration | batch |
|----|--------------------------------------|-------|
| 1 | 0001_01_000000_create_users_table | 1 |
| 2 | 2024_09_16_093138_create_roles_table | 2 |
| 3 | 2024_09_16_110536_create_posts_table | 3 |
| 6 | 2024_09_17_144817_change_posts_table | 4 |
| 7 | 2024_09_17_150235_change_users_table | 4 |

To je način kako nešto dodati bez da utječemo na podatke u tablicama. Kada dodajmo novi stupac a u bazi postoje zapisi mogu se pojaviti problemi oko default vrijednosti.

Ako probamo staviti `$table->string('phone');` zapisat će u bazu prazni string. To će napraviti bazu. Ako probamo staviti `$table->int('phone');` zapisat će kao default vrijednost `0`. To je sve u redu. Međutim, ako probamo zapisati datum s `$table->date('birth_date');` dojavit će grešku. U tom slučaju moramo pogledati u `migrations` i vidjeti da li je zapisao i da li možemo napraviti rollback. Moguće je kao u ovom slučaju da se nije zapisao niti zabilježio. Ako napravimo rollback poništiti ćemo krivu promjenu. Može se dogoditi da je migracija imala više elemenata koje je trebala napraviti npr:

```
$table->date('phone');
$table->date('birth_date');
```

Pokrenemo migraciju i dojavi grešku. Dakle puknuo je i nije napravio zapis u `migrations`. Međutim u `users` imamo `phone`. Dakle sve do greške je odrađeno. Dakle u ovom slučaju dodat će polje `phone` u `users`. Puknuo je na migraciji ali nije napravio automatski rollback. Rollback nije moguće ručno napraviti jer nema zapisanog drugog reda. Puknuo je prilikom kreiranja. Problem možemo riješiti tako da ispravimo red tako da bude

```
$table->date('birth_date')->nullable();
```

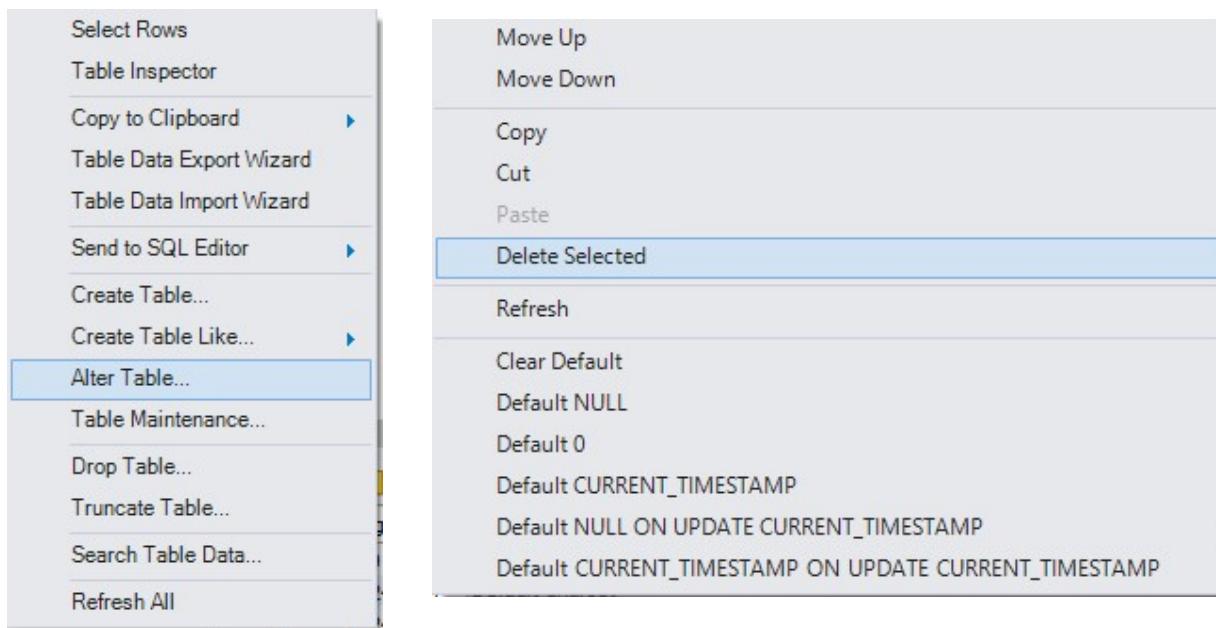
ili možemo postaviti neki specifičan datum:

```
$table->date('birth_date')->default('2024-09-17');
```

U ovom slučaju bolja je prva varijanta.

Ako sada pokrenemo migraciju dojavit će opet grešku. Column already exist iako prethodna migracija nije prošla. Ako napravimo rollback, dogodit će se da će zadnji batch vratiti nazad a to ne želimo. Nije jasno zašto ovo nije još uvijek implementirano kako treba. Dogodio se exception, ako je pokrenuta

transakcija i nije uspjela trebao se napraviti rollback a nije. U ovom slučaju je rješenje ili ručno obrisati stupac `phone` u `users` tablici sa `ALTER TABLE users DROP COLUMN phone;` ili



Drugi način što možemo probati je uzeti ime migracije, u ovom slučaju `xxx_change_users_table` i ručno ubaciti u `migrations` tablicu uz novi `batch` uvećan za 1. Nakon Apply popunit će se `id`. Možemo primjetiti da se pojavila rupa u id. To je zato što je on digao `id` ali ga je odbacio kada je pukao. Ako sada probamo pokrenuti rollback s php artisan migrate:rollback opet ćemo vidjeti grešku jer ne može dropati `birth_date` jer ne postoji. Možemo zakomentirati liniju koja pravi problem u `down()` metodi i to je rješenje. Na kraju uklonimo tu dodanu zadnju migraciju iz tablice `migrations`.

Napraviti ćemo još jednu ispravku tablice users, dodat ćemo polje `phone`:

```
php artisan make:migration change_users_table --table=users
```

```
<?php

use Illuminate\Database\Migrations\Migration;
use Illuminate\Database\Schema\Blueprint;
use Illuminate\Support\Facades\Schema;

return new class extends Migration
{
    /**
     * Run the migrations.
     */
    public function up(): void
    {
        Schema::table('users', function (Blueprint $table) {
            $table->string('phone')->nullable();
        });
    }

    /**
     * Reverse the migrations.
     */
    public function down(): void
    {
        Schema::table('users', function (Blueprint $table) {
            $table->string('phone');
        });
    }
}
```

```

    });

}

/**
 * Reverse the migrations.
 */
public function down(): void
{
    Schema::table('users', function (Blueprint $table) {
        $table->dropColumn('phone');
    });
}
;

```

Gledamo da novom stupcu damo default vrijednost.

Pokrenuli smo migraciju sa php artisan migrate i dobili tu kolonu `phone` u `users` tablici.

Kako smo dodali phone u `users` tablici, taj phone trebamo dodati i u model `Users.php`.u `$fillable`:

```

/**
 * The attributes that are mass assignable.
 *
 * @var array<int, string>
 */
protected $fillable = [
    'first_name',
    'last_name',
    'email',
    'password',
    'phone',
];

```

Relacije

Kada smo definirali modele, želimo definirati relacije između tih modela. To nam je bitno kako bi došli do svih potrebnih podataka. Npr. da dohvativamo sve postove koje je neki user objavio. Mogli bi smo saznati neki `user_id` i preko modela `Post.php` dohvatiti sve zapise gdje je stupac `user_id` taj određeni. Kada već imate user-a, želite za njega vezati i sve njegove postove. Imamo i role. Dakle dohvatili ste nekog user-a, tražit ćemo odmah i Role.php. To možemo postići uz pomoć relacija na modelima.

U dokumentaciji ćemo vidjeti kakve sve relacije postoje:

Jedan na jedan (engl. One to One)

Jedan prema više (engl. One to Many)

Jedan prema više (obrnuti postupak) / pripada (engl. One to Many (Inverse) / Belongs To)

Ima jednog od mnogih (engl. Has One of Many)

Ima jedan prolaz (engl. Has One Through)

Prolazi mnogo puta (engl. Has Many Through)

Idemo prvo pogledati odnose između user-a i role. Rekli smo da ćemo omogućiti da jedan user ima više različitih uloga. I jedna uloga može pripadati više različitih user-a.

Pogledajmo sada odnos između usera i post-a, user može imati više postova a jedan post pripada nekom user-u. Obrnuto bi bilo da jedan post pripada više usera, što nije kod nas slučaj. Za sada koristit ćemo Jedan na jedan, Jedan prema više i više prema više (engl. Many to Many).

Na modelu User.php ćemo definirati One to Many relaciju.

Rekli smo da ćemo omogućiti da jedan user ima više različitih uloga. Jedna uloga može pripadati više različitih usera tj. više različitih user-a može imati istu ulogu. Tu je definirana relacija many to many. To možemo vidjeti iz same relacije kada smo definirali tablicu `xxx_create_roles_table.php` onda smo kreirali tablicu `user_roles`:

```
Schema::create('user_roles', function (Blueprint $table) {
    $table->foreignId('role_id')->constrained()->cascadeOnDelete();
    $table->foreignId('user_id')->constrained()->cascadeOnDelete();
    $table->primary(['role_id', 'user_id']);
    $table->timestamps();
})
```

Idemo pogledati između user-a i post-a u `Users.php`. User ima više post-ova a jedan post pripada nekom user-u. Dakle Jedan prema više (engl. One to Many). Na `User.php` ćemo definirati funkciju `post()`:

```
/**
 * Get the user posts
 *
 * @return \Illuminate\Database\Eloquent\Relations\HasMany
 */
public function posts(): HasMany
{
    return $this->hasMany(Post::class);
}
```

Jako je bitno da je iznad uključen iz `use Illuminate\Database\Eloquent\Relations\HasMany;` Eloquent-a. To je `return` tip.

Kada dohvate user-a, možemo doći do njegovih postova jednostavno pozivajući se na relaciju `$this->hasMany(Post::class);`

Ono što Laravel napravi ispod haube pomoću magičnih metoda (jedna od njih se zove `__call`) koja automatski kada pozovete property. Sada ako pogledamo `web.php` gdje user model ima ovo svojstvo post:

```
Route::get('/', function() {
    dd(User::find(1)->posts());
});
```

Ako sada pogledamo ovo svojstvo `posts()`, User model nema nigdje ovo svojstvo `posts()`, ali će magic metoda `__call` će ispod haube aktivirati ovu metodu `posts()` i vratitiće relaciju i sve podatke dobivene kroz relaciju.

Ako direktno pozovemo metodu `posts` sa `dd(User::find(1)->posts());` iz `web.php` dobijemo relaciju ali ne i njezine podatke.

```
Illuminate\Database\Eloquent\Relations\HasMany {#940 ▼ // routes\web.php:9
  #query: Illuminate\Database\Eloquent\Builder {#237 ▶}
  #parent: App\Models\User {#990 ▶}
  #related: App\Models\Post {#229 ▶}
  #eagerKeysWereEmpty: false
  #foreignKey: "posts.user_id"
  #localKey: "id"
}
```

Da bi smo dobili te podatke trebamo pozvati dodatno metodu `get()` i tada red glasi `dd(User::find(1)->posts()->get());`. Dobijemo kolekciju koja je trenutno prazna (jer user s `id=1` nema ni jedan post).

```
Illuminate\Database\Eloquent\Collection {#989 ▼ // routes\web.php:9
  #items: []
  #escapeWhenCastingToString: false
}
```

Moguće je pozvati na `User` modelu metodu `posts()` koja vrati relaciju `hasMany` prema modelu `$related` i stranih ključeva `$foreignKey` i `$localKey` koji su po default-u `null`.

```

    'email_ver'
    'password'
];
}

/**
 * Get the user posts
 *
 * @return \Illuminate\Database\Eloquent\Relations\HasMany
 */
Codeumen: Refactor
public function posts()
{
    return $this->hasMany(\App\Models\Post::class);
}

```

Illuminate\Database\Eloquent\Concerns\HasRelationships::hasMany
Define a one-to-many relationship.

<?php
public function hasMany(\$related, \$foreignKey = null, \$localKey = null) { }
@param class-string<TRelatedModel> \$related
@param string|null \$foreignKey
@param string|null \$localKey
@return \Illuminate\Database\Eloquent\Relations\HasMany<TRelatedModel, \$this>
@template TRelatedModel of \Illuminate\Database\Eloquent\Model

return \$this->hasMany(\App\Models\Post::class);

Laravel zna gdje je relacija između tih stranih ključeva (`$foreignKey` i `$localKey`) po naming konvenciji. U dokumentaciji piše da će Eloquent automatski odrediti odgovarajući stupac stranog ključa za `Post` model (u slučaju njihovog primjera je to za `Comment` model). Prema konvenciji, Eloquent će uzeti naziv "snake case"³ nadređenog modela i dodati mu sufiks `_id`. Dakle, u ovom primjeru, Eloquent će pretpostaviti da je strani ključ stupac u `Post` modelu i to `user_id` (u slučaju primjera u dokumentaciji `Comment` modela pošto smo u Postu očekuje da ima `post_id`).

Drugim riječima, naš `User.php` model očekuje da na `Post.php` modelu postoji `user_id`. Ako pogledamo migraciju `xxx_create_post_table.php`, upravo tako se zove.

Ako se pridržavamo naming konvencije, ne moramo definirati stvari ručno ali možemo. Ako se naš origin key ne zove `user_id` i ako se lokalni ključ ne zove `id`, moguće je unijeti koji su to na sljedeći način:

```

public function posts(): HasMany
{
    return $this->hasMany(Post::class, 'user_id', 'id');
}

```

Kako se držimo konvencije, dovoljno je kako je bilo za automatsko povezivanje:

```

public function posts(): HasMany
{
    return $this->hasMany(Post::class);
}

```

³ Snake case (ponekad stilizirano autologijski kao snake_case) je konvencija imenovanja u kojoj se svaki razmak zamjenjuje znakom za podvlačenje (`_`), a riječi se pišu malim slovima

Vratimo se na dohvaćanje podataka u `web.php`. Ako se oslonimo na metodu `posts()` u `User.php`, ona vrati relaciju, a na toj relaciji postoji metoda `get()` (u `web.php`) koja kaže dohvatiti podatke putem relacije. Ako bi red u `web.php` prepravili da umjesto `dd(User::find(1)->posts()->get())` glasi: `dd(User::find(1)->posts()->toSql());` i tako pretvorili upit u SQL, dobit ćemo što to on šalje na bazu.

```
"select * from `posts` where `posts`.`user_id` = ? and `posts`.`user_id` is not null" // routes\web.php:9
```

To je upit koji se šalje na `post` tablicu. Preko `User` smo pozvali relaciju Post u modelu koji preko `hasMany` u principu kreira SQL upit prema `Post` tablici jer smo rekli da je to relacija preko modela `Post`. Sve stvari u `get()` Laravel napravi pomoću magic metode i dovoljno je pozvati nad modelom `User`. Svojstvo `posts` dolazi do metode `posts()`.

Praktično, magic metod `__call` u Laravelu kaže (prvenstveno u modelima) preko poziva svojstava dinamički rukuje metodom poziva u modelu i može pozvati metodu u nekom modelu. To znači ako pozovemo svojstvo `post`, Laravel ispod haube opet pozove `posts()` metodu i još napravi `get()`.

U `web.php` napraviti ćemo:

```
Route::get('/', function() {
    $user = User::find(1);
    dd($user->posts());
});
```

S ovime `posts` aktiviramo magic metod `__call` pozivamo metodu `posts`. Ovo `dd($user->posts())`; je ekvivalent `dd($user->posts()->get())`;

Nedostatak svih magic metoda u PHP-u je što podcrtava metodu i napiše „`Possible magic method call`“. Dakle može biti i ne mora. Često taj nedostatak konzistentnosti u kodu može biti nespretna. Zato neki programeri izbjegavaju magic metode.

Predavač kaže da je ljepši kod ako u modelu `User.php` napravimo:

```
public function getPosts()
{
    return $this->posts();
}
```

a iz `web.php` pozivamo sa:

```
Route::get('/', function() {
    $user = User::find(1);
    dd($user->getPosts());
});
```

Kada kasnije čitamo kod jasno je što se dešava. To je dobra praksa. Izbjegnemo magic metoda a sintaksa je jasna.

Dakle mogući oblici su:

```
Route::get('/', function() {
    $user = User::find(1);
    dd(User::find(1)->posts); // svi u početku koriste ovo
    dd(User::find(1)->posts()->get()); // a kad vide da nije pregledno koriste ovo
    dd(User::find(1)->getPosts()); // ili ovo
});
```

U `User.php` možemo taj `getPosts()` napraviti kao:

```
/**
 * Get the user posts
 *
 * @return \Illuminate\Database\Eloquent\Relations\HasMany
 */
public function posts(): HasMany
{
    return $this->hasMany(\App\Models\Post::class);
}

public function getPosts()
{
    return $this->posts()->get();
}
```

i onda ništa neće biti podcrtnato kao moguća greška. Imamo relaciju i nakon nje metoda koja je nad modelom. Kada pozovemo `getPosts()` to je to.

Sada ćemo vidjeti kako Laravel radi sa relacijama. Postoje dvije metode. To su **Lazy load** i **Eager load**.

Ako pogledamo pristup koji imamo trenutno u `web.php`, to je Lazy load.

```
Route::get('/', function() {
    $user = User::find(1);
    dd(User::find(1)->getPosts());
});
```

```
App\Models\User {#990 ▼ // routes\web.php:10
  #connection: "mysql"
  #table: "users"
  #primaryKey: "id"
  #keyType: "int"
  +incrementing: true
  #with: []
  #withCount: []
  +preventsLazyLoading: false
  #perPage: 15
  +exists: true
  +wasRecentlyCreated: false
  #escapeWhenCastingToString: false
  #attributes: array:11 [▶]
  #original: array:11 [▶]
  #changes: []
  #casts: array:2 [▶]
  #classCastCache: []
  #attributeCastCache: []
  #dateFormat: null
  #appends: []
  #dispatchesEvents: []
  #observables: []
  #relations: []
  #touches: []
  +timestamps: true
  +usesUniqueIds: false
  #hidden: array:2 [▶]
  #visible: []
  #fillable: array:5 [▶]
  #guarded: array:1 [▶]
  #authPasswordName: "password"
  #rememberTokenName: "remember_token"
}
```

```
Illuminate\Database\Eloquent\Collection {#989 ▼ // routes\web.php:11
  #items: []
  #escapeWhenCastingToString: false
}
```

Lazy load će prvo dohvatiti podatke o user-u, i nije svjestan relacije prema postovima tj. nema podataka iz te relacije, nije ih dohvatio.

```
Route::get('/', function() {
    $users = User::find(1);
    dump($users)
    dd(user->getPosts());
});
```

U principu nema podataka o relaciji zato što nisu odmah zatraženi nego naknadno. To se zove Lazy load. Možda izgleda odlično jer nema dodatnog opterećenja na početku, pa će to potrajati i u nekim slučajevima je to u redu.

Zamislimo slučaj da želimo sve user-e. I dalje smo u [web.php](#):

```
Route::get('/', function() {
    $users = User::all();
    dump($users);
```

```
});
```

Eloquent ORM ovdje vraća par stvari. Jedno je model (to smo vidjeli malo prije kada smo tražili pojedini podatak o nekom resursu). Ako tražimo sve user-e, dobit ćemo kolekciju tih modela.

```
Illuminate\Database\Eloquent\Collection {#990 ▼ // routes\web.php:10
    #items: array:1 [▼
        0 => App\Mode...\\User {#581 ▶}
    ]
    #escapeWhenCastingToString: false
}
```

Evo detaljnije:

```
Illuminate\Database\Eloquent\Collection {#990 ▼ // routes\web.php:10
    #items: array:1 [▼
        0 => App\Mode...\\User {#581 ▼
            #connection: "mysql"
            #table: "users"
            #primaryKey: "id"
            #keyType: "int"
            +incrementing: true
            #with: []
            #withCount: []
            +preventsLazyLoading: false
            #perPage: 15
            +exists: true
            +wasRecentlyCreated: false
            #escapeWhenCastingToString: false
            #attributes: array:11 [▶]
            #original: array:11 [▶]
            #changes: []
            #casts: array:2 [▶]
            #classCastCache: []
            #attributeCastCache: []
            #dateFormat: null
            #appends: []
            #dispatchesEvents: []
            #observables: []
            #relations: []
            #touches: []
            +timestamps: true
            +usesUniqueIds: false
            #hidden: array:2 [▶]
            #visible: []
            #fillable: array:5 [▶]
            #guarded: array:1 [▶]
            #authPasswordName: "password"
            #rememberTokenName: "remember_token"
        }
    ]
    #escapeWhenCastingToString: false
}
```

Ako sada pogledamo vidjet ćemo da matrica ima jedan item u kojoj se nalazi taj model. Kolekcija je ustvari proširena matrica sa zaista puno metoda. Sada imamo kolekciju usera. Zamislimo da imamo kolekciju od 100 user-a i trebamo na frontend-u ispisati podatke. Međutim nama trebaju i podaci o postovima za svakog pojedinog user-a. Želimo ispisati ime i prezime usera i broj postova koje je objavio. Dakle trebamo proći foreach petljom.

```
<?php

use App\Models\User;
use App\Models\Post;
use Illuminate\Support\Facades\Route;

Route::get('/', function () {
    //return view('welcome');
    $users = User::all();
    foreach ($users as $user) {
        echo "<h1>{$user->first_name} {$user->last_name}</h1>";
        echo "<p><p>Num of Posts: {$user->getPosts()->count()}</p>" ;
    }
    //dd($users->getPosts());
});
```

Pero Perić

Num of Posts: 0

Dohvati sve user-e, prođe foreach petljom po toj kolekciji. Dakle foreach dohvati svakog pojedinog usera kao model. Za svakog dohvativamo `first_name` i `last_name`. Preko relacije Laravel ode u bazu, pokupi sve njegove postove u kolekciju i onda na toj kolekciji pozovi `count()` metodu koja prebroji stavke u toj kolekciji. Kolekcije inače u sebi imaju [hrpu metoda koje možemo vidjeti u dokumentaciji](#).

Ovdje je problem jer se dešava nepotrebno otvaranje SQL upita. Ako imamo 100 korisnika dogodit će se 101 SQL upita na bazu. 1 dohvati sve usere i onda za svakog od ovih user-a ode po post-ove. To je nedostatak lazy load-a. U ovoj situaciji ovo nije pametna ideja.

Ako ovaj red echo `"<p><p>Num of Posts: {$user->getPosts()->count()}</p>"` pretvorimo u SQL upit da vidimo što radi echo `"<p><p>Num of Posts: {$user->posts()->toSql()}</p>"`; `web.php` će nam izgledati ovako:

```
<?php

use App\Models\User;
use App\Models\Post;
use Illuminate\Support\Facades\Route;

Route::get('/', function () {
    //return view('welcome');
    $users = User::all();
    foreach ($users as $user) {
```

```

        echo "<h1>{$user->first_name} {$user->last_name}</h1>";
        echo "<p><p>Num of Posts: {$user->posts()->toSql()}</p>" ;
    }
    dd($user->getPosts());
);

```

```
select * from 'posts' where 'posts'.user_id = ? and 'posts'.user_id is not null
```

Pero Perić

Num of Posts: select * from 'posts' where 'posts'.user_id = ? and 'posts'.user_id is not null

```

Illuminate\Database\Eloquent\Collection {#991 ▾ // routes\web.php:14
  #items: []
  #escapeWhenCastingToString: false
}

```

Taj SQL ide za jednog user-a. Za 100 korisnika sto takvih SQL upita. U takvima situacijama koristimo Eager load. Eager load umjesto 101 SQL upita napravi samo 2 SQL upita. Jedan dohvati sve user-e i napravi join sa postom i dobiti ćete automatski sve potrebne informacije.

Eager load se radi na više načina. Jedan od njih je da svaki puta kada hvatate podatke o user-u, želite dobiti informacije o njegovim postovima. To znači da možemo koristiti u User.php modelu koristiti `protected $with` iznad funkcije.

```
protected $with = ['posts'];
```

Laravel u modelu ima svojstvo koje se zove `$with` a 'posts' je ime relacije.

To znači da sada svaki puta kada idete po user-a, Laravel će automatski napraviti upit koji će napraviti `join` prema post tablici i odmah će dohvatiti sve postove. To je Eager load. Ako pogledamo u [uputstvu Laravela Eloquent ORM->Relationships->Eager Loading](#), vidjet ćemo detalje. Tamo piše da je kod upita dovoljno dodati `with` ili na modelu (u njihovom primjeru `$books = Book::with('author')->get();`), kako smo mi krenuli raditi. Između ta dva načina postoji razlika.

Kada jednom na modelu definiramo svojstvo `$with`, zacementirali smo Eager load. To znači da svaki puta hvatamo user-a i dobijamo postove. I to je u nekim slučajevima u redu.

Čest slučaj je da nam ovo ne odgovara i onda ovo nije dobra praksa. Ako nam ta relacija ne treba uvijek onda to ne koristimo.

Moguće je da želimo kada neko dohvati post da odmah želimo relaciju prema user-u. Ako i to stavimo pod `protected with`, user pozove post, post pozove user i stvori se beskonačna petlja. Dakle treba biti pažljiv s korištenjem s `$with` kao Eager load direktno na model.

Pogledajmo demonstraciju:

```

Route::get('/', function() {
    dd(User::find(1));
    $user =User::all;
});

```

```

        foreach ($users as $user) {
            echo "<h1>{$user->first_name}{$user->last_name}</h1>" ;
            echo "<p><p>Num of Posts: {$user->getPosts()->count()}</p>" ;
        }
        dd($user->getPosts());
    });
}

```

Ako sada ovo pokrenemo vidjet ćemo model koji će na sebi imati relaciju. Prvi puta relacija je bila prazna matrica. Ovaj puta relacija je posts i unutra je kolekcija:

```

App\Models\User {#1209 ▼ // routes\web.php:8
  #connection: "mysql"
  #table: "users"
  #primaryKey: "id"
  #keyType: "int"
  +incrementing: true
  #with: array:1 [▶]
  #withCount: []
  +preventsLazyLoading: false
  #perPage: 15
  +exists: true
  +wasRecentlyCreated: false
  #escapeWhenCastingToString: false
  #attributes: array:11 [▶]
  #original: array:11 [▶]
  #changes: []
  #casts: array:2 [▶]
  #classCastCache: []
  #attributeCastCache: []
  #dateFormat: null
  #appends: []
  #dispatchesEvents: []
  #observables: []
  #relations: array:1 [▼
    "posts" => Illuminate\Database\Eloquent\Collection {#1208 ▼
      #items: []
      #escapeWhenCastingToString: false
    }
  ]
  #touches: []
  +timestamps: true
  +usesUniqueIds: false
  #hidden: array:2 [▶]
  #visible: []
  #fillable: array:5 [▶]
  #guarded: array:1 [▶]
  #authPasswordName: "password"
  #rememberTokenName: "remember_token"
}

```

Eager load je automatski napravio upit na post i preko post relacije stvorio je join i povukao kolekciju koju možemo reći posts.

Ako sada prepravimo:

```

Route::get('/', function() {
    dd(User::find(1)->posts);
}

```

```
$user = User::all();
foreach ($users as $user) {
    echo "<h1>{$user->first_name}{$user->last_name}</h1>" ;
    echo "<p><p>Num of Posts: {$user->getPosts()->count()}</p>" ;
}
dd($user->getPosts());
});
```

Vidimo sada razliku između `posts` i `$user->getPosts()`.

Zamislimo sljedeći senarij u `web.php`:

```
Route::get('/', function () {
    //return view('welcome');
    $u1 = User::find(1);
    $posts = $u1->posts;
    dd($posts);
});
```

```
Illuminate\Database\Eloquent\Collection {#1208 ▼ // routes\web.php:11
    #items: []
    #escapeWhenCastingToString: false
}
```

Gledamo redove do praznog reda u `web.php`, zanemarimo ostatak. Ako sada odemo u `User.php` i maknemo `protected $with = ['posts'];`; tj. stavimo ispred komentar `//` i pogledamo opet `web.php` vidimo kolekciju (prazna je jer user nema postova). Ako skinemo komentar, opet radi, identično je.

Kada nema `protected $with = ['posts'];` na `User` modelu (u `Users.php`) smo pozvali svojstvo `posts`. Lavarel kaže nemamo to svojstvo `posts`, ali pošalje to svojstvo onoj magičnoj metodi `__call` koja na temelju tog imena potraži metodu u našem modelu. Ako postoji ta metoda onda je pozove i peko nje dohvati podatke, doslovno napravi upit na bazu i dohvati podatke i vrati ih nazad.

Ako je uključen `protected $with = ['posts'];` odnosno Eager load opet proba dohvati na user modelu `posts` atribut. On sada postoji. Relacija se automatski napunila i kreirala atribut `posts` i uhvatila podatke.

U prvom slučaju imamo dodatni upit na bazu a u drugom slučaju ne. U jednom i u drugom slučaju je pozvano svojstvo `posts` i to je problem koji zbujuje. Kada koristimo Eager load onda smo sigurni da imamo property, ne moramo zvati metodu. Ako imamo Lazy load onda nećemo zvati `protect` nego svoju metodu `getPosts()` ili metodu `post()->get()`. I to je razlika između Eager load i Lazy load.

Drugi način je kada želite povremeno upaliti Eager load (nije konstantno upaljen). U `web.php` nad modelom `User` pozvat ćemo metodu `with` i navesti ime relacije `posts` i dohvatiti podatak:

```
Route::get('/', function() {
    $u1 = User::with('posts')->find(1);
    $posts = $u1->posts;
    dd($posts);
});
```

Na `User` modelu je ugašen Eager load `// protected $with = ['posts'];` ali ga s `with` palimo po potrebi.

```
Illuminate\Database\Eloquent\Collection {#1207 ▼ // routes\web.php:11
    #items: []
    #escapeWhenCastingToString: false
}
```

Ponovo vidimo kolekciju. To je najčešće na početku da se napune relacije koje želimo.

Ako pogledamo model Users, vidjet ćemo da je `posts` na modelu u relaciji.

```
<?php

use App\Models\User;
use App\Models\Post;
use Illuminate\Support\Facades\Route;

Route::get('/', function () {
    $u1 = User::find(1);
    //$posts = $u1->posts;
    dd($u1);
});
```

```
App\Models\User {#990 ▼ // routes\web.php:11
  #connection: "mysql"
  #table: "users"
  #primaryKey: "id"
  #keyType: "int"
  +incrementing: true
  #with: []
  #withCount: []
  +preventsLazyLoading: false
  #perPage: 15
  +exists: true
  +wasRecentlyCreated: false
  #escapeWhenCastingToString: false
  #attributes: array:11 [▶]
  #original: array:11 [▶]
  #changes: []
  #casts: array:2 [▶]
  #classCastCache: []
  #attributeCastCache: []
  #dateFormat: null
  #appends: []
  #dispatchesEvents: []
  #observables: []
  #relations: []
  #touches: []
  +timestamps: true
  +usesUniqueIds: false
  #hidden: array:2 [▶]
  #visible: []
  #fillable: array:5 [▶]
  #guarded: array:1 [▶]
  #authPasswordName: "password"
  #rememberTokenName: "remember_token"
}
```

Vidimo da je `posts` kolekcija prazna jer smo isključili `$posts = $u1->posts;` da ne bi popunio ga sa Lazy Load. Išli smo s pristupom ne trebaju nam njegovi postovi, ako nam ipak trebaju:

```
$u1 = User::with('posts')->find(1);
```

Lazy load je odličan kada imamo jedan podatak koji smo dohvatili i treba nam njegova relacija. Kada imamo kolekciju podataka, treba koristiti Eager load. Eager load ne treba paliti bez potrebe (zbog `JOIN`), jer nepotrebno troši resurse.

Ako su potrebne relacije i podaci iz relacija upaliti Eager load.

Ostaje nam na `Post.php` definirati relaciju. Iz perspektive posta prema user-u, dobijemo user-a koji je vlasnik posta i to je `BelongsTo`.

Ako u dokumentaciji pogledamo relaciju [One to Many](#) i [HasMany](#). Postoji i One to Many (Inverse) gdje imamo obrnutu priču i `BelongsTo` (naš slučaj). Imena su intuitivna.

```
/** 
 * Get the user that owns the Post
 *
 * @return \Illuminate\Database\Eloquent\Relations\BelongsTo
```

```
 */
public function user()
{
    return $this->belongsTo(User::class);
}
```

Predavač voli kreirati metodu:

```
public function getUser(): string
{
    return $this->user()->get()->first();
```

Eloquent dohvati s `$this user()` relaciju, s `get()` sve podatke kao kolekciju i iz te kolekcije s `first()` prvu stavku (engl. item). Pošto znamo da će jedan post biti vezan za jednog usera u kolekciji ćemo imati 0 ili 1.

Složili smo user-post. Ostao nam je user-roll gdje imamo [Many to Many relaciju](#). To znači da se uvodi pivot tablica zato što kada pogledamo jedna user može imati više različitih rola a jedna rola može pripadati više različitih usera. Više različitih usera može imati istu rolu. Tu imamo `BelongsToMany`. Obrnuto je isto tako `BelongsToMany`.

Za definiranje ovog odnosa potrebne su tri tablice baze podataka: `users`, `roles` i `role_user`. Tablica `role_user` je izvedena iz abecednog reda povezanih naziva modela i sadrži `user_id` i `role_id` stupce. Ova se tablica koristi kao međutablica koja povezuje korisnike i uloge.

Zapamtite, budući da rola može pripadati mnogim korisnicima, ne možemo jednostavno postaviti `user_id` stupac na `roles` tablicu. To bi značilo da rola može pripadati samo jednom korisniku. Tablica `role_user` je potrebna kako bi se pružila podrška za dodjelu rola većem broju korisnika . Strukturu tablice odnosa možemo ovako sažeti:

users

```
id - integer
name - string
```

roles

```
id - integer
name - string
```

role_user

```
user_id - integer
role_id - integer
```

Na `User.php` modelu napisat ćemo:

```
public function roles()
{
    return $this->belongsToMany(Role::class);
}
```

Ako pogledamo pivot tablicu koju smo kreirali `xxx_create_roles_table.php` je `user_roles` a on očekuje singular.

Illuminate\Database\Eloquent\Concerns\HasRelationships::belongsToMany

Define a many-to-many relationship.

```
<?php
public function belongsToMany(
    $related,
    $table = null,
    $foreignPivotKey = null,
    $relatedPivotKey = null,
    $parentKey = null,
    $relatedKey = null,
    $relation = null
) { }
```

`user_roles` nije po konvenciji Larabela, trebalo je biti `roles_user`. Navest ćemo ovako:

```
/**
 * Get the user roles.
 *
 * @return BelongsToMany
 */
public function roles(): BelongsToMany
{
    return $this->belongsToMany(Roles::class, 'user_roles', 'user_id',
'role_id');
}
```

Želite li sve vaše role nad user-om možemo reći:

```
public function getRoles()
{
    return $this->roles()->get();
}
```

Ovo služi da se ne bunimo sa Eager load i Lazy load. Kada je `with` dodan u `web.php` lako je vidljiv i znamo da je Eager load. Kada vidimo poziv `getPosts()` ili `getRoles()` znamo da je Lazy load. Na roli je obrnuto gdje imamo relaciju prema userima gdje imamo pivot u igri koji će nam vraćati podatke.

Idemo na `Role.php`:

```
/**  
 * The user that belong to the role.  
 *  
 * @return BelongsToMany  
 */  
public function users(): BelongsToMany  
{  
    return $this->belongsToMany(User::class, 'user_roles', 'role_id',  
'user_id');  
}
```

Napisat ćemo i `getUsers()`:

```
public function getUsers(): Collection  
{  
    return $this->users()->get();  
}
```

return tip je `Collection`.

Ako sad pogledamo `web.php` u User imamo relaciju rolama što znači da želimo i role dohvatiti u Eager loadu.

```
Route::get('/', function () {  
    //return view('welcome');  
    $u1 = User::with(['posts', 'roles'])->find(1);  
    //$posts = $u1->posts;  
    dd($u1);  
});
```

Eager load će se upaliti i trebali bi vidjeti u atributu relations kolekciju.

```

App\Models\User {#1208 ▼ // routes\web.php:11
    #connection: "mysql"
    #table: "users"
    #primaryKey: "id"
    #keyType: "int"
    +incrementing: true
    #with: []
    #withCount: []
    +preventsLazyLoading: false
    #perPage: 15
    +exists: true
    +wasRecentlyCreated: false
    #escapeWhenCastingToString: false
    #attributes: array:11 [▶]
    #original: array:11 [▶]
    #changes: []
    #casts: array:2 [▶]
    #classCastCache: []
    #attributeCastCache: []
    #dateFormat: null
    #appends: []
    #dispatchesEvents: []
    #observables: []
    #relations: array:2 [▼
        "posts" => Illuminate\Database\Eloquent\Collection {#1210 ▼
            #items: []
            #escapeWhenCastingToString: false
        }
        "roles" => Illuminate\Database\Eloquent\Collection {#1209 ▼
            #items: []
            #escapeWhenCastingToString: false
        }
    ]
    #touches: []
    +timestamps: true
    +usesUniqueIds: false
    #hidden: array:2 [▶]
    #visible: []
    #fillable: array:5 [▶]
    #guarded: array:1 [▶]
    #authPasswordName: "password"
    #rememberTokenName: "remember_token"
}

```

Vidimo kolekcije koje su prazne.

Da ponovimo koraci su kad imate bazu, definirati modele, definirati na tim modelima što želite da je Mass assignment da možete puniti te podatke na modelu, da li je nešto `$hidden`, više je naglasak na relacijama. Trebate kreirati relacije i odnose između vaših modela. Treba gledati da su te relacije kreirane i na bazi.

Seeder-i

Nisu krucijalni u radu sa Laravelom. Služe da brzinski napunimo bazu. Factory je po arhitekturi neka vrsta patterna. Factory nema veze s factory patternom na refactoring.guru/design-patterns/.

Ugrađeni seeder `UserFactory.php`

Factory ovdje može kreirati setove podataka. Postoji ugrađeni fake koji generira podatke. Imamo već jedan izgrađen factory u `/database/factories/UserFactory.php`. Logika factory-ja je da se osloni na model i uz njegovu pomoć generira podatke koje zapiše u bazu. Bitno je da prilikom definiranja funkcije `definition()` da return ključevi ove asocijativne matrice budu atributi našeg User modela.

```
/**
 * Define the model's default state.
 *
 * @return array<string, mixed>
 */
public function definition(): array
{
    return [
        'first_name' => fake()->firstName(),
        'last_name' => fake()->lastName(),
        'email' => fake()->unique()->safeEmail(),
        'email_verified_at' => now(),
        'password' => Hash::make('password'),
        'remember_token' => Str::random(10),
    ];
}
```

Ako pogledamo atribute User modela `first_name`, `last_name`, `email`, `password`, `phone`. To su `$fillable` atributi. To nisu svi atributi na bazi ali ostale ne želimo popuniti sa factory-jem. Koji su svi atributi možemo vidjeti na migraciji. Poanta ako ne `nullable()` treba generirati. Password je mandatory pa ne možemo koristiti seeder-e.

Na faker-u postoje metode koje omogućavaju različite tipove podataka. `fake()` metoda je helper metoda. U email je `unique()` jer ne želimo više istih mailova a `safeEmail()` da bude siguran mail.

Za potrebe testiranja smo stavili da je password za sve generirane podatke doslovno riječ password, za testne svrhe. Mogli smo (ali nismo) napraviti:

```
'password' => static::password ??= Hash::make('password'),
```

Ne želimo generirati random password jer nećemo znati koji je i nećemo moći pristupiti. To generiranje korisnika je isključivo za potrebe testiranja.

Možemo kreirati vlastiti faker za Post.php. Treba nam factory. Ako želimo u seeder-u koristiti factory-je, a rekli smo da se factory oslanjaju na modele, da bi to radilo, model obavezno mora uključiti trait `HasFactory`. Trait-ove smo spomenuli [kod ugrađenog modela User.php](#). Kada pozovemo trait pomoću use unutar klase, sve funkcije koje se nalaze u trait-u su tada dostupne.

Kada imamo factory imamo model koji na sebi ima `HasFactory` i možemo `HasFactory` koristiti u seeder-u. Željeli bi smo kod `Post.php` kreirati faker koji bi trebao generirati nekakve postove. Postovi

su vezani za user-e. To znači da moramo kada se kreiraju user-i uzeti ID-ove i kreirati postove. To je malo kompleksnije nego kod `UserFactory.php`.

Što ćemo u `Post.php` ako gledamo tablicu `xxx_create_post_table.php` staviti. To je `user_id` koji je `constrained()`. Dakle postoji ograničenje nad tim stranim ključem i ako probamo gurnuti `user_id` koji ne postoji, u tablici user doći će do greške.

```
$table->foreignId('user_id')->constrained()->cascadeOnDelete();
```

To su česte situacije na koje bi trebali pripaziti kako bi generirali factory za `Post.php`.

Ako bi htjeli generirati novi model factory, koristit ćemo Artisan naredbu:

```
php artisan make:factory KorisničkiSeeder
```

Svi seederi bit će smješteni u `database/seeders` direktorij.

Database seeder je nešto što olakšava život kada treba testirati stvari. Ako pogledamo [dokumentaciju](#), Zaštita masovne dodjele (engl. Mass assignment protection) automatski se onemogućava tokom sijanja (engl. seeding) baze podataka. Dakle s ovime treba biti pažljiv i koristiti to na početku za popunjavanje tipova poslovnica, rola itd. Seeder-e na produkciji ne bi trebalo koristiti.

Seeder pokrenemo iz Artisana i kažemo da inserta podatke u bazu. Može koristiti Eloquent query builder ili Eloquent model factories, što je po predavaču bolje. U nekim situacijama to nema potrebe. Factories je zgodan kod hrpe nekakvih podataka jer brže može izgenerirati veliku količinu zapisa dok seeder je zgodan za jednostavnije stvari (npr. za role). Kod factories nema random-a. Factory će se osloniti na modele gdje pomoću fakera može kreirati tisuće zapisa.

Pogledajmo kako to sve skupa radi. Kada pokrenemo `php artisan db:seed` aktiviramo seeder. Možemo na više načina koristiti factory. Jedan je da unutar seeder-a pozovemo:

```
public function run(): void
{
    User::factory(10)->create();
}
```

Dakle daj nam 10 usera i kreiraj ih i faker će ih izgenerirati.

Ili možemo koristiti faker, unutar seeder-a jedan po jedan.

```
public function run(): void
{
    User::factory()->create([
        'name' => 'Test User',
        'email' => 'test@example.com'
    ]);
}
```

Odabrat ćemo prvu verziju. Evo cijelog koda:

```
<?php

namespace Database\Seeders;

use App\Models\User;
// use Illuminate\Database\Console\Seeds\WithoutModelEvents;
use Illuminate\Database\Seeder;

class DatabaseSeeder extends Seeder
{
    /**
     * Seed the application's database.
     */
    public function run(): void
    {
        User::factory(10)->create();
    }
}
```

Ova naredba pokreće `run()` unutar `DatabaseSeeder.php`.

```
php artisan db:seed
```

```
$ php artisan db:seed
INFO  Seeding database.
```

Ovime se popunjava baza podataka testnim ili inicijalnim podatcima koji su statički ili generirani korištenjem factory-ja (kao u ovom slučaju). Izvršavaju se seeder-i koji su definirani u projektu. U ovom trenutku pokrećemo seeder klasu koja je definirana u zadoanoj datoteci `DatabaseSeeder.php` ili drugim seed klasama koje ste kreirali (za sada ih nemamo). Seederi često koriste Laravelove factories i alat Faker za generiranje lažnih, ali uvjerljivih podataka (npr. nasumična imena, email adrese itd.).

Nakon ovoga u tablici `users` treba biti 10 novih user-a. Pero ima id 1 ali idućih 10 su izgenerirani. Svi imaju hash-irane passworde ali svi maju password koji je definiran kao password. Tako vrlo brzo možemo izgenerirati user-e.

| <code>id</code> | <code>first_name</code> | <code>last_name</code> | <code>email</code> | <code>email_verified_at</code> | <code>password</code> | <code>remember_token</code> | <code>created_at</code> | <code>updated_at</code> |
|-----------------|-------------------------|------------------------|---------------------------|--------------------------------|------------------------------------------------------------|-----------------------------|-------------------------|-------------------------|
| 1 | Pero | Perić | pero@pero.com | NULL | 123456 | NULL | 2024-09-22 23:45:33 | 2024-09-22 23:45:33 |
| 2 | Wayne | Herzog | metz.percival@example.net | 2024-09-22 23:45:31 | \$2y\$12\$4bG6MYG4jYBPaHupWYDlEOnnZubChV... p90NNBX0m9 | 2024-09-22 23:45:33 | 2024-09-22 23:45:33 | 2024-09-22 23:45:33 |
| 3 | Kiel | Mohr | joesph.hills@example.net | 2024-09-22 23:45:31 | \$2y\$12\$zuAYF10cxHdflJU400b.qgIG3Em6df... PINzUy4f7 | 2024-09-22 23:45:33 | 2024-09-22 23:45:33 | 2024-09-22 23:45:33 |
| 4 | Nathaniel | Bergnaum | winfield87@example.org | 2024-09-22 23:45:31 | \$2y\$12\$C/N1Qz0Nid6rSTSXYvCoM.yQN9exEUjt... SMis4TuRMV | 2024-09-22 23:45:33 | 2024-09-22 23:45:33 | 2024-09-22 23:45:33 |
| 5 | Raleigh | Gerlach | ludie.heaney@example.net | 2024-09-22 23:45:31 | \$2y\$12\$hxEV6exhuzm47m.Bv6dTTe10NQd6V... FWPAXUEw4t | 2024-09-22 23:45:33 | 2024-09-22 23:45:33 | 2024-09-22 23:45:33 |
| 6 | Marianne | Schaden | koch.eldred@example.org | 2024-09-22 23:45:32 | \$2y\$12\$huwFqiuJSfDUY/zqvVldOkeqjKTQdNZ/A... GWvdQGS0gm | 2024-09-22 23:45:33 | 2024-09-22 23:45:33 | 2024-09-22 23:45:33 |
| 7 | Aleen | Ortiz | elise.kling@example.com | 2024-09-22 23:45:32 | \$2y\$12\$ZHP1SP.Rciw2uJYKLSFuEB.DwKQTTr... W3no1puMxu | 2024-09-22 23:45:33 | 2024-09-22 23:45:33 | 2024-09-22 23:45:33 |
| 8 | Myriam | Schuster | ozella76@example.org | 2024-09-22 23:45:32 | \$2y\$12\$xe6fp7Rgozffao.Sq3X0Putghjs98UwH... Y7TJGeMnnR | 2024-09-22 23:45:33 | 2024-09-22 23:45:33 | 2024-09-22 23:45:33 |
| 9 | Kristoffer | Walsh | ruby.volkmann@example.org | 2024-09-22 23:45:33 | \$2y\$12\$3X126oGEHv16FhyIMmhNep3ZssO/s.... 2ovysYFEB | 2024-09-22 23:45:33 | 2024-09-22 23:45:33 | 2024-09-22 23:45:33 |
| 10 | Aubrey | O'Hara | ledner.olin@example.com | 2024-09-22 23:45:33 | \$2y\$12\$pPhiGM004bs1/SdpArfb6eDOZPRgc4C... dMJAJgxEZo | 2024-09-22 23:45:33 | 2024-09-22 23:45:33 | 2024-09-22 23:45:33 |
| 11 | Judd | Prosacco | brendon62@example.com | 2024-09-22 23:45:33 | \$2y\$12\$ql.nLtrNxu.9KGa5lZmR.a5FZrkdeEUldZ... DaM22qrpbR | 2024-09-22 23:45:33 | 2024-09-22 23:45:33 | 2024-09-22 23:45:33 |

Da smo imali više seedera i želi smo još 10 usera, dovoljno je pokrenuti:

```
php artisan db:seed --class=DatabaseSeeder
```

Seeder Post

Sada idemo napraviti custom factory. Pokazat ćemo kako ga napraviti, kako ga napuniti i kako ga povezati s User-om.

```
php artisan make:factory Post -model=Post
```

Ova naredba kreira novu datoteku u direktoriju `database/factories`, nazvanu `PostFactory.php`. Ta generirana datoteka sadržavat će osnovni kod za kreiranje lažnih podataka koji se mogu koristiti za popunjavanje tablice povezane s modelom Post. Flag `--model=Post` osigurava da factory bude povezan s modelom Post. To znači da će factory automatski biti konfiguriran da kreira instancu modela `Post` sa svim potrebnim poljima.

Factory se zove `Post`. Veže se za taj model `Post`. Izgenerirao se `PostFactory.php`, naravno u `database/factories`. U njemu imamo klasu `PostFactory` i funkciju `definition()`. Dodat ćemo polja u metodu `definition()`, a koja odlučit ćemo tako da pogledamo u migraciju `xxx_create_posts_table.php` gdje je definirana tablica `posts`:

```
namespace Database\Factories;

use App\Models\User;
use Illuminate\Database\Eloquent\Factories\Factory;

/**
 * @extends \Illuminate\Database\Eloquent\Factories\Factory<\App\Models\Post>
 */
class PostFactory extends Factory
{
    /**
     * Define the model's default state.
     *
     * @return array<string, mixed>
     */
    public function definition(): array
    {
        $users = User::all()->pluck('id')->toArray();

        return [
            "title" => fake()->sentence(),
            "content" => fake()->paragraph(),
            "image" => fake()->imageUrl(),
            "slug" => fake()->unique()->slug(),
        ];
    }
}
```

```

        "user_id" => fake()->random_int(2, 11),
    ];
}
}

```

Ovaj `PostFactory.php` služi za generiranje lažnih podataka za model `Post.php`, uključujući naslove, sadržaj, slike, slug-ove i povezivanje s korisnicima (modelom User). Ovaj factory nalazi se u namespacu `Database\Factories`. `use App\Models\User` omogućava pristup modelu `User` kako bi se kasnije mogao koristiti unutar factory-ja za dohvatanje korisničkih ID-jeva. `use factory` nasljeđuje osnovnu funkcionalnost iz Laravelove Factory klase. Klasa `PostFactory` nasljeđuje osnovnu Laravelovu factory klasu (`Factory`) i koristi se za generiranje lažnih podataka za model `Post`. U komentarima je i naglašeno da je to factory za model `Post`.

Red `$users = User::all()->pluck('id')->toArray();` dohvata sve korisnike (`User::all()`), uzima samo njihove ID-jeve (`pluck('id')`) i pretvara ih u matricu (`toArray()`). Ovo se koristi za slučajni odabir korisničkog ID-a za svaki kreirani post. `return` unutar `definition()` metode vraća asocijativnu matricu koji sadrži lažne podatke za model `Post`: geneira se slučajna rečenica za naslov posta, nasumični URL adrese slike (spremamo samo putanje do slike a ne BLOB ili BASE64), jedinstveni slug koji će se koristiti kao identifikator za post i nasumični ID iz matrice svih korisničkih ID-ova (u stvarnoj situaciji pravimo ga iz naslova). To povezuje post s nekim korisnikom. `slug` je jedinstven.

Da bi smo to mogli pozvati u `DatabaseSeeder.php` dodat ćemo red ispod postojećeg koji kreira 10 usera `User::factory(10)->create();`:

```
Post::factory(10)->create();
```

Prva naredba kreira 10 korisnika (u `User` tablici), dok će druga kreirati 10 postova (u Post tablici).

Ako sada ponovo pokrenemo seeder on će izgenerirati 10 usera i tako svaki puta. Zgodno je obrisati podatke ako nam stari podaci ne trebaju za testove. Bitan je redoslijed kako brišemo. Ako pogledamo tablicu `database/migrations/xxx_create_posts_table.php` vidjet ćemo da imamo ograničenje (engl. constrained) `cascadeonDelete()` na stranom ključu `user_id`.

```
$table->foreignId('user_id')->constrained()->cascadeonDelete();
```

To znači da ako obrišemo usera kaskadno će obrisati i postove. To nije dobro jer ne želimo gubiti podatke. Dakle usera brišemo sa softdelete. Če na kod sada izgleda ovako:

```

<?php

namespace Database\Seeders;

use App\Models\Post;
use App\Models\User;
use Illuminate\Database\Seeder;

class DatabaseSeeder extends Seeder

```

```
{  
    /**  
     * Seed the application's database.  
     */  
    public function run(): void  
    {  
        User::query()->delete();  
        User::factory(10)->create();  
        Post::query()->delete();  
        Post::factory(10)->create();  
  
        $this->call([  
            RoleSeeder::class,  
        ]);  
    }  
}
```

Pokrenut ćemo ponovo:

```
php artisan db:seed
```

Dogodila se greška gdje je dojavio da je došlo do kršenja ograničenja integriteta jer nije moguće dodati ili ažurirati podređeni red jer ograničenje stranog ključa ne uspijeva.

```
Integrity constraint violation: 1452 Cannot add or update a child row: a foreign  
key constraint fails...
```

Useri u `users` tablici su dodani ali u `posts` tablici nisu. Puknuo je na brisanju post-ova.

```
User::query()->delete();  
User::factory(10)->create();  
//Post::query()->delete();  
Post::factory(10)->create();
```

U migraciji imamo kao što smo već spomenuli:

```
$table->foreignId('user_id')->constrained()->cascadeOnDelete();
```

Generirali smo Id usera od 2 do 11. Kako smo ih obrisali a on je generirao nove ID od 12 do 21. Kod inserta je puknuo random ubacivanje u `user_id` u `/database/factories/PostFactory.php`.

Red `"user_id" => fake()->random_int(2, 11)`, treba zamijeniti s:

```
"user_id" => fake()->randomElement($users),
```

Razlika je što u prvoj `user_id` koji je nasumičan, ali se oslanja na specifične brojeve `user_id` vrijednosti u rasponu od 2 do 11 `user_id` unutar tog opsega a ovo drugo rješenje koristi `user_id` koji se dodjeljuje za usere i stvarno postoji u bazi podataka. Pukao je zato što je random integer bio od 2 do 11 a naši `user_id` su bili u rasponu od 12 nadalje i puknulo je na ograničenju (engl. constrain). Ne može dodati `user_id` koji ne postoji. Dakle bitno je da id postoji.

Ako sada probamo `php artisan db:seed` sve radi. Ako pogledamo u tablicu users vidjet ćemo da su od 1 do 10 obrisani i generirani novi:

| ID | first_name | last_name | email | email_verified_at | password | remember_token | created_at | updated_at | deleted_at | phone |
|----|------------|------------|---------------------------------|---------------------|-----------------------------------------------|----------------|---------------------|---------------------|------------|-------|
| 11 | Frieda | Thompson | vrippin@example.net | 2024-09-23 12:08:43 | \$2y\$12\$zQcaejxuq3hr9eUHj8DyfJN0qj_68USj... | LbxvYH5jt2 | 2024-09-23 12:08:46 | 2024-09-23 12:08:46 | NULL | NULL |
| 12 | Alessandra | Herzog | gledoman@example.net | 2024-09-23 12:08:44 | \$2y\$12\$z1yJULuqAfj6js5wKuiZ.0L062pp2B... | J0Wal6xw7 | 2024-09-23 12:08:46 | 2024-09-23 12:08:46 | NULL | NULL |
| 13 | Chester | Smith | sawayin.ressie@example.org | 2024-09-23 12:08:44 | \$2y\$12\$bb5QLLekzMATo2CTb.GCA.kkYrmA6b1... | fDoVlxorRx | 2024-09-23 12:08:46 | 2024-09-23 12:08:46 | NULL | NULL |
| 14 | Erling | Lesch | zhar.mann@example.org | 2024-09-23 12:08:44 | \$2y\$12\$hd8nxzKcJhRONZhk8sfCauUC7Cvk1D... | r8Yj#Pcoo | 2024-09-23 12:08:46 | 2024-09-23 12:08:46 | NULL | NULL |
| 15 | Cletus | Feehey | pledner@example.org | 2024-09-23 12:08:45 | \$2y\$12\$P.1hmKCW7awICa.EpoIveQd1g9B... | qo62Y0tyRr | 2024-09-23 12:08:46 | 2024-09-23 12:08:46 | NULL | NULL |
| 16 | Jensen | Beatty | jaron.botsford@example.org | 2024-09-23 12:08:45 | \$2y\$12\$5f.69QV/Wyyxf4v2a.1DDetd5HBGdv5... | pSweepglm0 | 2024-09-23 12:08:46 | 2024-09-23 12:08:46 | NULL | NULL |
| 17 | Corallie | Reilly | considine.roosevelt@example.net | 2024-09-23 12:08:45 | \$2y\$12\$sp/1YkQqlz9mQLz9mQLz9mQLz9mQLz... | iwRUtwYqJ5 | 2024-09-23 12:08:46 | 2024-09-23 12:08:46 | NULL | NULL |
| 18 | Shayne | Dicki | cshinner@example.com | 2024-09-23 12:08:45 | \$2y\$12\$Oh+QnWIBKEusErFMVOEvgos1Gs... | RplU8lJ7N | 2024-09-23 12:08:46 | 2024-09-23 12:08:46 | NULL | NULL |
| 19 | Vicente | Kuphal | eula.schiller@example.net | 2024-09-23 12:08:46 | \$2y\$12\$Kfc7gmOgNbG8o7Ab4juK4k6Fa0s... | y4N1yYQhVv | 2024-09-23 12:08:46 | 2024-09-23 12:08:46 | NULL | NULL |
| 20 | Tatyana | Stiedemann | sct53@example.org | 2024-09-23 12:08:46 | \$2y\$12\$hlXtHNY1QEm.yhQbomghRMOWhD2VR... | 85hoFQ0sq | 2024-09-23 12:08:46 | 2024-09-23 12:08:46 | NULL | NULL |

Također u posts vidimo nove postove od 11 do 20 (stari su obrisani) i vezu prema stvarnim userima:

| ID | Title | Content | Slug | Image | User ID | Created At | Updated At | Delete |
|----|----------------------------------------------------|-------------------------------------------------------|---------------------------------------------------|------------------------------------------------|---------|---------------------|---------------------|--------|
| 11 | Labore voluptas officiis sunt sed. | Saepe ure aut nobis sapiente voluptate dolores... | ulam-illum-autem-adipisci-eum-accusamus... | https://via.placeholder.com/640x480.png?077... | 14 | 2024-09-23 12:08:46 | 2024-09-23 12:08:46 | 0 |
| 12 | Architecte quibusdam voluntatis ut beatas as... | Eveniet facere rerum quis commodi. Quos sol... | veritatis-voluptatum-eurum-ipsum | https://via.placeholder.com/640x480.png?033... | 15 | 2024-09-23 12:08:46 | 2024-09-23 12:08:46 | 0 |
| 13 | Sit delenit doloris et et. | Dolorum enim est. Eveniet quibusdam conse... | vite-et-sit-reqendis-et-potius-provident | https://via.placeholder.com/640x480.png?001... | 14 | 2024-09-23 12:08:46 | 2024-09-23 12:08:46 | 0 |
| 14 | Eligendi beatas ut cum. | Distinctio vel nam et velit. Natus eum qui qua aut... | aut-dolorem-est-in-esse-cumque-et-molestiae | https://via.placeholder.com/640x480.png?066... | 16 | 2024-09-23 12:08:46 | 2024-09-23 12:08:46 | 0 |
| 15 | Maiores ratione exercitationem qui deserunt eu... | Et consequatur voluptates mollitia. Repellendus... | minus-est-at-nunquam-nostrum-commodi-lo | https://via.placeholder.com/640x480.png?011... | 14 | 2024-09-23 12:08:46 | 2024-09-23 12:08:46 | 0 |
| 16 | Libero laboriosam ea qui fugit alias opto. | Adipiso inviduit natus qui consequeatur. Sed ... | xpedita-vitae-officia-eaque-quod | https://via.placeholder.com/640x480.png?044... | 16 | 2024-09-23 12:08:46 | 2024-09-23 12:08:46 | 0 |
| 17 | Veniam exercitationem officiis odit voluptate... | Assumenda nihil non tenetur deserunt aliquam... | nemo-adipisci-nunquam-voluptatebus | https://via.placeholder.com/640x480.png?099... | 12 | 2024-09-23 12:08:46 | 2024-09-23 12:08:46 | 0 |
| 18 | Ullam debitis nunquam voluptas nulla impedit no... | Cum quid quod. Est ilium perferendis aut inc... | obis-inviduit-mod-harum-dicta-qui-qui | https://via.placeholder.com/640x480.png?088... | 14 | 2024-09-23 12:08:46 | 2024-09-23 12:08:46 | 0 |
| 19 | Facilis odit vero minima qui ut dolors. | Quidem est nesciuit ut libero sed consecutus... | et-repellat-corrupt-set-peum | https://via.placeholder.com/640x480.png?088... | 14 | 2024-09-23 12:08:46 | 2024-09-23 12:08:46 | 0 |
| 20 | Explicabo qui rem autem voluptatem sint. | Quia et voluptas ut aut voluptatem. Atque do... | explicabo-aut-distinctio-et-optio-laudentum-an... | https://via.placeholder.com/640x480.png?099... | 18 | 2024-09-23 12:08:46 | 2024-09-23 12:08:46 | 0 |

Random je povukao tako da neki useri imaju više postova.

Metoda `pluck()`

Ako želite dobiti ID-eve svih korisnika iz baze, možete koristiti `pluck()`:

```
$users = User::all()->pluck('id');
```

Rezultat je kolekcija:

```
[1, 2, 3, 4, 5, 6, ...]
```

U našem primjeru:

```
$users = User::all()->pluck('id')->toArray();
```

Ovo izvlači `id` stupac iz rezultata, praveći kolekciju s ID-evima korisnika.

`pluck()` se koristi kada želimo izvući samo jednu kolonu iz baze podataka:

```
$emails = User::all()->pluck('email'); // ovo dohvata emailove
```

Seeder `RoleSeeder`

Želimo kreirati novi seeder

```
php artisan make:seeder RoleSeeder
```

Možemo za svaku tablicu formirati pojedinačan seeder i u svakom seederu pozivati se na factory. Za to nema potrebe jer možemo sve staviti u `DatabaseSeeder.php`.

Kreirao se `RoleSeeder.php` sa klasom `RoleSeeder` i klasu `run()`. Idemo ga popraviti:

```
<?php

namespace Database\Seeders;

use Illuminate\Database\Console\Seeds\WithoutModelEvents;
use Illuminate\Database\Seeder;
use App\Models\Role;
use App\Models\User;

class RoleSeeder extends Seeder
{
    /**
     * Run the database seeds.
     */
    public function run(): void
    {
        Role::query()->delete();
        Role::insert([
            ['name' => 'Admin'],
            ['name' => 'User'],
            ['name' => 'Author'],
        ]);
    }
}
```

`RoleSeeder.php` je klasa koja se koristi za punjenje baze podataka s početnim podacima za uloge (eng. roles). Ovaj se seeder koristi u okviru Laravelovog sistema migracija i seedera za generiranje testnih podataka. `WithoutModelEvents` ima mogućnost isključivanja događaja modela prilikom sjemenja, iako ovdje nije korišten. `Seeder` je osnovna klasa iz koje se nasljeđuje `RoleSeeder`. `Role` i `User` su modeli koji predstavljaju tablice `roles` i `users` u bazi podataka.

`Role::query()->delete();` briše sve postojeće unose u tablici `roles`, osiguravajući da se novi podaci dodaju bez sukoba s postojećim. Dodaju se tri nove uloge (role) u tablicu `roles`: `Admin`, `User` i `Author`. `Insert` prima višedimenzionalnu matricu.

Naredba `php artisan db:seed` se koristi u Laravelu za izvođenje seeder klase koje su odgovorne za punjenje baze podataka s početnim podacima. Kada se koristi zajedno s imenom specifičnog seeder-a, ova naredba omogućava izvođenje samo tog seeder-a umjesto svih seedova koji su definirani u aplikaciji.

```
php artisan db:seed --class=ImeSeeder
```

Ako imamo desetke seedera suludo je da svaki puta to pozivamo ponovo i ponovo. Mi želimo samo poslati `DatabaseSeeder.php` i da se sve izvrši. Zato ćemo unutar `DatabaseSeeder.php` unutar metode `run()` napraviti callable `$this->call([])` dati listu svih seedera koje želimo da se pokrenu. `$this->call([])` se koristi unutar seeder klasa za pozivanje drugih seeder klasa. Ova metoda omogućava da se unutar jednog seeder-a izvrše drugi seeder-i, što može pomoći u organizaciji i strukturi seeder-a podataka.

```
$this->call([
    RoleSeeder::class,
]);
```

Sada ovo nije potrebno ručno pisati u Artisan u.

Sada ako pokrenemo `php artisan db:seed` radi seeding.

```
$ php artisan db:seed
INFO  Seeding database.
Database\Seeders\RoleSeeder ..... RUNNING
Database\Seeders\RoleSeeder ..... 119 ms DONE
```

Ponovo se kreiraju useri i postovi ali ovaj puta i role.

Ostalo je još vidjeti kako povezati usere s nekom ulogom. Ako pogledamo kako je vezan user i rola, možemo uzeti sve user-e i proći po njima, te svakom useru preko relacije reći kreiraj neku ulogu ili više uloga. Mi ćemo dodijeliti jednu i to nasumično svakom korisniku u bazi podataka. To ćemo napraviti tako da ne znamo `user_id` i `role_id`, dakle nećemo hardkodirati. U `RoleSeeder.php` dodat ćemo dio za povezivanje korisnika s rolama.

```
User::all()->each(function ($user) {
    $user->roles()->attach(Role::inRandomOrder()->first());
});
```

`User::all()` uzima sve korisnike iz tablice `users`. `User` ovdje je Laravelov model `User`. Metoda `all()` dohvaca sve zapise (korisnike) iz tablice `users`. Kao rezultat toga, dobit ćete kolekciju (zbirku) svih korisnika u bazi podataka. Metoda `each()` je funkcija koja se koristi za iteraciju kroz svaki element u kolekciji. U ovom slučaju, to su svi korisnici koje je `User::all()` dohvatiša iz baze. Unutar `each()` funkcije definiramo callback (anonimnu funkciju) koja će se izvršiti za svakog korisnika u kolekciji. Parametar `$user` predstavlja pojedinačnog korisnika. Za svakog korisnika će se unutar ove funkcije izvršiti definirana logika.

`$user->roles()` poziva relaciju između modela `User` i `Role`. Kako smo u modelu `User` definirali many to many vezu s modelom `Role` koristeći metodu `roles()`, što podrazumijeva da korisnik (engl. user)

može imati više uloga. Laravel koristi pivot tablicu za ovu vrstu relacije, koja povezuje korisnički ID s ID-om role u bazi.

Ako pogledamo definiciju te relacije u modelu `User.php` vidjet ćemo da smo definirali:

```
public function roles(): BelongsToMany
{
    return $this->belongsToMany(Role::class, 'user_roles', 'user_id',
'role_id');
}
```

Vratimo se na analizu koda. U dijelu `Role::inRandomOrder()->first()` se koristi model `Role`, koji predstavlja tablicu roles u bazi podataka. `inRandomOrder()` metoda uzima sve uloge iz baze i vraća ih u nasumičnom redoslijedu. Koristi SQL funkciju `ORDER BY RAND()` (ili sličnu, zavisno o bazi). Nakon što su uloge sortirane nasumično, metoda `first()` vraća prvu ulogu iz tog nasumično poredanog skupa. To znači da svaki put kada se ova funkcija pozove, dobit ćete nasumičnu ulogu iz tablice `roles`.

Ova skripta osigurava da svaki korisnik iz baze dobije nasumično dodijeljenu ulogu. Funkcija prolazi kroz sve korisnike, uzima nasumičnu ulogu iz tablice `roles` i povezuje je s korisnikom kroz pivot tablicom koja povezuje korisnike i uloge u mnogostrukom (engl. many to many) odnosu.

S ovime je seeder zadovoljio sve naše potrebe. Sada ako pokrenemo:

```
php artisan db:seed
```

U bazi ćemo vidjeti generirane nove usere, generirane nove postove i generirane role. Popunjena je i pivot tablica `user_roles`. Sada svaki puta kada pozovemo seedere, vidjet ćemo nove podatke u bazi.

Vidimo da kombinacija factory-ja i seeder-a radi.

Definiranje kontrolera

U Laravelu, kontroleri (eng. Controllers) su klase koje organiziraju poslovnu logiku i odgovaraju na HTTP zahtjeve. Umjesto da svu logiku rukovanja HTTP zahtjevima smjestite direktno u rute, koristi se kontroler kako bi se logika bolje organizirala, modulirala i postala lakša za održavanje.

Kontroleri će kontrolirati ulazne zahtjeve i kontrolirali što treba raditi. U direktoriju `App\Http\Controllers` imamo bazni kontroler `Controller.php` koji nasljeđuju svi kontroleri. Ako želite natjerati sve svoje kontrolere da implementiraju neku od tih metoda dovoljno je da je stavimo u klasu `Controller`. Npr ako stavimo `abstract public function index();` unutra time smo natjerali da svaki naš kontroler koji napravimo mora implementirati tu metodu `index()`. Apstraktna klasa osim što je neki temelj, ona može poslužiti da kontrolira sve klase koje nasljeđuju apstraktну klasu, što moraju implementirati.

```
<?php

namespace App\Http\Controllers;
```

```
abstract class Controller
{
    abstract public function index();
}
```

Osnovni (engl. basic) kontroler možemo generirati pomoću Artisan naredbe:

```
php artisan make:controller PrimjerController
```

Ova naredba automatski kreira novu klasu `PrimjerController` u direktoriju `app/Http/Controllers`.

On će biti prazan i mi ćemo ga morati puniti.

Ako pogledamo nešto što se naziva **Resource Controllers**, vidjet ćemo da su to posebna vrsta kontrolera koja omogućava lakše i organiziranje upravljanje **CRUD** (Create, Read, Update, Delete) operacijama. Umjesto ručnog definiranja svake metode u rutu, Resource Controller nudi preddefiniran skup RESTful metoda za najčešće operacije koje se koriste prilikom rada s resursima (kao što su korisnici, postovi, proizvodi itd.). Na taj način ne moramo pisati ni metode ali ni rute jer Laravel omogućava `Route::resource()` metodu koja automatski generira sve rute za ove operacije. On u toj metodi raspoznaće različite request-ove i zna ih rasporediti na ispravnu metodu u kontroleru.

Da bi kontroler odgovarao na HTTP zahtjeve, trebate ga povezati s rutom koji preusmjerava zahtjev na kontroler. Ovo se može napraviti u Laravelu unutar datoteka u direktoriju routes (najčešće `routes/web.php` za web rute) tako da za svaku pojedinu rutu definiramo neku akciju. Moguće je definirati osnovni (engl. basic) kontroler i u njemu metodu i kasnije u rutu definirati rutu prema toj metodi i tom kontroleru.

Ako imamo klasičnu CRUD aplikaciju onda je to nepotrebno pisati. Zato Laravel nudi Resource Controllers koji ima svoja pravila. Pogledajmo kako Resource Controllers rukuje sa akcijama, URI-jima i kako rukuje s metodama kojima se šalje zahtjev (engl. verb).

| HTTP Metoda | URI | Akcija | Koristi metodu | Ime rute |
|-------------|-------------------|-----------------------------|----------------|----------------|
| GET | /resurs | Prikaz svih | index() | resurs.index |
| GET | /resurs/create | Forma za kreiranje | create() | resurs.create |
| POST | /resurs | Kreiranje novog | store() | resurs.store |
| GET | /resurs/{id} | Prikaz pojedinačnog resursa | show() | resurs.show |
| GET | /resurs/{id}/edit | Forma za uređivanje | edit() | resurs.edit |
| PUT/PATCH | /resurs/{id} | Ažuriranje resursa | update() | resurs.update |
| DELETE | /resurs/{id} | Brisanje resursa | destroy() | resurs.destroy |

U rutama nećemo vidjeti putanje, u njima je sakriveno 7 različitih putanja.

Riješit ćemo sada prijavu korisnika. Za to nam neće trebati Resource Controller nego običan kontroler. Nazvat ćemo ga `AuthController.php`.

```
php artisan make:controller AuthController
```

On će automatski kreirati kontroler u našem direktoriju `app/Http/Controllers`.

Ako ne želimo da bude tamo nego negdje drugdje, morali bi to i navesti. Npr.

```
php artisan make:controller Auth/AuthController
```

Ova naredba bi stvorila `AuthController.php` u `app/Http/Controllers/Auth` direktoriju.

Pogledajmo što je izgenerirano:

```
<?php

namespace App\Http\Controllers;

use Illuminate\Http\Request;

class AuthController extends Controller
{
    //
}
```

Primijetit ćemo da je automatski ekstendao bazni kontroler. To znači da smo u baznom kontroleru nešto definirali. Da smo imali onaj `abstract public function index();` u `Controllers.php` onda će `AuthController` zahtijevati da implementiramo tu metodu. Vidjeli smo da `index` metoda možda baš nije pametna, zbog Resource Controller-a. Ali recimo može to biti metoda koja trigerira servis koji preuzima odgovornost i radi sve što treba.

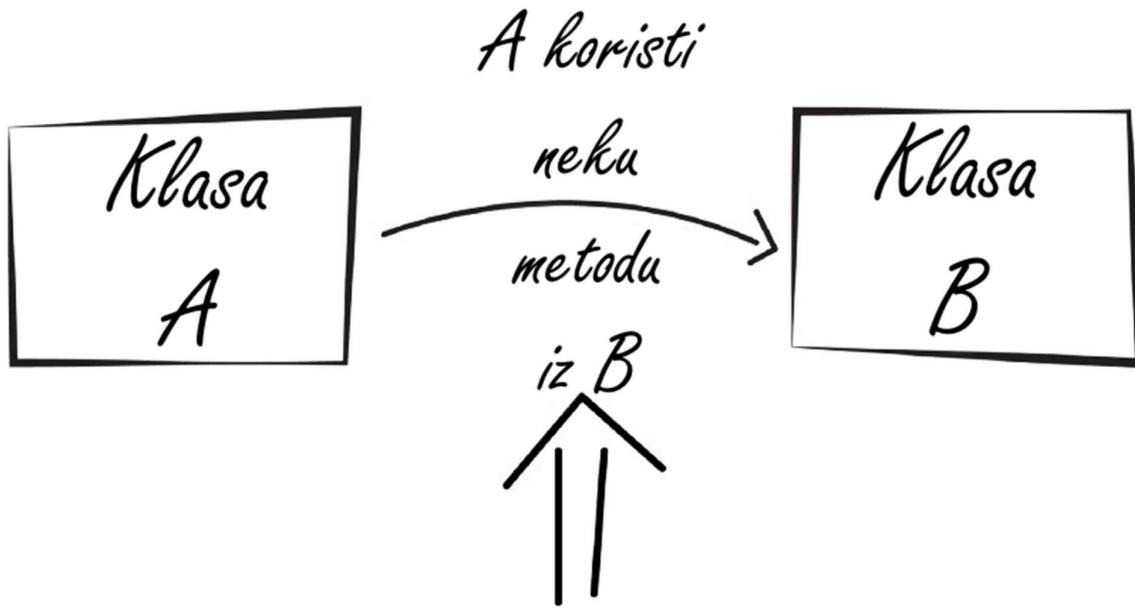
Ovdje ćemo pokazati klasičan pristup a kod API-ja kako uhvati dodatni, servisni sloj.

U `AuthController.php` napisat ćemo metodu `login()`:

```
public function login(Request $request)
{
}
```

Parametar `Request $request` koristi se za prihvatanje HTTP zahtjeva. `Request` je Laravelov objekt koji sadrži sve informacije o dolaznom HTTP zahtjevu (kao što je podatak iz forme, URL parametara, kolačić, itd.). U ovom slučaju, podaci iz zahtjeva nisu eksplicitno korišteni u metodi, ali su dostupni ako zatrebaju.

Laravel kontroleri automatski rade **dependency injection**, što znači da možete direktno u kontroler metodu "ubrizgati" potrebne objekte. U Laravelu, dependency injection se koristi na način da framework automatski osigurava potrebne objekte kada su definirani kao parametri metode ili konstruktora. To znači da će ruta prosljediti zahtjev na login. Laravel je već napunio `$request` objekt i onda vam ga gurnuo u ovu metodu ako vi to zahtijevate. Vi to ne morate. Dakle, možemo ako nam je potrebno pozvati `login()` i bez parametara, ako nam je to potrebno. Do `$request` možemo doći na neki drugi način. Naravno, mi ćemo koristiti `$request`.



U ovom slučaju, objekat `Request` je zavisnost metode `login()`. Laravel automatski "ubrizgava" instancu `Request` klase kad god pozove ovu metodu. To znači da mi kao programeri ne moramo ručno kreirati ili dohvatiti objekat `Request`. Laravelov **service container** to radi za nas. `Request` je klasa koja pruža sve podatke o trenutnom HTTP zahtjevu (`GET` ili `POST`, parametri, kolačići, itd.). Kad Laravel prepozna da metoda prima parametar tipa `Request`, automatski stvara i proslijeđuje instancu klase `Illuminate\Http\Request` u tu metodu.

Laravel koristi Service Container, koji je osnovni mehanizam za rješavanje i upravljanje zavisnostima u aplikaciji. Kada servisna klasa ili metoda treba neku zavisnost, kao što je `Request`, Laravelova arhitektura prepoznaće tip zavisnosti i automatski je osigurava koristeći servisni kontejner.

Unutar funkcije `public function login(Request $request)` unijet ćemo:

```

public function login(Request $request)
{
    dump($request);
    dd(request());
}
  
```

Da bi smo ovo pokrenuli, moramo pokrenuti Artisan server, već poznatom naredbom:

```
php artisan serve
```

Rute

Nitko nije proslijedio zahtjev na ovaj kontroler. Za to su zadužene rute. Ruta će moći preusmjeriti zahtjev kontroleru. Ako pogledamo naše rute u `routes/web.php`. Vidjeli smo da imamo neku rutu, koja ne preusmjerava zahtjev kontroleru, nego Laravel ima mogućnost i u samoj ruti kao drugi argument da pošaljete callback funkciju koja se izvrši. Dakle, zahtjev se može poslati callback funkciji i može se poslati kontroleru. U `web.php` ćemo dodati:

```
Route::get('/login', []);
```

Šaljemo matricu. Osim matrice može biti, callable (funkcija array gdje ćemo poslati kontroler i metodu za taj kontroler) string (gdje ćemo poslati samo kontroler, a onda mora biti resource). Jedan od načina je samo matrica, što znači da njemu možemo poslati `AuthController` kao prvi element te matrice a kao drugi element te matrice je `'login'` u tom kontroleru. Ruta će preusmjeriti taj zahtjev na `Login`.

```
Route::get('/login', [AuthController::class, 'login']);
```

Ovdje je `[AuthController::class, 'login']` callable tip, gdje se koristi kao pokazivač na metodu `login` unutar `AuthController` klase.

Kada bi pogledali `localhost:8000/login`, trebali bi vidjeti Request objekt dva puta.

```

Illuminate\Http\Request {#43 ▶ // app\Http\Controllers\AuthController.php:22
+attributes: Symfony\...\ParameterBag {#48 ▷}
+request: Symfony\...\InputBag {#44 ▷}
+query: Symfony\...\InputBag {#51 ▷}
+server: Symfony\...\ServerBag {#46 ▷}
+files: Symfony\...\FileBag {#50 ▷}
+cookies: Symfony\...\InputBag {#49 ▷}
+headers: Symfony\...\HeaderBag {#45 ▷}
#content: null
#languages: null
#Charsets: null
#encodings: null
#acceptableContentTypes: null
#pathInfo: "/login"
#requestUri: "/login"
#baseUrl: ""
#basePath: null
#method: "GET"
#format: null
#session: Illuminate\...\SymfonySessionDecorator {#220 ▷}
#locale: null
#defaultLocale: "en"
-preferredFormat: null
-isHostValid: true
-isForwardedValid: true
-isSafeContentPreferred: ? bool
-trustedValuesCache: []
-isIisRewrite: false
#json: null
#convertedFiles: null
#userResolver: Closure($guard = null) {#187 ▷}
#routeResolver: Closure() {#196 ▷}
basePath: ""
format: "html"
}

```

`Illuminate\Http\Request {#43 ▷}`

Primijetit ćete `#43` dva puta jer se radi o istom objektu.

Došli smo do u `routes/web.php` i reda `Route::get('/login', [AuthController::class, 'login']);` Laravel odlazi na `AuthController.php` i **dependency injection**-om je ubacio ovdje request ili došao do requesta pomoću `request()` helper metode. To je jedan te isti objekt.

Unutra se nalazi hrpa podataka. Nema atributa. Puno informacija je moguće izvući iz ovog objekta.

Kako smo sada vidjeli što se nalazi u objektu, obrisat ćemo ta dva reda.

```

dump($request);
dd(request());

```

Ono što nemamo je prompt dio gdje bi omogućili korisniku da upiše username i password. Idemo vidjeti da li Laravel može handle-ati autentifikaciju korisnika, idemo vidjeti možemo li doći do podataka o korisniku. Iz `$request` ako pozovemo `user()`, iz funkcije `login()`:

```
dd($request->user());
```

Dobit ćemo:

```
null // app\Http\Controllers\AuthController.php:22
```

Nema trenutno niti jednog autentificiranoj user-a ali iz `$request()` možemo dobiti informaciju koji je to user autentificiran. Ta informacija će nam bitna u trenutku kada želimo određene informacije štititi. Netko je napravio request, zanima nas tko je to i koje dozvole ima. To možemo napraviti na više načina. Nije dovoljno vidjeti da li je korisnik prijavljen nego i tko je on u sistemu. Upravljanje policy-ima tj. permission-ima je vrlo bitno.

Laravel može koristiti i `auth()` helper metodu kojom može doći do user-a. U svakom trenutku možemo bilo gdje u aplikaciji pitati da li je tekući korisnik autentificiran.

```
dd(auth()>user);
```

Ne ovisimo o `request`-u, nego možemo bilo kada provjeriti korisnika. Također može koristiti različite metode za autentifikaciju korisnika, provjeru statusa, odjavu itd. Neke od najčešćih metoda uključuju:

- `auth()>check()` - Provjerava je li korisnik prijavljen.
- `auth()>id()` - Dohvata ID trenutno prijavljenog korisnika.
- `auth()>attempt($credentials)` - Pokušava autentifikaciju s proslijedjenim akreditivima (npr. kombinacija emaila i lozinke).

Pogledi

Ruta je preusmjerena na kontroler. Međutim, što ako trebamo imati vezu na nekom svom pogledu tj. nekom html-u ako pričamo o monolitnoj aplikaciji, gdje ćemo stvoriti link koji će odvesti na login.

Ako pogledamo `resources/views/` gdje su pogledi i otvorimo `welcome.blade.php` koji je ogroman. On nam ne treba pa ćemo ga obrisati.

U originalnom [uputstvu o pogledima](#) piše da kreiramo html ručno ili pomoću Blade-a. Blade nudi dodatne fukcionalnosti. Za naprednije stvari koristi se React ili Vue na frontendu. Postoje starter kitevi koji se mogu prilagoditi da koriste Vue ili React. View se može kreirati pomoću Artisana i to kao [Blade template](#). Jedan blade već imamo - `welcome.blade.php`. Blade nam nudi mogućnost korištenja direktiva. Kada želite nešto ispisati iz PHP koristimo dvostrukе vitičaste zagrade `{}{}`, to je u Blade-u naredba `echo` u PHP-u. Prije nego dođe taj `echo` prođe kroz funkciju za sanetizaciju. Laravel nudi tu direktivu kao jedan nivo sigurnosti:

Ako u `resources/views/welcome.blade.php` upišemo:

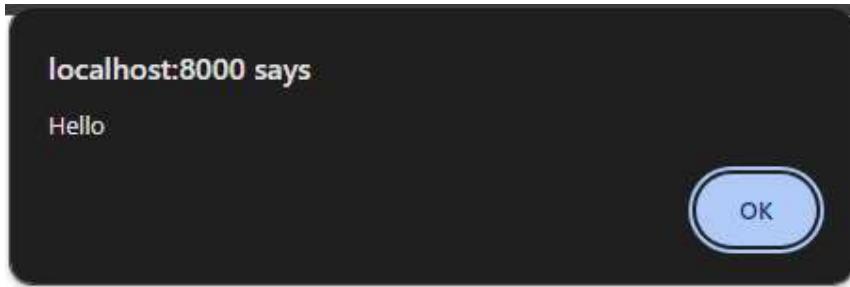
```
{{ '<script>alert("Hello");</script>' }}
```

```
<script>alert("Hello");</script>
```

Laravel će sanetizirati i neće browseru predati JavaScript i pokrenuti ga.

Ako imate potrebu izbjegići sanetizaciju postoji direktiva i za to `{!! !!}`

```
{!! '<script>alert("Hello");</script>' !!}
```



Kreiranje linkova u Blade

Idemo sada reći kako kreirati link na `welcome.blade.php`.

Hardkodirane rute

Možemo hardkodirati rutu kao što to radimo u html-u:

```
<a href="/Login">Login</a>
```

Ovaj link preusmjero nas je na `localhost:8000/Login`.

U nekom trenutku odlučimo da ruta bude u `routes/web.php` umjesto `Route::get('login', [AuthController::class, 'login']);`, da bude `Route::get('auth/login', [AuthController::class, 'login']);`. Mi promijenimo rutu ali npr. u Blade-u u `welcome.blade.php` zaboravimo i tada vidimo:

404 | NOT FOUND

Imenovane rute

Puno je bolje u Laravelu koristiti imenovane rute tj. imenovati svaku rutu. Želimo li ruti dati ime trebamo napisati:

```
Route::get('login', [AuthController::class, 'login'])->name('login');
```

Gdje god nam treba link sada možemo koristiti dvostrukе zagrade, metodu `route()` i unutra ime rute:

```
<a href="{{route('login')}}">Login</a>
```

Dovoljno je da u Blade direktivi za ispis pozovemo helper metodu `route` i ime rute. Laravel će sam prilikom generiranja ovog Blade template u klasičnu PHP datoteku raditi kako treba.

Ako sada u `web.php` napišemo `Route::get('auth/login', [AuthController::class, 'login']);` link i dalje radi, vodi na `localhost:8000/auth/login`. Ostaviti ćemo tako.

Dakle treba koristi imenovane rute a ne oslanjati se na URL. Ako promijenimo URL na rutu onda to moramo raditi i na Blade-u.

Layout opcija

Pogledamo Blade-ov layout. U `resources/views` otvorimo `layout.blade.php`. To je nešto što je zajedničko svim template-ovima. Kostur svakog HTML je isti (zaglavlje, navigacija, podnožje itd.). Zašto bi to morali svaki put dodavati u svakom `.blade.php`. Ovo je zajedničko svim blade template-ima.

Iskopirat ćemo kompletan `welcome.blade.php` i prebacit ćemo ga u `layout.blade.php`.

```
<!DOCTYPE html>
<html lang="{{ str_replace('_', '-', app()->getLocale()) }}>
    <head>
        <meta charset="utf-8">
        <meta name="viewport" content="width=device-width, initial-scale=1">

        <title>
            @yield('title', 'Laravel Blog')
        </title>
    </head>
    <body>
        @yield('content')
    </body>
</html>
```

Red sa `str_replace` dobiva trenutni jezični kod aplikacije (npr. en_US, hr_HR) i zamjenjuje donje crte s crtama, (npr. en_US postaje en-US) jer je takva konvencija u HTML-u. Red sa meta definira kako će stranica izgledati na mobilnim uređajima `width=device-width`: širina stranice će odgovarati širini uređaja. `initial-scale=1`: početni nivo zumiranja je 1 (normalna veličina).

Između `body` tagova dodat ćemo Blade direktivu `@yield` koja definira kontejner u koji u bilo kojem Blade-u koji naslijedi layout ima mogućnost da ubacimo neki content. Inače većina direktiva u Blade-i počinje s `@`. To znači da ne morate pisati ostatak nego u `welcome.blade.php` možemo prebaciti cijeli layout a u content želimo ubaciti samo link. U `welcome.blade.php` obrišemo sve i kažemo:

```
@extends('layout')

@section('content')
    <h1>Welcome to Laravel Blog!</h1>
    <p>
        <a href="{{ route('login') }}>Login</a>
    </p>
@endsection
```

`@extends` kao i kod klase kažemo da Blade nasljeđuje `'Layout'`. To je ime našeg layouta. Da smo napisali `@extends('layouts.layout')` to bi značilo da uđe u direktorij `resources/views` i potraži direktorij `layouts` i u njemu `layout.blade.php`. Ono što smjestimo unutar direktiva `@section('content')` i `@endsection`, Laravel će prilikom kompajliranja Blade template datoteke ubaciti u prostor u `layout.blade.php` gdje smo naveli Blade direktivu `@yield`

U `welcome.blade.php` ubacili smo u `@section('content')` pozdrav i link za login.

Dolazimo do pitanja `<title>` koji je na svim stranicama isti. Ako imamo harkodiran naslov, sve stranice će imati isti `<title>`. I tu želimo napraviti `@yield` i unijeti default vrijednost ako neka stranica ne želi poslati vrijednost:

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

Login link radi ali nemamo nikakav action unutar metode jer je metoda prazna.

Ispod reda `<meta charset="utf-8">` dodat ćemo:

```
<meta name="viewport" content="width=device-width, initial-scale=1">
```

Također iza `</title>` taga dodat ćemo:

```
<link
href="https://cdn.jsdelivr.net/npm/bootstrap@5.2.3/dist/css/bootstrap.min.css"
rel="stylesheet" integrity="sha384-
rbsA2VBKQhggwzxH7pPCaAq046Mgn0M80zW1RWuH61DGLwZJEdK2Kadq2F9CUG65"
crossorigin="anonymous">
```

Saa imamo uključen i Bootstrap za CSS-ove.

Login forma (obrazac)

Sada ćemo generirati login. Želimo neku formu s username i password i želimo da se možemo prebaciti na registar stranicu da se korisnik može registrirati. Za to možemo koristiti Arisan za generiranje Blade-ova:

```
php artisan make:view login
```

ili ručno kreiranjem datoteke `login.blade.php` u direktoriju `resources/views`. Predavač kaže da to radi ručno. Predavaču je ovo izgenerirao M\$ Copilot:

```
@extends('layout')

@section('content')
    <h1>Login</h1>
    @error('credentials')
        <p>{{ $message }}</p>
    @enderror
    <form method="POST" action="{{ route('post.login') }}">
        @csrf
        <div>
            <label for="email">Email</label>
            <input type="email" name="email" id="email" value="{{ old('email') }}"/>
        @error('email')
```

```

<p>{{ $message }}</p>
@enderror
</div>
<div>
    <label for="password">Password</label>
    <input type="password" name="password" id="password">
    @error('password')
        <p>{{ $message }}</p>
    @enderror
</div>
<button type="submit">Login</button>
</form>
<p>Don't have an account? <a href="{{ route('register') }}">Register</a></p>
@endsection

```

Blade direktiva `@extends` nalazi se tu kako bi se naslijedila osnovni layout (šablona) iz datoteke `layout.blade.php`. Blok `@section('content') ... @endsection` koristi Blade direktivu da označi početak dijela stranice koji popunjava `content` sekciju iz layouta. Sve što je unutar ovog bloka će se umetnuti u layout na mjesto gdje je definirano `@yield('content')` u `layout.blade.php`. Direktiva `@error('credentials')` provjerava postoji li greška vezana za `credentials` (npr. pogrešna kombinacija emaila i lozinke). Ako postoji greška, ispisuje se poruka o grešci pomoću varijable `$message`, koja sadrži grešku validacije. `<form method="POST" action="{{ route('post.login') }}">` prikazuje **HTML formu** koja služi za prijavu korisnika. Koristimo `POST` metodu. `action="{{ route('post.login') }}"` postavlja **action** URL forme koristeći Laravel-ovu `route()` funkciju za generiranje rute prema kontroleru koji obrađuje login (ruta nazvana '`post.login`' jer ne može biti '`login`' jer ne može poslati na istu rutu i vratiti se nazad). To bi bilo preusmjeravanje na rutu `GET` a forma ide s `POST`. Direktiva `@csrf` umetne [CSRF token \(engl. Cross-Site Request Forgery token\)](#)⁴ koji Laravel koristi za zaštitu od CSRF napada. Svaka forma koja koristi `POST` metodu treba imati ovaj token. Ona automatski u formu implementira skriveno polje (engl. hidden field) koji se zove `_token` i koji generira `value`.

⁴ CSRF (Cross-site request forgery) je falsificiranje zahtjeva na Internet stranicama koji su na istom site-u i koji omogućava napadaču da navede korisnika da izvrši radnju koju ne namjerava. Mora postojati relevantno djelovanje (neko privilegirano djelovanja ili djelovanje na podacima korisnika, npr. promjena lozinke), rukovanje sesijom temeljeno na kolačićima (izvođenje djelovanja barem jednog HTTP zahtjeva a da se aplikacija oslanja samo na kolačiće seseije za identifikaciju korisnika koji je podnio zahtjev) i da nema nepredvidivih parametara zahtjeva (nešto što napadač ne može odrediti ili pogoditi).

The screenshot shows a browser window with the URL `localhost:8000/auth/login`. On the left, there is a login form with fields for Email and Password, and a Login button. Below the form are links for 'Don't have an account?' and 'Register'. On the right, the developer tools' Elements tab is open, displaying the HTML structure of the page. The HTML includes a header, a body containing an `<h1>Login</h1>`, a `<form method="POST" action="http://localhost:8000/auth/login">`, and an `<input type="hidden" name="_token" value="S3mv4x oNheJnt2WmeTXk3eu0fAet94heUDg9D2y1" autocomplete="off">`. The `<input type="email" name="email" id="email" value>` field is highlighted in blue, indicating it is selected.

Ako to ne stavimo `@csrf`, baca grešku `419 PAGE EXPIRED`. Iako je debuger upaljen, nema opisa greške. Dakle dovoljno je samo upaliti `@csrf`.

419 | PAGE EXPIRED

`@error('email')` provjerava postoji li greška validacije za polje email. Ako postoji greška, prikazuje se odgovarajuća greška u `$message`. Ugrađena metoda `old()` vrati vrijednost nakon validacije (za detalje pogledati [Ponovo popunjavanje forme](#)). `$request->old()` (u našem slučaju `old('email')`) se upravo koristi da vrati vrijednost nakon validacije u samo polje, tako da korisnik vidi što je unio. Vidjet će tada ako je bio tipfeler. Ovo nije za korištenje s passwordom jer ga ionako ne vidi. `<label for="password">Password</label>` je labela za polje lozinke. `<input type="password" name="password" id="password">` je ulazno polje za unos lozinke. `type="password"` sakriva znakove unosa (koristi se obično za lozinke). `name="password"` ime za backend obradu lozinke. `@error('password')` provjerava postoji li greška za polje `password` i prikazuje je ako postoji. `<button type="submit">Login</button>` je dugme za slanje forme. `<p>Don't have an account? Register</p>` prikazuje link za korisnike koji nemaju račun, vodi na `route('register')`, što je ruta prema stranici za registraciju.

Ostalo nam je da u `routes/web.php` kreiramo tu rutu.

Ispod postojećeg:

```
Route::get('auth/login', [AuthController::class, 'login'])->name('login');
```

Upišemo:

```
Route::post('auth/login', [AuthController::class, 'authenticate'])->name('authenticate');
```

Dakle može biti isto ime što je čest slučaj. Razlikuje se po metodi po kojoj se šalje zahtjev (engl. request). Ako imamo problema s tim možemo drugačije nazvati rute. Da ne bi zbunjivalo predavač je promijenio ime u `post.login`.

```
Route::post('auth/login', [AuthController::class, 'authenticate'])->name('post.login');
```

Ovo znači da nam treba metoda `authenticate` na kontroleru kako bi zaprimila zahtjev tj. request objekt u kojem se nalaze informacije koje su upisane u formu (obrazac) ili nisu (moramo ih validirati).

Imamo jednostavnu formu, imamo rutu na koju će kada se pritisne dugme Submit, action će preusmjeriti na `AuthController authenticate` i Laravel će s **dependency injection-om** ubaciti to u request objekt i ubaciti u tu metodu koju ćemo moći iskoristiti i nešto napraviti.

U `app/Controllers/AuthController.php` u login metodi idemo vratiti `view`:

```
public function login(Request $request)
{
    return view('login');
}
```

Sada ćemo napisati `authenticate` metodu koja će izvući sve podatke koji su došli iz forme:

```
public function authenticate(LoginRequest $request)
{
    return ($request->all());
}
```

Pitanje je smijemo mi to raditi – odgovor je ne, jer podaci nisu validirani. Druga stvar je da vidimo podatke koje ne bi smjeli vidjeti.

```
{
    "_token": "S3mv4xoNheJnt2WmeTXk3eu0fAet94heUDg9D2y1",
    "email": "branko@primjer.com",
    "password": "password"
}
```

Vidimo `_token` od `@csrf` koji je već razriješen.

Dakle trebamo ovo izmjeniti:

```
public function authenticate(LoginRequest $request)
{
    dd($request->get('email'));
}
```

```
"branko@primjer.com" // app\Http\Controllers\AuthController.php:27
```

Jedini ispravan način je `$request->validated` jer želimo samo validirane podatke iz request-a.

```
null // app\Http\Controllers\AuthController.php:27
```

Ovo znači da nijedan podatak nije validiran.

Na backend-u smo **UVIJEK DUŽNI IMPLEMENTIRATI VALIDACIJU PODATAKA**. Kada oni dođu na requestu, mi moramo pozvati ovaj `$request->validated`.

Da ponovimo imamo kontroler, imamo rutu, vidjeli smo kako forma pozove ime rute i onda se kreira URL, a na temlju toga se aktivira kontroler i metoda u njemu. Sljedeći korak je validacija podatka ako je sve u redu i nastavimo proces autentifikacije korisnika. Ako nešto nije u redu, validator će zaustaviti daljnje izvođenje i generirati greške, te napuniti error bag koji će nas onda vratiti (redirektat) na stranicu odakle smo došli ali će error bag biti napunjeno sa greškama koje ćemo moći dohvatiti i ispisati korisniku. To su direktive `@error` o kojima još nismo govorili.

Validacija i autentifikacija korisnika

Ako pogledamo `app\Http\Controllers\AuthController.php` koji smo napravili i metodu `login`:

```
public function login(Request $request)
{
    return view('login');
}
```

Nju smo svakako mogli napisati i ovako:

```
public function login(Request $request)
{
    return View::make('auth.login');
}
```

U gornjem primjeru koristili smo helper metodu `view()`. Nju smo mogli zamijeniti glavnim servisom a to je `View::make`, gdje uzme Blade i stvori PHP (u Blade-u postoje direktive koje PHP ne razumije) i onda ova heper metoda (funkcija) zamjenjuje servis. Ako nemamo dependency injection onda imamo fasadu i uvijek možemo pristupiti Laravelovim servisima.

Validacija podataka

Došli smo do dijela gdje smo pričali o validaciji podataka koji pristižu u kontroler. Laravel kreira `Request` objekt i praktično ga automatski gura kroz parametar metode kao argument kroz **dependency injection** imate pristup tom objektu ili kroz **dependency injection**. Vidjeli smo kada navedemo `$request->all()` možemo povući sve podatke koji su pristigli u requestu, u onom dijelu koji je zadužen za podatke, da li je to u form data obliku ili je to query string parametar, ovisi da li koristimo `POST` ili `GET`. Rekli smo da to baš i nije dobra praksa i da je zgodnije koristiti `$request->get` ali opet nismo te podatke validirali.

Ako pogledamo naš pogled [resources/views/login.blade.php](#), gdje smo rekli da upravo kroz direktive možemo pristupiti nekakvim validacijskim pogreškama tj. porukama koje možemo korisniku ispisati (npr. neispravan mail). Moguće je provjeriti da li taj email već postoji u bazi. Sve što se tiče autentifikacije (provjere identiteta) - ne treba javljati informacije tipa ovaj email ne postoji ili ovaj password nije ispravan. Dovoljno je javiti neispravni podaci za autentifikaciju, ostalo je problem korisnika, neka unese ispravno podatke. Ako npr. javimo korisniku da ima e-mail u sistemu, može koristiti brute force da nađe password itd.

Postoji više načina za validiranje podataka. Validacija (eng. validation) je proces provjere da li podaci koje je korisnik unio u formu ili koje aplikacija obrađuje ispunjavaju određene kriterije ili pravila. Cilj validacije je osigurati da su podaci ispravni i u odgovarajućem formatu prije nego što se dalje koriste ili pošalju na server. Svi podaci koje želimo imati kasnije za obradu, uvijek moraju proći validaciju. Ponekad je dovoljno da je obavezno unijeti podatke tj. da se polje ne može preskočiti ([email](#) i [password](#) su mandatory). Kontroler ih može dobiti kroz [\\$request](#) i pomoću svojstva [validate](#).

```
public function authenticate(LoginRequest $request)
{
    dd($request->validate);
}
```

Idemo vidjeti gdje ćemo napisati validaciju. Postoji više načina kako validirati podatke. Jedan način je direktno u kontroleru ([AuthController.php](#) kod nas), preko [\\$request->validate](#):

```
public function authenticate(LoginRequest $request)
{
    $request->validate([
        'email' => 'required|email',
        'password' => 'required'
    ]);
;
    dd($request->validate);
}
```

Predavač kaže da ovaj tip validacije ne koristi, već koristi [validaciju zahtjeva forme \(engl. form request\)](#). Preko njih možemo napraviti validaciju podataka. Možemo koristiti [FormRequest](#) umjesto [Request](#) koji ga ekstenda. Na [FormRequest](#) postoje dodatne metode i svojstva koja nam mogu poslužiti. Jedno od njih je [validate](#). PHP Storm ih ne razumije

```
public function authenticate(FormRequest $request)
{
    $request->validate([
        'email' => 'required|email',
        'password' => 'required'
    ]);
;
```

```
        dd($request->validate);
    }
```

`FormRequest` je nešto što nasljeđuje `Request`. Kada radimo custom Request, možemo naslijediti taj `FormRequest`. Unutra možemo definirati pravila validacije, gdje za svako polje u toj formi (atribut u requestu) možemo validirati po određenim pravilima. Druga stvar koju ovdje možemo raditi je da prije nego što dođe zahtjev za kontroler, jedan od mehanizama je da li ćemo dozvoliti da request prođe dalje. Postoji mehanizam kojim možemo provjeriti da li korisnik ima uopće pravo napraviti taj request. To je jedan od načina. Ima još middleware i policy koje još koristi.

`authorize` metoda je odgovorna za utvrđivanje može li trenutno autentificirani korisnik izvršiti radnju predstavljenu request-om, dok metoda rules vraća pravila validacije koja bi se trebala primijeniti na podatke zahtjeva.

Sve troje i middleware i policy i validacija zahtjeva forme (engl. form request) ima mogućnost da preusmjeri zahtjev. Middleware na samoj ruti, prije nego što dođe na kontroler može uhvatiti request i policy koji može implementirati unutar samog kontrolera, mogu određene blokove (engl. chunk) koda samo provjeriti (engl. check) s tim policy-jem i vidjeti ćemo gdje je to zgodno koristiti.

Da samo spomenemo, možemo dohvatiti validirane ulazne podatke s `$validated = $request->validated();` ili dohvatiti dio validiranih ulaznih podataka.

```
$validated = $request->safe()->only(['name', 'email']);
$validated = $request->safe()->except(['name', 'email']);
```

Više u dokumentaciji o [kreiranju requesta forme](#). Prvi slučaj je da ako od više polja trebamo samo dva polja. Nije problem ako izvučemo više polja koliko ako ih ne validiramo.

Možemo dodati [custom validatore](#) i raditi [dodatne validacije](#). Možemo prilagoditi poruke koje idu van, itd.

Predavač kaže da treba poštovati strukturu kontrolera i arhitekturu, držati se dobre prakse i ne ubacivati nepotrebno u kontroler. Poslovna logika nema što raditi u kontroleru, zahtjev treba prihvatiti i preusmjeriti u servis. Na servisu razvijati poslovnu logiku.

Pogledajmo što bi treba napraviti da možemo validirati podatke. Trebat ćemo vlastiti request. Nazvat ćemo ga `LoginRequest` pa ćemo pomoću Artisana automatski ekstendati potrebu klasu kako bi sve radio.:)

```
php artisan make:request LoginRequest
```

Ono što trenutno nemamo a dobili smo je direktorij `app/Http/Requests`. Svi request-ovi će se pohraniti u taj direktorij, osim ako drugačije ne navedemo (isto kao i pogledi i kontroleri). Ako pogledamo kod vidimo:

```
<?php
namespace App\Http\Requests;
```

```
use Illuminate\Foundation\Http\FormRequest;

class LoginRequest extends FormRequest
{
    /**
     * Utvrdi je li korisnik ovlašten za podnošenje ovog request-a.
     */
    public function authorize(): bool
    {
        return true;
    }

    /**
     * Dohvati pravila provjere validnosti koja se odnose na zahtjev.
     *
     * @return array<string,
    \Illuminate\Contracts\Validation\ValidationRule|array<mixed>|string>
     */
    public function rules(): array
    {
        return [
            //
        ];
    }
}
```

Klasa `LoginRequest` ekstenda `FormRequest`, koji ekstenda `Request`. Imamo `authorize()`, koji po default-u vraća `false`. Ako ovo ostavimo, svi request-ovi će biti odbijeni.

Ispod imamo validacijska pravila gdje imamo matricu sa pravilima koja želimo za naše atributе, to će biti za `email` i `password`.

Trebamo povezati naš request `FormRequest.php` s kontrolerom `AuthController.php`. Jednostavno, s dependency injection, u `authenticate` ćemo umjesto `FormRequest` reći ćemo da želimo koristiti `LoginRequest`.

```
public function authenticate(LoginRequest $request)
{
    //return $this->authService->authenticate($request);
}
```

Ne treba zaboraviti iznad dodati:

```
use App\Http\Requests\LoginRequest;
```

Sada u kontroleru umjesto `FormRequest` tj. običnog requesta koristimo svoj custom `LoginRequest`. Kada se preusmjeri request prema kontroleru, onda ćemo proći kroz `LoginRequest` i prvo što će se dogoditi, odbit će nas request zato što njegova metoda `authorize()` vraća `false`. Ona neće pustiti da request dalje prođe.

403 | THIS ACTION IS UNAUTHORIZED.

401 će se koristiti kao status kada korisnik nije autentificiran (nije prijavljen u sistem). To znači da bilo kakvu logiku možemo definirati u `app/Http/Requests/LoginRequests.php` u metodi `authorize()`. Evo primjera samo za demonstraciju:

```
public function authorize(): bool
{
    if (random_int(1,20) === 10) {
        return true;
    }
    return false;
}
```

Ako je slučajni broj 10 onda će pustiti korisnika da napravi request. Logika ne mora imati veze sa userom, kao u ovom slučaju. 403 Dva statusna koda koja se najčešće susreću su 401 (neovlašteno) i 403 (zabranjeno). Iako se na prvi pogled mogu činiti sličnim, služe različitim svrhama. Kada server vrati statusni kod 401, to zapravo znači: " Ne prepoznajem te ." Statusni kod 403 ima drugačiju implikaciju. Kaže klijentu: " Znam tko ste, ali vam nije dopušteno ovdje." Ostaviti ćemo ovako:

```
public function authorize(): bool
{
    return true;
}
```

Idemo popuniti što validiramo:

```
/**
 * Get the validation rules that apply to the request.
 *
 * @return array<string,
 \Illuminate\Contracts\Validation\ValidationRule|array<mixed>|string>
 */
public function rules(): array
{
    return [
        'email' => 'required|email',
        'password' => 'required',
    ];
}
```

Validiramo `email` i `password`, drugo ovdje nije potrebno validirati. Čak nismo morali tražiti validaciju `email`.

Idemo provjeriti kako radi. Pokrenut ćemo `http://localhost:8000/` i otići na Login link, te ostaviti prazan Email i Password i pritisnuti dugme Login. Desit će se ovo:

Login

Email

The email field is required.

Password

The password field is required.

Don't have an account? [Register](#)

Ispisao je ove poruke zato što je validator tj. request završio u našem `LoginRequest` gdje smo definirali pravila, tj. gdje smo rekli da `email` i `password` moraju biti uključeni. Kada se dogodila validacija, Laravel je generirao error objekt (zove se error tag) koji napuni neke informacije, a to su greške i možemo ih ispisati ili ne. Poruke validatora grešaka su ugrađene i mogu biti različite za različite jezike. Možemo definirati svoje poruke tj. prilagoditi ih:

```
/**
 * Get the error messages for the defined validation rules.
 *
 * @return array<string, string>
 */
public function messages(): array
{
    return [
        'email.required' => 'Email is required',
        'email.email' => 'Email is invalid',
        'password.required' => 'Password is required',
    ];
}
```

Ovo nije dobra praksa zbog višejezičnosti.

Zbog ovoga je bolje koristiti customizirani form request nego raditi validaciju unutar kontrolera i koristiti custom validaciju u samom kontroleru.

Ako sada upišemo u `app/Http/Controllers/AuthController.php` u metodu `authenticate`:

```
public function authenticate(LoginRequest $request)
```

```
{
    dd($request->validated());
}
```

```
array:2 [▼ // app\Http\Controllers\AuthController.php:26
  "email" => "test@test.com"
  "password" => "testtest"
]
```

Dobit ćemo `email` i `password` koje smo unijeli. Tako smo sigurni da smo dobili validirane podatke.

Idemo sada pogledati kako koristiti vlastite servise koje ćemo ubaciti u servisne kontejnere Laravela. Ti servisi ne moraju biti dostupni svuda. To će biti servisi za poslovnu logiku koju ćemo koristiti u kontroleru. Zato ćemo mu metodi `authenticate` reći samo:

```
public function authenticate(LoginRequest $request)
{
    return $this->authService->authenticate($request);
}
```

Dakle ovdje zaprima request koji dolazi kroz rutu i preusmjerava ga servisu na neku metodu. Poslovna logika na tom servisu što to mi trebamo napraviti. Ako pogledamo Artisan:

`php artisan` vidjet ćemo da nigdje nema servisa. Pojam servisa u Laravelu nije jedinstven. Servis je nešto što možemo imati u vidu sloja, kada pogledamo kontroler, servis, model, gdje ćemo mi biti taj koji će potencijalno komunicirati sa modelima i sa samom bazom.

Zna doći još jedan servis a to je repozitorij između sloja servisa i database test modela. Početnici obično sve ubace u kontroler ali bilo bi dobro odmah se naučiti dobroj praksi i ne stavljati u kontroler poslovnu logiku.

Stroga tipizacija je dobra. Kada imamo samouke probleme obično nisu dobro istražili dobre prakse a stroga tipizacija na to utiče. Stroga tipizacija tjera da se ispravno radi.

Servisi

Ručno ćemo napraviti direktorij `app/Services`. Sa kontrolera ide na servis pa kako nije dio `Http`, stavit ćemo ga odvojeno. Dakle, napraviti ćemo servis i preusmjeriti ga. Napisat ćemo `AuthService.php`.

```
<?php

namespace App\Services;

class AuthService
{
    public function authenticate(LoginRequest $request)
    {
    }
}
```

```
}
```

Ako pogledamo kako iz kontrolera `app/Http/Controllers/AuthController.php` šaljemo request i pošto se radi o `LoginRequest` Copilot je prepostavio da će to biti i u `AuthService.php`. Međutim, mi možemo umjesto `LoginRequest` koristiti `FormRequest`. To će raditi zato što `LoginRequest` ekstenda `FormRequest`. Ovdje nam ne trebaju validacijska pravila. Request se proslijedio iz `AuthController.php` dalje u `AuthService.php`. Ako ne želimo slati kompletan request na svoj servis, nego samo podatke, onda ćemo izvući podatke koje trebamo poslati.

U metodu u `AuthController.php` možemo izmijeniti metodu `authenticate()`:

```
public function authenticate(LoginRequest $request)
{
    'credentials' = $request->safe()->only('email', 'password');
    return $this->authService->authenticate($credentials);
}
```

Sada se ne moramo boriti sa `$request` nego šaljemo podatke koje želimo. U ovom slučaju `$credentials` umjesto `$request`.

Predavač kaže da on ne radi na kontroleru jer ne želi nikakvu logiku na kontroleru. Validator provjeri podatke i ako je sve u redu, request se proslijedi na servis `app/Services/AuthService.php`,

Na servisu `authenticate()` ćemo zaprimiti podatke iz tog requesta:

```
public function authenticate(FormRequest $request)
{
    return = $this->authService->authenticate($request);
}
```

S `$request->safe(['email', 'password'])` dohvaćamo samo validirane podatke iz `FormRequest` objekta koristeći `safe` metodu, koja osigurava da su samo navedena polja iz zahtjeva dostupna, čime se smanjuje rizik od ranjivosti kao što je **mass assignment vulnerability**.

Na rutu se dogodi preusmjeravanje na metodu, metoda odradi **dependency injection**, kroz parametar metode napuni request a taj request je onaj koji je prošao validaciju u `LoginRequest.php`. Tamo provjerimo da li postoji `email` i `request`. Laravel ima više načina kako rukovati s **dependency injection**. Jedan od njih je da u metodu `authenticate()` u `app/Http/Controllers/AuthController.php` ubacimo parametar kao parametar `AuthService $authService` i onda pozovemo taj `$authService`.

```
public function authenticate(FormRequest $request, AuthService $authService)
{
    return = $authService->authenticate($request);
}
```

Imamo pristup svome servisu i šaljemo mu request. U `authenticate()` metodi u `app/Services/AuthService.php`, zaprimamo taj request i servis izvači te podatke i treba ih zapisati.

Sada ovaj `FormRequest` ima pristup svim ovim informacijama i validator nam ne treba. `FormRequest` je ovdje bolje iskoristiv jer ako neko promjeni `LoginRequest` u `AuthController.php` u npr. `AuthRequest`, na našem servisu u `AuthService.php` naš `FormRequest` imat će pristup jer na našim servisima nismo vezani za original klasu, nego ovu koja je naslijeđena.

Pogledajmo što se desi ako u `AuthService.php` dodamo `dd($credentials)`:

```
public function authenticate(FormRequest $request)
{
    'credentials' = $request->safe(['email', 'password']);
    dd($credentials);
}
```

```
array:2 [▼ // app\Services\AuthService.php:18
  "email" => "a@10mail.org"
  "password" => "asdfasdf"
]
```

Nedostatak ovog načina vidjet ćemo u ovoj idućoj metodi u `AuthController.php`, `public function register` koja isto treba koristiti `AuthService`. Opet će morati gurati `AuthService` kroz parametar. Ako nam je zamorno da imamo više metoda u našem kontroleru **dependency injection** kroz parametar te metode, možemo koristiti konstruktor. Kada se učita kontroler, konstruktor automatski napuni svojstvo sa servisom. Kažemo u `AuthController.php`, odmah iznad definicije klase:

```
class AuthController extends Controller
{
    private $authService;

    public function __construct(AuthService $authService)
    {
        $this->authService = $authService;
    }

    public function authenticate(LoginRequest $request)
    {
        return $this->authService->authenticate($request);
    }
}
```

Vidimo definiran `__construct`. Konstruktor korz svoj parametar zaprima `AuthService` i napunit će ovo svojstvo. Sada svaki puta u našoj metodi umjesto da imamo u parametru `AuthService $authService` kao injection, on se već dogodio gore. Dovoljno je reći samo u metodi `authenticate`:

```
return $this->authService->authenticate($request);
```

Sada ne moramo u kontroleru u našim metodama, kada gradimo novu metodu, dodati još taj parametar našeg servisa da bi se dogodio injection. Dependency injection se već dogodio. Predavač preporuča ovakav pristup. Kaže da je ovo praktičnije.

Kad se instancira klasa u objekt, aktivira se konstruktor. Kada pozivamo bilo koju metodu dobivamo objekt.

Naš kontroler je ovisan o servisu i ne može raditi bez servisa. Najčešće se radi s ugovorm (engl. contract). Contract je interface. Potpisanim ugovorom vežu se servis i interface. Dakle u konstruktoru bi koristili `AuthServiceInterface`. Sada možemo bilo kojoj drugoj klasi reći implementiraj `AuthServiceInterface` i praktički ga možemo koristiti u konstruktu. Kada smo vezani uz interface tj. `_contract`, onda možemo mijenjati za razliku kada smo vezani uz klasu. O contract-ima ćemo govoriti kasnije.

Idemo završiti autentifikaciju s metodom `authenticate()`. Nakon što smo dobili potrebne i validirane podatke iz request-a, sada ćemo ispitati da li možemo korisnika autentificirati.

```
public function authenticate(FormRequest $request)
{
    $credentials = $request->safe(['email', 'password']);

    if (Auth::attempt($credentials)) {
        $request->session()->regenerate();

        return redirect()->intended(route('dashboard'));
    }

    return back()->withErrors([
        'credentials' => 'The provided credentials do not match our records.',
    ]);
}
```

Metoda `Auth::attempt($credentials)` pokušava autentificirati korisnika koristeći unešen `$credentials`. Ako `$credentials` odgovaraju korisniku u bazi, vraća `true`.

Nakon uspješne prijave, s `$request->session()->regenerate()` regeneriramo sesiju kako bi se zaštitio od session fixation napada. Nakon što se korisnik prijavi (tj. nakon autentifikacije), aplikacija treba regenerirati sesijski ID kako bi prekinula vezu između prethodnog sesijskog ID-a (koji je mogao biti kontroliran od strane napadača) i novog, sigurnog ID-a. To osigurava da zlonamjerni korisnici ne mogu koristiti iste sesijske ID-e nakon što se korisnik prijavi.

U redu s `return redirect()->intended(route('dashboard'))` koristimo `intended` kako bi se korisnik preusmjerio na stranicu na kojoj je prethodno htio pristupiti prije prijave. Za sada nemamo taj `dashboard`.

Ako autentifikacija nije uspješna s `return back()->withErrors()`, vraćamo korisnika natrag na formu za prijavu s porukom o grešci. Napravili smo poruku koju treba prikazati na view-u `resource/views/login.blade.php`. Dodat ćemo ovaj kod na početku:

```
@error('credentials')
<p>{{ $message }}</p>
```

```
@enderror
```

Dodat ćemo još dio na dnu koji pita korisnika ima li račun i ako nema da može kliknuti na link.

```
<p>Don't have an account? <a href="{{ route('register') }}>Register</a></p>
```

Kada neko pokuša pristupiti zaštićenom dijelu aplikacije a nije autentificiran, bit će redirektan na login stranicu. Onda ako on nema račun, ima dole ovaj dio sa `register`. Ako ovo sad pokrenemo, dojavit će grešku jer nemamo rutu `register`. Da smo rekli `/register` onda bi radilo ali ovo je bolje za validaciju rute. Idemo u `routes/web.php`.

```
Route::get('register', [AuthController::class, 'showRegister'])->name('register');
```

Napisali smo `showRegister` jer ćemo `register` ostaviti za `post.register`. Ruta sada postoji. Međutim ako kliknemo na dugme Register, dobit ćemo grešku jer nemamo metodu.

U `resources/views/login.blade.php` imamo formu, u `app/Controllers/AuthController.php` imamo kontroler kojim se šalju podaci, servis `app/Services/AuthService.php` koji će ih obraditi.

Idemo vidjeti da li će se javiti greška nakon što unesemo krive podatke.

Login

The provided credentials do not match our records.

Email

Password

Don't have an account? [Register](#)

Idemo sada završiti registraciju korisnika. Postoji mogućnost da ne aktiviramo korisnika odmah dok ne potvrdi svoju adresu. Tada šaljemo neki kod na njegovu adresu u linku. Laravel dolazi s ugrađenim sistemom za verifikaciju email adresa putem `email/verify` mehanizma (korisnički model mora implementirati `MustVerifyEmail`).

U `routes/web.php` dodat ćemo post rutu:

```
Route::post('register', [AuthController::class, 'register'])->name('post.register');
```

U kontroleru `app/Http/Controllers/AuthController.php` ubacit ćemo metodu `showRegister()` i `register()`:

```
public function showRegister()
{
```

```

        return view('register');
    }

    public function register(RegisterRequest $request)
    {
        return $this->authService->register($request);
    }

```

Treba nam pogled `resources/view/register.blade.php` jer ćemo imati formu. Ručno ćemo kreirati:

```

@extends('layout', ['admin' => false])

@section('title', 'Register')

@section('content')
    <h1>Register</h1>
    @error(['credentials', 'error'])
        <p>{{ $message }}</p>
    @enderror
    <form method="POST" action="{{ route('post.register') }}">
        @csrf
        <div>
            <label for="name">First Name</label>
            <input type="text" name="first_name" id="first_name" value="{{ old('first_name') }}">
            @error('first_name')
                <p>{{ $message }}</p>
            @enderror
        </div>
        <div>
            <label for="name">Last Name</label>
            <input type="text" name="last_name" id="last_name" value="{{ old('last_name') }}">
            @error('last_name')
                <p>{{ $message }}</p>
            @enderror
        </div>
        <div>
            <label for="email">Email</label>
            <input type="email" name="email" id="email" value="{{ old('email') }}">
            @error('email')
                <p>{{ $message }}</p>
            @enderror
        </div>

```

```

<div>
    <label for="password">Password</label>
    <input type="password" name="password" id="password">
    @error('password')
        <p>{{ $message }}</p>
    @enderror
</div>
<div>
    <label for="password_confirmation">Confirm Password</label>
    <input type="password" name="password_confirmation"
id="password_confirmation">
    @error('password_confirmation')
        <p>{{ $message }}</p>
    @enderror
</div>
<button type="submit">Register</button>
</form>
<p>Already have an account? <a href="{{ route('showLogin') }}>Login</a></p>
@endsection

```

Linija `@extends('layout', ['admin' => false])` koristi Blade-ovu `@extends` direktivu kako bi proširila osnovni `layout.blade.php` (koji je smješten u `resources/views/layout.blade.php`). U ovom slučaju `layout.blade.php` koristi se kao temelj na koji se nadograđuje sadržaj specifičan za registraciju. Također, prenosi se varijabla `'admin' => false`, koja može biti korištena unutar layouta za prikaz različitih dijelova ovisno o statusu korisnika (običan korisnik naspram administratora).

`@section('title', 'Register')` definira naslov stranice. Blade koristi `@section` direktivu da bi ubacio specifičan sadržaj unutar sekcije definirane u rasporedu. U ovom slučaju, naslov je `'Register'`.

Sadržaj stranice za registraciju je unutar `@section('content')`. Sekcija contentće biti umetnuta u mjesto predviđeno za sadržaj unutar `layout.blade.php`.

```

@error(['credentials', 'error'])
    <p>{{ $message }}</p>
@enderror

```

Ovaj kod koristi `@error` direktivu da bi prikazao grešku povezanu s ključevima `'credentials'` i `'error'`. Ovi ključevi se mogu koristiti za prikaz grešaka kao što je neuspješna autentifikacija ili drugih grešaka prilikom registracije.

Ako je prisutna bilo kakva greška u validaciji za zadane ključeve, prikazuje se poruka greške unutar `<p>` tog.

U formi za registraciju `<form method="POST" action="{{ route('post.register') }}">` `method="POST"` postavlja metodu HTTP zahtjeva na `POST`, što znači da će podaci iz forme biti poslani serveru za obradu. `action="{{ route('post.register') }}"` koristi `route()` helper kako bi dinamički generirao URL rute koja obrađuje POST zahtjev za registraciju. Ova ruta vodi do kontrolera

koji obrađuje podatke i kreira korisnika. `@csrf` je Blade-ova direktiva koja generira CSRF token kako bi se spriječili CSRF (Cross-Site Request Forgery) napadi.

Pogledajmo polja unutar forme.

```
<div>
    <label for="name">First Name</label>
    <input type="text" name="first_name" id="first_name" value="{{ old('first_name') }}>
    @error('first_name')
        <p>{{ $message }}</p>
    @enderror
</div>
```

Ovaj blok koda definira polje za unos imena korisnika.

`value="{{ old('first_name') }}"` koristi `old()` funkciju koja automatski popunjava vrijednost koju je korisnik prethodno unio u slučaju da dođe do greške, kako bi korisnik video što je ranije unio. `@error('first_name')` prikazuje grešku ako postoji validacijska greška povezana s unosom imena.

Slično je sa unosom prezimena i email-a. Kreira se polje za unos email adrese. Tip polja je email, što znači da HTML browser očekuje ispravnu adresu e-pošte. `old('email')` ponovno popunjava polje s prethodno unesenom email adresom ako se dogodi greška. `@error('email')` prikazuje grešku povezanu s poljem e-pošte.

Lozinka i potvrda lozinke:

```
<div>
    <label for="password">Password</label>
    <input type="password" name="password" id="password">
    @error('password')
        <p>{{ $message }}</p>
    @enderror
</div>
```

Polje za unos lozinke koristi tip `password` kako bi unos lozinke bio skriven. `@error('password')` prikazuje grešku ako validacija lozinke ne uspije.

```
<div>
    <label for="password_confirmation">Confirm Password</label>
    <input type="password" name="password_confirmation" id="password_confirmation">
    @error('password_confirmation')
        <p>{{ $message }}</p>
    @enderror
</div>
```

Potvrda lozinke osigurava da korisnik unese lozinku dva puta kako bi se spriječile greške prilikom unosa lozinke.

Dugme za slanje forme definiran je s `type="submit"`. Na kraju postoji i link koji vodi na stranicu za prijavu. Taj link koristi rutu `route('showLogin')` koja vodi do login stranice.

Na `resources/view/welcome.blade.php` dodali smo:

```
<a href="{{ route('register') }}>Register</a>
```

Osmi linka koji vodi na stranicu za prijavu, koristeći `route('showLogin')`, dodali smo link vodi na `route('register')`.

U `routes/web.php` imamo rutu koja vodi na kontroler `app/Http/Controller/AuthController.php`, i s nje na `resources/views/register.blade.php`. Zatim imamo `register()` u njemu. Ako pogledamo ponovo rutu `routes/web.php` i `register`, koji takođe vodi u `app/Http/Controller/AuthController.php` i na servis `register()` metodu i šalje `$request`. Ostalo je na `app/Services/AuthService.php` dodati register. Morat ćemo dodati validator kako bi mogli validirati ulazne podatke. Napraviti ćemo metodu `register` na servisu u `app/Services/AuthService.php`:

```
public function register(FormRequest $request)
{
}
```

Sada će se na kontroleru `app/Http/Controller/AuthController.php` maknuti greška (podvlačenje crvenim). Vratimo se u `app/Services/AuthService.php` gdje nam ostaje kreirati request kao i login. Želimo raditi validaciju iz requesta doći do podataka koji nisu validirani. To ćemo napraviti uz pomoć Artisana:

```
php artisan make:request RegisterRequest
```

Laravel će kreirati novu PHP datoteku `RegisterRequest.php` unutar direktorija `app/Http/Requests`. U klasi `RegisterRequest` možemo definirati pravila validacije unutar metode `rules()`. U formi od podataka koje šaljemo imamo ime, prezime, email, password i potvrdu passworda. Username i password su stringovi a max. dužina je 255. Trebamo provjeriti što smo definirali u bazi, što znači da moramo pogledati migraciju `xxx_change_users_table.php`. Tamo smo dodali telefon, koji nam ne treba. Trebamo pogledati migraciju `xxx_create_users_table.php`. Tamo imamo `first_name`, `last_name`, `email` koji je `unique()`, `email_verified_at` koji je `nullable()` (dakle može biti i nedefinirana vrijednost) i `password`. Podaci koji se šalju iz forme uvijek su string.

Kod registracije ne treba tražiti od korisnika previše informacija. Najjednostavnije je username i password a za daljnje bodove, bonuse i sl. korisnik mora popuniti više podataka (telefon, ime, datum rođenja).

Idemo otvoriti `app/Http/Requests/RegisterRequest.php` koji smo kreirali. Idemo popuniti `rules()`.

```
<?php

namespace App\Http\Requests;
```

```

use Illuminate\Foundation\Http\FormRequest;

class RegisterRequest extends FormRequest
{
    /**
     * Determine if the user is authorized to make this request.
     */
    public function authorize(): bool
    {
        return true;
    }

    /**
     * Get the validation rules that apply to the request.
     *
     * @return array<string,
\Illuminate\Contracts\Validation\ValidationRule|array<mixed>|string>
     */
    public function rules(): array
    {
        return [
            'first_name' => ['required', 'string', 'max:255'],
            'last_name' => ['required', 'string', 'max:255'],
            'email' => ['required', 'email', 'max:255', 'unique:users'],
            'password' => ['required', 'string', 'min:8', 'confirmed'],
        ];
    }
}

```

'unique:users' provjerava u tablici users da li ima takvog maila. 'password' mora biti 'confirmed'. Ovo je bitno zbog pogleda register.blade.php. Polje `password` i `password_confirmation` su polje `password`. U validatoru možemo vidjeti ako validiramo polje password, koje treba biti potvrđeno, očekuje polje `password_confirmation` i vrijednost da bude ista kao u polju password. Sve te validacije možemo vidjeti u [dokumentaciji](#). Ako se polje zove `password` onda polje za potvrdu se mora zвати `password_confirmation`. Na taj način će validator znati što treba usporediti.

Možemo zaključiti da je završen validator. Ako je to u redu, tek tada će validator pustiti na kontroler i preusmjeriti na naš servis.

Sljedeći korak je na kontroleru poslati naš validator tj. naš `RegisterRequest`. Ispravit ćemo `register()` metodu u `app/Http/Controllers/AuthController.php`:

```

public function register(RegisterRequest $request)
{
    return $this->authService->register($request);
}

```

Ako izaberemo iz menija `RegisterRequest` onda Visual Studio Code sam iznad na potrebno mjesto ubaci:

```
use App\Http\Requests\RegisterRequest;
```

Sada smo sigurni da ćemo u servisu dobiti validirane podatke koje možemo koristiti za daljnju registraciju. Tu bi mogli testirati da li validator radi i sve probati.

Probat ćemo unijeti neki mail koji već postoji u bazi:

Register

First Name

Last Name

Email

The email has already been taken.

Password

Confirm Password

Already have an account? [Login](#)

Vidimo da validacija dobro radi. Greška „The email has already been taken.“ ukazuje na to. Neki koriste username koji mora biti jedinstven a mail je popratna informacija. Opet je nezgodno ako imamo više istih mailova. Trebamo još provjeriti ako ne unesmo isti password:

Register

First Name

Last Name

Email

Password

The password field confirmation does not match.

Confirm Password

Already have an account? [Login](#)

Vidimo da password i njegova potvrda se obrišu i ne ostaju, dok ime, prezime i email ostaju. Dakle i to radi.

Ostala nam je još registracija logike korisnika. Idemo u `app/Services/AuthService.php`. U praznu metodu register staviti ćemo `dd` i pogledati što je u `$request->validate()`.

```
public function register(FormRequest $request)
{
    dd($request->validated());
}
```

```
array:4 [▼ // app\Services\AuthService.php:32
  "first_name" => "Pero"
  "last_name" => "Perić"
  "email" => "juliana.abernath@example.net"
  "password" => "password"
]
```

Mogli smo koristiti i `dd($request->validated());` ali to nam ne treba:

```
array:6 [▼ // app\Services\AuthService.php:32
  "_token" => "uBTACxU4wIDyJkappFF3xJQC46pE2tQJ3VIykyln3"
  "first_name" => "Pero"
  "last_name" => "Perić"
  "email" => "juliana.abernath@example.net"
  "password" => "password"
  "password_confirmation" => "password"
]
```

```
public function register(FormRequest $request)
{
    $data = $request->safe(['first_name', 'last_name', 'email', 'password']);
    $user = User::create($data);
    Auth::login($user);

    return redirect()->intended(route('dashboard'));

}
```

Ovdje se iz zahtjeva uzimaju samo polja koja su navedena: `first_name`, `last_name`, `email` i `password`.

U `$user = User::create($data);` se koristi model `User` za kreiranje novog korisnika u bazi podataka s podacima iz zahtjeva. Polja koja se kreiraju su ona prikupljena u prethodnom koraku.

Novi korisnik se odmah prijavljuje pomoću `Auth::login($user)`.

Korisnik se nakon registracije preusmjerava na željenu stranicu (u ovom slučaju na `dashboard`) s `return redirect()->intended(route('dashboard'));`

Na `routes/web.php` ćemo hardkodirati dashboard.

```
Route::get('dashboard', function () {
    dd(auth->user());
})->middleware('auth')->name('dashboard');
```

Kasnije ćemo definirati kontroler. Nismo vratili `view('dashboard')`; s `return`, nego ispisali `dd(auth->user())`; autentificiranog user-a.

Ako probamo ručno doći na stranu `http://localhost:8000/dashboard` ispisat će `null`, jer niti jedan user nije prijavljen. Želimo se logirati da unesemo username i password od korisnika u bazi, da authenticate kontroler to proslijedi servisu autentifikaciju i preusmjeri na dashboard. Na dashboardu bi trebali dobiti model user-a koji je prijavljen.

```
App\Models\User {#1235 ▶ // routes\web.php:29
    #connection: "mysql"
    #table: "users"
    #primaryKey: "id"
    #keyType: "int"
    +incrementing: true
    #with: []
    #withCount: []
    +preventsLazyLoading: false
    #perPage: 15
    +exists: true
    +wasRecentlyCreated: false
    #escapeWhenCastingToString: false
    #attributes: array:11 [▼
        "id" => 21
        "first_name" => "Jarred"
        "last_name" => "Carroll"
        "email" => "juliana.abernathy@example.net"
        "email_verified_at" => "2024-09-24 07:59:38"
        "password" => "$2y$12$W38TvedJQsW.ur0hvfvNVguNcI/3CH1.xqgCM3K0bYzibG1MK.dxma"
        "remember_token" => "AZmU3kBMyqy4eisGuLLZsLuo6yennKpnR20RUlgNsK3o8TFHegRWIYfCy3AW"
        "created_at" => "2024-09-24 07:59:43"
        "updated_at" => "2024-09-24 07:59:43"
        "deleted_at" => null
        "phone" => null
    ]
    #original: array:11 [▶]
    #changes: []
    #casts: array:2 [▶]
    #classCastCache: []
    #attributeCastCache: []
    #dateFormat: null
    #appends: []
    #dispatchesEvents: []
    #observables: []
    #relations: []
    #touches: []
    +timestamps: true
    +usesUniqueIds: false
    #hidden: array:2 [▶]
    #visible: []
    #fillable: array:5 [▶]
    #guarded: array:1 [▶]
    #authPasswordName: "password"
    #rememberTokenName: "remember_token"
}
```

Možemo zaključiti da login radi. Register isto tako radi bez problema.

Vidimo user-a jer register servis je pokupio validirane podatke, zapisao u bazu novog user-a koristeći user model i create metodu koja se oslanja na fillable attribute (mass assignment). Nakon što je `create` kreirao novog usera, izbacio nam je te podatke van. Mi smo na temelju tih podataka i ugrađenog outh servisa prijavili korisnika.

Ako pogledamo u browseru `Inspect>Application>Cookies` možemo vidjeti da `laravel_session`. Upravo taj session osigurava da je korisnik stalno autoriziran

| Name | Value | D... | P... | Ex... | Size | H... | Se... | S... | P... | Cr... | Pr... |
|------------------------|--------------------------------|-------|------|-------|------|------|-------|------|------|-------|-------|
| PHPSESSID | ti31od8nnc9k5392v8dc... | lo... | / | S... | 35 | | | | | | M... |
| XSRF-TOKEN | eyJpdil6ljdaVmM4SzFhZ... | lo... | / | 2... | 352 | | | | Lax | | M... |
| laravel_session | eyJpdil6lkVIODdiTDRF... | lo... | / | 2... | 357 | ✓ | | | Lax | | M... |

Ako ga obrišemo i refreshamo stranicu vidjet ćemo `null` jer više nismo logirani. Taj session se kreirao kod `app/Services/AuthService.php` i u njemu metoda `authenticate()`:

```
$request->session()->generate();
```

Ista priča je i kad se dogodio login:

```
Auth::login($user);
```

U principu flow je vrlo jednostavan. Ruta `web.php`, kontroler `AuthController.php`, servis `AuthService.php` koji onda koristi ili ne modele. Na kraju imamo poglede `welcome.blade.php`.

Middleware

Middleware u Laravelu je sloj logike između HTTP zahtjeva i odgovora koji omogućava filtriranje i manipuliranje zahtjeva prema aplikaciji. Možete ga zamisliti kao vrstu „čuvara“ (engl. guard) koji provjerava ili modificira zahtjeve prije nego što stignu do kontrolera. Na primjer, middleware može provjeriti je li korisnik prijavljen, ima li određene dozvole ili prilagoditi zaglavla (headers).

Primjer korištenja middleware-a je ograničavanje pristupa određenoj ruti samo prijavljenim korisnicima, postavili biste middleware koji provjerava autentifikaciju. Middleware može bilo što provjeravati u requestu, može promijeniti bilo što jer je to ponekad potrebno. Npr. možemo provjeriti određene stvari, modifikaciju requesta iz ERP sistema koji nije usklađen. Na middleware-u presretнемo

request i prilgodimo ga našem modelu, nabindamo ga na naš model i pustimo ga dalje prema kontroleru. Potencijalno tako možemo raditi s više ERP sistema.

Osnovna struktura middleware-a

Middleware se obično nalazi u direktoriju `app/Http/Middleware/`. Laravel dolazi s nekoliko ugrađenih middleware-a, kao što su `auth`, `throttle` i drugih.

U Laravelovom middleware `'auth'` je izgrađena logiku gdje očekuje da imamo rutu `login`. To je još jedan razlog zašto korisiti ime rute. Ako nemate ime rute, neće raditi ovaj redirektor, gdje štitimo rutu. Naravno to se može promjeniti.

Možemo reći da je nedostatak ugrađenih middleware-a je da moramo poštivati naming konvenciju. Kada dođemo na našu stranicu i pritisnemo Login vidjeli smo da na `routes/web.php` imamo middleware koji kaže `'guest'`:

```
Route::get('auth/login', [AuthController::class, 'login'])->name('showLogin')->middleware('guest');
```

Middleware provjeri da li smo možda prijavljeni u sistem, ako jesmo on će napraviti redirekciju u `dashboard`:

```
Route::get('dashboard', function () {
    return view('admin.dashboard');
    dd(auth()->user());
})->middleware('auth')->name('dashboard');
```

Pokušamo li doći na `localhost:8000/auth/login` nakon logiranja sistem će nas automatski preusmjeriti na `localhost:8000/dashboard`. Middleware će u handleru pitati da li je korisnik koji je napravio request prijavljen, ako jeste preusmjerit će nas na `dashboard`.

Ako nemamo dashboard rutu, nego npr. imamo greškom upisano gore `name('dashboardd')`, tada ako probamo na `localhost:8000/auth/login` opet ćemo biti preusmjereni na `dashboard`. Očito ne ne ide na rutu nego na URL, što je u redu. Ako promjenimo URL u `localhost:8000/auth/login` vratiće nas na početnu stranu i kada god pritisnemo Login stalno nas vraća na početnu stranu (jer smo logirani i ne zna gdje je `dashboard`).

Ako kao neprijavljeni korisnik pokušamo na `dashboard`, on nas vraća na login. Ruta se mora zvati login.

Ako želimo drugaćiji middleware možemo napisati vlastiti middleware, promijeniti njegovu logiku i nadvladati (engl. override) middleware. Nekada je nezgodno što i kako nadvladati. Ono što nikako ne bi smjeli dirati nikako je u direktoriju `vendor`. Ako napravimo update, pregazit ćemo promjenu koda. Nadvladavanje middlewarea mora biti u našem direktoriju. Predavač će pokazati kako napisati vlastiti middleware. Nećemo nadvladati `'auth'` i `'guest'` iz `web.php` već ćemo ih registrirati kao `'custom_auth'` i `'custom_guest'` ili tako nekako i moći ćemo ih koristiti nad postojećima.

Ono što još nismo napravili je u servisu `app/Services/AuthService.php` kod registracije iz requesta smo izvukli podatke, kreirali smo user-a, logirali smo usera i redirektali na `dashboard`. Svaki korisnik mora imati dodijeljenu rolu. Kada smo gradili bazu rekli smo da svaki korisnik ima ulogu. Ta uloga će određivati kasnije smije što vidjeti. Možemo pustiti korisnika koji se prijavio u sistem na temelju toga da middleware to dopusti jer `middleware('auth')` ne pita koju rolu ima.

Ono što middleware još ima je da middleware može definirati i tip role. Naravno pitanje je kako smo složili sistem. Možemo npr. pitati ima li user admin ulogu `middleware('auth:admin')`. Kako je admin uloga i gdje je definirana. Ako nismo korisniku prilikom registracije dodijelili niti jednu ulogu a štitimo naše rute i kasnije resurse uz policy-ja, izbacivat će grešku da nemamo prava. Dakle trebamo dodijeliti ulogu kada se korisnik registira. Postavlja se pitanje gdje je ta registracija. Da li je to registracija koju radi admin ili je to registracija koja je public i bilo koji korisnik se sam registrira. Kod naše aplikacije je ovaj drugi slučaj. Ne možemo dozvoliti da korisnik bira sam ulogu već mora dobiti ulogu User. Osim toga u tablici `roles` imamo i Admin i Author. Kako ćemo doći do te uloge i gdje ćemo je kreirati? Prije nego što prijavimo user-a u sistem, želimo doći do uloge i dodijeliti ulogu samom korisniku uz pomoć modela iz baze rolu `Role::where('name', 'User')->first()->getKey();` Ne može biti više od jedne. Idemo pričati o Eloquentu. Idemo umjesto `first()` reći `get()`.

```
public function register(Request $request)
{
    try {
        $data = $request->safe(['first_name', 'last_name', 'email',
'password']);

        $user = User::create($data);
        $role =
            Role::where(
                'name',
                Roles::USER->value
            )
            ->first()
            ->getKey();

        $user->roles()->attach($role);

        Auth::login($user);

        return redirect()->intended(route('dashboard'));
    } catch (\Exception $e) {
        Log::error($e);
        return back()->withErrors([
            'error' => 'An error occurred while registering the user.',
        ]);
    }
}
```

Napravit ćemo rutu test u `routes/web.php` gdje ćemo vrtiti testne stvari, kako ne bi morali paliti i gasiti druge servise:

```
Route::get('test', function () {
    $role = role::where('name', 'User')->get();
    dd($role );
});
```

Metoda `get()` dohvaca sve rezultate upita i vraća ih kao kolekciju modela. Ako odemo na test stranicu to ćemo i vidjeti. `get()` ne zna da je npr. name unique polje. Ako očekujete više rezultata i želite ih sve dohvatiti, koristite `get()`. Ako iza `get()`, stavimo `first()` dobit ćemo iz kolekcije prvu stavku (engl. item) tj. model. Zato ćemo ubaciti `first()`. Dakle metoda `first()` dohvaca prvi rezultat upita, te ga vraća kao jedan model (objekt) ili `null` ako rezultat ne postoji. `getKey()` izvlači ID.

```
Route::get('test', function () {
    $role = Role:: where('name', 'User')->first()->getKey();
    dd($role );
});
```

```
App\Models\Role {#1229 ▼ // routes\web.php:38
  #connection: "mysql"
  #table: "roles"
  #primaryKey: "id"
  #keyType: "int"
  +incrementing: true
  #with: []
  #withCount: []
  +preventsLazyLoading: false
  #perPage: 15
  +exists: true
  +wasRecentlyCreated: false
  #escapeWhenCastingToString: false
  #attributes: array:1 [▼
    "name" => "User"
  ]
  #original: array:1 [▶]
  #changes: []
  #casts: []
  #classCastCache: []
  #attributeCastCache: []
  #dateFormat: null
  #appends: []
  #dispatchesEvents: []
  #observables: []
  #relations: []
  #touches: []
  +timestamps: true
  +usesUniqueIds: false
  #hidden: []
  #visible: []
  #fillable: array:2 [▼
    0 => "name"
    1 => "permissions"
  ]
  #guarded: array:1 [▼
    0 => "*"
  ]
}
```

Vraćamo se u `app/Service/AuthService` i promijenit ćemo iz `get()` u `Role::where('name', 'User')->first();`. Korisnik se povezuje s određenom ulogom pomoću `attach` metode na `roles` vezi (definiranoj kao relacija "many-to-many" u modelu `User`). Jednostavno vezujemo dva modela jedan uz drugog u pivot tablici, jedan je user kojeg smo kreirali i rolu koju smo dobili tako da smo željenu rolu izvukli i napravili `attach`.

Dodali smo u biti samo dva reda u `AuthService.php` na početku pripreme.

```
$role = Role::where('name', 'User')->first()->getKey();
$user->roles()->attach($role);
```

Nakon ovoga idemo sa `login`. Na svakom modelu kada ga dobijemo (ne bitno je ako je on ključ) moramo navesti što želimo, u našem slučaju `getKey()` vrati primarni ključ jer inače dobijemo cijeli model.

Ako pogledamo zadnjeg korisnika u tablici `user_roles` nemamo informaciju koje role pripadaju tom zadnjem useru jer ih nismo dodjeli kod upisa korisnika. Registrirat ćemo novog korisnika i vidjeti da li će mu biti dodijeljena rola.

Registrirat ćemo novog korisnika i vidjeti da li mu je dodijeljena rola.

| <code>id</code> | <code>first_name</code> | <code>last_name</code> | <code>email</code> | <code>email_verified_at</code> | <code>password</code> | <code>remember_token</code> | <code>created_at</code> | <code>updated_at</code> | <code>deleted_at</code> | <code>phone</code> |
|-----------------|-------------------------|------------------------|-------------------------------|--------------------------------|-------------------------------------------------------------------------------------|-----------------------------------------|-------------------------|-------------------------|-------------------------|--------------------|
| 21 | Jarred | Carroll | juliana.abernathy@example.net | 2024-09-24 07:59:38 | \$2y\$12\$W38TvedJQsW.uOHvfvNgUNcl/3CHi.x...4hlbQm8yeeflwLBfmeR7yCeATv7eOdtE964A... | 4hlbQm8yeeflwLBfmeR7yCeATv7eOdtE964A... | 2024-09-24 07:59:43 | 2024-09-24 07:59:43 | NULL | NULL |
| 22 | Davin | Paucek | armstrong.janis@example.com | 2024-09-24 07:59:39 | \$2y\$12\$epBkP71AdeW35ImoNq8X5GeGCCOF0...j3WAmLPz | j3WAmLPz | 2024-09-24 07:59:43 | 2024-09-24 07:59:43 | NULL | NULL |
| 23 | Martina | Rath | leora77@example.com | 2024-09-24 07:59:39 | \$2y\$12\$pfIgcYuzMLAMyVnKloCWEcG6eaOZ...KhdfQJFzg7w | KhdfQJFzg7w | 2024-09-24 07:59:43 | 2024-09-24 07:59:43 | NULL | NULL |
| 24 | Corene | Nader | friedrich37@example.com | 2024-09-24 07:59:40 | \$2y\$12\$LNM2oQspQcn8AYokj0g.ee4\$9khlmxM...q3HFBMd8 | q3HFBMd8 | 2024-09-24 07:59:43 | 2024-09-24 07:59:43 | NULL | NULL |
| 25 | Octavia | Krajcik | bqibson@example.net | 2024-09-24 07:59:40 | \$2y\$12\$yrlOT66uNBn92e15zuDOOFk.Y0z...kohACoCx2d | kohACoCx2d | 2024-09-24 07:59:43 | 2024-09-24 07:59:43 | NULL | NULL |
| 32 | Ana | Anić | a@primjer.com | NULL | \$2y\$12\$wIGjq50PAetnXTonV5gELB3e7PrOb3...NULL | NULL | 2024-10-02 14:48:17 | 2024-10-02 14:48:17 | NULL | NULL |

| <code>id</code> | <code>first_name</code> | <code>last_name</code> | <code>email</code> |
|-----------------|-------------------------|------------------------|-------------------------------|
| 21 | Jarred | Carroll | juliana.abernathy@example.net |
| 22 | Davin | Paucek | armstrong.janis@example.com |
| 23 | Martina | Rath | leora77@example.com |
| 24 | Corene | Nader | friedrich37@example.com |
| 25 | Octavia | Krajcik | bqibson@example.net |
| 32 | Ana | Anić | a@primjer.com |
| NULL | NULL | NULL | NULL |

| role_id | user_id | created_at | updated_at |
|---------|---------|------------|------------|
| 7 | 21 | NULL | NULL |
| 7 | 22 | NULL | NULL |
| 7 | 24 | NULL | NULL |
| 7 | 26 | NULL | NULL |
| 8 | 28 | NULL | NULL |
| 8 | 29 | NULL | NULL |
| 8 | 30 | NULL | NULL |
| 8 | 31 | NULL | NULL |
| 8 | 32 | NULL | NULL |
| 9 | 23 | NULL | NULL |
| 9 | 25 | NULL | NULL |
| 9 | 27 | NULL | NULL |
| NONE | NONE | NONE | NONE |

Vidimo da radi, što znači da je dovoljno predati samo model bez `getKey()` jer to Laravel zna sam napraviti. Ostavat ćemo za sada s `getKey()`.

Vlastiti middleware

Kreirat ćemo novi middleware sa `php` naredbom:

```
php artisan make:middleware AuthMiddleware
```

Artisan će u `app/Http/Middleware` direktoriju kreirati `AuthMiddleware.php`. Middleware treba biti registriran u servisnom kontejneru kojem moramo imati pristup u aplikaciji. Kada ga registriramo, moći ćemo ga koristiti u našim rutama, kako bi mogli upravljati dolaznim zahtjevima gdje u `handle` metodu ulaze `Request` i `Closure`. Closure se koristi kao funkcija koja omogućava dolazak do sljedećeg `$request`-a.

```
<?php

namespace App\Http\Middleware;

use Closure;
use Illuminate\Http\Request;
use Symfony\Component\HttpFoundation\Response;

class AuthMiddleware
{
    /**
     * Handle an incoming request.
     *
     * @param \Closure(\Illuminate\Http\Request):
     * (\Symfony\Component\HttpFoundation\Response) $next
     */
}
```

```
public function handle(Request $request, Closure $next): Response
{
    return $next($request);
}
```

Sada trebamo definirati logiku u middleware-u. Možemo injektirati bilo koju ovisnost ako želimo. Bitno je da se middleware appenda. Ako je predugačko ime klase, možemo koristiti alias. Mi ćemo registrirati middleware tako da ga appendamo u servisni kontejner. U `routes/app.php` dodat ćemo `withMiddleware`:

```
->withMiddleware(function (Middleware $middleware) {
    // $middleware->append(AuthMiddleware::class);
})
->withExceptions(function (Exceptions $exceptions) {
    //
})->create();
```

Middleware na ruti ne moramo ni prijaviti, samo ako ćemo imati alias.

Sada bi htjeli vidjeti da li u middleware-u `app/Http/Middleware/AuthMiddleware.php` zaprimamo request:

```
public function handle(Request $request, Closure $next): Response
{
    dump($request);
    return $next($request);
}
```

Želimo vidjeti request objekt a to je moguće ako je registiran u ruti. U `routes/app.php` u test ruti napisat ćemo:

```
Route::get('test', function () {
    Log::info('Test log');
})->middleware(AuthMiddleware::class);
```

Kako nemamo alias moramo napisati full qualified class name. Request treba proći kroz middleware i trebamo vidjeti objekt.

```

Illuminate\Http\Request {#43 ▶ // app\Http\Middleware\AuthMiddleware.php:19
    +attributes: Symfony\...\\ParameterBag {#48 ▶}
    +request: Symfony\...\\InputBag {#44 ▶}
    +query: Symfony\...\\InputBag {#51 ▶}
    +server: Symfony\...\\ServerBag {#46 ▶}
    +files: Symfony\...\\FileBag {#50 ▶}
    +cookies: Symfony\...\\InputBag {#49 ▶}
    +headers: Symfony\...\\HeaderBag {#45 ▶}
    #content: null
    #languages: null
    #Charsets: null
    #encodings: null
    #acceptableContentTypes: null
    #pathInfo: "/test"
    #requestUri: "/test"
    #baseUrl: ""
    #basePath: null
    #method: "GET"
    #format: null
    #session: Illuminate\\SymfonySessionDecorator {#1203 ▶}
    #locale: null
    #defaultLocale: "en"
    -preferredFormat: null
    -isHostValid: true
    -isForwardedValid: true
    -isSafeContentPreferred: ? bool
    -trustedValuesCache: []
    -isIisRewrite: false
    #json: null
    #convertedFiles: null
    #userResolver: Closure($guard = null) {#1161 ▶}
    #routeResolver: Closure() {#1170 ▶}
    basePath: ""
    format: "html"
}

```

I vidimo ga. U Middleware-u imamo pristup request objektu. Ovaj middleware treba provjeriti da li je korisnik prijavljen ili ne, treba štititi rutu od neprijavljenih korisnika.

Napisat ćemo u `handle()` metodi:

```

public function handle(Request $request, Closure $next): Response
{
    dump($request);
    if (!Auth::check()) {
        return redirect()->route('showLogin');
    }
    // $request->merge(['title' => 'Dashboard']);
    return $next($request);
}

```

Ovime provjeravamo da li je korisnik prijavljen uz pomoć `Auth` servisa. Pokušamo li doći do `dashboard` neregistrirani, vratit će nas na `login`.

U `$request` koji je došao ubacujemo u middleware podatak koji je biti dostupan unutar `dashboard` rute `routes/web.php` i usera bi trebali vidjeti unutar rute `app/Http/Middleware/AuthMiddleware.php`:

```
public function handle(Request $request, Closure $next): Response
{
    dump($request);
    if (!Auth::check()) {
        return redirect()->route('showLogin');
    }
    // $request->merge(['title' => 'Dashboard']);
    return $next($request);
}
```

U postojeći request merge-amo neki novi podatak...

Pogledajmo što sve možemo raditi s post-om. Ako želimo request kontrolirati tko ima pravo što raditi trebamo koristi policy.

Na `welcome.blade.php` iskoristit ćemo direktive `@auth`:

```
@extends('layout', ['admin' => false])

@section('content')
    <h1>Welcome to Laravel Blog!</h1>
    @auth
        <p>Welcome, {{ auth()->user()->first_name }}!</p>
        {{-- <p><a href="{{ route('post.create') }}">Create a new post</a></p> --}}
        <p><a href="{{ route('logout') }}">Logout</a></p>
    @else
        <p>
            <a href="{{ route('showLogin') }}">Login</a>
            <a href="{{ route('register') }}">Register</a>
        </p>
    @endauth
@endsection
```

Ovdje imamo razliku da li je korisnik prijavljen ili nije.

Dodat ćemo rutu za logout u `routes/web.php`. Pošto želimo da i ova ruta ide preko `AuthMiddleware`, grupiramo sve rute koje želimo poslati preko middleware i ne moramo ih sve pojedinačno pozvati.

```

Route::group(['middleware' => 'guest', 'prefix' => 'auth'], function () {
    Route::get('login', [AuthController::class, 'login'])->name('showLogin');
    Route::post('login', [AuthController::class, 'authenticate'])-
>name('post.login');

    Route::get('register', [AuthController::class, 'showRegister'])-
>name('register');
    Route::post('register', [AuthController::class, 'register'])-
>name('post.register');
});

Route::group(['middleware' => AuthMiddleware::class], function () {
    Route::get('auth/logout', [AuthController::class, 'logout'])->name('logout');

    Route::group(['prefix' => 'admin'], function () {
        Route::get('dashboard', function () {
            return view('admin.dashboard');
        })->name('dashboard');

        Route::resource('posts', PostController::class);
    });
});

```

Sada ćemo grupirati rute u `routes/web.php` za `middleware` (kao što smo to već radili za naš `AuthMiddleware`).

`login` i `register` su unutar middleware-a `'guest'`. Dakle niti jednoj od tih ruta ne može se pristupiti ako je korisnik autentificiran tj. prijavljen u sistem jer nema što ići na te rute.

```

Route::group(['middleware' => 'guest', 'prefix' => 'auth'], function () {
    Route::get('login', [AuthController::class, 'login'])->name('showLogin');
    Route::post('login', [AuthController::class, 'authenticate'])-
>name('post.login');

    Route::get('register', [AuthController::class, 'showRegister'])-
>name('register');
    Route::post('register', [AuthController::class, 'register'])-
>name('post.register');
});

```

Kada kliknemo na dugme Register dobijemo grešku. Razlog tome je zato što ruta login nije definirana tj. probali smo doći na `localhost:8000/auth/register`. Razlog je što je promijenjeno ime rute ali to nije napravljeno na dijelu kod blade tj. view-a. Bitno je kad mijenjamo imena ruta u `routes/web.php` da promijenimo ime i na ostalim mjestima. Kod Visual Studio Code to moramo raditi ručno, što nije zgodno. Ovdje konkretno moramo promijeniti u `resources/views/register.blade.php` rutu tako da umjesto rute '`login`' koristimo postojeću rutu '`showLogin`'. '`login`' smo postavili u poglavlju [Layout opcija](#).

```
<p>Already have an account? <a href="{{ route('showLogin') }}>Login</a></p>
```

Nakon ovoga dugme Register radi.

Ispravljanje pogrešne rute

Zamislimo stotine ovakvih blade ruta gdje može doći do problema. Bolje je koristiti ovaj pristup `Neki link` umjesto `Neki link` jer takav pristup ne prikazuje grešku. Korisnik dobiva `404 NOT FOUND` što nećemo vidjeti u `.log` datoteci. Taj exception se ne zapisuje. Zapisuje se samo reportable exception a ovo je renderable exception i on se samo prikazuje korisniku. Dakle ovo je bug koji uzrokuje error.

Odglogiranje s aplikacije dodavanjem Logout dugmeta

U ruti `routes/web.php` smo namontirali logout. Time session token se neće invalidirati (neće se sljedeći puta dodijeliti ponovo i to moramo ručno napraviti). CSRF (Cross-site request forgery, falsificiranje zahtjeva na Internet stranicama koje se preklapaju) se isto tako neće invalidirati pa i trebamo generirati novi kod sljedeće prijave. O invalidaciji ćemo vidjeti kod JWT kako ne bi imali session hijacking⁵. Bez obzira što smo se odjavili, sistem može dodijeliti isti token.

```
Route::group(['middleware' => AuthMiddleware::class], function () {
    Route::get('auth/logout', fn() => AuthController::logout()->name('logout'));
    Route::get('dashboard', function () {
        dd(request());
    })->name('dashboard');
});
```

⁵ Session hijacking (otmica sesije) se odnosi na čin preuzimanja kontrole nad korisnikovom web sesijom. Napadač krade ili malipunira tokoenom kako bi dobio neovlašteni pristup informacijama ili uslugama. Proces otmice obično počinje kada napadač presretne ovaj token, što se može usporediti s tajnim rukovanjem između korisnika i web stranice. Nakon što je u posjedu ovog tokena, napadač dobiva mogućnost maskiranja u legitimnog korisnika, potencijalno uzrokujući pustoš. Metode presretanja mogu varirati, u rasponu od mrežnog prisluškivanja do sofisticiranih phishing napada. U većini aplikacija, nakon uspješne otmice sesije, napadač dobiva potpuni pristup svim korisničkim podacima i dopušteno mu je obavljanje operacija umjesto korisnika čija je sesija oteta. Može koristiti Brute Force, računanje i ukrasti ID sesije.

Rutu nećemo definirati u callback funkciji `fn() => AuthController::logout()`, što je iz PHP-a i nije vezano za Laravel. Radi se o arrow funkcijama (uvedenim u verziji 7.4) koje imaju sintaksu `fn(parametri) => izraz;` izraz predstavlja vrijednost koju funkcija vraća bez potrebe za korištenjem ključne riječi `return`. `fn` predstavlja function. Uklonit ćemo taj dio i ubaciti `[AuthController::class]`. To ispiše punu putanju gdje se nalazi naš `AuthController` kontroler. To daje string. Možemo provjeriti s:

```
Route::get('test', function () {
    dd(AuthController::class);
    $test = [];
});
```

`"App\Http\Controllers\AuthController"` // routes\web.php:37

To je string tj. putanja do kontrolera koja se oslanja na definiran namespace. To se naziva **full qualified class name** ili **FQCN**.

Drugi element ove matrice je ime metode. U našem slučaju to je `'Logout'`. Sada red glasi:

```
Route::get('auth/logout', [AuthController::class, 'logout'])->name('logout');
```

Nakon ovoga idemo u kontroler `app/Http/Controllers/AuthController.php`. Tu želimo pozvati servis i metodu koju ćemo sada napraviti `logout()`:

```
public function logout(Request $request)
{
    return $this->authService->logout( $request);
}
```

To znači da idemo na `$authService` servis. Ta varijabla je deklarirana na početku klase `AuthController`. U `__construct` vidimo `AuthService`. Ako kliknemo na njega s tipkom miša + Control, otvorit će se `app/Services/AuthService.php` i vidimo klasu `AuthService`. Na kraju te klase dodat ćemo metodu `logout()`:

```
public function logout(Request $request)

{
    Auth::logout();

    return redirect()->route('home');
}
```

Vidimo logout i redirekciju. Na nama je kuda ćemo redirektati. Ovdje nemamo potrebe za `intended()` metodom, kao u `authenticate()` metodi iznad. `intended()` prilikom prijave korisnika preusmjerava na početni URL (prije prijave) i tako poboljšava korisničko iskustvo. Ako izvorni URL nije dostupan,

korisnik se preusmjeravao na zadalu rutu `dashboard`. Na `Logout` za tim nema potrebe, redirektamo korisnika na `home`, koju još nemamo.

Kako nemamo rutu `home`, dodat čemo je u `routes/web.php` na dio koji već imamo:

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

Tako da dodamo ime na kraju reda `->name('home')`:

```
Route::get('/', function () {
    return view('welcome');
})->name('home');
```

Kada to ne bi imali dobili bi grešku zato `route('home')` što ne postoji.

Vratit čemo se na `app/services/AuthService.php`. Nakon što korisnika odjavimo invalidiramo sesiju i regeneriramo token. U dokumentaciji pod [Security>Authenitcation>Logging Out](#), piše da „Uz pozivanje `Logout` metode, preporučuje se da korisnikovu sesiju poništite i regenerirate njegov CSRF token.“ Moramo uhvatiti `$request` u `Logout` metodi. To znači da kako bi mogli validirati sesiju, moramo doći do `request-a`. Ima više načina, jedan je da ga prihvativmo kao dependency injection. Laravel u svaku metodu u našem kontroleru može uz dependency injection ubaciti objekt (to je predavač pokazao).

Predavač savjetuje da se držimo jednog te istog pravila kod programiranja. Ili koristiti dependency injection ili helper funkcije i biti konzistentan u korištenju Laravelovih funkcionalnosti. Loš primjer je ako npr. kažemo u servisu smo i nemam pristup `$request`, niti ga hvatam u kontroleru (ne radim dependency injection), pa će zaobići to pravilo pa će pozvati helper funkciju. Radit će program ali nećemo biti dosljedni. To kombiniranje kasnije otežava razumijevanje koda. Laravel također može raditi promjene koda i nešto će biti zastarjelo. Npr. helper funkcije se mogu izbaciti. Tada moramo mijenjati kod bez potrebe.

Ovo znači da čemo u `app/Http/Controllers/AuthController.php` napraviti običan request (nećemo napraviti vlastiti request)

```
public function logout(Request $request)
{
    return $this->authService->logout( $request);
}
```

Ovaj Request dolazi iz `Illuminate\Http\Request`. Treba biti pažljiv jer dolazi i `Request` iz `Illuminate\Support\Facades` i još nekoliko. Trebamo odabratи prvi.

```
on logout(Request $request)
  ↪ Request [use Illuminate\Http\Request]
    ↪ Request [GuzzleHttp\Psr7]
      ↪ Request [Illuminate\Http\Client]
        ↪ Request [Illuminate\Support\Facades]
          ↪ Request [Symfony\Component\HttpFoundation]
            ↪ Required
              ↪ RequiredIf
                ↪ RequirePass
                  ↪ Requirement [PHPUnit\Metadata\Version]
                    ↪ Requirement [PharIo\Manifest]
                      ↪ Requirement [Symfony\Component\Routing\Requirement]
                        ↪ RequiresPhp [PHPUnit\Framework\Attributes]
```

Ovdje ne trebamo `FormRequest`, zato što nemamo nikakve podatke koje dolaze u Request-u dolaze od strane korisnika kroz nekakvu formu, pa ih moramo validirati ili nešto s njima raditi. Uostalom `FormRequest` je samo nadogradnja na `Request`. Ovdje nam to ne treba jer nemamo nikakve validacije. Ovdje će nam Request poslužiti za slanje na naš servis `app/Services/AuthService.php`.

Sada imamo pristup `$request` objektu i na njemu možemo pozvati `$request->session()->invalidate();`

```
public function logout(Request $request)
{
    Auth::logout();

    $request->session()->invalidate();

    $request->session()->regenerateToken();

    return redirect()->route('home');
}
```

Ako pogledamo u `app/Services/AuthService.php`, `back()` i `redirect()` su helper funkcije, koje smo mogli zaobići ali ako se kod redirekta držimo toga da koristimo helper onda se toga držimo a ne koristimo fasadu.

```
$request->session()->regenerateToken();
```

Dobit ćemo novi CSRF token. Evo cijele `logout()` metode:

```
public function logout(Request $request)
{
    Auth::logout();

    $request->session()->invalidate();
```

```

    $request->session()->regenerateToken();

    return redirect()->route('home');

}

```

Bacanje iznimke (engl. query exception) u aplikaciji

Prije nego što idemo dalje, pogledajmo postoji li mogućnost za potencijalnim problemima koji se mogu dogoditi. Mi ne znamo što Laravel radi ispod haube u npr. `app/Services/AuthService.php` u metodi `create()`. Npr. ne znamo što ako predamo `create()` prazne podatke. To ćemo napraviti sa:

```
$user = User::create([]);
```

Dolazi je do bacanja iznimke (query exception). Mi ne upravljamo s time. Debugger upravlja s time ako je uključen. Debugger možemo ugasiti na `.env` i reći `APP_DEBUG=false` i tada ćemo dobiti samo grešku `500 SERVER ERROR` bez detalja. Mi tada ne znamo što se dogodilo. Laravel u `storage/logs` pohranjuje što se dogodilo u logove. Tu možemo u formatu `Laravel-godina-mjesec-datum.log` vidjeti da li je bilo problema u aplikaciji. To Laravel radi ispod haube.

U nekim trenutcima htjet ćemo raditi s iznimka. Pitanje je tko tu iznimku obrađuje i na koji način. Imamo naravno više načina. Custom iznimke prijavljujemo `withExceptions` u `bootstrap/app.php`. Sve iznimke koje se događaju u našoj aplikaciji možemo ili reportati ili ih možemo renderirati, ovisno što želimo. Možemo i potpuno zaobići globalni sistem i ručno u svakoj metodi definirati `try` i `catch`. Pogledajmo to u `app/Services/AuthService.php`:

```

public function register(FormRequest $request)
{
    try {
        $data = $request->safe(['first_name', 'last_name', 'email',
'password']);

        $user = User::create($data);
        $role =
            Role::where(
                'name',
                Roles::USER->value
            )
            ->first()
            ->getKey();

        $user->roles()->attach($role);

        Auth::login($user);
    }
}

```

```

        return redirect()->intended(route('dashboard'));
    } catch (\Exception $e) {
        Log::error($e);
        return back()->withErrors([
            'error' => 'An error occurred while registering the user.',
        ]);
    }
}

```

Mi ne idemo specifično jer ne znamo koje iznimke se mogu dogoditi. Pokrenut ćemo registraciju korisnika i pogledati da li će se to zapisati u exception. Nema tog reporta jer ništa nije zapisao – mi smo preuzeli kontrolu. Početnici grieše jer kad preuzmemmo kontrolu do Laravla više ne dolaze informacije o exception-u. Dakle ako ispišemo da se desila greška, mi želimo u logu imati detalje. Debuger je dobar samo kada razvijamo aplikaciju. Zato iza catch trebamo ručno napraviti log kako bi mogli zalogirati error, kao u slučaju iznad. Predamo mu cijeli exception \$e ili samo ime greške. Tako ćemo u logu imati zapisanu grešku i vidimo informaciju da nešto nije bilo u redu. Ako krenete upravljati s Exception u baznoj klasi hvatamo sve iznimke. Ako ne želimo hvatati iznimke koje hvata Laravel, onda to odredimo ovdje. Savjet predavača je da koristimo iznimke i kontroliramo kako kod radi.

Blade direktive

dashboard stranica

Idemo vidjeti `dashboard`, kako bi se mogli prijaviti. Iza toga posveti ćemo se razvoju svojstva posta, gdje ćemo proći kompletan CRUD. Vidjet ćemo kako funkcionira kontroler resursa (engl. Resource Controller) i koje su mu prednosti i mane da ne gradimo rutu po rutu. On može odhendlati sve kroz jednu rutu. Da ne guramo sve u direktorij `resources/views` otvorit ćemo poddirektorij `admin`. `Layout.blade.php` možemo podijeliti na više layouta, npr. default, public itd. Mi ćemo zadržati `Layout.blade.php` a u `resources/views/admin` otvorit ćemo `dashboard.blade.php`:

```

@extends('layout', ['admin' => true])

@section('title', 'Dashboard')

@section('content')
    <div class="container">
        <h1>Dashboard</h1>
        <p>Welcome to the admin dashboard.</p>
    </div>
@endsection

```

`@extends` direktiva se koristi kako bi naslijedila glavni layout iz datoteke `Layout.blade.php`. Čim kažemo `@extends` Blade templating sistem gleda u direktorij `views`, bez obzira gdje se nalazimo. Ovdje se također prosljeđuje parametar `'admin' => true`, što znači da ovaj layout prepoznaće da je korisnik

u administrativnom okruženju. Na taj način, layout može prilagoditi izgled stranice za administrativne korisnike, na primjer prikazivanjem različitih menija ili opcija.

Direktivom `@section` se definira sadržaj za sekciju nazvanu `title` koja će biti umetnuta u odgovarajuće mjesto unutar layout-a. Ovdje se naslov stranice postavlja na `Dashboard`. Layout datoteka ima `<title>` HTML tag gdje se dinamički postavlja ovaj naslov.

`@section('content')` je glavna sekcija koja se odnosi na sadržaj stranice. Sve unutar ove sekcije će biti prikazano u layoutu na odgovarajućem mjestu. Unutar `<div class="container">` se nalazi naslov `Dashboard` i paragraf s tekstom `Welcome to the admin dashboard.`, što predstavlja glavni sadržaj admin dashboarda.

`@endsection` direktiva označava kraj svake sekcije. Laravel zna gdje započinje i gdje završava određeni dio stranice.

nav.blade.php stranica

Treba nam navigacija da bi smo se kretali po stranicama. Moguće je izdvojiti komadiće HTML i onda ih uključivati include-om.

Napravit ćemo novu datoteku `nav.blade.php` u `resources/views/admin`. Logično bi bilo da navigaciju ubacimo u `layout.blade.php`. `layout.blade.php` se koristi ne samo za admin nego i ostale stranice: za login, register i welcome. Laravel omogućava kroz direktive da ima uvjetovano uključivanje što native PHP ne omogućava. Pogledamo kako ćemo popuniti `nav.blade.php`:

```
<nav>
  <p>
    <a href="{{ route('dashboard') }}>Dashboard</a> | 
    <a href="{{ route('posts.index') }}>Posts</a> | 
    <a href="{{ route('logout') }}>Logout</a> |
  </p>
  <hr>
</nav>
```

Ovaj Blade šablon `nav.blade.php` kreira jednostavnu navigacijsku traku za admin panel u Laravel aplikaciji. Pogledajmo detalje:

`<nav>` element koristi se za označavanje sekcije koja sadrži navigacijske linkove. To je semantički tag koji pomaže u organizaciji i strukturiranju stranice.

Unutar `<p>` (paragraf tag), postavljena su tri linka koja koriste Blade direktivu `{{ route() }}` za generiranje URL-ova:

- `Dashboard` link generira URL za rutu nazvanu `dashboard` koristeći funkciju `route()`. Klikom na ovaj link korisnik će biti preusmjeren na `dashboard` stranicu.
- `Posts` link generira URL za rutu `posts.index`, što će prikazati stranicu s popisom svih postova.
- `Logout` link vodi do rute `Logout`, što će pokrenuti postupak odjave korisnika iz aplikacije.

Znak | između svakog linka služi kao vizualni razdjelnik među njima.

<hr> je horizontalna linija koja vizualno odvaja navigacijski meni od ostatka sadržaja stranice.

Trebamo razmisliti gdje ćemo uključiti ovu navigaciju. U `dashboard.blade.php` ili `layout.blade.php`. Modificirat ćemo routu da vrati blade dashboard a kasnije ćemo to prebaciti na kontroler i servis. Idemo na `routes/web.php`, napraviti kako to radi i welcome, na `dashboard` ćemo reći `return view('admin.dashboard');`. Ako kažemo 'dashboard', helper metoda view gleda u direktorij `view`.tj. kreće iz nje kao root. Zato moramo reći `'admin.dashboard'`. Kada prođemo mišem vidjet ćemo punu putanju i odvest će nas do te datoteke a to je `resources/views/admin/dashboard.blade.php`.

```
default laravel-algebra-blog/resources/views/admin/dashboard
group(['prefix'=>'admin'])
te::get('dashb Follow link (ctrl + click)
return view('admin.dashboard');
```

Moguće je u Laravelu promijeniti i default postavka za `view` direktorij. Savjet predavača je da ne mijenjamo default podešene stvari.

Pogledajmo kako izgleda `dashboard` za administratora.

| <code>id</code> | <code>name</code> | <code>permissions</code> | <code>created_at</code> | <code>updated_at</code> | <code>deleted_at</code> |
|-----------------|-------------------|--------------------------|-------------------------|-------------------------|-------------------------|
| 7 | Admin | NULL | NULL | NULL | NULL |
| 8 | User | NULL | NULL | NULL | NULL |
| 9 | Author | NULL | NULL | NULL | NULL |
| NULL | NULL | NULL | NULL | NULL | NULL |

| <code>role_id</code> | <code>user_id</code> | <code>created_at</code> | <code>updated_at</code> |
|----------------------|----------------------|-------------------------|-------------------------|
| 7 | 21 | NULL | NULL |
| 7 | 22 | NULL | NULL |
| 7 | 24 | NULL | NULL |
| 7 | 26 | NULL | NULL |
| 8 | 28 | NULL | NULL |
| 8 | 29 | NULL | NULL |
| 8 | 30 | NULL | NULL |
| 8 | 31 | NULL | NULL |
| 8 | 32 | NULL | NULL |
| 9 | 23 | NULL | NULL |
| 9 | 25 | NULL | NULL |
| 9 | 27 | NULL | NULL |

| id | first_name | last_name | email | email_verified_at |
|----|------------|-----------|-------------------------------|---------------------|
| 21 | Jarred | Carroll | juliana.abernathy@example.net | 2024-09-24 07:59:38 |
| 22 | Davin | Paucek | armstrong.janis@example.com | 2024-09-24 07:59:39 |
| 23 | Martina | Rath | leora77@example.com | 2024-09-24 07:59:39 |
| 24 | Corene | Nader | friedrich37@example.com | 2024-09-24 07:59:40 |
| 25 | Octavia | Krajcik | bgibson@example.net | 2024-09-24 07:59:40 |
| 26 | Agustina | Hickle | oreilly.paolo@example.org | 2024-09-24 07:59:41 |
| 27 | Patrick | Lockman | ariel.goodwin@example.com | 2024-09-24 07:59:41 |
| 28 | Janelle | Kreiger | weber.reta@example.net | 2024-09-24 07:59:41 |
| 29 | Adrain | Pollich | mae.hackett@example.net | 2024-09-24 07:59:42 |
| 30 | Edna | Boyle | xspencer@example.com | 2024-09-24 07:59:42 |
| 31 | Pero | Perić | petar.petrovic@algebra.net | NULL |
| 32 | Ana | Anić | a@primjer.com | NULL |

Vidimo da su 21, 22, 24 i 26 Admin-i. Password za sve je `password`.

Login

Email

Password

Don't have an account? [Register](#)

Probajmo se logirati kao korisnik 26.

Dashboard

Welcome to the admin dashboard.

Stranica `dashboard` radi. Samo po sebi se nameće da u `dashboard.blade.php` uključimo navigaciju.

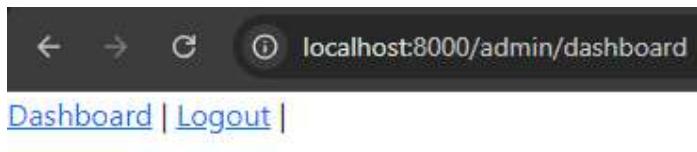
```
@extends('layout', ['admin' => true])

@section('title', 'Dashboard')

@section('content')
    <div class="container">
        @include('admin.nav')

        <h1>Dashboard</h1>
        <p>Welcome to the admin dashboard.</p>
    </div>
@endsection
```

Kako trenutno ne postoji ruta `admin.users` iz `resources/views/admin/nav.blade.php` treba je isključiti dok je ne definiramo, jer će nam u suprotnom Laravel javljati grešku. Dovoljno je staviti `{{--}}` na početku i `--}}` na kraju.



Dashboard

Welcome to the admin dashboard.

Imamo `Dashboard` i `Logout`. Post ćemo vratiti kada kreiramo rutu za njega. Sada možemo svuda ubacivati `@include('admin.nav')` da bezuslovno uključimo navigaciju. Dashboard ekstenda `Layout`. Svaki idući Blade će ekstendati `Layout`. Imamo situaciju da nisu svi Blade-ovi admin Blade-ovi. Postaje Blade-ovi koji ekstendaju layout a to su `login.blade.php`, `register.blade.php` `welcome.blade.php`. Nije onda baš zgodno da uzmemmo iz `dasboard.blade.php` red `@include('admin.nav')` i ubacimo ga u `<body>` u `layout.blade.php`, da vidimo kako radi.

Idemo na stranicu `localhost:8000/auth/register`. Gore nam opet ispisuje `Dashboard` i `Logout` kao linkove. To radi ali mi to ne želimo. Ne želimo imati tu registraciju na `login.blade.php`, `register.blade.php` i `welcome.blade.php`. Rekli smo da ovo želimo imati isključivo na admin Blade-ovima u admin direktoriju. Dakle kreirat ćemo odvojen layout za admina. To ima smisla ako će daj layout biti različit. Međutim, ako je identičan layout, onda je bolje [uključiti Blade Including Subviews tj. podpreglede](#). Iako će uključeni pogled (engl. view) naslijediti sve podatke dostupne u nadređenom pogledu, također možete proslijediti matricu dodatnih podataka koji bi trebali biti dostupni uključenom pogledu. Ako pokušate pristupiti `@include` pogledu koji ne postoji, Laravel će izbaciti grešku. Ako želite uključiti pogled koji može ili ne mora biti prisutan, trebali bi smo upotrijebiti `@includeIf` direktivu. Ako želite `@include` vidjeti da li dati Booleov izraz daje `true` ili `false`, možete koristiti `@includeWhen` i `@includeUnless` direktive.

Ove stvari nam mogu olakšati jer možemo u `resources/views/layout.blade.php` staviti odmah iza `<body>` taga `@includeWhen` :

```
<body>
  @includeWhen($admin, 'admin.nav')
  @yield('content')
</body>
```

Kada je `$admin = true` onda uključi a kada je `false` onda ne uključi. Ako sada pokrenemo program, dobit ćemo grešku da je nedefinirana varijabla `$admin` tj. `Undefined variable $admin`. Možemo poslati default vrijednost s `@includeWhen($admin = true, 'admin.nav')`. Ako na `resources/views/login.blade.php` na početku kažemo `@extends('layout', ['admin' => false])` umjesto `@extends('layout')`. Kao argument proslijede layout-u `admin => false`. Dakle `@extend` nudi da si pošaljemo podatke u obliku matrice, `admin` je ključ koji će završiti u varijabli `admin`. Pravilo je da ključ u ovoj matrici mora se zvati identično kao varijabla na našem layoutu. U ovom

slučaju, admin je postavljen na `false`, što znači da se prikazuje regularni korisnički layout. Vrijednost se prebaci na varijablu `$admin` u `resources/views/Layout.blade.php`. Ovisno da li je definirano `$admin = true` ili `$admin = false` vidimo ili ne vidimo navigaciju na `localhost:8000/auth/login`.

Na svakom našem Blade-u koji ekstenda layout možemo kontrolirati želimo li neki dio uključiti ili ne. U ovom slučaju je to navigacija. Imamo i default opciju ako na `resources/views/login.blade.php` ne pošaljemo ništa. Ako pošaljemo onda će postaviti varijablu koju šaljemo na default. To je nedostatak PHP-a. Ako stvarno želimo default vrijednost onda na početku `resources/views/layout.blade.php` postavimo `@php` direktivu u kojoj napišemo:

```
@php  
    $admin = $admin ?? false;  
@endphp
```

Nullish operator (`??`) u PHP-u koristi se za postavljanje vrijednosti varijable samo ako je ta varijabla null. Ako nije null, zadržava svoju trenutnu vrijednost. Ako `$admin = null`, nakon izvršavanja ovog koda, `$admin` će biti false. Ako `$admin = true`, vrijednost će ostati `true`.

Predavač savjetuje da ne petljamo sa tim defoltom nego uključujemo `@php` direktivu. Ako je ovo prekompleksno dignemo novi layout za admina. Mi smo na `resources/views/login.blade.php` stavili `@extends('layout', ['admin' => false])` a sada ćemo i na `welcome.blade.php`, `register.blade.php` i `login.blade.php`. Na taj način kontroliramo na layout-u i palimo i gasimo navigaciju.

Admin interface-u neće pristupati samo administrator. Na njega gledamo kao backoffice. Kako imamo tri različite role, trebamo znati da li autor može pristupiti admin interface-u ili ne.

Na početku `resources/views/layout.blade.php` možemo u `@php` direktivu staviti helper `request()` metodu:

```
@php  
    $admin = request()->is('admin/*');  
@endphp
```

`request()` metoda poziva globalni Request objekt u Laravelu, koji sadrži informacije o trenutnom HTTP zahtjevu (uključujući URL, metode, parametre itd.). Metoda `is()` provjerava da li trenutni URL (ili putanja) odgovara zadatom obrascu. U ovom slučaju, provjerava se da li trenutni URL počinje na `admin`. To je wildcard `*`.

Ako je URL točno `admin`, metoda vraća `true`, u suprotnom vraća `false`.

CRUD u Laravelu

Pozabavit ćemo se postovima i zato nam treba CRUD. Predavač takve zadatke naziva CRUDalice. Skraćenica CRUD označava Laravelovo kreiranje (engl. Create), čitanje (engl. Read), ažuriranje (engl. Update) i brisanje (engl. Delete).

Kontroler resursa (engl. resource controller)

Prvo ćemo pogledati u Kontrolerima tzv. Resource Controllers tj. Kontroleri resursa. Laravel rutanje resursa dodjeljuje uobičajenim CRUD rutama za kontrolu s jednim redom koda. Možemo upotrijebiti

opciju Artisan naredbe `make:controller ImeKontrolera --resource` za kreiranje kontrolera koji upravlja ovim akcijama:

```
php artisan make:controller MojKontroler --resource
```

Ova naredba će generirati kontroler na `app/Http/Controllers/MojKontroler.php`. Kontroler će sadržavati metodu za svaku od dostupnih operacija resursa. Zatim možete registrirati rutu resursa koja upućuje na kontroler:

```
use App\Http\Controllers\MojController;

Route::resource('photos', PhotoController::class);
```

Mi ne definiramo o kojoj se metodi radi (`GET`, `PUT`, `POST`, `PATCH`, `DELETE`), ne definiramo URL, imena rutama, ni metode kojima šaljemo request sa routera na kontroler. O tome brine Laravel ispod haube. Imamo doslovno jednu rutu sa 7 metoda i automatski će sve biti povezano s jednom rutom.

Radnje kojima upravljaju kontroleri resursa

| HTTP metoda | URI | Akcija (ime metode u kontroleru) | Ime rute |
|-------------|----------------------|----------------------------------|----------------|
| GET | /photos | index | photos.index |
| GET | /photos/create | create | photos.create |
| POST | /photos | store | photos.store |
| GET | /photos/{photo} | show | photos.show |
| GET | /photos/{photo}/edit | edit | photos.edit |
| PUT/PATCH | /photos/{photo} | update | photos.update |
| DELETE | /photos/{photo} | destroy | photos.destroy |

Moguće je vezati model rute i metode kontrolera resursa (engl. Resource Controller) upisati u instancu modela, kao i uputiti Artisan da generira klase zahtjeva forme za metode pohrane i ažuriranja kontrolera:

```
php artisan make:controller NazivKontrolera --model=Photo --resource --requests
```

Ovo automatski kreira dva requesta, jedan za update a drugi za create. Moguće je kreirati i kontroler za API što je zgodno je na API nema pogleda i nepotrebni su endpoint-ovi za create i edit jer ne vraćate nikakvu formu.

Upravo ovo ćemo napraviti na našem primjeru:

```
php artisan make:controller PostController --model=Post --resource --requests
```

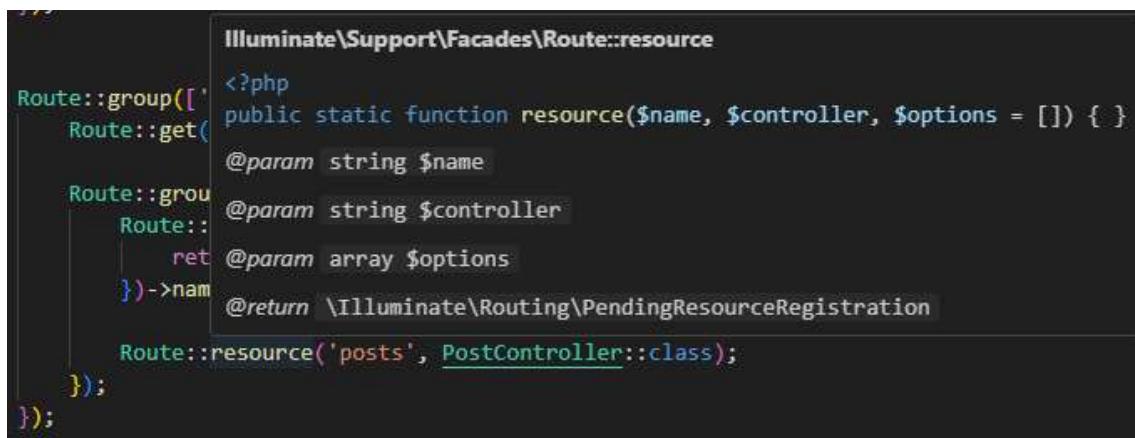
Kreirao je nekoliko stvari: request-ove u `app/Http/Requests/StorePostRequest.php` i u istom direktoriju `UpdatePostRequest.php` i `PostController.php`.

Idemo u `routes/web.php` dodati rutu ka `PostController`: `Route::resource('posts', PostController);` Tu bi funkcioniralo i sa `Route::resource('posts', PostController::class);`. Sa navodnicima ga ne moramo uključiti gore. U stringu ne znamo gdje je kontroler smješten niti je definiran u dijelu `use`. Kada koristimo FQCN onda vidimo putanju. Pritisnemo Ctrl+klik tipku i prebacit će nas u `app/Http/Controllers/PostController.php`, ako to trebamo.

Ako pogledamo rute onda ćemo imati `/posts` (prvi slučaj u tablici). Staviti ćemo ispred `admin`, tako da će red glasiti:

```
Route::resource('admin.posts', PostController::class);
```

Ako pogledamo `resource` kontroler prvi parametar mu je `name`, ime resursa iz kojeg kreira URI, iz kojeg kreira ime rute.



```
Illuminate\Support\Facades\Route::resource
<?php
public static function resource($name, $controller, $options = []) {
    @param string $name
    @param string $controller
    @param array $options
} ->name(@return \Illuminate\Routing\PendingResourceRegistration
    Route::resource('posts', PostController::class);
});;
});;
```

Grupiranje ruta

Zato ovo neće raditi i neće pretvoriti u ispravan URI. Kada radimo s `resource`, ipak želimo imati `admin/post`. Ono što Laravel nudi je grupiranje naših ruta. Ako imamo iste rute koje počinju s `auth` u našem slučaju, sve ih možemo smjestiti u istu grupu koja počinje s `auth`.

U middleware možemo dodati i prefix `auth`, što znači da u ruti ne moramo navesti '`auth/login`' ili '`auth/register`' ili neki drugi. Sada će ruta middleware glasiti:

```
Route::group(['middleware' => 'guest', 'prefix' => 'auth'], function () {
    Route::get('login', [AuthController::class, 'login'])->name('showLogin');
    Route::post('login', [AuthController::class, 'authenticate'])-
>name('post.login');

    Route::get('register', [AuthController::class, 'showRegister'])-
>name('register');
    Route::post('register', [AuthController::class, 'register'])-
>name('post.register');
});;
```

Ovo je praktičniji način grupiranja. Ovo nema u dokumentaciji.

Ako sada probamo, rute normalno funkcioniraju.

Unutar grupe middleware ne možemo raditi grupiranje po prefiksnu ali možemo kreirati grupu unutar grupe:

```
Route::group(['middleware' => AuthMiddleware::class], function () {
    Route::get('auth/logout', [AuthController::class, 'logout'])->name('logout');

    Route::group(['prefix' => 'admin'], function () {
        Route::get('dashboard', function () {
            return view('admin.dashboard');
        })->name('dashboard');

        Route::resource('posts', PostController::class);
    });
});
```

Sada smo i na svoj kontroler resursa (engl. Resource Controller) nadodali da ima `admin/post`, i to je nedostatak kontrolera resursa da ne možemo na njemu imati `resource`. Sada sve funkcionira što znači da `LoginRequest`, `RegisterRequest` i `StorePostRequest` trebamo popuniti s validatorima.

Ponovo uključivanje Posts rute

Nakon što smo sve rute kontrolirali, idemo na `resources/views/admin/nav.blade.php` ukloniti komentar s `Posts` | gdje ime rute neće biti `admin.posts`. Laravel na osnovu svoje naming konvencije (vidi tablicu iznad) kaže da ruta `admin/post` gađa metodu na kontroleru `index`, a ruta se zove `posts.index`. Potencijalno imamo problem, sa istim imenom ruta jer potencijalno imamo postove koji su unutar admina i koji su izvan admina. Ne treba zaboraviti da imamo jedan kontroler resursa (engl. resource controller) koji je u adminu. Što se tiče imenovanja ruta, možemo napraviti dva post kontrolera od kojih je jedan za admin dio. Možemo razdvojiti (isto kao što smo poglede stavili u direktorij admin) direktorije i `PostControllers.php` možemo staviti u direktorij admin i onda unutar `Controllers` možemo imati dva `PostController`-a uz različiti Namespace.

Vratimo se na `resources/views/admin/nav.blade.php`. Rekli smo da na temelju imena u rutu kreira imena svih ruta koje će složiti. Kako u ruti `routes/web.php` imamo `Route::resource('posts', PostController::class)`; onda možemo napisati `route('posts.index')`:

```
<a href="{{ route('posts.index') }}>Posts</a> |
```

On će sada kada kliknemo na Post, odvesti nas na `app/Http/Controllers/PostController.php` i njegovu metodu `index()`. Ako otvorimo taj `PostController.php`, vidjet ćemo da u njemu imamo sve metode `index()`, `create()`, `store()` povezan već s `$request` koji je stvorio, `show()` povezan s

modelom, `edit()` povezan s modelom, `update()` povezan s `$request`-om i s modelom, tako da kroz dependency injection imamo razriješeno sve. Ne moramo tražiti koji post treba ažurirati u bazi, Laravel je to već razriješio, dobit ćemo tu model. Na kraju imamo `destroy()` metodu.

```
<?php

namespace App\Http\Controllers;

use App\Models\Post;
use App\Http\Requests\StorePostRequest;
use App\Http\Requests\UpdatePostRequest;

class PostController extends Controller
{

    /**
     * Prikaži popis izvora.
     */
    public function index()
    {

    }

    /**
     * Prikaži formu za kreiranje novog izvora.
     */
    public function create()
    {
        //
    }

    /**
     * Pohrani novokreirani resurs u skladište(engl. storage).
     */
    public function store(StorePostRequest $request)
    {
        //
    }

    /**
     * Display the specified resource.
     */
    public function show(Post $post)
    {
        //
    }
}
```

```
}

/**
 * Prikaži određeni resurs.
 */
public function edit(Post $post)
{
    //
}

/**
 * Ažuriraj određeni resurs u skladište (engl. storage).
 */
public function update(UpdatePostRequest $request, Post $post)
{
    //
}

/**
 * Ukloni navedeni resurs iz skladišta (engl. storage).
 */
public function destroy(Post $post)
{
    //
}
}
```

Ispravak absolutnih ruta u relativne

Ako sada provjerimo da li radi, vidjet ćemo da program dojavi grešku. To je zato što je u `app/Services/AuthService` u `authenticate()` metodi ostala absolutna ruta umjesto relativne. Red `return redirect()->intended('dashboard');` trebamo ispraviti u `return redirect()->intended(route('dashboard'));`. Ovo je naravno moguće izvući u varijablu. Može i ovako ako nam nije potrebna informacija (kao nama). Ovakvu promjenu trebamo napraviti svuda gdje koristimo absolutne rute (u istoj datoteci u `Register()` metodi).

Nakon login-a bilo kojeg korisnika preusmjereni smo na <http://localhost:8000/admin/dashboard> što smo i željeli.

Ako odemo na [admin/posts](#) pomoću dugmeta Posts, vidjet ćemo da još nemamo ništa jer otišao do metode u kontroleru koja je prazna.

Da ponovimo, grupirali smo [admin](#) rute uz pomoć '[prefix](#)' zbog [resource](#) kontrolera (engl. Resource Controller) i [resource](#) rute koja nema mogućnost da kontrolira URI nego on kontrolira URI na temelju imena '[posts](#)' (i kontrolirat će sve URL-je na temelju tog imena). Htjeli smo dodati [admin](#) ispred i to smo riješili sa prefiksom.

Nakon toga smo vidjeli problem s Dashboard-om, gdje smo željeli koristili ime rute a ne URI i to smo popravili. Sada ćemo pogledati kontroler [app/Http/Controllers/PostController.php](#). Metode su prazne. Ako želimo mijenjati nazive metoda, onda treba kemijati nad [Resource](#) rutom. Savjet predavača je to ne dirati.

Sljedeći korak je kao što smo imali [app/Services/AuthService.php](#), kreirati [PostService.php](#) na istoj lokaciji. Taj servis neće imati nikakvu logiku. Želimo dobiti sve postove iz baze. Kreirat ćemo metodu [getAllPosts\(\)](#). U servisu ne može biti metoda s nazivom [index\(\)](#) jer to nema logike. Ako pogledamo [app/Services/AuthService.php](#), tu smo redirektali i radili obradu grešaka ako je došlo do njih. To možemo i ovdje ali moramo vratiti pogled sa podacima. Ovo ćemo kasnije proširiti zbog paginacije⁶.

```
<?php

namespace App\Services;

class PostService
{
    public function getAllPosts()
    {
        return Post::all()
    }
}
```

⁶ **Paginacija** je proces podjele velikih setova podataka na manje dijelove (stranice) radi lakše obrade i prikaza u pogledu (engl. view). Paginacija omogućava korisnicima da pregledaju podatke postepeno, kroz numerirane stranice, umjesto da učitaju i prikažu sve podatke odjednom, što bi moglo značajno usporiti aplikaciju. Paginaciju možemo raditi u modelu ili kontroleru ili u pogledu.

```

    }
}
```

Nema potrebe ovdje vraćati pogled jer će to odraditi pogled `app/Http/Controllers/PostController.php`, čija je zadaća da primi request, obradi request i vrati odgovor. Najčešće kontroler vraća pogled. U `index()` možemo doći do nekih podataka:

```

public function index()
{
    $posts = Post::all();
}
```

Umjesto da imamo `Post::all()`; dakle da nemamo servis, nama treba servis i napisat ćemo `PostService::` jer moramo dobiti objekt jer `Post` nije statička metoda u `app/Services/PostService.php` u `getAllPosts()` i ne možemo reći `PostService::get()`, nego moramo koristiti dependency injection ili direktno u metodu `index()` u `app/Http/Controllers/PostController.php` ili u konstruktoru nešto što će riješiti odmah dependency injection i servis injektati kroz konstruktor. Mi ćemo ga zapisati u naš property i kroz taj property imati pristup cijelo vrijeme servisu.

```

class PostController extends Controller
{
    private $postService;

    public function __construct(PostService $postService)
    {
        $this->postService = $postService;
    }
}
```

`index()` ćemo sada ispraviti. `return` bi bio neki `view` gdje bi poslali ove podatke. Za sada neka ostane ovako:

```

public function index()
{
    $posts = $this->postService->getAllPosts()
        dd($posts);
}
```

Zahtjev dolazi u kontroler. Kontroler pita servis da da postove. Za sada je tamo logika jednostavna ali logika može biti i kompleksna. Moguće je u requestu slati atribute po kojima možemo filtrirati stvari. To se može poslati `getAllPosts()` metodi i onda ona na temelju tih atributa napravi filtriranje podataka i vrati samo podatke koje ste tražili. To znači da možemo imati opciju npr. sortiranja po

različitim kriterijima ili filtriranje po autoru. Zato je bitno onda to odvojiti u servis. Ako ćemo kasnije trebati ubaciti nešto onda imamo razrađenu poslovnu logiku i ne radimo je na kontroleru.

```
Illuminate\Pagination\LengthAwarePaginator {#1265 ▼ // app\Http\Controllers\PostController.php:30
  #items: Illuminate\Collection {#1258 ▼
    #items: array:5 [▼
      0 => App\Mode...\\Post {#1272 ▶}
      1 => App\Mode...\\Post {#1273 ▶}
      2 => App\Mode...\\Post {#1274 ▶}
      3 => App\Mode...\\Post {#1275 ▶}
      4 => App\Mode...\\Post {#1276 ▶}
    ]
    #escapeWhenCastingToString: false
  }
  #perPage: 5
  #currentPage: 1
  #path: "http://localhost:8000/admin/posts"
  #query: []
  #fragment: null
  #pageName: "page"
  +onEachSide: 3
  #options: array:2 [▼
    "path" => "http://localhost:8000/admin/posts"
    "pageName" => "page"
  ]
  #total: 10
  #lastPage: 2
}
```

Sada bi trebali vidjeti sve postove koje imamo u bazi. Vidimo kolekciju modela. U svakom modelu imat ćemo informacije.

Ono što ovdje ne vidimo je koji je user, dakle ne ID. Sada imamo dvije opcije, koristiti **Lazy load** i **Eager load**. Kada hvatamo više podataka iz baze trebali bi koristiti Eager load. Prilikom requesta tj. query-ja na bazu s original podacima dohvati sve podatke o relaciji. Ako pogledamo Lazy load i kolekciju vratimo na pogled, u nekoj tablici trebamo ispisati Title posta i ime autora. U modelu nema imena autora. Ako pogledamo relaciju `#relations: []`, ona je prazna, ne možemo doći do podatka jer ga nismo povukli. Moramo naknadno pitati bazu za podatak o useru. Imamo prvi query na bazu i onda još 5 query-ja (ili koliko ih već ima) za svaki podatak, kako bi za svaki post dobili informaciju o useru. Zato koristimo ovdje Eager load.

Zato u servisu `app/Services/PostService.php` u metodi `getAllPost()` ćemo umjesto vraćanja `Post::all()` napraviti relaciju :

```
public function index()
{
    return Post::with('user')->get();
}
```

Odmah u ovoj kolekciji u modelu post dobit ćemo informaciju o useru koji je vlasnik.

```

#relations: array:1 [▼
  "user" => App\Models\User {#1254 ▼
    #connection: "mysql"
    #table: "users"
    #primaryKey: "id"
    #keyType: "int"
    +incrementing: true
    #with: []
    #withCount: []
    +preventsLazyLoading: false
    #perPage: 15
    +exists: true
    +wasRecentlyCreated: false
    #escapeWhenCastingToString: false
    #attributes: array:11 [▼
      "id" => 24
      "first_name" => "Corene"
      "last_name" => "Nader"
      "email" => "friedrich37@example.com"
      "email_verified_at" => "2024-09-24 07:59:40"
      "password" => "$2y$12$LNW2oQqsQcsn8AYokj0g.ee45KpNnxM0W21vC7DWW794WIu5wiHY."
      "remember_token" => "qr3HfBMdk8"
      "created_at" => "2024-09-24 07:59:43"
      "updated_at" => "2024-09-24 07:59:43"
      "deleted_at" => null
      "phone" => null
    ]
    #original: array:11 [▶]
    #changes: []
    #casts: array:2 [▶]
  ]
]

```

Vratimo se na [app/Http/Controllers/PostController.php](#). Nemamo višestruke query-je na bazi. Dodat ćemo pogled `admin.post.index`. `compact` kreira maticu koja sadrži varijable i njihove vrijednosti. `compact` će kolekciju pretvoriti tj. serijalizirati. Bitno je da je prebrojiva (engl. enumerable) i možemo po njoj proći petljom.

```

public function index()
{
    $posts = $this->postService->getAllPosts()

    return view('admin.posts.index', compact('posts'));
}
}

```

`compact` će na kraju kreirati matricu. Prilikom ulaska će morati `$posts` serijalizirati u matricu. Kolekciju u tren oka možemo pretvoriti u maticu, sam ona kraj reda dodamo `$posts = $this->postService->getAllPosts()->toArray();`

```
array:13 [▼ // app\Http\Controllers\PostController.php:30
  "current_page" => 1
  "data" => array:5 [▶]
  "first_page_url" => "http://localhost:8000/admin/posts?page=1"
  "from" => 1
  "last_page" => 2
  "last_page_url" => "http://localhost:8000/admin/posts?page=2"
  "links" => array:4 [▶]
  "next_page_url" => "http://localhost:8000/admin/posts?page=2"
  "path" => "http://localhost:8000/admin/posts"
  "per_page" => 5
  "prev_page_url" => null
  "to" => 5
  "total" => 10
]
```

Možemo pretvoriti i u JSON sa `$posts = $this->postService->getAllPosts()>toJson();`

```
"{"current_page":1,"data":[{"id":21,"title":"Quod placeat placeat rem debitis.", "content":"Quia id totam magni quo quo dolor. Ea quis sed doloribus ducimus eius. Ipsum laborum et possimus voluptas voluptatem dolorem explicabo.", "slug":"dolores-voluptas-autem-rerum-voluptatis-id-laborum","image":"https://via.placeholder.com/640x480.png/00aadd?text=saepe","user_id":24,"created_at":"2024-09-24T07:59:43.000000Z","updated_at":"2024-09-24T07:59:43.000000Z","deleted_at":null,"user":{"id":24,"first_name":"Corene","last_name":"Nader","email":"friedrich37@example.com","email_verified_at":"2024-09-24T07:59:40.000000Z","created_at":"2024-09-24T07:59:43.000000Z","updated_at":"2024-09-24T07:59:43.000000Z","deleted_at":null,"phone":null}},"id":22,"title":"Aut rem laboriosam quis quaerat dolore accusamus.", "content":"Eaque perferendis eveniet maiores voluptatem. In recusandae enim nihil est sit. Dignissimos et atque sit non et dolor.", "slug":"sint-suscipit-quo-apерiam-sint-quia","image":"https://via.placeholder.com/640x480.png/005522?text=molestias","user_id":23,"created_at":"2024-09-24T07:59:43.000000Z","updated_at":"2024-09-24T07:59:43.000000Z","deleted_at":null,"user":{"id":23,"first_name":"Martina","last_name":"Rath","email":"leora77@example.com","email_verified_at":"2024-09-24T07:59:39.000000Z","created_at":"2024-09-24T07:59:43.000000Z","updated_at":null,"phone":null}},"id":23,"title":"Dolor similiique ut asperiores adipisci cupiditate amet expedita.", "content":"Doloribus et voluptas soluta fuga eos. Nostrum est voluptatibus ut laboriosam. Aperiam reiciendis et est nam vitae non atque.", "slug":"consequatur-blanditiis-assumenda-consequuntur","image":"https://via.placeholder.com/640x480.png/00ff55?text=quas","user_id":24,"created_at":"2024-09-24T07:59:43.000000Z","updated_at":"2024-09-24T07:59:43.000000Z","deleted_at":null,"user":{"id":24,"first_name":"Corene","last_name":"Nader","email":"friedrich37@example.com","email_verified_at":"2024-09-24T07:59:40.000000Z","created_at":"2024-09-24T07:59:43.000000Z","updated_at":null,"phone":null}},"id":24,"title":"Aspernatur veritatis ab quisquam est placeat eum provident.", "content":"Veniam quia inventore voluptatem iure aperiam asperiores fugiat. Inventore qui voluptatem qui occaecati voluptatum quia. Omnis occaecati nam consecetur a. Quaerat error et id.", "slug":"animi-veritatis-perferendis-quasi","image":"https://via.placeholder.com/640x480.png/0077aa?text=fugit","user_id":22,"created_at":"2024-09-24T07:59:43.000000Z","updated_at":"2024-09-24T07:59:43.000000Z","deleted_at":null,"user":{"id":22,"first_name":"Davin","last_name":"Paucek","email":"armstrong.janis@example.com","email_verified_at":"2024-09-24T07:59:39.000000Z","created_at":"2024-09-24T07:59:43.000000Z","updated_at":null,"phone":null}},"id":25,"title":"Eos iusto asperiores eius incident earum inventore culpa.", "content":"Modi odio eum et sed delectus deleniti vero voluptatibus. Expedita voluptas quam labore. Repellendus atque non amet aut ullam molestias.", "slug":"totam-soluta-tenuit-sit-et-debitis-perspicillatis-nulla","image":"https://via.placeholder.com/640x480.png/0088ff?text=facilis","user_id":24,"created_at":"2024-09-24T07:59:43.000000Z","updated_at":"2024-09-24T07:59:43.000000Z","deleted_at":null,"user":{"id":24,"first_name":"Corene","last_name":"Nader","email":"friedrich37@example.com","email_verified_at":"2024-09-24T07:59:40.000000Z","created_at":"2024-09-24T07:59:43.000000Z","updated_at":null,"phone":null}},"first_page_url": "http://localhost:8000/admin/posts?page=1", "from": 1, "last_page": 2, "last_page_url": "http://localhost:8000/admin/posts?page=2", "links": [{"url": null, "label": "\u2190 Previous"}, {"url": "http://localhost:8000/admin/posts?page=1", "label": "1", "active": true}, {"url": "http://localhost:8000/admin/posts?page=2", "label": "2", "active": false}, {"url": "http://localhost:8000/admin/posts?page=2", "label": "Next \u2192", "active": false}], "next_page_url": "http://localhost:8000/admin/posts", "per_page": 5, "prev_page_url": null, "to": 5, "total": 10}
```

Ako radimo s API-jima treba nam JSON. Kolekcija koristi i implementira [Enumerable Contract](#) tj. Enumerable interface. Enumerable omogućava serijalizaciju iz jednog tipa u drugi, što je jako bitno kod rada s API.

Eventualno ćemo dodati dugme Kreiraj novi a pored svakog posta Izmjeni postojeći i Obriši post. Podsjetimo se da HTML ne podržava Delete i PUT i Patch tj. podržava samo Post i GET. Pitanje je kako to Laravel hendla u monolitu. Najčešće je problem zbog Delete jer kroz pritisak na dugme treba poslati delete request ili kroz formu u HTML a forme to ne podržavaju. O ovome ćemo govoriti idući puta.

Kreirat ćemo taj `admin.posts.index` tj u pogledu `resources/views/admin/` kreirat ćemo direktorij `posts` gdje ćemo ih prikazati. Napravit ćemo `index.blade.php` koji do sada imali:

```
@extends('layout', ['admin' => true])
```

```
@section('title', 'Posts')
```

```

@section('content')
    <div class="container">
        <h1>Posts</h1>
        <p><a href="{{ route('posts.create') }}>Create a new post</a></p>
        <table class="table table-striped">
            <thead>
                <tr>
                    <th>Title</th>
                    <th>Author</th>
                    <th>Published</th>
                    <th>Actions</th>
                </tr>
            </thead>
            <tbody>
                @foreach ($posts as $post)
                <tr>
                    <td>{{ $post->title }}</td>
                    <td>{{ $post->user->first_name }} {{ $post->user->last_name
}}</td>
                    <td>{{ $post->created_at->format('d.m.Y. H:i') }}</td>
                    <td>
                    </td>
                </tr>
                @endforeach
            </tbody>
        </table>
        <div class="mx-auto my-2">
            {{-- {{ $posts->links() }} --}}
        </div>
    </div>

```

Imamo naslov i imamo dugme `Create a new post` koje će voditi na rutu gdje će biti forma za kreiranje. Zatim se kreirata tablica s četiri stupca: Title, Author, Published i Actions. Nemamo polje Published, nego `title`, `content`, `image` (možemo to vidjeti u `app/Models/Post.php`) i `created_at` i `updated_at`. Nadalje imamo `@foreach` direktivom koja prolazi kroz `$post` matricu. Odmah iza `@section` direktive, odat ćemo:

```

@php
    dd($posts);
@endphp

```

Pogledat ćemo što se nalazi u varijabli `$post`. Kada pogledamo kontroler `app/Http/Controllers/PostController.php`, koji je vratio pogled i postove koje je dohvatio pomoću servisa je smjestio u matricu uz pomoć metode `compact`. Proslijedio je nazad kolekciju i imamo pristup kolekciji.

```

Illuminate\Pagination\LengthAwarePaginator {#1265 ▼ // resources\views\admin\posts\index.blade.php
  #items: Illuminate\Collection {#1258 ▼
    #items: array:5 [▼
      0 => App\Model\Post {#1272 ▶}
      1 => App\Model\Post {#1273 ▶}
      2 => App\Model\Post {#1274 ▶}
      3 => App\Model\Post {#1275 ▶}
      4 => App\Model\Post {#1276 ▶}
    ]
    #escapeWhenCastingToString: false
  }
  #perPage: 5
  #currentPage: 1
  #path: "http://localhost:8000/admin/posts"
  #query: []
  #fragment: null
  #pageName: "page"
  +onEachSide: 3
  #options: array:2 [▶]
  #total: 10
  #lastPage: 2
}

```

Dobili smo kolekciju na Blade-u. Obrisat ćemo @php direktivu.

Kolekcija ima implementiran inteface enumerable, @foreach direktiva (ili obična foreach petlja) će znati kako proći po svakom elementu kolekcije, dobit ćemo \$post. A taj \$post je naš model. Kada imamo model (umjesto matrice), lakše je doći do atributa modela. Model je objekt a kada pretvorimo u matricu imamo višedimenzionalnu matricu i unutra asocijativnu matricu do čijih elemenata moramo moći s nespretnjom sintaksnom i nečitkije. Zato predavač proeporuča raditi s modelima kao objektima umjesto s matricom.

Došao je do modela i dohvatio title s \$post->title. Želi doći do usera i dohvatio first_name i last_name, te created_at. Nema Lazy load-a jer su sada je već kolekcija i model napunjeni relacijom user, tako da možemo doći do first_name i last_name.

Sada se možemo kretati između dashboard i post.

Podešavanje CRUD

Trebamo riješiti uređivanje postova, dodati post, urediti post i izbrisati post. Koristit ćemo soft delete.

Resource rute su praktične jer imamo samo jednu. 'posts' je ime rute a ne URI. To ime se razriješi na nivou Laravela u URI za svaku pojedinu rutu. Ako se držimo naming konvencije, Laravel poveže svaku rutu sa ispravnom metodom u app/Http/Controller/PostController.php. Ako pogledamo tamo, vidjet ćemo metode index(), create(), store(), show(), edit(), update() i destroy(). Te metode su actions kako ih Laravel zove. Te metode se pozivaju u ovisnosti gdje se dogodio request. Dakle nismo pisali za svaku metodu ručno kao u routes/web.php.

Kada se dogodi request na URI na admin/posts i ide metodom GET aktivira se u našem podkontrolelu app/Http/Controllers/PostController metoda index(). Tamo izvučemo sve postove iz baze uz pomoć servisa getAllPosts čija je zadaća da preko modela app/Services/PostService.php dohvati sve postove i pomoću Eager loada uključi sve informacije o useru u svaki post kako ne bi za svakog Authora tj. usera raditi request prema bazi.

Kada servis vrati sve postove iz baze, kontroler te podatke koje je zaprimio preusmjeri na pogled `admin.posts.index` i krira matricu koja sadrži varijable i njihove vrijednosti (`compact()`). Na pogledu su ti podaci dostupni i bit će u varijabli `posts`. Pogled se nije morao zvati tako ali **uvriježeno je da se za pogled koriste identični nazivi metodama**. U `resources/views/admin/posts/index.blade.php` ekstendamo layout i kažemo da se radi o admin dijelu naše aplikacije kako bi layout na osnovu ove aplikacije mogao uključiti navigaciju. Ubacili smo neki `title` i u sekciju '`content`' neka HTML struktura glavnog sadržaja stranice. `<div class="container">` ide Bootstrap komponenta koja koristi fiksnu širinu koja se prilagođava ovisno o veličini ekrana. Cilj je da se sadržaj stranice pravilno centririra i ne "rastegne" preko cijele širine ekrana. Tu je HTML koji kreira strukturu za prikaz liste postova. U `@foreach` ćemo napraviti još neke prilagodbe.

Idemo u `app/Services/PostService.php` i prepraviti ćemo metodu `getAllPosts()`. Sada vraćamo `return Post::with('user')->get();`:

```
public function getAllPosts()
```

```
{  
  
    return Post::with('user')->where ('title', 'Bla')->get();  
}
```

Ovime dohvativamo sve postove gdje je title jednako `Bla`. Kada napravimo refresh, ne vidimo ništa jer nemamo takve postove.

U `resources/views/admin/posts/index.blade.php`, `@foreach` direktiva koja se pretvori u foreach petlju kaže da prođemo po `$post` varijabli je prazna. Ovdje sada ne znamo da li to radi i htjeli bi korisniku dati nekakvu informaciju. Možemo napraviti provjeru i ako je `$posts` prazna, možemo ispisati neki tekst. U Laravelu je zgodno kada u situacijama kada foreach moramo proći po nekoj kolekciji, možemo koristiti [@forelse direktivu](#). Ta petlja će proći po matrici ako nešto ima u njoj. Taj tip podatka je enumerable. Postoji i direktiva `@empty` gdje ako nema ništa u matrici ona će se aktivirati. Tako da ne moramo korisiti uslov i provjeravati da li je `$posts->isEmpty` itd. Puno je ljepešće napraviti:

```
<tbody>  
    @forelse ($posts as $post)  
        <tr>  
            <td>{{ $post->title }}</td>  
            <td>{{ $post->user->first_name }} {{ $post->user->last_name }}</td>  
            <td>{{ $post->created_at->format('d.m.Y. H:i') }}</td>  
            <td>  
            </td>  
        </tr>  
    @empty  
        <tr>  
            <td colspan="4">No posts found.</td>  
        </tr>  
    @endforelse  
</tbody>
```

Ovime nemamo dodatni `if` i unutar same petlje tj. prije ulaska u nju ispiše ako nema postova.

Sada kada smo Blade doradili, treba dodati action dio, gdje ćemo imati dugmad za editiranje i brisanje. To ćemo napraviti kasnije.

Kreiranje novog posta

Idemo napraviti create tj. probati kreirati novi post. Tu bi trebali uključiti i sliku. Laravel ima ugrađeno upravljanje sa datotekama. On to zove storage. Taj storage je agnostičan tj. može lokalno pohranjivati datoteke ili koristiti neke servise. Npr. od Amazon Simple Storage Service (S3). Trenutno je to integrirano u Laravelu. Neki drugi servisi se moraju ručno napisati ([DigitalOcean](#)). Kreirat ćemo novi post i za to nam treba nova forma u koju će korisnik unositi podatke u novi post. Slug ćemo provjeriti da li ćemo koristiti. Laravel ima svoj helper za slug. Provjerit ćemo da li je slug unique. Pogledat ćemo tablicu posts u laravel-blog bazi:

Table: posts

Columns:

| | |
|-------------------|------------------------|
| id | bigint(20) UN AI PK |
| title | varchar(255) |
| content | text |
| slug | varchar(255) |
| image | varchar(255) |
| user_id | bigint(20) UN |
| created_at | timestamp |
| updated_at | timestamp |
| deleted_at | timestamp |

Vidimo da je **slug** podebljan (boldan). To znači da je upaljen neki indeks nad njime. Odmah možemo prepostaviti da je **user_id** indeksiran radi relacije radi stranog ključa (engl. foreign key) dok je slug radi jedinstvenosti jer je upaljen unique nad njim. To znači da možemo imati post s istim naslovom. Ako iz naslova generiramo slug, što trenutno nije bio slučaj kada smo to radili. Slug je jedinstven zapis koji je potreban za SEO friendly URL-ove. Naš posao je omogućiti da se slug generira iz naslova posta. Teoretski, imamo mogućnost da title bude isti, dakle nije jedinstven. Slug mora biti jedinstven.

Slug ima unutar sebe funkcije. U uputstvu funkcije za rad s njima možemo vidjeti pod [Digging Deeper>Strings>Available Methods](#). Tu se nalazi kako puno metoda a jedna od njih je `Str::slug()`. Metoda `Str::slug` generira slug prilagođen URL-u iz zadanog stringa. Nažalost on ne gleda uniqueness, ostaje na nama da kada upisujemo slug u bazu to provjerimo. Možemo dodati na kraj -1, -2, -3 itd. Postoje za to već gotovi paketi. Takav paket je jako jednostavno implementirati. Za ovo trebamo title i korisnika koji je prijavljen u sistem – on je autor. Može post mijenjati i neki superadministrator ali on ne može napraviti post umjesto nekog drugog.

Otvoriti ćemo novu datoteku `resources/views/admin/posts/create.blade.php`:

```
@extends('layout', ['admin' => true])

@section('title', 'Create a new post')

@section('content')
    <div class="container">
        <h1>Create a new post</h1>
        @if (session('post-created'))
            <div class="alert alert-success">
                {{ session('post-created') }}
            </div>
        @endif
        <form method="post" action="{{ route('posts.store') }}"
        enctype="multipart/form-data">
```

```

@csrf


<label for="title">{{ __('forms.title') }}</label>
<input type="text" class="form-control" id="title" name="title"
value="{{ old('title') }}"
@error('title')
    <div class="alert alert-danger">{{ $message }}</div>
@enderror
</div>


<label for="content">Content</label>
<textarea class="form-control" id="content" name="content">{{ old('content') }}</textarea>
@error('content')
    <div class="alert alert-danger">{{ $message }}</div>
@enderror
</div>


<label for="image">Image</label>
<input type="file" class="form-control" id="image" name="image">
</div>
@error('image')
    <div class="alert alert-danger">{{ $message }}</div>
@enderror
<button type="submit" class="btn btn-primary">Create</button>
</form>
</div>
@endsection


```

Kao i kod drugih administrativnih stranica, ovaj template nasljeđuje osnovni layout (`layout.blade.php`). Parametar `'admin' => true` je tu kako bi omogućio rad administratora i njegovih funkcionalnosti. `@section` direktiva postavlja naslov stranice na "Create a new post". Unutar `@section ('content')` nalazi se HTML koji definira formu za kreiranje novog posta. `<div class="container">` je Bootstrap klasa koja postavlja širinu elementa na odgovarajući način, omogućavajući da sadržaj bude centriran i responzivan, ovisno o veličini ekranra. Ona kreira fiksni raspored za stranice. Da li postoji u sesiji vrijednost pod ključem `'post-created'` provjeravamo u idućem dijelu. Ako postoji, prikazuje se uspješna poruka korisniku unutar elementa sa klasom `alert alert-success`. U `session('post-created')` koristi se sesijska poruka koja je postavljena nakon uspješnog kreiranja posta.

U formi za kreiranje posta kažemo da će forma koristiti `POST` metodu za slanje podataka. Klasa `<div class="form-group">` grupira elemente forme (labela i unosa) u jedan blok. Koristi se za postizanje urednog rasporeda u formama. Obično stvara razmak između elemenata forme. U `action="{{ route('posts.store') }}"` podaci forme se šalju na rutu `posts.store`, koja je odgovorna za

pohranjivanje novog posta u bazu podataka. Vidimo poveznicu između naziva rute i naziva metode u kontroleru. Dakle ovo će biti preusmjerena na kontroler `app/Http/Controllers/PostController.php` na `store()` metodu. U `enctype="multipart/form-data"` vrijednost omogućava prijenos datoteka (slika) pomoću forme. Forma bez toga radi kao encoded tekst i kada odaberemo sliku praktično šalje tekst. S ovime omogućavamo formi da šalje podatke binarno tj. u formatu BASE64. Slika se pohrani u temp direktorij a zatim mi trebamo prenijeti u neki naš direktorij.

`@csrf` direktiva generira skriveni CSRF token koji štiti formu od napada Cross-Site Request Forgery.

U poljima za unos naslova `<label>` za polje title koristi `__('forms.title')` za prijevod naslova koristeći lang datoteke. `<input>` je polje za unos naslova posta. Ako je prethodni unos neuspješan, `old('title')` vraća vrijednost koja je unešena u prethodnom pokušaju. `@error('title')` direktiva ako postoji greška u unosu naslova (npr. validacija ne uspije), prikazat će se poruku o grešci unutar `alert alert-danger`.

Nakon toga imamo polje za unos sadržaja gdje je `<textarea>` polje za unos sadržaja posta. Ponovno, koristimo `old('content')` kako bi prikazali prethodno unešeni sadržaj u slučaju greške. `@error('content')` prikazuje grešku ako validacija sadržaja ne uspije. Ako želimo implementirati rich text editore tj. WYSIWYG editor, potrebno je poznavanje Java Script-a. Za ovo možemo koristiti [TinyMCE](#) ili [CKEditor](#).

U polju za unos slike `<input type="file">` omogućava odabir slike koju korisnik želi upload-ovati. Ovdje nije postavljena opcija `old('image')`, jer se datoteke ne pamte kao tekstualni input. `@error('image')` prikazuje grešku ako upload slike ne uspije. Ako image nije nullable polje, onda validator ne smije ne validirati tj. da ga nije briga ako slike nema. Dakle validator mora to otkriti. Tako je i za `title` i za `content`. Zadnje polje je polje zareške, bez njega neće validator imati gdje ispisati.

Bootstrap klase `form-control` stilizira unose (kao što su tekstualna polja, tekst area polja, i upload polja) tako da zauzimaju cijelu širinu roditeljskog elementa i izgledaju uniformno. Omogućavaju responzivne i estetski ujednačene inpute, čime se olakšava kreiranje formi. Bootstrap `btn` stilizira dugme prema osnovnim stilovima Bootstrap-a. `btn-primary` klase dodaje stilizaciju primarnog dugmeta (obično plave boje), što signalizira važnu akciju. Bootstrap alert klase je osnovna klasa za prikazivanje poruka upozorenja ili statusa u obliku obojenih okvira. `alert-success` i `alert-danger` mijenjaju boju okvira, `alert-success` koristi se za prikazivanje uspješnih poruka (zeleno), `alert-danger` koristi se za prikazivanje poruka o greškama (crveno).

Dugmetom za slanje forme korisnik šalje podatke forme.

Idemo na kontroler `app/Http/Controllers/PostController.php` popuniti metodu `create()`:

```
public function create()
{
    return view('admin.posts.create');
}
```

To je dovoljno za prikazati formu.

[Dashboard](#) | [Posts](#) | [Logout](#) |

Create a new post

Title

Content

Image

Choose File No file chosen

Create

Idemo vidjeti što se dešava kada korisnik popuni formu i stisne **Create** dugme. Prvo što gledamo na `routes/web.php` postoji li neki middleware. Možemo vidjeti da za sve rute postoji middleware AuthMiddleware. Korisnik za sada mora biti prijavljen.

```
Route::group(['middleware' => AuthMiddleware::class], function () {
```

Nakon toga request ide na naš custom request `app/Http/Requests/StorePostRequest.php`. Tu ćemo metodu `authorize()` odmah prebaciti u `true`:

```
public function authorize(): bool
{
    return true;
}
```

Nećemo raditi autorizaciju, pustit ćemo ga samo dalje. Nad samim requestom nikada nećemo raditi provjeru. To ćemo kasnije raditi s policy, koji ćemo definirati na resouce post, tko što s njim smije raditi (vidjeti sve, vidjeti pojedinačno, kreirati, raditi update) . Nakon toga pišemo `rules()` tj. pravila. Obavezno moramo vidjeti u bazi kako je definirano. Ako validator ne validira i prođe podatak a u bazi je drugačije, desit će se greška. Pogledati u migracijama (ovdje u `xxx_create_posts_table.php`) i vidjet ćemo da je `slug unique` a `image nullable` tj. optionalan. Imali smo i tablicu s izmjenom (`xxx_change_posts_table.php`), pa pogledajmo i tamo. Ubacili smo `softDeletes()`. Idemo napisati metodu `rules()`:

```
public function rules(): array
{
    return [
        'title' => ['required', 'string', 'max:255'],
        'content' => ['required', 'string'],
    ];
}
```

```
    'image' => ['image', 'max:5000'],
];
}
```

Vidimo da je `'image'` definiran kao `'image'`, da ne bi uvalio drugu datoteku osim slike (jpg, jpeg, png, bmp, gif, svg i webp). Moguće je i definirati samo određeni tip s: `=> 'mimes:jpg,bmp,png'`. S `'max:5000'` smo ograničili sliku na 5 Mb.

Postoji paket [Intervention Image](#) koji je PHP biblioteka za obradu slika. Omogućuje jednostavan način za uređivanje slika i podržava dvije najčešće PHP biblioteke za obradu slika **GD Library** i **Imagick**. On nije posebno za Laravel ali postoji paket image-laravel koji implementira ovo u Laravel. Kako ga integrirati pogledati na <https://image.intervention.io/v3/introduction/frameworks>

Idemo u `app/Http/Controllers/PostController.php` i metodu `store()` staviti ćemo:

```
public function store(StorePostRequest $request)
{
    dd ($request);

    return redirect()->route('posts.index');
```

Pogledajmo kako validator radi:

Create a new post

Title

The title field is required.

Content

The content field is required.

Image

 Choose File No file chosen

The image field must be an image.

Create

Odabrao sam datoteku koja nije slika i validator je prepoznao prazno polje naslova, sadržaja i slike. Idemo provjeriti da li radi s ispravnim podacima:

```
App\Http\Requests\StorePostRequest {#1242 ▶ // app\Http\Controllers\PostController.php:52
+attributes: Symfony\Component\HttpFoundation\RequestBag {#1244 ▶}
+request: Symfony\Component\HttpFoundation\RequestBag {#1241 ▶}
#parameters: array:3 [▼
    "_token" => "twygLYODZQUjzVPLWf1R2zAwkEDkO10o35R6WiZe"
    "title" => "Proba"
    "content" => "Ovo je sadržaj"
]
}
+query: Symfony\Component\HttpFoundation\RequestBag {#1243 ▶}
+server: Symfony\Component\HttpFoundation\RequestBag {#1247 ▶}
+files: Symfony\Component\HttpFoundation\RequestBag {#1246 ▶}
#parameters: array:1 [▼
    "image" => Symfony\Component\HttpFoundation\File\UploadedFile {#41 ▶
        -test: false
        -originalName: "IMG_20240731_104032.jpg"
        -mimeType: "image/jpeg"
        -error: 0
        -originalPath: "IMG_20240731_104032.jpg"
        path: "C:\xampp\tmp"
        filename: "phpE7E1.tmp"
        basename: "phpE7E1.tmp"
        pathname: "C:\xampp\tmp\phpE7E1.tmp"
        extension: "tmp"
        realPath: "C:\xampp\tmp\phpE7E1.tmp"
        aTime: 2024-10-10 23:00:11
        mTime: 2024-10-10 23:00:10
        cTime: 2024-10-10 23:00:10
        inode: 3377699721247756
        size: 3483886
        perms: 0100666
        owner: 0
        group: 0
        type: "file"
        writable: true
        readable: true
        executable: false
        file: true
        dir: false
        link: false
        linkTarget: "C:\xampp\tmp\phpE7E1.tmp"
    }
]
}
+cookies: Symfony\Component\HttpFoundation\RequestBag {#1245 ▶}
+headers: Symfony\Component\HttpFoundation\RequestBag {#1248 ▶}
#content: ""
#languages: null
#Charsets: null
#encodings: null
#acceptableContentTypes: null
#pathInfo: null
#requestUri: null
#baseUrl: null
#basePath: null
#method: null
#format: null
#session: Illuminate\Session\SessionManager {#1240 ▶}
#locale: "en"
#defaultLocale: "en"
-preferredFormat: null
-isHostValid: true
-isForwardedValid: true
-isSafeContentPreferred: ? bool
-trustedValuesCache: []
-isIISRewrite: false
#json: Symfony\Component\HttpFoundation\RequestBag {#1249 ▶}
#convertedFiles: array:1 [▶]
#userResolver: Closure($guard = null) {#1162 ▶}
#routeResolver: Closure() {#1171 ▶}
#container: Illuminate\Foundation\Application {#4 ...44}
#redirector: Illuminate\Redirector {#1239 ▶}
#redirect: null
#redirectRoute: null
#redirectAction: null
```

U request imamo tri parametra. Vidimo da slike nema nego je ona izdvojena u drugi direktorij. Formu možemo kreirati i bez slike, pa ako pošaljemo vidjet ćemo da će files biti empty dok će u requestu i dalje biti popunjeno. Validator ne validira polje slike.

```
App\Models\User {#1232 ▼ // app\Http\Controllers\PostController.php:52
  #connection: "mysql"
  #table: "users"
  #primaryKey: "id"
  #keyType: "int"
  +incrementing: true
  #with: []
  #withCount: []
  +preventsLazyLoading: false
  #perPage: 15
  +exists: true
  +wasRecentlyCreated: false
  #escapeWhenCastingToString: false
  #attributes: array:11 [▼
    "id" => 21
    "first_name" => "Jarred"
    "last_name" => "Carroll"
    "email" => "juliana.abernathy@example.net"
    "email_verified_at" => "2024-09-24 07:59:38"
    "password" => "$2y$12$W38TvedJQsW.ur0hvfvNVguNcI/3CH1.xqgCM3K0bYzibG1MK.dxma"
    "remember_token" => "mWxkXQgxcbcOuVJzU6Ei02C2sgwwwPoRhGMTZB0ebcUYyLpK6AiHskatss091"
    "created_at" => "2024-09-24 07:59:43"
    "updated_at" => "2024-09-24 07:59:43"
    "deleted_at" => null
    "phone" => null
  ]
  #original: array:11 [▶]
  #changes: []
  #casts: array:2 [▶]
  #classCastCache: []
  #attributeCastCache: []
  #dateFormat: null
  #appends: []
  #dispatchesEvents: []
  #observables: []
  #relations: []
  #touches: []
  +timestamps: true
  +usesUniqueIds: false
  #hidden: array:2 [▶]
  #visible: []
  #fillable: array:5 [▶]
  #guarded: array:1 [▶]
  #authPasswordName: "password"
  #rememberTokenName: "remember_token"
}
}
```

Sada kada smo riješili validator, pozvat ćemo servis [app\Http\Controllers\PostController.php](#) i proslijediti podatke koji su nam potrebni ili ćemo izvući cijeli request. Idemo vidjeti što nam treba s `validated()`:

```
public function store(StorePostRequest $request)
{
    dd ($request->validated());
}
```

```
array:3 [▼ // app\Http\Controllers\PostController.php:52
  "title" => "Proba"
  "content" => "Ovo je sadržaj"
  "image" => Illuminate\UploadedImage {#1261 ▼
    -test: false
    -originalName: "IMG_20240731_104032.jpg"
    -mimeType: "image/jpeg"
    -error: 0
    -originalPath: "IMG_20240731_104032.jpg"
    #hashName: null
    path: "C:\xampp\tmp"
    filename: "php6D4F.tmp"
    basename: "php6D4F.tmp"
    pathname: "C:\xampp\tmp\php6D4F.tmp"
    extension: "tmp"
    realPath: "C:\xampp\tmp\php6D4F.tmp"
    aTime: 2024-10-10 23:14:56
    mTime: 2024-10-10 23:14:56
    cTime: 2024-10-10 23:14:56
    inode: 10977524091845430
    size: 3483886
    perms: 0100666
    owner: 0
    group: 0
    type: "file"
    writable: true
    readable: true
    executable: false
    file: true
    dir: false
    link: false
    linkTarget: "C:\xampp\tmp\php6D4F.tmp"
  }
]
```

Ako na kontroleru koristimo izvlačenje podataka validiranih iz requesta, i šaljemo ih servisu, moramo biti svjesni da tamo iz requesta doći do usera, nego ćemo morati koristiti `auth` helper. To je samo zato što smo poslali samo validated podatke. Imamo mogućnost da izvučemo request usera i dobit ćemo usera koji je prijavljen u sistem i slati odatle. Predavač kaže da je bolje da to servis radi. Servis će izvući što mu treba i kontroler će reći:

```
public function store(StorePostRequest $request)
{
    $this->postService->storePost($request);
    return redirect()->route('posts.index');
}
```

Dakle ovime posljeđujemo cijeli `$request`. Na nama je da odlučimo gdje ćemo vratiti korisnika – možemo ga vratiti na index a možemo ga vratiti na create, kada se izkreira post.

Idemo kreirati servis koji zaprima request i vidjeti što nam sve treba kako bi mogli pohraniti post u bazu. Trebat će nam i fotografija. Laravel ima file system za upravljanje datotekama, odnosno storage za kompletno upravljanje datotekama. Idemo na `app/Service/Postservice.php`:

```
<?php

namespace App\Services;
```

```
use App\Models\Post;
use Illuminate\Foundation\Http\FormRequest;

class PostService
{
    public function getAllPosts()
    {
        return Post::with('user') ->get();
    }

    public function storePost($request)
    {

    }
}
```

Metoda `getAllPosts()` vraća sve postove `storePost()` služi za spremanje novog posta.

Predavač predlaže strogu tipizaciju s `storePost(FormRequest $request)` zato što svi request-ovi koji idu iz forme prolaze kroz ovaj request a praktično svaka naša forma ekstenda `Request`, praktično na kraju ekstenda i ovaj `FormRequest`.

`Illuminate\Foundation\Http\FormRequest`

Request represents an HTTP request.

```
<?php
class FormRequest extends Request implements ValidatesWhenResolved
{
    use ValidatesWhenResolvedTrait;
}
```

The methods dealing with URL accept / return a raw path (% encoded):

- `get basePath`
- `get baseUrl`
- `get PathInfo`
- `get Request Uri`

`FormRequest $request)` Tomislav Kešcec, 4 weeks ago • Predavanje 6

```
storePost(Request $request)
{
    $data = $request->validated()
}
```

Poanta je da iz tog requesta možemo izvući podatke s `$request->validated()`. Metoda `validated()` je dostupna samo unutar klase `FormRequest` i služi za vraćanje validiranih podataka prema pravilima definiranim unutar `FormRequest`, onda možemo napisati:

```
public function storePost(FormRequest $request)
```

```
{
    $data = $request->validated();
}
```

Izvukli smo nekakve podatke van koji su validirani. Prvo što trebamo s njima raditi je otvoriti slug. U \$data možemo ubaciti slug sa helperom:

```
public function storePost(FormRequest $request)
{
    $data = $request->validated();
    $data['slug'] = Str::slug($data['title']);
    $data['user_id'] = $request->user()->getAuthIdentifier();
}
```

Mogli smo izvući `user_id` i s `$data['user_id'] = $request->user()->id()` ali možda je bolje ovako s `getAuthIdentifier()` jer u prvom pristupu idemo da se naš stupac u tablici zove `id`. Ako se držimo Laravelove konvencije, to je u redu ali predavač to ne preporuča ako se promjeni primarni ključ. `getAuthIdentifier()` daje jedinstveni identifikator. Možda je ipak najbolje koristiti `getKey()` metodu koja dobiva vrijednost primarnog ključa modela. Vraća vrijednost primarnog ključa modela (obično `id`), što može biti bilo koji atribut koji je definiran kao primarni ključ u bazi podataka. Svaki model na sebi ima metodu `getKey()`.

Idemo u kontroleru `app\Http\Controllers\PostController.php` staviti u store metodu na početak `dd($request->user()->getKey())` da to i vidimo.

21 // app\Http\Controllers\PostController.php:52

Dobit ćemo vrijednost primarnog ključa. Radi na sva tri načina. Kada smo to utvrdili obrišimo `dd` red.

Sada smo si sami napunili podatke `$data`. `slug` ćemo puniti uz pomoć paketa jer imamo problem jedinstvenosti, koja se ovdje ne prati. Ostaje nam još datoteka sa slikom koju trebamo prenijeti i pohraniti, dobiti putanju i to zapisati u bazu.

Ponovimo. Imamo formu u `resources/views/admin/posts/create.blade.php` i u njemu tri polja, te validator `app\Http\Requests\StorePostRequest.php`, u njemu validator requesta `rules()` za svako polje, dolazak u kontroler `app\Http\Controllers\PostController.php`. Kontroler taj request šalje servisu `app\Services\PostService.php`, servis obrađuje request i izvlači validirane podatke i vraća nazad neku informaciju (kompletan post koji je kreiran, ID posta ili boolean true).

Trebamo početi koristiti try-cache blokove da hendlamo potencialne except-ione. Potencijalni problemi se pojavljuju kada zapisujemo podatke u različite tablice. Tada se desi da se kod raspadne jer se zapis ne napravi do kraja a ne desi se roll back. Za to koristimo **transakcije**.

Prva stvar koju ćemo riješiti je **try-catch blok**.

```
public function storePost(FormRequest $request)
{
    try {
        $data = $request->validated();
        $data['slug'] = Str::slug($data['title']);
        $data['user_id'] = $request->user()->getAuthIdentifier();
    } catch (\Exception $e) {
        throw \Exception($e->getMessage());
    }
}
```

Na nama je što ćemo i kako hendlati. Npr možemo logirati tu grešku. Isto tako `\Exception` koji uhvatimo, možemo reći `use \Exception` iznad.

```
public function storePost(FormRequest $request)
{
    try {
        $data = $request->validated();
        $data['slug'] = Str::slug($data['title']);
        $data['user_id'] = $request->user()->getAuthIdentifier();
    } catch (\Exception $e) {
        Log::error($e->getMessage());
    }
}
```

Vidjet ćemo da će u nekom trenutku zbog slag-a doći do problema. Imamo podatke koje smo prikupili. Želimo vidjeti da li na `$data` imamo sliku. Ako imamo, onda su podaci validirani. Sa time možemo raditi s datotekama kao i na request-u. Idemo vidjeti što je unutra. Iznad `catch` stavit ćemo `dd($data)`:

```
public function storePost(FormRequest $request)
{
    try {
        $data = $request->validated();
        $data['slug'] = Str::slug($data['title']);
        $data['user_id'] = $request->user()->getAuthIdentifier();
        dd($data);
    } catch (\Exception $e) {
        Log::error($e->getMessage());
    }
}
```

Dakle vidimo image i kompletnu informaciju o datoteci.

```
array:5 [▼ // app\Services\PostService.php:38
  "title" => "abc"
  "content" => "abcdef"
  "image" => Illuminate\UploadedImage {#1261 ▶}
  "user_id" => 21
  "slug" => "abc"
]
```

Mogli smo umjesto `dd($data);` reći `dd($data['image']);` i time bi dobili objekt UploadedFile iz kojeg dalje možemo koristiti njegove metode itd.

```
Illuminate\Http\UploadedFile {#1261 ▼ // app\Services\PostService.php:38
  -test: false
  -originalName: "k.png"
  -mimeType: "image/png"
  -error: 0
  -originalPath: "k.png"
  #hashName: null
  path: "C:\xampp\tmp"
  filename: "phpE162.tmp"
  basename: "phpE162.tmp"
  pathname: "C:\xampp\tmp\phpE162.tmp"
  extension: "tmp"
  realPath: "C:\xampp\tmp\phpE162.tmp"
  eTime: 2024-10-11 16:47:51
  mTime: 2024-10-11 16:47:51
  cTime: 2024-10-11 16:47:51
  inode: 4785074604212977
  size: 545975
  perms: 0100666
  owner: 0
  group: 0
  type: "file"
  writable: true
  readable: true
  executable: false
  file: true
  dir: false
  link: false
  linkTarget: "C:\xampp\tmp\phpE162.tmp"
}
```

Međutim ne možemo na koji to iz request-a možemo provjeriti, koristiti `$request->hasFile()`, jer ova metoda gleda da li nad request-om postoji slika (image). Ako postoji, možemo ga izvući i pohraniti. Kada ga pohranimo, pospremljeni path možemo staviti u image.

```
if ($request->hasFile('image')) {
    $data['image'] = $request->file('image')->store('images',
'public');
}
```

Konkretno, `hasFile()` metoda provjerava postoji li datoteka koja je poslana putem HTTP POST zahtjeva pomoću HTML forme s `enctype="multipart/form-data"`. U ovoj `storePost` metodi, koristit ćemo `hasFile()` da bi smo provjerili da li je korisnik prilikom kreiranja posta učitao datoteku (tj. sliku). Laravel koristi Symfony komponentu za rukovanje HTTP zahtjevima, pa `hasFile()` u stvari pristupa objektu koji predstavlja formu i provjerava `$_FILES` matricu u PHP-u (gdje se pohranjuju informacije o učitanim datotekama). Ako korisnik nije priložio datoteku ili je priložena datoteka prazna, `hasFile()` vraća `false`. Ako je datoteka uspješno poslana pomoću forme, metoda vraća `true`.

Ako postoji datoteka slike, tj. prisutna je, koristi se metoda `file()` da bi se pristupilo toj datoteci, a zatim se pohranjuje pomoću `store('images', 'public')` u direktorij `public/images`, a putanja slike se spremi u bazu.

Pokrenimo [Postman-a](#). Nije bitno da se prijavimo ali možemo dijeli workspace-ove sa projektima na kojima radimo, tako da imamo uvijek dostupne projekte.

Odabrat ćemo POST metodu i <http://localhost:8000/admin/posts>. U body ćemo staviti form-data.

The screenshot shows the Postman application interface. At the top, there are tabs for Home, Workspaces, and API Network. Below that, a search bar and various settings icons. The main area shows a list of requests. One request is selected, showing the URL <http://localhost:8000/admin/posts>. The method is set to POST. Under the Body tab, the content type is set to form-data. A table is visible with columns for Key, Value, Description, and Bulk Edit. The table has two rows: one for Key and one for Value. The Response section is collapsed at the bottom.

Ako ne pošaljemo ništa i pritisnemo dugme Send. Dobit ćemo u Preview grešku [419 Page Expired](#).

The screenshot shows the Postman interface with the Preview tab selected. At the top, there are tabs for Body, Cookies (1), Headers (7), and Test Results. On the right, there is a status indicator showing a globe icon and the text "419 unk". Below the tabs, there are buttons for Pretty, Raw, Preview, and Visualize. The main area displays the error message "419 | PAGE EXPIRED".

Nemamo CSRF token i ne možemo krivotvoriti prilaz. Preko postmana možemo poslati što želimo, ne moramo koristiti formu.

Sada imamo sliku i trebali bi neki proces pohrane na kompjuter ili van na neki AWS. Pogledajmo u dokumentaciju gdje se to sve može podesiti. To je u Digging Deeper>[File Storage](#). Ima toga dosta oko pohrane datoteka. Konfiguracija se nalazi u `config/filesystems.php`. Ako otvorimo tu datoteku vidjet ćemo unaprijed definirane ključeve za `local`, `public` i `s3`. Pod ključem `local` imam `'root' => storage_path('app')`, što znači da će spremati u `storage/app` a ako pod ključem `public` imam `'root' => storage_path('app/public')`, što znači da će spremati u direktorij

`storage/app/public`. Neke stvari ne želimo da su dostupne javno. Kada pohranimo sliku, možemo doći do nje tako da u browser unesemo putanju. Ona je javno dostupna. Ako ne želimo da datoteke budu javno dostupne, onda pohranjujemo u `storage/app`. Direktorij `storage/framework` namijenjen je za sam framework. Tu ne bi trebali nešto mijenjati. U `storage/logs` nalaze se svi logovi.

Ono što Laravel ima zgodno je direktorij `public` je jedini javno izložen na serveru. Kada smo govorili o Apache serveru onda smo naveli root mapu servera (sve izvan toga je nedostupno kroz browser). Bio je to direktorij `xampp/htdocs/Algebra`. Tako je u biti taj `public` direktorij root i sve van tog direktorija nije dostupno korisnicima.

Razlog ovome je što pokušavamo sakriti što više jer je PHP interpreter i sve je vidljivo.

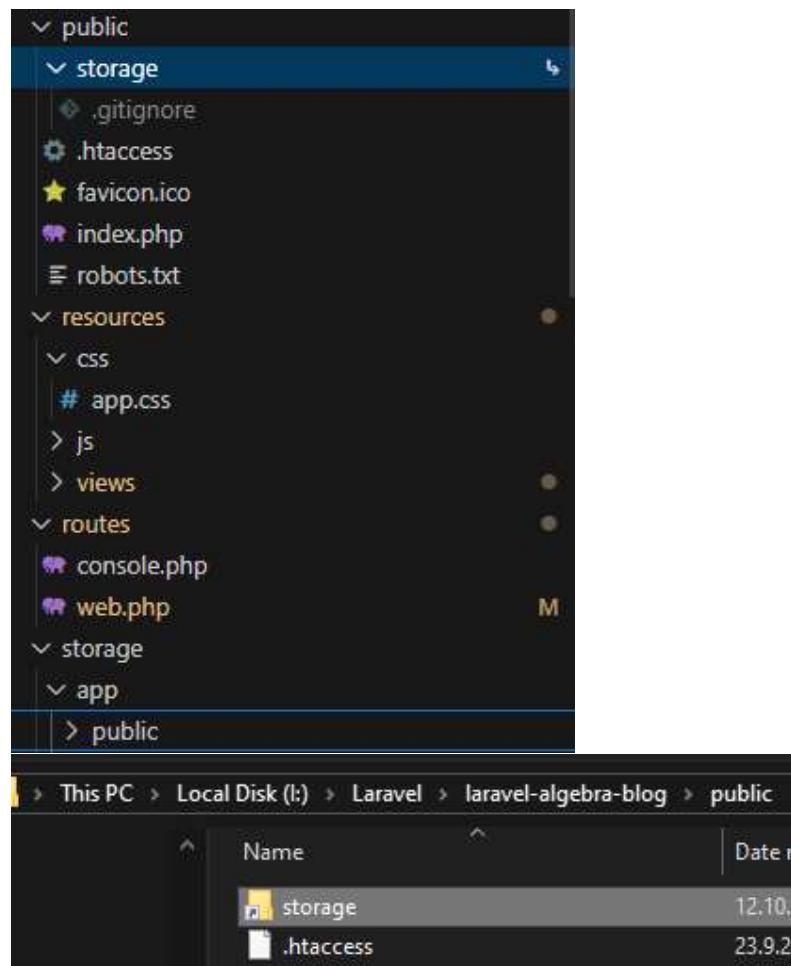
Poanta je da slike koje želimo prikazati u browseru, onda imamo tom `public` dijelu. Pogledamo li `storage`, on nije u `public` dijelu, ali u `storage` direktoriju postoji poddirektorij `app/public`. Da bi to mogli od tamo koristiti moramo koristiti **sym linkove ili symbolic linkove**. Direktorij `public` i poddirektorij `storage`, uz pomoć sym linka se veže s direktorijem `storage/app/public`. Kada mi pohranimo naše fotografije u `storage/app/public` one će biti vidljive i u `public` direktoriju.

Za kreiranje simboličkog linka možemo korisiti:

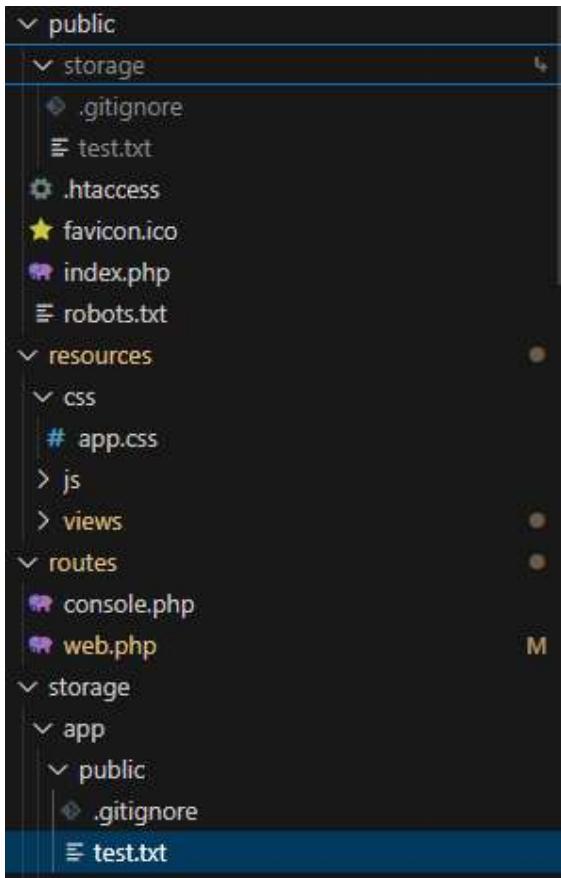
```
php artisan storage:link
```

```
$ php artisan storage:link
[INFO] The [I:\Laravel\laravel-algebra-blog\public\storage] link has been connected to [I:\Laravel\laravel-algebra-blog\storage\app\public].
Administrator@DESKTOP-4J5DF41 MINGW64 /i/Laravel/laravel-algebra-blog (master)
$ |
```

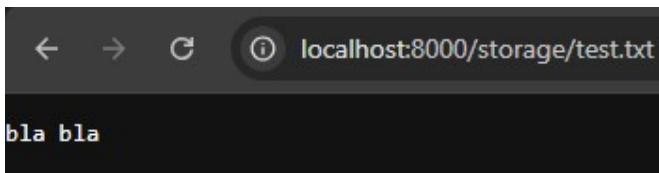
Vidimo u `public` direktoriju kreirao se symbolic link `storage` i vezao ga je za direktorij `storage/app/public`.



Ako u direktorij `storage/app/public` ubacimo `test.txt` datoteku, vidjet ćemo je i u `public/storage`:



To znači da do njega možemo doći i kroz browser.



Da nema sym linka ne bi mogli pristupiti datoteci. Tako ćemo pohranjivati slike.

Sada kada znamo gdje ćemo pohranjivati, trebamo vidjeti koji potencijalni problem može postojati s datotekama. To mogu biti ista imena. U Windowsima ili drugim operativnim sistemima, dodaje se - copy. Kada dodate sliku s istim imenom, postojaće se pregaziti. Dakle ne možemo se osloniti na ime.

Vratimo se u `app/Services/PostService.php` i metodu `storePost` kojom kreiramo novi post. Dodali smo provjeru da li postoji datoteka slike:

```
public function storePost(WebRequest $request)
{
    try {
        $data = $request->validated();
        $data['slug'] = Str::slug($data['title']);
        $data['user_id'] = $request->user()->getAuthIdentifier();
```

```

if ($request->hasFile('image')) {
    $data['image'] = $request->file('image')->store('images', 'public');
}
dd($data);
} catch (\Exception $e) {
    Log::error($e->getMessage());
}
}

```

`$request->file('image')` izvuče datoteku iz `$request`. Ako postoji, pohranjuje se pomoću `store('images', 'public')` u direktorij `images` i na disk `public`, a putanja slike se spremi u bazu. `store` će vratiti putanju gdje je pohranio datoteku ili će vratiti `false`. `store` će puknuti ako nema datoteke, zato postoji uslov iznad.

Kada pokrenemo program, dobijemo:

```

array:5 [▼ // app\Services\PostService.php:42
  "title" => "abc"
  "content" => "abcdef"
  "image" => "images/UXUvGh81Nv69mng9U9YKvXWEhbkNC1SbVsvz6bMG.jpg"
  "user_id" => 21
  "slug" => "abc"
]

```

Ne uzima original ime nego generira neki set znakova. Slika se sada nalazi u `public/storage/images`.

Zadnji korak je da sve pohranimo u bazu preko modela. Metoda `storePost` izgleda ovako:

```

public function storePost(FormRequest $request)
{
    try {
        $data = $request->validated();
        $data['user_id'] = $request->user()->getKey();
        $data['slug'] = Str::slug($data['title']);
        dd($data['image']);

        if ($request->hasFile('image')) {
            $data['image'] = $request->file('image')->store('images',
'public');
        }

        Post::create($data);

        return true;
    } catch (\Exception $e) {

```

```
        Log::error($e->getMessage());
        return false;
    }
}
```

Probat ćemo pohraniti. Ne pohranjuje ali ne dojavljuje ni grešku. Ako pogledamo u `Logs` direktorij, zadnji zapis vidimo grešku da polje `'slug'` nema default vrijednost.

Uz pomoć mass assignment, štitimo model da ne ubacimo podatak u neko polje (stupac) u koje ne želimo da se to dogodi. Ako pogledamo `app/Models/Post.php` nemamo polje slug i zato se model štiti:

```
protected $fillable = [
    'title',
    'content',
    'image',
];
```

Nema slug i zato se model štiti. `slug` smo ubacili u podatke koje smo poslali modelu na metodu create, a create je rekao samo fillable atributi a ostale je odbacio van. Baza je rekla da ne može zapisati jer slug nema default vrijednost.

Dovoljno je da na modelu dodamo polje `'slug'`. Isti problem je i s `'user_id'`, pa ćemo i njega dodati. Sada `$fillable` u `app/Models/Post.php` izgleda ovako:

```
protected $fillable = [
    'title',
    'content',
    'image',
    'slug',
    'user_id',
];
```

| Title | Author | Published | Actions |
|-------------------------------------------------------------|----------------|-------------------|---------|
| abc | Jarred Carroll | 12.10.2024. 23:31 | |
| Quod placeat placeat rem debit | Corene Nader | 24.09.2024. 07:59 | |
| Aut rem laboriosam quis quaerat dolore accusamus. | Martina Rath | 24.09.2024. 07:59 | |
| Dolor simili | Corene Nader | 24.09.2024. 07:59 | |
| Aspernatur veritatis ab quisquam est placeat eum provident. | Davin Paucek | 24.09.2024. 07:59 | |

U bazi, tablica posts nakon upisa izgleda ovako:

| id | title | content | slug | image | user_id | created_at | updated_at | deleted_at |
|----|------------------------------------------------|-----------------------------------------------------------------------------------------------------|----------------------------------------------------|---------------------------------------------|---------------------|---------------------|---------------------|---------------------|
| 28 | Qui sapiente recusandae quis dolorum illo sit. | Voluptatem omnis voluptatem nesciunt temporibus... et accusantium ullam-et-nesciunt-omnis-dolore... | https://via.placeholder.com/640x480.png/0088... | 27 | 2024-09-24 07:59:43 | 2024-09-24 07:59:43 | | NULL |
| 29 | Cumque cumque reiciendis ex perferendis. | Rerum labore ab ea modi. Molestiae similique illo... | https://via.placeholder.com/640x480.png/0000... | 23 | 2024-09-24 07:59:43 | 2024-09-24 07:59:43 | | NULL |
| 30 | Qui ab qui sit consequatur unde sequi qui. | Eius dolore omnis possimus minus quaerat non qu... | similique-voluptatem-maiores-voluptatem-velit-t... | 30 | 2024-09-24 07:59:43 | 2024-09-24 07:59:43 | | NULL |
| 31 | abc | abcdef | abc | images/k0hJeS0KENd6stsbtJ4Jo4DG7CZtM7oRf... | 21 | 2024-10-12 23:31:07 | 2024-10-12 23:31:07 | 2024-10-12 23:31:07 |

| image | user_id |
|---------------------------------------------------------------------------------------------------------------|---------|
| https://via.placeholder.com/640x480.png/0088... | 27 |
| https://via.placeholder.com/640x480.png/0000... | 23 |
| https://via.placeholder.com/640x480.png/00dd... | 30 |
| images/k0hJeS0KENd6stsbtJ4Jo4DG7CZtM7oRf... | 21 |

Iako sve izgleda OK, imamo problem u kršenju ograničenja ([Integrity constraint violation](#)) a to je unique polje za slug. To znači da smo imali dva puta isto. Korisniku bi htjeli nešto vratiti. To može biti nakon zapisivanja u log:

```
public function storePost(WebRequest $request)
{
    try {
        $data = $request->validated();
        $data['user_id'] = $request->user()->getKey();
        $data['slug'] = Str::slug($data['title']);

        if ($request->hasFile('image')) {
            $data['image'] = $request->file('image')->store('images',
'public');
        }
    }
}
```

```

        Post::create($data);

    } catch (\Exception $e) {
        Log::error($e->getMessage());
        return redirect()->back()->with('error', 'An error occurred while
saving the post');
    }
}

```

Možemo uhvatiti taj error na `resources/views/adimin/posts/create.blade.php`, i ispisati tu grešku. Ispod `<h1>` taga napišemo:

```

@error('error')
<div class="alert alert-danger">{{ $message }}</div>
@enderror

```

Kada se u `app/Services/PostService` dogodi `catch`, zapišemo sebi što se dogodilo, da imamo konkretnu informaciju a korisniku kažemo da se greška dogodila.

Ovako kako smo napravili iz `storePost()` metode u servisu `app/Services/PostService.php` ide `redirect()`, a servis treba vratiti `true` ili `false`, tako da `storePost` moramo preformulirati:

```

public function storePost(FormRequest $request)
{
    try {
        $data = $request->validated();
        $data['user_id'] = $request->user()->getKey();
        $data['slug'] = Str::slug($data['title']);

        if ($request->hasFile('image')) {
            $data['image'] = $request->file('image')->store('images',
'public');
        }

        Post::create($data);

        return true;
    } catch (\Exception $e) {
        Log::error($e->getMessage());
        return false;
    }
}

```

Na temelju toga što servis vratio (`true` ili `false`), u kontroleru `app/Http/Controllers/PostController.php` možemo iskontrolirati:

```
public function store(StorePostRequest $request)
{
    if (!$this->postService->storePost($request)) {
        return redirect()->back()->with('post-created', 'Post not created');
    }

    return redirect()->route('posts.index');
}
```

Sada kada pogledamo servis `app/Services/PostService.php` u `try` bloku vratí `true` ako je uspješno napravio, a ako uđe u `catch` vratí `false`, došlo je do greške i nismo uspješno kreirali post. Kontroler `app/Http/Controllers/PostController.php` će tu informaciju primiti. Ako `storePost()` vratí `true`, uspješno je pohranio negacija `!` će okrenuti u `false` i vratiti će ga s `redirect()`. Ako nije uspio pohraniti, negacija će ga pretvoriti u `true` i vratiti će ga s greškom. Greška koja ne dolazi iz validatora izvučena je s `with('post-created', 'Post not created');`.

Ako pokrenemo, ovo ne radi jer direktiva `@error` je samo za validator. Iz `create.blade.php` izbacujem:

```
@error('post')
    <div class="alert alert-danger">{{$message}}</div>
@enderror
```

Tu mora doći:

```
@if (session('post-created'))
    <div class="alert alert-success">
        {{ session('post-created') }}
    </div>
@endif
```

Idemo na `app/Http/Controllers/PostController.php` i mijenjamo metodu `store()`, tako da with koristi `'post-created'`:

```
public function store(StorePostRequest $request)
{
    // TODO: uncomment this line if method create is changed in the policy
    // Gate::authorize('create', Post::class);

    if (!$this->postService->storePost($request)) {
        return redirect()->back()->with('post-created', 'Post not created');
    }
}
```

```
    }

    return redirect()->route('posts.index');
}
```

Dakle ako probamo koristi isti slug, dojavit će grešku. U logu je zapisano zašto se greška dogodila. Ne možemo ovo ovako ostaviti jer postoji mogućnost da se to dogodi. Idemo to rješiti.

Ponovimo. Kontroler `app/Http/Controllers/PostController.php` je zaprimio zahtjev i u `StorePostRequest` se odradila validacija. Šaljemo servisu i servis nam vrati da li je uspio ili nije uspio odraditi. Ako nije uspio mi korisniku odgovaramo s porukom da Post nije kreiran. Ako je uspio, korisnika vraćamo na index stranicu.

Više ne želimo ručno raditi na servisu sa slug. To znači da ćemo iz `app/Services/PostService.php` ukloniti red gdje se spominje `'slug'`:

```
//$data['slug'] = Str::slug($data['title']);
```

Maknut ćemo ga iz `app/Models/Post.php` iz `$fillable`:

```
protected $fillable = [
    'title',
    'content',
    'image',
    'user_id',
];
```

To ćemo napraviti zato što ne želimo da se može ubaciti iz requesta slug prilikom kreiranja posta. To znači da i da nismo izkomentirali red sa `slug` u `app/Services/PostService.php`, red će biti ignoriran.

Automatski ćemo riješiti tako da na google potražimo laravel sluggable. Predavač kaže da je uvijek koristio ovaj <https://github.com/cviebrock/eloquent-sluggable>. U uputstvu niže možemo vidjeti da ga je potrebno instalirati s:

```
composer require cviebrock/eloquent-sluggable
```

Potrebno je prebaciti iz tog vendor-a pokrenuti ServiceProvider koji je odraditi određene stvari (prebaciti konfiguracijsku datoteku, prebacite poglede ili neke rute koji će biti dostupne cijelom Laravelu)

```
php artisan vendor:publish --provider="Cviebrock\EloquentSluggableServiceProvider"
```

Ako trebamo modifikaciju, npr. kako želimo da se slug-ovi ponašaju, postoji opcija i za to. Paket se naravno neće dirati u vendoru, nego možemo prebaciti u određeni direktorij.

Pogledajmo kako aktivirati slug da se kači na model. Vidjet ćemo da on u principu radi Eloquent modela. Trebamo koristiti `use Sluggable;` Idemo vidjeti kako radi:

Pokrenuli smo oba reda. Drugi je iskopirao iz vendor-a u `config/sluggable.php`. Sada možemo konfigurirati paket bez bojazni da će kada se paket upgrade-a pregaziti vrijednosti. Ako ga otvorimo, imamo različite vrijednosti za podešavanje.

U `app/Models/Post.php` dodajemo unutar klase `Post`:

```
use HasFactory, Sluggable;
```

Taj trait `use Sluggable` tjera naš model da implementira metodu `sluggable`. U trait-u je zadana apstraktna funkcija `sluggable`. To znači da kada smo u svoju klasu uključili trait, ovaj abstract tjera našu klasu da imamo tu metodu.

```
public function sluggable(): array
{
    return [
        'slug' => [
            'source' => 'title'
        ]
    ];
}
```

Više ne moramo brinuti o slug-u. Idemo provjeriti da li radi. Sve je OK.U `app/Services/PostService.php` umjesto mass assignment mogli smo manualno nafilati model i pospremiti:

```
$post = new Post();
$post->title = $data['title'];
$post->content = $data['content'];
$post->image = $data['image'];
$post->slug = $data['slug'];
$post->user_id = $data['user_id'];
$post->save();
```

To on radi za `slug`, napuni model samo s podacima iz `$fillable` a sa `sluggable()` ručno ažurira slug. Ovo naravno nećemo koristiti.

Dugme za editovanje postova

Kod kreiranja novog posta mogli bi smo staviti dugme za vraćanje nazad ali to je podizanje korisničkog iskustva, pa se predavač nije htio s tim zamarati.

Umjesto toga, stavit ćemo kod stupca Actions dugme koje omogućava editiranje tekućeg posta. Otvorit će se slična forma kao i za kreiranje novog forma. Razlika je u tome što obrazac treba biti popunjena sa Title i Content. Logično je ako neko editira sadržaj da vidi dosadašnji sadržaj. Prije nego stavimo dugmad u tablicu, otvorit ćemo novu datoteku `resources/views/admin/posts/edit.blade.php`:

```
@extends('layout', ['admin' => true])

@section('title')
    Edit post - {{ Str::limit($post->title, 20) }}
@endsection

@section('content')
    <div class="container">
        <h1>Edit post</h1>
        @if (session('post-updated'))
            <div class="alert alert-success">
                {{ session('post-updated') }}
            </div>
        @endif
        <form method="post" action="{{ route('posts.update', ['post' => $post->id]) }}"
            enctype="multipart/form-data">
            @csrf
            @method('put')
            <div class="form-group">
                <label for="title">Title</label>
                <input type="text" class="form-control" id="title" name="title"
                    value="{{ old('title', $post->title) }}>
                @error('title')
                    <div class="alert alert-danger">{{ $message }}</div>
                @enderror
            </div>
            <div class="form-group">
                <label for="content">Content</label>
                <textarea class="form-control" id="content" name="content">{{ old('content', $post->content) }}</textarea>
                @error('content')
                    <div class="alert alert-danger">{{ $message }}</div>
                @enderror
            </div>
            <div class="form-group">
                <label for="image">Image</label>
```

```

        <input type="file" class="form-control" id="image" name="image">
    </div>
    @error('image')
        <div class="alert alert-danger">{{ $message }}</div>
    @enderror

    <div>
        title }}" style="max-width: 200px;" class="thumbnail">
    </div>
    <button type="submit" class="btn btn-primary">Update</button>
</form>
</div>
@endsection

```

Vidimo da je `title` ograničen `Str::limit($post->title, 20)` na 20 znakova. Vidimo da forma ima metodu `post`. Kada gledamo kako Laravel na nivou Larabela gleda zahtjeve za update, to je da su sve rute `PUT` ili `PATCH`. Ako pogledamo kod, imamo `POST` i `GET` metode. To su HTTP metode. HTML je ograničen i pozna samo `POST` i `GET`. Dakle HTML ne može napraviti `PUT` ili `PATCH` metode. Razlika između `PUT` ili `PATCH` je što `PUT` update-a sve i ako se podatak nije promijenio. `PATCH` će update-ati polja koja su se promjenila. Danas kod modernih načina izrade aplikacija frontend je izrađen pomoću Angular, React i Vue.js. Oni handlaju te zahtjeve gdje šalju zahtjeve koji se trebaju update-ovati i kreirati. Oni šalju kompletan model i zato ih je bolje slati metodom `PATCH`, gdje se šalje samo ono što je promijenjeno.

Laravel nudi `PUT` i `PATCH` metode i možemo ih obje koristiti. Dakako request mora doći putem `PUT/PATCH`. Laravel je to riješio tako da je u formu „podvalio“ jedno skriveno polje, u kodu `@method()`, gdje se navodi radili li se o `'put'`, `'put'` ili `'delete'`. To je direktiva. Kod handla s `old('title')` i `$post->title`). Kada prvi puta dođemo na formu on će povući iz baze. To je razlika od `create.blade.php` gdje nema zapisa u bazi. Ovdje ostaje `old` jer korisnik može promijeniti `Title`.

Ubacili smo dio koji prikazuje postojeću sliku:

```

<div>
    title }}"
style="max-width: 200px;" class="thumbnail">
</div>

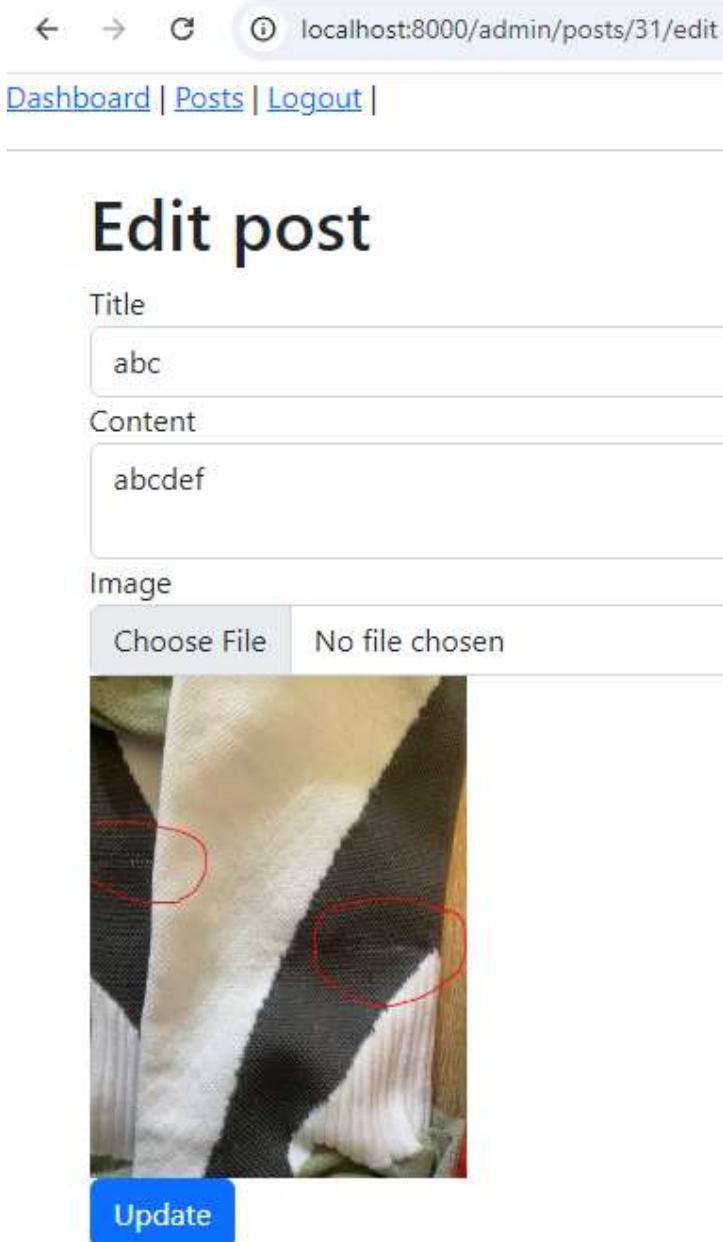
```

`asset()` helper funkcija će krenuti iz `public/storage`. `$post->image` vraća ime neke slike.

Pogledajmo kako ovo radi. Otvorit ćemo `localhost:8000/admin/posts/31/edit` jer sam vidio u tablici posts da tamo ima slika. Vidimo da je ispravna ruta po `GET` u tablici [Radnje kojima upravljaju kontroleri resursa](#). Dinamički parametar koji postoji na ruti se proslijeđuje kontroleru u metodu kao parametar u tu metodu. Ako smo koristili resource kontrolere, imamo dva načina kako možemo slati parametar: jedan je da šaljemo ID i onda ga zaprimimo u parametru metode (pogledati u

`app/Http/Controllers/PostController.php` metoda `edit()`. Taj dinamički parametar koji se mijenja, on može doći u metodu `edit()`. Tamo je ID a u `edit` dobivamo model zbog dependency injection-a. Ispod haube, kada router preusmjeri na `edit` metodu kontrolera i pošalje taj ID, `edit` metoda očekuje Post model. To znači da prije nego što dođe na `edit` metodu, mora se riješiti taj model. A riješi se preko ID. Kada imamo post, ne moramo ništa raditi niti mijenjati. Možemo reći da se vrati na post koji se već kreirao:

```
public function edit(Post $post)
{
    return view('admin.posts.edit', compact('post'));
}
```



Vidimo da sve radi.

Edit post

Title
abc

Content
abcdef

Image
Choose File No file chosen



Update

Ako nema slike trebali bi provjeriti `$post->image` ima li ga ili ne.

Ovdje smo ručno unosili ID. Želimo kreirati dugme koje bi uzelo ID resursa i izkreiralo ID. To ćemo napraviti uz pomoć helper funkcije `route()`. U `edit.blade.php` možemo vidjeti kako metoda `route()` binda podatak koji kasnije možemo izvući: `route('posts.update', ['post' => $post->id])` iz tog dinamičkog route parametra. Što znači ako

opet pogledamo `update()` u `app/Http/Controllers/PostController.php`, da ponovo dolazi taj `Post`:

```
public function update(UpdatePostRequest $request, Post $post)
{
    if (!$this->postService->updatePost($request, $post)) {
        return redirect()->back()->with('post-updated', 'Post not updated');
    }

    return redirect()->route('posts.index');
}
```

Naravno to je zato što mora znati što update-a tj. koji post će update-ati i podaci koji su promijenjeni u requestu kako bi se vezali za model i spremili u bazu.

Idemo na `resource/views/admin/posts/index.blade.php`. Tu smo imali `@forelse` direktivu:

```
@forelse ($posts as $post)
    <tr>
        <td>{{ $post->title }}</td>
        <td>{{ $post->user->first_name }} {{ $post->user->last_name
}}</td>
        <td>{{ $post->created_at->format('d.m.Y. H:i') }}</td>
        <td>

    </td>
    </tr>
@empty
    <tr>
        <td colspan="4">No posts found.</td>
    </tr>
@endforelse
```

Između `<td>` i `</td>` ubacit ćemo:

```
<a href="{{ route('posts.edit', ['post' => $post->id]) }}>Edit</a>
```

Identična je logika kao i na onoj formi, samo što je ovdje update. Uzimamo ID od posta. Ispod haube se ID binda u model. Ako ne želemo da se vidi broj možemo hash-irati ID. Postoji paket koji to radi i zove se [HashId](#).

Ako sada pokrenemo imamo dugme Edit:

| Title | Author | Published | Actions |
|-------------------------------------------------------------------|----------------|-------------------|---------|
| abc | Jarred Carroll | 13.10.2024. 22:54 | Edit |
| abc | Jarred Carroll | 12.10.2024. 23:31 | Edit |
| Quod placeat placeat rem debitis. | Corene Nader | 24.09.2024. 07:59 | Edit |
| Aut rem laboriosam quis quaerat dolore accusamus. | Martina Rath | 24.09.2024. 07:59 | Edit |
| Dolor similiique ut asperiores adipisci cupiditate amet expedita. | Corene Nader | 24.09.2024. 07:59 | Edit |

Sada korisnik koji se prijavio vidi sve postove. Nismo implementirali da korisnik koji nije admin ne vidi postove koji nisu njegovi. Kasnije ćemo definirati policy što smije netko vidjeti i mijenjati.

Iduće ćemo napraviti update u `create.blade.php`.

Iz `resources/views/admin/posts/create.blade.php` iskopirat ćemo dio:

```
@if (session('post-created'))
    <div class="alert alert-success">
        {{ session('post-updated') }}
    </div>
@endif
```

i to prebaciti u `resources/views/admin/posts/edit.blade.php` na isto mjesto samo sa `post-updated`:

```
@if (session('post-updated'))
    <div class="alert alert-success">
        {{ session('post-updated') }}
    </div>
@endif
```

Idemo istom logikom na `app/Http/Controllers/PostController.php`. Iskopiramo sadržaj `store` metode i prebacimo ga u `update` metodu. Neće biti `storePost` već `updatePost`. Poslat ćemo osim `$request` i `Post $post`.

```
public function update(UpdatePostRequest $request, Post $post)
```

```
{
    if (!$this->postService->updatePost($request, $post)) {
        return redirect()->back()->with('post-updated', 'Post not updated');
    }

    return redirect()->route('posts.index');
}
```

Nemoj neku veliku poslovnu logiku na kontroleru. Prije nego ovo ode na servis, moramo definirati `UpdatePostRequest`. Moramo pustiti `authorize` na true jer je po default na `false`. Ako kliknemo na `UpdatePostRequest` otvorit će se `app/Http/Requests/UpdatePostRequest.php` gdje riješimo `authorize()`:

```
public function authorize(): bool
{
    return true;
}
```

U `rules()` ćemo autorizirati:

```
public function rules(): array
{
    return [
        'title' => ['required', 'string', 'max:255'],
        'content' => ['required', 'string'],
        'image' => ['image', 'max:5000'],
    ];
}
```

Sa title koji je `required` ne može se desiti da neko obriše Title i pritisne update jer će validator javiti da ne može. Oslanjam se na bazu i ako je tamo required onda će title biti `nullable`.

Riješili smo kontroler da primi podatke iz requesta, koji prolaze kroz ovaj update post request i validiraju se ova pravila. Ako je sve OK, request će skupa s postom koji se treba update-ovati (u requestu imamo nove podatke, POST model je već tu jer smo ga bind-ali iz baze) i model trebamo spojiti s novim podacima i snimiti.

Sada se logika seli na servis `app/Services/PostService.php`. Kreiramo `updatePost` funkciju.

```
public function updatePost(WebRequest $request, Post $post)
{
    try {
        $data = $request->validated();
        // $user_id = $request->user()->getKey();

        if ($request->hasFile('image')) {
            $oldImage = $post->image;
```

```

        if ($oldImage) {
            //unlink(storage_path('app/public/' . $oldImage));
            Storage::disk('public')->delete($oldImage);
        }
        $data['image'] = $request->file('image')->store('images',
'public');
    }

    $post->update($data);

    return true;

} catch (\Exception $e) {
    Log::error($e->getMessage());
    return false;
}
}

```

Izvuče podatke iz requesta koji su validirani, vidi da li ima sliku, spremi tu novu sliku i napravi update podataka. Mi moramo izbrisati staru sliku. Da bi je se riješili moramo znati putanju. A putanja se nalazi u modelu post:

```
$oldImage = $post->image;
```

Sada kada imamo tu sliku trebamo provjeriti postoji li jer je `image` polje neobavezno. Nakon toga, dođe nova slika.

```

        if ($oldImage) {
            unlink(storage_path('app/public/' . $oldImage));
            //Storage::disk('public')->delete($oldImage);
        }
        $data['image'] = $request->file('image')->store('images',
'public');
    }
}

```

Ako se radi o postu koji ima u polju `image`, `null`, provjeri ako nema stare slike onda ne treba `unlink()`. Nakon toga doda novu sliku. Imamo return `true` a ako dođe do greške `false`. `unlink()` radi na nivou lokalnog servisa. Možemo koristiti ili jedan ili drugi pristup (zakomentiran).

[Dashboard](#) | [Posts](#) | [Logout](#) |

Edit post

Title

Content

The content field is required.

Image

Choose File No file chosen

Dugme za brisanje postova

U `resources/views/admin/posts` dodamo dio koji umeće Delete dugme, tako da ukupno imamo:

```
<td>
    <a href="{{ route('posts.edit', ['post' => $post->id]) }}"
}'' class="btn btn-sm btn-primary">Edit</a>
    <form method="post" action="{{ route('posts.destroy',
['post' => $post->id]) }}" style="display: inline;">
        @csrf
        @method('delete')
        <button type="submit" class="btn btn-sm btn-
danger">Delete</button>
    </form>
</td>
```

Da smo umjesto ove forme (obrasca) iznad i dugmeta Delete, pokušali staviti:

```
<a href="{{ route('posts.destroy', ['post' => $post->id]) }}" class="btn btn-sm
btn-primary">Delete</a>
```

Zamaskirali smo ovo kao običan link koji bi trebao ići na `destroy` rutu. Međutim on ide na `app/Http/Controllers.php show()` metodu.

Ruta (URI) je identična `/posts/{post}` ali se razlikuje u metodi kojom se šalje request. To su `GET` ili `DELETE`. To možemo vidjeti u tablici [Radnje kojima upravljaju kontroleri resursa](#). Link koji smo izgenerirali uz pomoć imena `posts.destroy` izgenerirali URI `/posts/{post}` koji je identičan URI `GET`-a. Kad smo ovo odradili, trebali bi na kontroleru app/Http/Controllers/PostController.php nad `destroy()` prekopirati doslovno `update()` metodu. Promijenit ćemo `updatePost` u `deletePost`.

```
public function destroy(Post $post)
{
    if (!$this->postService->deletePost($post)) {
        return redirect()->back()->with('post-deleted', 'Post not deleted');
    }

    return redirect()->route('posts.index');
}
```

`'post-deleted'` je ključ a poruka je `'Post not deleted'`.

Idemo u servis `app/Services/PostService.php` dodati `deletePost`:

```
public function deletePost(Post $post)
{
    try {

        $post->delete();

        return true;

    } catch (\Exception $e) {
        Log::error($e->getMessage());
        return false;
    }
}
```

Laravel neke stvari zapisuje u log-ove a neke ne. Logiranjem potencijalnih grešaka, lakše se otklanjavaju greške. Soft-delete je upaljen nad modelom i to mu treba reći. Na `app/Models/Post.php` treba dodati `SoftDeletes` iz Eloquenta.

```
use HasFactory, Sluggable, SoftDeletes;
```

Sada servis dodaje u polje `deleted_at` polje kada je obrisano i u indeksu taj podatak ne vidimo (u [/admin/posts](#)). Na migracijama moramo upaliti taj stupac, kako bi ga mogao provjeravati. Na modelu moramo upaliti trait `SoftDeletes`. Automatizmom kada pozovemo metodu `delete()`, podatak se neće fizički izbrisati.

Moguće je napraviti poseban pogled na smeće i pozvati ga. Pokazat ćemo kako to radi na metodi `getAllPosts()` u `app/Services/PostService.php`, bez ideje da to zadržimo:

```
public function getAllPosts()
{
    return Post::with('user')->onlyTrashed()->get();
}
```

| deleted_at |
|-------------|
| NULL |
| 2024-10-... |
| 2024-10-... |

Posts

[Create a new post](#)

| Title | Author | Published | Actions |
|--------|----------------|-------------------|---------------------------------------------|
| šdžčč | Jarred Carroll | 15.10.2024. 23:31 | Edit Delete |
| šdžččš | Jarred Carroll | 15.10.2024. 23:32 | Edit Delete |

Umjesto Edit i Delete ovdje mi napravili Restore dugme koje bi pozivalo `restorePost()` metodu i u njoj `$post->restore();`. Također postoji `hardDelete()` koja poziva `$post->forceDelete();`.

Umjesto `onlyTrashed()` metode koja prikazuje samo smeće, imamo i `withTrashed()` koja prikazuje i podatke i smeće.

Sortiranje podataka

Kada imamo veliku količinu podataka u bazi, dohvaćanje podataka zauzima velike resurse. U tom slučaju koristimo paginaciju. Ono što možemo napraviti prvo u `getAllPost()` možemo napraviti sortiranje od najnovijih postova do najstarijih.

```
public function getAllPosts()
{
    return Post::with('user')->orderBy('created_at', 'desc')->get();
}
```

Korištenje paginatora

Paginaciju kreiramo tako da umjesto `get()` koristimo `paginate(x)`. Dakle ne kreira se standardna kolekcija već kolekcija s dodatnim informacijama. Ovdje trebamo uključiti i neku navigaciju.

U `resource/views/admin/posts/index.blade.php` dodat ćemo pred kraj `<div>` tag (koji inače služi kao kontejner za HTML):

```
</table>
<div class="mx-auto my-2">
    {{-- {{ $posts->links() }} --}}
</div>
</div>
@endsection
```

Laravel će automatski izgenerirati kompletan HTML s linkovima koji rade ispravne requestove. `$posts` je ovdje ta proširena kolekcija. Ovo je osnovni paginator. Možemo vidjeti da kroz query u zaglavlju šalje parametar.

 localhost:8000/admin/posts?page=2

Ako pogledamo [Database>Pagination u uputstvu](#), možemo vidjeti ostale mogućnost. U uputama ima odlomak vezan za korištenje Bootstrap CSS-a. Za korištenje ovih prikaza umjesto zadanih Tailwind pogleda, možete pozvati paginator `use BootstrapFive` metode unutar boot metode vaše `App\Providers\AppServiceProvider` klase:

```
use Illuminate\Pagination\Paginator;

/**
 * Bootstrap bilo kojeg aplikacijskog servisa.
 */
public function boot(): void
{
    Paginator::useBootstrapFive();
}
```

Otići ćemo u naš `app/Providers/AppServiceProvider.php` i u metodu `boot()` dodati:

```
public function boot(): void
{
    Paginator::useBootstrapFive();
}
```

Čim to uključimo vidjet ćemo strelice i brojeve za paginator.

Showing 1 to 5 of 12 results



Ograničavanje mijenjanja i pogleda na postove s rolama

Moramo gledati tko što smije. Možemo to gledati na nivou uloge a možemo na nivou permission-a. Npr. možemo reći da rola administrator može sve a rola user nešto – to znači da svi koji imaju istu rolu

mogu isto. Mi možemo granulirati tako da pojedina rola ili pojedini user mogu imati permission-e. Sam korisnik može imati ulogu, i dozvolu te uloge i vlastite dozvole. Taj koncept može biti kompleksan i moramo paziti tko ima prioritet. Mi ćemo to napraviti jednostavnije, da pojedini user ima pojedinu rolu. Htjeli bi provjeriti da li je user u nekoj roli. Znači mehanizam koji će javiti `true` ili `false`. user inroll(Admin) a odgovor je `true` ili `false`, ili user inroll(User) a odgovor je `true` ili `false`.

Prvo ćemo napraviti enumerator. Napraviti ćemo novi direktorij i datoteku `app/Enums/Roles.php` i napisati:

```
<?php

namespace App\Enums;

enum Roles: string
{
    case ADMIN = 'Admin';
    case USER = 'User';
    case AUTHOR = 'Author';
}
```

U biti, cilj nam je izvući vrijednosti iz ovoga enumeratora. Kada budemo htjeli pitati da li je neki korisnik u toj ulozi, onda ćemo nad modelom User pozvati metodu inroll i moći ćemo poslati vrijednost kao string (ne kao enumerator). Zato je i enumerator definiran kao string. Trenutno sistem dodjeljuje prilikom registracije automatski dodjeljuje rolu user. U back office-u administrator može mijenjati userima uloge. To se događa u `app/Services/AuthService.php`. Hardkodirano je ime role:

```
$role = Role::where('name', 'User')->first()->getKey();
```

Ako promijenimo ime da ne bude 'User' u bazi onda moramo primijeniti vrijednost u `app/Enums/Roles.php` i automatski će se aplicirati u kodu. Predavač govori o principu „single source of true“ tako da s jednog mesta možemo izvršiti promjenu. Promijenit ćemo kod da bude u više redova jer je preglednije. Umjesto `('name', 'User')` napisat ćemo `('name', Roles::USER->value)`:

```
$role =
Role::where(
    'name',
    Roles::USER->value
)
->first()
->getKey();
```

Izbjegli smo hard kodiranje. Na user modelu želimo kreirati metodu koja će jednostavno provjeriti u kojoj ulozi je korisnik. Na kraju user modela `app/Models/User.php` kreirat ćemo `hasRole` metodu:

```
public function hasRole(string $role): bool
{
    return $this->roles()->where('name', $role)->exists();
}
```

Pošaljemo li string a vezana je uloga za korisnika metoda će vratiti true, ako nije vratiti će false. Ovime se policy se veže sa post i daje dozvolu ili ne. Time štitimo resource. Ako odemo u dokumentaciju i pod [Security>Authorization](#), možemo vidjeti da postoji Gates. Gates su jednostavno closures koji određuju je li korisnik ovlašten izvršiti određenu radnju. Iskoristit ćemo taj Gates i pomoću policy-ja ćemo provjeriti ima li korisnik na to pravo ili nema. Gates možemo pisati [pomoću User modela](#). Unutar servisa nad userom možemo pitati što on smije ili ne smije.

```
class PostController extends Controller
{
    /**
     * Kreiraj post.
     */
    public function store(Request $request): RedirectResponse
    {
        if ($request->user()->cannot('create', Post::class)) {
            abort(403);
        }

        // Kreiraj post...

        return redirect('/posts');
    }
}
```

Drugi način je [pomoću Gate fasade](#):

```
Gate::authorize('update', $post);
```

Treći način je preko middleware, što je malo nezgodno ako koristite resource rutu;

```
Route::put('/post/{post}', function (Post $post) {
    // Tekući korisnik može ažurirati post...
})->middleware('can:update,post');
```

Kako policy radi, to je ništa drugo nego definicija pravila po metodama, koje mi možemo definirati: `view`, `viewAny`, `update`, `create`, `delete` i `forceDelete`. Generiranje policy-ja je vrlo jednostavno:

```
php artisan make:policy PostPolicy
```

Tu vežemo policy uz model i time vežemo određeni resurs. Naredba `make:policy` će generirati praznu klasu pravila. Ako želite generirati klasu s primjerima metoda politike koje se odnose na pregled, stvaranje, ažuriranje i brisanje resursa, možete dati `--model` opciju prilikom izvršavanja naredbe:

```
php artisan make:policy PostPolicy --model=Post
```

Ovo ćemo pokrenuti. Kreirat će se `app/Policies` i datoteka `PostPolicy.php`. `viewAny` određuje kada korisnik smije vidjeti kolekciju modela. `view` određuje kada korisnik smije vidjeti jedan model, `create` određuje kada korisnik smije kreirati novi zapis, `update` kada može ažurirati, `delete` kada obrisati, `restore` kada vratiti i kada raditi `forceDelete`, tu se definira logika.

Administratori smiju sve. Pisati policy logiku u svakoj metodi je redundantno. Ako u dokumentaciji pogledamo [Security>Authorization>Writing Policies>Policy Filters](#) gdje za određene korisnike možda ćete htjeti autorizirati sve radnje unutar dane politike. Da biste to postigli, definirajte `before` metodu na politici. Metoda `before` će se izvršiti prije bilo koje druge metode na politici, dajući vam priliku da autorizirate radnju prije nego što se namjeravana metoda politike stvarno pozove. Ova se svojstvo najčešće koristi za autoriziranje administratora aplikacije za izvođenje bilo koje radnje. To znači da ćemo u `app/Policies/PostPolicy.php` dodati na početku metodu `before()`:

```
public function before(User $user): ?bool
{
    if ($user->hasRole(Roles::ADMIN->value)) {
        return true;
    }

    return null;
}
```

Provjerimo dinamički da li je rola administratora ako je vrati `true` a ako nije vrati `null`. Zato je na vrhu metode `?bool` jer je ili `bool` ili `null`.

U `viewAny` staviti ćemo:

```
public function viewAny(User $user): bool
{
    return false;
}
```

U našem servisu `app/Services/PostService.php` kada hvatamo podatke, možemo koristiti u `getAllPosts()`:

```
Gate::authorize('viewAny'), Post::class);
```

Treba pripaziti da uzmemo ispravan `Gate [Illuminate\Support\Facades]`.

Uz pomoć `Gate` se vršiti autorizacija i gledat ćemo ima li taj korisnik pravo da vidi. Ako korisnik nije administrator korisnik ne može doći do postova. Ako nije korisnik ne bi smio doći do postova. Korisnik bi ipak trebao vidjeti svoje postove i zato treba koristiti `if`:

```

public function getAllPosts()
{
    if(auth()->user()->cannot('viewAny', Post::class)) {
        return Post::with('user')
            ->where('user_id', auth()->id())
            ->orderBy('created_at', 'desc')
            ->paginate(5);
    }

    return Post::with('user')->orderBy('created_at', 'desc')->paginate(5);
}

```

Ako autorizirani korisnik nema pravo vidjeti tada vrati post-ove koji su njegovi, složeni i paginirani. Ako pogledamo sada, vidjet ćemo samo post korisnika prijavljenog u sistem. Ako stisnemo Edit dugme, možemo to napraviti ali možemo napraviti i s drugim postom koji nije njegov. I to moramo štititi da se ne dogodi.

Pojedini post zaštitit ćemo iz `view()` metode, naravno u `app/Policies/PostPolicy.php`.

```

public function view(User $user, Post $post): bool
{
    return $user->id === $post->user_id;
}

```

`view()` se koristi da zaštitimo jedan post za razliku od `viewAny()`. Kod `viewAny()` nismo slali post a kod `view()` tom moramo kako bi provjerili ima li user pravo. Jedini koji smije post vidjeti je `$user->id` koji je identičan `$post->user_id`. Kako primijeniti policy u servisu, kontroleru ili negdje drugdje je vrlo jednostavno.

Prvo ćemo u `app/Services/PostService.php` u metodi `getAllPosts()` riješiti `try-catch` blok:

```

public function getAllPosts()
{
    try {
        if(auth()->user()->cannot('viewAny', Post::class)) {
            return Post::with('user')
                ->where('user_id', auth()->id())
                ->orderBy('created_at', 'desc')
                ->paginate(5);
        }

        return Post::with('user')->orderBy('created_at', 'desc')->paginate(5);
    }
}

```

```

    } catch (\Exception $e) {
        Log::error($e->getMessage());
        return null;
    }
}

```

Vraćamo bullian `true` ili `false`. U slučaju da dođe do greške, vraćamo `null`. Kada pogledamo kontroler `app/Http/Controllers/PostController.php` i metodu `store()` onda nam je jasno zašto vraćamo `true` ili `false`, jer u njoj dodatno provjeravamo da li je post kreiran. U `index()` nemamo taj podatak. Ako vratimo `null` ako se dogodi iznimka, dakle `$posts` je `null`.

```

public function index()
{
    $posts = $this->postService->getAllPosts();
    return view('admin.posts.index', compact('posts'));
}

```

Taj `null` odlazi na view u `resources/views/admin/posts/index.blade.php`. Pošto tu nema ništa, dobit ćemo `No posts found`. To je pogrešno jer postova ima ali ih on ne vidi – došlo je do greške. Zato umjesto da u servisu `app/Services/PostService.php` u `getAllPost()` metodi, na kraju `catch` umjesto da vratimo `return null`, možemo zaustaviti ili isto reći `return false` pa onda na kontroleru u `app/Http/Controllers/PostController.php` u metodi `store()` staviti nešto. Možemo i ostaviti tamo `return null` a ovdje u `PostController.php` u `store()` napisati:

```

public function index()
{
    if(!$posts = $this->postService->getAllPosts()) {
        abort(400);
    }

    return view('admin.posts.index', compact('posts'));
}

```

Što ćemo dobiti ako u `routes/web.php` u onoj testnoj rutu napišemo:

```

Route::get('test', function () {
    $test = [];
    dd($test != null);
});

```

```
false // routes\web.php:38
```

Prazna matrica nije različita od `null`. To je isto. U našem slučaju `!$posts = $this->postService->getAllPosts()`, ako `getAllPosts()` vrati `null`, `!` će taj null pretvoriti u `true`. `getAllPosts()` može vratiti praznu matricu. Naš servis `app/Services/PostService.php` vraća kolekciju tj. u našem slučaju paginator:

```
return Post::with('user')
    ->where('user_id', auth()->id())
    ->orderBy('created_at', 'desc')
    ->paginate(5);
```

To nije prazno iako unutra nema niti jedan naš podatak, uvijek će postojati neki podatak. To je zato što ako napišemo u `routes/web.php` u onoj testnoj ruti:

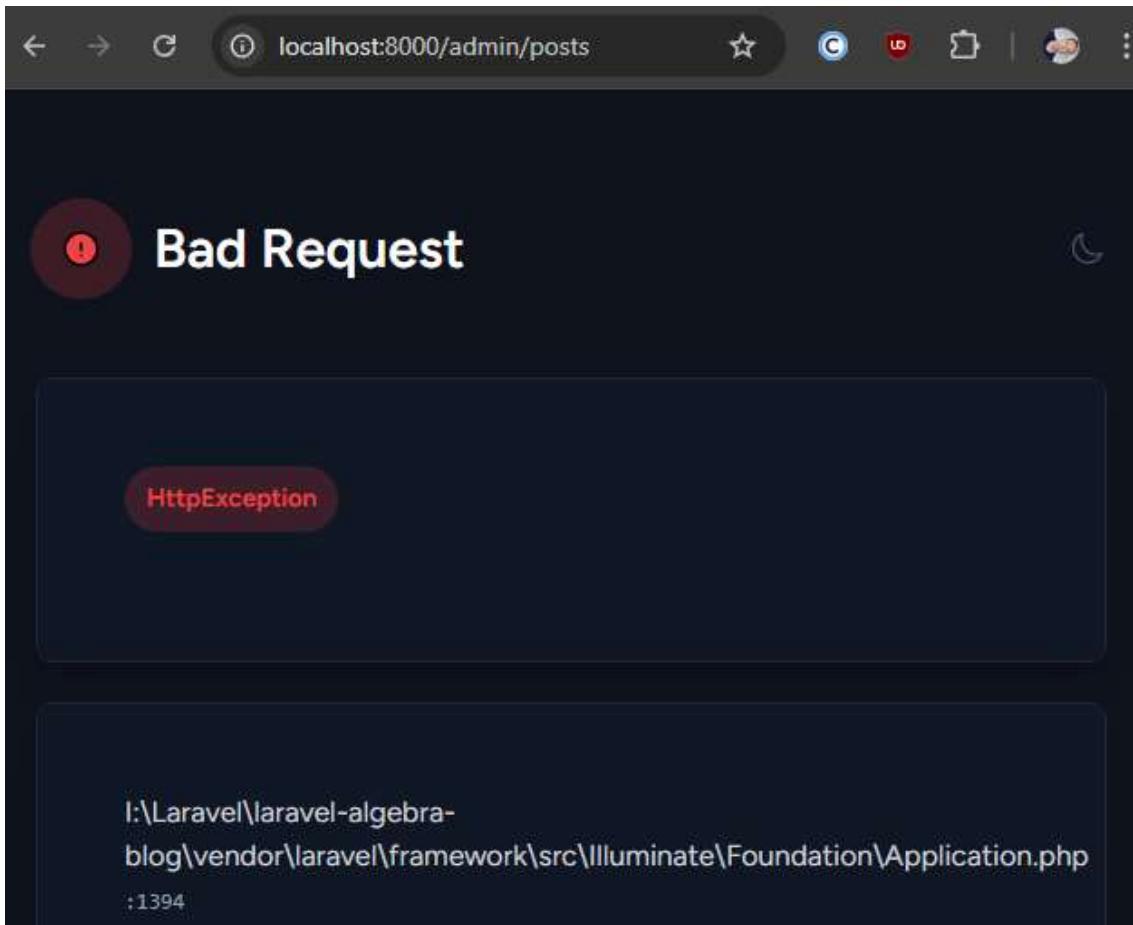
```
Route::get('test', function () {
    $test = [];
    dd(collect() == null);
});
```

```
false // routes\web.php:38
```

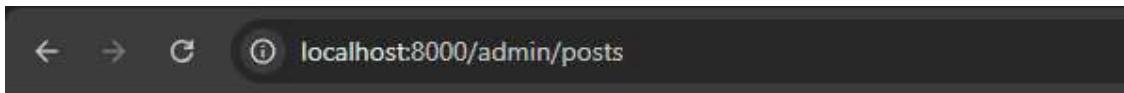
Prazna kolekcija nije isto što i `null` ali je prazna matrica isto što i `null`. Da `getAllPost()` vraća matricu imali bi smo problem. `getAllPost()` vraća postove ali nema ni jednog posta, dobije praznu matricu, prazna matrica je `false`, `!` će okrenuti u `true` i dobit ćemo `abort(400)` dakle nešto nije u redu s aplikacijom. Ali to nije točno, mi samo nemamo postova u bazi.

Ovo sada sve radi. Pogledajmo što ako ubacimo neki `exception` u `app/Services/Postservice.php`. U `getAllPost()` iznad `return` ubacit ćemo `throw`:

```
Throw new \Exception('User does not have premission to view posts');
```



Ako bi bio ugašen APP_DEBUG u `.env`.



Sada vidimo da se dogodila greška, dogodio se `Exception` i mi smo ga uhvatili. Mi smo u `app/Services/PostService.php` u `catch` zapisali u `Log`. Vratili smo `null`, kontroler

`app/Http/Controllers/PostController.php` u `index()` je uzemo taj null i zapisao ga u `$posts`, ! operator je to okrenuo to u true, i nakon toga išao je abort(400) tj. generiran je `Bad Request`.

Isti problem ćemo imati ako u `app/Services/PostService.php` gdje je sada:

```
return Post::with('user')->orderBy('created_at', 'desc')->paginate(5);
```

izmjenimo u

```
return Post::with('user')->orderBy('created_at', 'desc')->skip(300)->take(5)->get();
```

Dakle preskoči prvih 30 zapisa, uzmi 5 i to mi vrati. Mi ne bi smjeli dobiti postove, ali ne bi smjeli niti imati grešku ali ne smije dohvati ni jedan post.

Trebamo u `resources/views/admin/posts/index.blade.php` zakomentirati red:

```
 {{ $posts->links() }}
```

Ako nije spomenuto do sada, u VSC kod se zakomentira s Ctrl+K+C a komentar se može ukloniti s Ctrl+K+U. Ovo radi s označenim blokom i s pojedinim označenim redom.

Posts

[Create a new post](#)

| Title | Author | Published | Actions |
|-----------------|--------|-----------|---------|
| No posts found. | | | |

Dodat ćemo novi servis u `app/Services/PostService.php` i to `getPost()`:

```
public function getPosts()
{
    try {
        if(auth()->user()->cannot('view', Post::class)) {
            return null;
        }

        return $post;
    } catch (\Exception $e) {
        Log::error($e->getMessage());
        return null;
    }
}
```

Pogledajmo ovdje dohvati post i vratimo post. Nema svrhe da to radimo. Ipak, korisnika moramo autorizirati. Na nivou kontrolera obično ovo ne rješavamo, jer obično tamo ne želimo poslovnu logiku.

```
App\Models\Post {#1249 ▼ // app\Http\Controllers\PostController.php:65
  #connection: "mysql"
  #table: "posts"
  #primaryKey: "id"
  #keyType: "int"
  +incrementing: true
  #with: []
  #withCount: []
  +preventsLazyLoading: false
  #perPage: 15
  +exists: true
  +wasRecentlyCreated: false
  #escapeWhenCastingToString: false
  #attributes: array:9 [▶]
  #original: array:9 [▶]
  #changes: []
  #casts: array:1 [▶]
  #classCastCache: []
  #attributeCastCache: []
  #dateFormat: null
  #appends: []
  #dispatchesEvents: []
  #observables: []
  #relations: []
  #touches: []
  +timestamps: true
  +usesUniqueIds: false
  #hidden: []
  #visible: []
  #fillable: array:4 [▶]
  #guarded: array:1 [▶]
  #forceDeleting: false
}
```

Ako smo neki drugi user (koji nije administrator ili nije napravio taj post), dobit ćemo poruku:

403 | THIS ACTION IS UNAUTHORIZED.

Predavač kaže da koristi pristup nad userom can i cannot, najčeće kod ovih All postova, gdje trebamo razdvojiti postove po pojedinom korisniku. Kada autoriziramo viewAny tamo moramo gledati puštamo li ga i ili ne da vidi sve postove. U `app/Policies/PostPolicy.php` u `viewAny` je teško odraditi neku logiku jer nemamo post koji je poslan. Ovdje odraditi neku logiku je dosta nespretno. Možemo provjeriti ima li korisnik neku ulogu ili pravo i na temelju toga vratiti `true` ili `false`. U drugim metodama gdje imamo `(User $user, Post $post)` a, možemo provjeravati da li je user vlasnik resourса i da li ćemo dozvoliti useru da kreira post ili ne.

Možemo potpuno izbjegići ovaj servis i obrisati ga. Ipak ćemo u kontroleru `app\Http\Controllers\PostController.php` vratiti post nazad i prije toga uz pomoć `Gate` poslati na

autorizaciju post meti `view`. Prije nego što dođe do metode `view()`, policy (onaj `before()`) će provjeriti radi li se o administratoru, on smije sve vidjeti:

```
public function show(Post $post)
{
    Gate::authorize('view', $post);
    //dd($post);
    return view('admin.posts.show', compact('post'));
}
```

Gate autorizira ili ne autorizira korisnika na temelju policy-ja. Proslijedio je metodi `view $post` koji se treba kao resource pogledati. Policy `app/Policies/PostPolicy.php` tj `view()` je rekao `$user` koji je napravio request (`Gate::authorize` je poslao user-a). Sve metode u policies imaju user model zato što na temelju njega provjeravamo da li user koji je napravio request, smije ili ne smije nešto. Ovdje u policy u `view` smo poslali `$post`. Pitamo da li id user-a koji je napravio request se podudara s `user_id` poljem u post modelu.

Ovaj post još nemamo pa ćemo pogledati `dd($post)`. Za određeni post npr `29`, otvaramo link `http://localhost:8000/admin/posts/29` vidimo ga samo ako smo administrator ili ako smo taj user koji je napravio post. `Gate::authorize('view', $post);` smo mogli prebaciti u servis što bi bilo zgodno jer bi tamo radilo sve i hendlati iznimku. Ovaj `authorize` baci svoj interni exception `AuthorizationException`.

Pogledajmo `create()` u `app/Http/Controllers/PostController.php`. `create()` gleda da li korisnik smije kreirati post - ne smije. Ako ne smije spremiti novi post, onda ne treba vidjeti formu. Pogledajmo kako će onda glasiti policy u `app/Policies/PostPolicy.php`. Ako pogledamo `before()` on gleda samo da li se radi o adminu. `view()` uspoređuje da li je user vlasnik posta. Kod `create()` dozvolit ćemo da user kreira post:

```
public function create(User $user): bool
{
    return
        $user->hasRole(Roles::USER->value) ||
        $user->hasRole(Roles::AUTHOR->value);
}
```

Ako je user u ulozi user-a ili autora onda vraća `true`. Možemo jednostavnije napisati da za sve uloge vraća `true`:

```
public function create(User $user): bool
{
    return true;
}
```

Ako nemamo nikoga za izostaviti tko ne smije kreirati onda ćemo jednostavno vratiti `true` i ne gledati user-a. Middleware će odraditi provjeru da li je netko prijavljen u sistem. Bez obzira ćemo u `create()`

```
u      app/Http/Controllers/PostController.php      staviti      Gate::authorize('create',  
Post::class);  
  
    public function create()  
    {  
        Gate::authorize('create', Post::class);  
  
        return view('admin.posts.create');  
    }
```

Dajemo mu nad kojim resursom se primjenjuje ne konkretan resurs nego općenito, klasu mu dajemo, potpis resursa to je post model. Pitamo da li je autoriziran. Koji god da je user, nije problem. Ako bi nekoj grupi usera, zabranili da kreiraju nove postove, dovoljno je da ostavimo `$user->hasRole(Roles::AUTHOR->value)` ili `$user->hasRole(Roles::USER->value)`, ovisno što propuštamo. Dobit ćemo 403 This action is unauthorized. Time možemo iskontrolirati što želimo.

403 | THIS ACTION IS UNAUTHORIZED.

Dakle ako nije uključeno u `app/Http/Controllers/PostController.php` u metodi `create()` neće raditi:

```
Gate::authorize('create', Post::class);
```

Drugim riječima promjene na policy u `app/Policies/PostPolicy.php` u `create()` neće biti moguće napraviti bez toga.

Istu priču bi ponovili `updated`, `view`, `delete`, `restore` i `forceDelete` u `app/Policies/PostPolicy.php`.

Kod `delete`, treba napomenuti da ne bi trebalo dozvoliti da se podatak obriše već treba postaviti potvrdu da li ste sigurni da želite obrisati.

Višejezičnost

Laravel je prilagođen višejezičnosti. To ne znači da ćemo automatizmom prevoditi sadržaj. Do sada smo hardkodirali sve tekstove u našem programu za upravljanje postovima.

U `.env` datoteci postoji `APP_LOCALE=en`. Ako promjenimo jezik `APP_LOCALE=hr`. Ako `APP_FALLBACK_LOCALE=en`, postavimo na `si`, poruke nećemo vidjeti tj. vidjet ćemo u validacijama `validation.required`. Ako ne pronađe hr jezik, backup mu je `si`. Poruke bi se trebale nalaziti u `lang/jezik/messages.php`.

```
php artisan lang:published
```

Dakle svi stringovi trebaju biti izdvojeni u datoteku. Kada odradimo ovu naredbu pojavit će se `resources/Lang/en` i unutra `auth.php`, `pagination.php`, `passwords.php` i `validation.php`. Mi ručno možemo kreirati sličan direktorij za bilo koji jezik.

Možete dohvatiti stringove prijevoda iz svojih jezičnih datoteka pomoću `__` helper funkcije. Ako koristite kombinacije tipki za definiranje nizova prijevoda, trebali biste proslijediti datoteku koja sadrži ključ i sam ključ funkciji `__` koristeći "točka" sintaksu. Na primjer, dohvativamo `welcome` string prijevoda iz `Lang/en/messages.php` jezične datoteke:

```
echo __('messages.welcome');
```

Ako navedeni string prijevoda ne postoji, `__` funkcija će vratiti ključ niza prijevoda. Dakle, korištenjem gornjeg primjera, `__` funkcija bi vratila `messages.welcome` ako prevedeni string ne postoji.

Dakle kod pozivanja navodimo ime datoteke i ključ u matrici, vraća se poruka.

Jezik možemo u Laravelu primijeniti kroz `App::setLocale($locale);`

Poanta je gradili ili ne gradili višejezičnost, dobra ideja je odmah krenuti s višejezičnošću.

Možemo otvoriti svoju datoteku, npr. `forms.php`. Struktura je:

```
<?php

return [
    'title' => 'Title'
];
```

U pozivamo ga s npr.

```
<label for="title">{{ __('forms.title') }}</label>
```

Kada korisnik klikne na `localhost:8000/auth/login` promijenio se URL. Ako pogledamo našu rutu `routes/web.php`, Ako pogledamo na kod, vidimo prefiks `auth` i `login` (ime rute `showLogin`), dakle `localhost:8000/auth/login`. Ako idemo u kontroler `app/Http/Controllers/AuthController.php` i metodu `login` u njemu, ona vraća `'login'` pogled. U `'login'` pogledu imamo formu (obrazac) u koju korisnik može upisati email i password. Ista stvar je ako idemo na `register`. Prefiks je i dalje `auth`, ruta `showRegister` (a ime rute `register`). Ako pogledamo u `app/Http/Controllers/AuthController.php` možemo vidjeti metodu `showRegister()`, u kojem je opet forma (obrazac) s nekim poljima koja smo definirali kao obavezna ili ne. Nakon što se vratio `'register'` pogled. U tom pogledu je forma koja kada je submitt-amo ide na rutu s imenom `post.register`. Ta ruta isto ima prefiks auth i rutu `register`. To znači da kada stisnemo dugme Register, odlazimo u kontroler `app/Http/Controllers/AuthController.php` i dolazimo do `register` metode. Tu imamo dependency injection `RegisterRequest $request` objekta koji će napraviti validaciju podataka koji su pristigli u formi. Ta pravila nalaze se u `app/Http/Requests/RegisterRequest.php`. Ako pravila nisu zadovoljena neće doći do returna u register. Kada se metoda pokreće i kada se instancirala, dogodi se exception i iznimka obradi grešku i

vratiti nazad poruku u nekom error bag objektu. Ako pogledamo `resources/views/register.blade.php` vidimo tu direktivu koja izvlači greške. Ako register prođe, vraća se rezultat `register` metode unutar `authService` servisa. Na nivou kontrolera napravili smo dependency injection. Svaki puta kada se kontroler `AuthController` instancira. U konstruktor se injekta `AuthService` i mi to zapišemo u `$authService`. Sada svaki puta kada se kontroler `AuthController` instancira dobit ćemo pristup `app/Services/AuthService.php`. U tom `AuthService` mi trebamo doći do metode `register`. Ona prima podatke iz requesta, poslali smo sve podatke. Na tom servisu smo izvukli `first_name`, `last_name`, `email` i `password`, tj. samo validirane podatke. Nakon toga idemo na `create` uz pomoć modela. Mass assignment mora biti omogućen na modelu. To se omogući s `app/Models/User.php` i `$fillable`. To znači da `create` iz `register` metode iz `app/Services/AuthService.php` prima podatke i te podatke iz `$fillable` pohranjuje u bazu. Nakon toga izvlačimo rolu koju želimo dodijeliti useru, tako da uz pomoć modela iz baze izvučemo točno taj objekt i attach-amo user-u, uz pomoć relacije. Relacija je na modelu prema drugim modelima `belongsToMany`. Dakle u user-u imamo relaciju prema rolama jer je to many-to-many relacija. Preko toga želimo reći attach-aj rolu. Kada smo to sve odradili, pomoću `Auth::login` prijavljujemo korisnika u sistem. Ako je nešto u tom procesu pošlo po zlu, zapišemo (bilo koja iznimka iz bazne klase `\Exception`) to u Log i vraćamo korisnika na `dashboard`. Ista stvar je kod login-a. Šaljemo u servis i obrađujemo podatke i validiramo u requestu, servis će ih obraditi. Vadimo email i password, radimo attempt uz pomoć `$credentials` i izvlačimo van iz request-a session i regeneriramo `session` novi token, te redirektamo na `dashboard`.

Flow je ruta-kontroler-servis.

API (Aplikacijski programski interface) za rad s postovima

[Što je API?](#)

API (Aplikacijski programski interface) je skup definicija i protokola za izgradnju koji omogućavaju različitim softverskim aplikacijama da međusobno komuniciraju. Ponekad se naziva ugovorom između davaljatelja informacija (engl. information provider) i korisnika informacija -postavljajući sadržaj koji se traži od korisnika (poziv, engl. request) i sadržaj koji zahtjeva proizvođač (engl. producer) (odgovor, engl. response). Na primjer, dizajn API-ja za meteorološke usluge mogao bi specificirati da korisnik unese poštanski broj i da proizvođač odgovori dvodijelnim odgovorom, prvi je viša temperatura, a drugi niža.

Drugim riječima, ako želite komunicirati s kompjuterom ili sistemom kako biste dohvatali informacije ili izvršili funkciju, API vam pomaže priopćiti ono što želite tom sistemu kako bi on mogao razumjeti i ispuniti zahtjev. API omogućuje komunikaciju dva kompjutera preko web-a. Npr. aplikacija za dostavu može koristiti Google Maps API za podršku praćenju lokacije umjesto da izgradi svoju. Ili npr. web stranica s receptima može koristiti AI model da predloži alternativne recepte na temelju istih sastojaka koristeći ChatGPT REST API.

API možete zamisliti kao posrednika između korisnika ili klijentata i resursa ili web usluga koje žele dobiti. To je također način na koji organizacija može dijeliti resurse i informacije uz održavanje sigurnosti, kontrole i autentifikacije—određivanje tko čemu može pristupiti.

Što je RESTful API?

RESTful API je interface za programiranje aplikacija (engl. application programming interface - API) koje slijedi principe dizajna REST arhitektonskog stila. REST je skraćenica za reprezentativni prijenos stanja i skup je pravila i smjernica o tome kako biste trebali izgraditi web API.

REST je skup arhitektonskih ograničenja, a ne protokol ili standard. API programeri mogu implementirati REST na razne načine.

Kada se zahtjev klijenta podnese putem RESTful API-ja, on prenosi prikaz stanja resursa podnositelju zahtjeva ili krajnjoj točki. Ove informacije ili prikaz isporučuju se u jednom od nekoliko formata putem HTTP-a: JSON (Javascript Object Notation), HTML, XML, Python, PHP ili čisti tekst. JSON je općenito najpopularniji format datoteke jer, bez obzira na svoj naziv, ne ovisi o jeziku, a čitljiv je i za ljude i za strojeve.

Još nešto što treba imati na umu: zaglavla i parametri također su važni u HTTP metodama RESTful API HTTP zahtjeva, budući da sadrže važne identifikatorske informacije o metapodacima zahtjeva, autorizaciji, jedinstvenom identifikatoru izvora (URI), kešu, kolačićima i više. Postoje zaglavla zahtjeva i zaglavla odgovora, svako sa svojim podacima o HTTP vezi i statusnim kodovima.

Da bi se API smatrao RESTful, mora biti u skladu sa ovim kriterijima:

- Arhitektura klijent-server sastavljena od klijenata, poslužitelja i resursa, sa zahtjevima kojima se upravlja pomoću HTTP-a.
- Komunikacija klijent-poslužitelj bez stanja , što znači da se informacije o klijentu ne pohranjuju između zahtjeva za dobivanje i svaki zahtjev je odvojen i nepovezan.
- Podaci koji se mogu keširati koji pojednostavljaju interakcije klijent-server.
- Uniformni interface između komponenti tako da se informacije prenose u standardnom obliku.

To zahtjeva sljedeće:

- zatraženi resursi su prepoznatljivi i odvojeni od prikaza poslanih klijentu.
- resursima klijent može manipulirati putem reprezentacije koju prima jer reprezentacija sadrži dovoljno informacija za to.
- samoopisne poruke vraćene klijentu imaju dovoljno informacija za opis kako bi ih klijent trebao obraditi.
- hipertekst/hipermedij je dostupan, što znači da bi nakon pristupa resursu klijent trebao moći koristiti hiperveze za pronalaženje svih drugih trenutno dostupnih radnji koje može poduzeti.
- Slojeviti sistem koji organizira svaku vrstu servera (one odgovorne za sigurnost, balansiranje opterećenja itd.) uključivao je dohvaćanje traženih informacija u hijerarhije, nevidljive klijentu.
- Kod na zahtjev (izborno): mogućnost slanja izvršnog koda s poslužitelja klijentu kada se to zatraži, proširujući funkcionalnost klijenta.

Iako REST API ima ove kriterije s kojima se mora uskladiti, još uvijek se smatra lakšim za korištenje od propisanog protokola kao što je [SOAP \(Simple Object Access Protocol\)](#), koji ima specifične zahtjeve kao što su razmjena XML poruka i ugrađena sigurnost i usklađenost transakcija koje ga čine sporijim i težim.

Nasuprot tome, REST je skup smjernica koje se mogu implementirati po potrebi, čineći REST API-je bržim i lakšim, s povećanom skalabilnošću—savršeno za Internet stvari (IoT)⁷ i razvoj mobilnih aplikacija.

Osnove REST API-ja – 4 stvari koje treba znati

REST API je najčešći standard koji se koristi između klijenata (osoba ili aplikacija) koji žele pristupiti informacijama s weba s servera (aplikacija ili baza podataka) koji imaju pristup tim informacijama.

Interface za programiranje aplikacija (engl. application programming interface - API) je način na koji dva kompjutera međusobno komuniciraju kroz web. Na primjer, aplikacija za dostavu može koristiti Google Maps API za podršku praćenja lokacije umjesto da je izrađuje od nule. Ili web stranica s receptima može koristiti AI model za predlaganje alternativnih recepata na temelju istih sastojaka pomoću ChatGPT REST API-ja.

API koji slijedi REST standard naziva se RESTful. Evo dobrih praksi pri dizajniranju i korištenju RESTful API-ja na primjeru aplikacije za ankete.

Osnovni format

- HTTP metoda kojoj se šalje zahtjev (glagol, engl. verb)
- URL krajnje točke
- Statusni kodovi

Osnovni format

Format RESTful API-ja ima tri glavne komponente: 1) krajnju točku URL-a; 2) HTTP metodu kojom se šalje zahtjev (glagol, engl. verb); i 3) Tijelo.

Krajnja točka URL-a: je URL veza koja predstavlja resurse kojima želimo pristupiti ili ih poslati. Resursi (izvori) mogu biti tekst, slike, dokumenti ili bilo koji unos podataka. Na primjer, [primjer.com/surveys](#).

HTTP metoda kojom se šalje zahtjev (engl. verb): Reci serveru što želimo učiniti s resursom krajnje točke URL-a. Na primjer, [POST](#) zahtjev znači da želimo stvoriti novi predložak ankete, a [GET](#) zahtjev znači da želimo vidjeti postojeće podatke sa servera.

Tijelo poruke: Neobavezni je prilagođeni sadržaj koji sadrži poruku sa svojstvima i vrijednostima koje želimo koristiti za stvaranje ili ažuriranje određenog resursa.

⁷ IoT (engl. Internet of Things) ili Internet stvari odnosi se na mrežu fizičkih objekata, uređaja, vozila i drugih stvari koje su opremljene senzorima, softverom i drugim tehnologijama kako bi se povezale i razmjenjivale podatke putem interneta. IoT omogućava tim uređajima da prikupljaju, šalju i primaju podatke, što im omogućava automatsko djelovanje bez potrebe za ljudskom intervencijom. Primjeri IoT uređaja uključuju pametne termostate, sigurnosne kamere, pametne satove i industrijske senzore. Ova tehnologija se koristi u raznim područjima, kao što su pametne kuće, zdravstvena industrija, proizvodnja i transport.

HTTP metode kojih se šalje zahtjev (glagol, engl. verb)

Postoji 5 osnovnih naredbi kada se šalje HTTP zahtjev (glagol, engl. verb):

GET: napravi zahtjev samo za čitanje za pregled jednog ili popisa više izvora

POST: kreiraj novi resurs na temelju sadržaja danog u tijelu zahtjeva

DELETE: izbriši dani resurs na temelju navedenog ID-a

PUT: ažuriraj cijela polja resursa na temelju zadanog tijela zahtjeva ili kreirajte novo ako već ne postoji, na serveru, najčešće u bazi podatka

PATCH: ažuriraj samo polja resursa ako postoji, na serveru, najčešće u bazi podatka

Većina aplikacija i resursa će podržati sve ove naredbe. Ovo se obično naziva CRUD aplikacija:

| Skraćenica | HTTP metoda |
|--------------------------|-------------|
| Kreranje (engl. Create) | POST |
| Čitanje (engl. Read) | GET |
| Ažuriranje(engl. Update) | PUT i PATCH |
| Brisanje (engl. Delete) | DELETE |

Kada pogledamo [POST metodu po MDN dokumentaciji](#), HTTP **POST** metoda šalje podatke serveru. Tip tijela zahtjeva označen je zaglavljem **Content-Type**.

Razlika između **PUT** i **POST** je u tome što je **PUT idempotentan**: pozivanje jednom ne razlikuje se od pozivanja nekoliko puta uzastopno (nema nuspojava). Uzastopni identični **POST** zahtjevi mogu imati dodatne efekte, kao što su kreiranje istog naloga nekoliko puta. Ako pogledamo na freeCodeCamp [JavaScript POST zahtjev – Kako poslati HTTP POST zahtjev u JS-u](#), možemo kako poslati **POST** metodom. Možemo JavaScriptom poslati tijelo poruke. U jednom trenutku naići ćemo da se spremanje podataka na server ne radi **POST** metodom nego **PUT** metodom. U Restful-u se za to koristi **POST**. U principu se na serverskoj strani za pohranu treba koristiti **PUT** a ne **POST**. To nije slučaj u Restful-u i to treba zapamtiti. Vidjeli smo kod Laravela Rest Resource rutu (onaj Resource Controller [app/Http/Controllers/PostController.php](#)) koji za store koristi **POST**. Dakle treba primijetiti da se spremi s **PUT** a mi koristimo **POST**. To je postavljeno tako zato što ako nema podatka **PUT** ga može stvoriti, a **PATCH** ažurirati.

URL krajnje točke

Krajnja točka URL-a u RESTful API-ju predstavlja bilo koji objekt, podatak ili uslugu kojom API može pristupiti. Na primjer, [primjer.com/surveys](#) predstavlja podatke za sve predloške ankete i [primjer.com/surveys/123/responses](#) podatke za sve odgovore dane ankete.

Krajnje točke URL-a trebale bi biti grupirane kao imenice u množini oko poslovnih podataka i objekata, a ne oko metoda kojima se šalje zahtjev. Na primjer, [primjer.com/surveys](#) a ne [primjer.com/getAllSurveys](#).

Krajnje točke URL-a trebale bi uključivati jedinstveni identifikator kao put iz imenice u množini prilikom pregledavanja, ažuriranja ili brisanja pojedinačne stavke. Na primjer, [primjer.com/surveys/123](#).

Organizirajte zbirke URL-ova u logičnu hijerarhiju na temelju odnosa. Na primjer, korisnici će najvjerojatnije pristupiti odgovorima na dani predložak ankete. Ovo bi bilo predstavljeno kao primjer.com/surveys/123/responses.

Ovdje je sažetak krajnjih točaka URL-a dobre prakse i njihova interakcija s HTTP metodama kojoj se šalje zahtjev (glagol, engl. verb):

| URL krajnje točke resursa | URL | GET | POST | PUT | DELETE |
|---------------------------|-------------------------------------|------------------------------------|--------------------------------------------------------------|-------------------------------------------------------|-------------------------------------|
| /surveys | Dohvati sve ankete | Napravi novu anketu | Skupno ažuriranje anketa (ne preporučuje se) | Ukloni sve ankete (ne preporučuje se) | Dohvati sve ankete |
| /surveys/123 | Dohvati detalje za istraživanje 123 | Greška | Ažuriraj detalje ankete 123 ako postoje | Ukloni anketu 123 | Dohvati detalje za istraživanje 123 |
| /surveys/123/responses | Dohvati sve odgovore za anketu 123 | Napravi novi odgovor za anketu 123 | Skupno ažuriranje odgovora za anketu 123 (ne preporučuje se) | Ukloni sve odgovore za anketu 123 (ne preporučuje se) | Dohvati sve odgovore za anketu 123 |
| /responses/42 | Dohvati detalje za odgovor 42 | Greška | Ažurirajte detalje odgovora 42 ako postoji | Ukloni odgovor 42 | Dohvati detalje za odgovor 42 |

Statusni kodovi

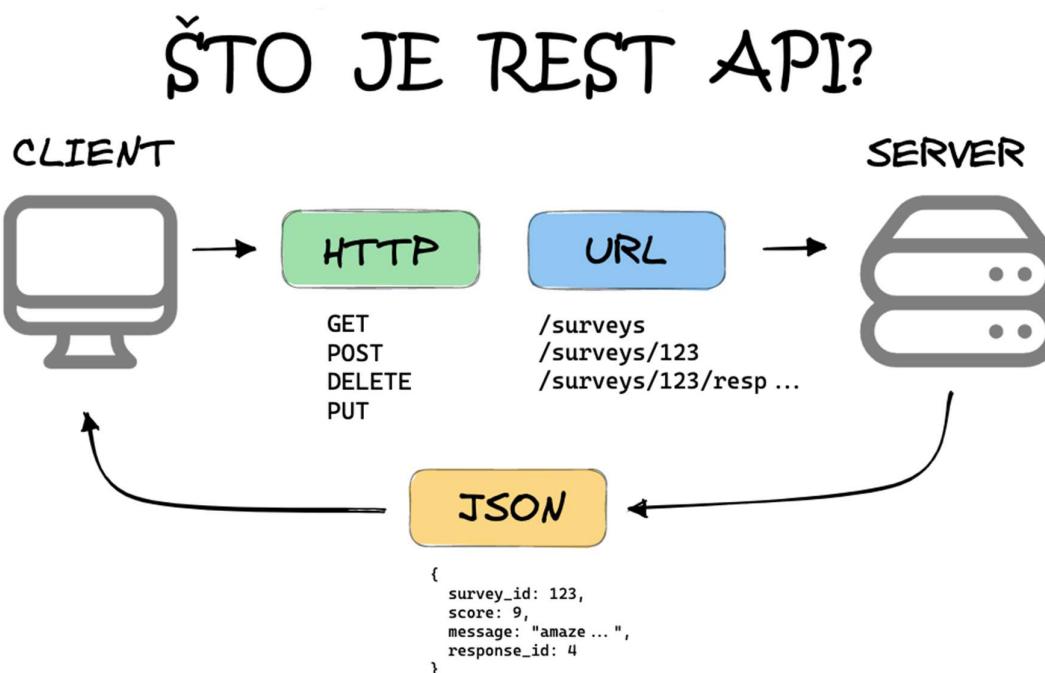
Po primitku HTTP zahtjeva u osnovnom RESTful formatu, server će vratiti HTTP statusni kod zajedno sa svim dodatnim JSON korisnim sadržajima. Ovdje su najčešći HTTP statusni kodovi i njihovo značenje:

| Šifra stanja | Značenje |
|----------------------------------------------------------|----------------------------------------------------------------------------------------------------|
| 200 OK | Zahtjev je bio uspješan. |
| 301 Preseljen za stalno (engl. Moved Permanently) | Za potrebe SEO-a kada je stranica premještena i potrebno je proslijediti sav trošak linka. |
| 401 Neovlašteno (engl. Unauthorized) | Server zahtijeva provjeru autentičnosti. |
| 403 Zabranjeno (engl. Forbidden) | Klijent je autenticiran, ali nema dopuštenja za pregled resursa. |
| 404 Nije pronađeno (engl. Not Found) | Stranica nije pronađena jer nema rezultata pretraživanja ili je možda nema na zalihu. |
| 500 Interna greška servera (engl. Internal Server Error) | Greška na strani servera. Obično zbog grešaka i iznimaka na strani servera. |
| 503 Server nedostupan (engl. Server Unavailable) | Greška na strani servera. Obično zbog hostinga platforme, preopterećenja i problema s održavanjem. |

Kreiranje REST API-ja u Laravelu s podrškom za JSON format podataka uključuje nekoliko koraka, od postavljanja ruta do implementacije kontrolera i modela.

Kada je neki problem na backend-u i API vrati kao error i vrati poruku sa statusom 200 je ono što trebamo izbjegići. Ne može biti sve OK, a vratiti grešku. RESTful API treba vratiti JSON. Status kodove treba hand-lati a ne ostanjati se na interne poruke. To je dobra praksa. Moguće je vratiti interne kodove kada se dogodi [400 Bad Request](#).

Danas svi dobri programeri imaju razrađen sistem poruka i ne pokazuju korisniku što se dogodilo. U monolitu smo upravljali pogledom. Ovdje response može biti u obliku poruke ili podataka ali je status kod jako bitan.



<https://mannhowie.com/rest-api>

Klijent radi neki zahtjev ([GET](#), [POST](#), [DELETE](#), [PUT](#)) na neki end point (tj. URL) na serveru. Server vraća neki JSON kao odgovor. Najčešće u tom zahtjevu može biti kao payload koji se šalje serveru da bi ga pohranili u zahtjevu. To se šalje u tijelu zahtjeva.

Ostali API u Laravelu

GraphQL

GraphQL je upitni jezik (query language) za API-je, koji služi za učinkovitu interakciju s podacima. Razvijen od strane Facebooka, GraphQL nudi fleksibilan način dohvatanja i manipulacije podataka u odnosu na tradicionalne REST API-je. Frontend ne može postati postati podatke bazi i napraviti upit. Frontend mora poslati request backendu a on onda poslati ga bazi. Frontend sa GraphQL može napistti SQL i poslati ga bazi. Ovaj API je agnostičan naspram podataka i end point-a.

Ključne prednosti GraphQL-a:

1. **Preciznost podataka:** Klijenti mogu točno specificirati koja im polja podataka trebaju, što smanjuje preopterećenje podacima. Kod REST-a, odgovori API-ja često uključuju previše ili premalo informacija, dok u GraphQL-u klijent dobiva upravo ono što traži.
Primjer: Umjesto da dobije cijelog korisnika s previše informacija, klijent može tražiti samo korisničko ime i email adresu.
2. **Jedinstvena krajnja točka:** GraphQL koristi jednu jedinstvenu krajnju točku za sve upite, za razliku od REST-a gdje svaki resurs obično ima svoju specifičnu krajnju točku.
3. **Mogućnost dohvaćanja povezanih podataka:** U jednom upitu moguće je dohvatiti više povezanih podataka, smanjujući potrebu za višestrukim zahtjevima. Na primjer, možeš dobiti korisničke podatke zajedno s podacima o njegovim narudžbama u jednoj interakciji.
4. **Snažna dokumentacija:** GraphQL dolazi s ugrađenim alatima za introspekciju, što omogućava automatsko generiranje dokumentacije i lakšu integraciju.

GraphQL API funkcioniра tako što klijent šalje upit u formi JSON-a prema serveru, koji odgovara strukturiranim JSON podacima. GraphQL se koristi kada je potrebno efikasnije upravljanje podacima, posebno u složenim aplikacijama gdje je potrebno dohvaćati više povezanih entiteta u jednom zahtjevu, ili smanjiti preopterećenje podacima koje često dolazi s REST API-jem.

Da bi koristio GraphQL s Laravelom, trebao bi integrirati neki paket koji omogućava rad s GraphQL API-jima. Dva najpopularnija paketa za to su **Rebing's GraphQL** i **Laravel Lighthouse**, koji pružaju implementaciju GraphQL servera u Laravel aplikaciji.

[SOAP API](#)

SOAP (Simple Object Access Protocol) je XML-bazirani protokol za razmjenu informacija preko mreže. Iako je manje popularan od RESTful API-ja, SOAP se još uvijek koristi u nekim starijim sistemima i aplikacijama koje zahtijevaju sigurnost na razini transakcija.

Za razliku od REST API, koji je konfigurabilan, SOAP API se oslanja na XML i uspostavljenom šemom, pa je lako moguće čupati podatke van. SOAP se teže konfigura, treba naložiti novu šemu.

Laravel nema ugrađenu podršku za SOAP, ali se može koristiti s vanjskim paketima kao što su Artisan Soap ili putem PHP-ovih ekstenzija za SOAP.

Obično se koristi u aplikacijama koje komuniciraju s naslijeđenim sistemima (engl. legacy systems), posebno u finansijskom sektoru. Koristi se kod fiskalizacije.

[gRPC \(engl. Google Remote Procedure Call\) API](#)

gRPC je moderni open-source **RPC (Remote Procedure Call)** okvir razvijen od strane Googlea. Omogućuje brzu komunikaciju i podržava binarni prijenos podataka (umjesto JSON-a). Koristi Protocol Buffers (protobufs) za serijalizaciju podataka.

Laravel trenutno nema izravnu podršku za gRPC, ali se može integrirati putem vanjskih biblioteka kao što su Grpc PHP ili prilagođenim rješenjima.

Koristi se za visoko performansne aplikacije koje trebaju brz prijenos podataka, kao što su microservices arhitekture.

JSON-RPC API

Opis: JSON-RPC je lagani protokol koji omogućuje pozivanje udaljenih funkcija korištenjem JSON-a. Sličan je RESTful API-ju, ali koristi udaljene procedure umjesto tradicionalnih HTTP metoda kao što su GET i POST.

Laravel nema nativnu podršku za JSON-RPC, ali se može dodati pomoću vanjskih biblioteka.

Koristi se za jednostavne i brze udaljene procedure, gdje je potrebna mala potrošnja resursa.

WebSocket API

WebSocket omogućava dvosmjernu komunikaciju između servera i klijenta, što je idealno za aplikacije u realnom vremenu, kao što su chat aplikacije ili notifikacijski sistemi.

Laravel nudi podršku za WebSockets putem paketa kao što je Laravel Websockets, koji pruža alat za implementaciju real-time aplikacija.

Koristi se za aplikacije koje zahtijevaju real-time interakcije, kao što su chatovi, live nadzor ili gaming aplikacije.

FaaS (Function as a Service) API

FaaS omogućava kreiranje "serverless" funkcija koje se pokreću kao odgovor na događaje. Ovi API-jevi omogućuju da se pojedine funkcije koriste kao usluge koje mogu obavljati određene zadatke bez potrebe za održavanjem cijelog servera.

Laravel može koristiti FaaS arhitekturu koristeći AWS Lambda, Google Cloud Functions ili druge cloud providere za serverless funkcionalnosti.

Koristi se za aplikacije gdje su potrebne izolirane funkcije, na primjer, pozivanje udaljenih servisa na osnovu događaja.

Pusher API (Real-time API)

Opis: Pusher omogućuje real-time komunikaciju između frontend-a i backend-a kroz kanale i događaje. Koristi se za slanje notifikacija, chat poruka ili bilo kakve real-time akcije bez potrebe za stalnim osvježavanjem stranice.

Laravel ima ugrađenu podršku za Pusher kroz Broadcasting kanal, što omogućava slanje real-time podataka kroz WebSockets.

Koristi se za aplikacije koje zahtijevaju notifikacije u stvarnom vremenu, kao što su aplikacije za chat ili notifikacijski sistemi.

SWAPI API (Star Wars API)

SWAPI API (Star Wars API) je besplatan i otvoren API koji omogućava pristup podacima iz Star Wars svemira. Služi kao RESTful API (engl. Representational State Transfer API) za dohvaćanje informacija o likovima, filmovima, planetima, vozilima, vrstama i brodovima iz Star Wars franšize.

Programeri koriste SWAPI API kako bi dohvatali ove podatke i integrirali ih u svoje aplikacije ili web stranice. Na primjer, možete poslati request (zahtjev) za podatke o Darth Vaderu i dobiti response

(odgovor) s detaljnim informacijama o njemu, poput njegovog identiteta, visine, težine i povezanih filmova.

SWAPI API je odličan alat za vježbanje rada s API-jem, jer omogućava testiranje i implementaciju API poziva u stvarnim aplikacijama.

API backend u Laravelu

Laravel također može poslužiti kao API backend za JavaScript jednostraničnu aplikaciju ili mobilnu aplikaciju. Na primjer, možete koristiti Laravel kao API backend za svoju Next.js aplikaciju. U tom kontekstu, možete koristiti Laravel za provjeru autentičnosti i pohranjivanje/dohvaćanje podataka za svoju aplikaciju, dok također iskorištavate Laravelove moćne usluge kao što su redovi čekanja (engl. queues), e-pošta, notifikacije i više toga. Možemo koristiti ugrađeni Laravel Sanctum ili JWT koji ćemo mi koristiti.

Da bi API radio u Laravelu, moramo ga natjerati da se ponaša kao API a ne kao monolit.

Laravel Sanctum

Laravel Sanctum je sistem provjere autentičnosti za SPA (engl. single page applications, aplikacije na jednoj stranici), mobilne aplikacije i jednostavne API-je temeljene na tokenu. Sanctum omogućuje svakom korisniku vaše aplikacije generiranje više API tokena za svoj račun. Ovim tokenima mogu se dodjeliti sposobnosti/opsezi koji određuju koje radnje tokeni smiju izvoditi. Predavač kaže da ga do sada nije niti jednom koristio jer nije koristio njegove API tokene.

JWT

JWT (JSON Web Token) je kompaktni, URL-sigurni format za predstavljanje podataka kao JSON objekta između dviju strana. Najčešće se koristi za autentifikaciju i autorizaciju u web aplikacijama. JWT omogućuje prijenos podataka na siguran način, osiguravajući da se može potvrditi njihova autentičnost i da podaci nisu izmijenjeni. Nema tehnologije ni jezika koji nema podršku za JWT.

Kako funkcioniра JWT?

JWT se sastoji od tri dijela:

Zaglavljive

Nosivi dio(korisni podaci)

Potpis

Ovi dijelovi su odvojenim točkama (.) i predstavljeni su kao jedan niz (string), što čini JWT lako prenosivim preko URL-ova, POST tijela i HTTP zaglavlja.

1. Zaglavje

Zaglavje sadrži metapodatke o tokenu, uključujući:

- Tip tokena (uvijek "JWT")
- Algoritam za potpisivanje (npr. HMAC SHA256 ili RSA)
- Primjer zaglavja (u JSON formatu):

```
{  
  "alg": "HS256",  
  "typ": "JWT"  
}
```

2. Nosivost (korisni podaci)

Payload sadrži korisne podatke (claims). To su informacije koje želite prenijeti u tokenu, kao što su korisnički ID, vrijeme isteka tokena i drugi podaci. Mogu biti dvije vrste:

- **Registrirane tvrdnje:** Standardizirane informacije (npr. `iss` - izdavač, `exp` - datum isteka, `sub` - subjekt).
- **Javna potraživanja:** Proširive informacije koje sami možete definirati (npr. korisnički podaci kao što je `user_id`).
- **Privatni zahtjevi:** Koriste se za podatke specifične za aplikaciju, koje mogu biti dogovorene između strana.

Primjer korisnog opterećenja-a:

```
{  
  "sub": "1234567890",  
  "name": "John Doe",  
  "admin": true,  
  "exp": 1700000000  
}
```

3. Potpis (potpis)

Potpis služi za osiguranje integriteta tokena i provjera autentičnosti. Generira se kombiniranjem:

- Encodiranog zaglavja
- Encodiranog nosivosti-a
- Tajnog ključa ili privatnog ključa (ovisno o algoritmu)

Primjer potpisivanja (korištenjem HMAC SHA256):

```
HMACSHA256(  
  base64UrlEncode(header) + "." + base64UrlEncode(payload),  
  secret  
)
```

Potpis osigurava da se sadržaj tokena nije mijenjao. Ako bilo tko promijeni korisni teret, potpis će se promijeniti i token neće biti valjan.

Kako JWT radi u praksi?

Prijava: Kada korisnik unese ispravne pristupne podatke, poslužitelj generira JWT koji sadrži korisničke podatke.

Zaštita: JWT se obično vraća korisniku na odgovor i može biti pohranjen u localStorage , sessionStorage ili HTTP kolačiće .

Slanje zahtjeva: Prilikom svakog budućeg zahtjeva, klijent šalje JWT u zaglavje (obično kao `Authorization: Bearer <token>`).

Provjera autentičnosti : Server prima zahtjev, provjerava token i potpis kako bi osigurao da je validan i nije promijenjen, te na temelju podataka u tokenu dopušta ili odbija pristup.

Prednosti JWT-a

Sigurnost: Podaci unutar tokena su potpisani, pa se može potvrditi njihova autentičnost.

Samostalnost: Token sadrži sve potrebne informacije za autentifikaciju. Nema potrebe da server održava stanje (stateless).

Prijenosnost: Kompaktan format, lako se prenosi u URL-ovima i zaglavljima HTTP zahtjeva.

Nedostaci JWT-a

Veliki korisni teret : ako se u JWT stave veliki podaci, token može postati prevelik za prijenos.

Ne možete se opozvati : Kada je JWT izdan, ne možete ga lako opozvati (osim ako koristite "blacklisting" ili kraće vrijeme trajanja tokena).

Primjer JWT-a

Evo jednog JWT-a (kodirano base64):

```
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOiIxMjM0NTY3ODkwIiwibmFtZSI6IkpvaG4gRG91IiwiZW1haWFOIjoxNTE2MjM5MDIyfQ.SfIKxwRJSMeKKF2QT4fwpMeJf36P0k6yJV_adQssw5c
```

Ovaj JWT se sastoji od:

Zaglavljia: `eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9`

Nosivih podataka (korisnih podataka):

```
eyJzdWIiOiIxMjM0NTY3ODkwIiwibmFtZSI6IkpvaG4gRG91IiwiZW1haWFOIjoxNTE2MjM5MDIyfQ
```

Potpisa: `SfIKxwRJSMeKKF2QT4fwpMeJf36P0k6yJV_adQssw5c`

JWT tokeni vrlo su korisni za moderne aplikacije gdje se želi eliminirati potreba za držanjem stanja na poslužitelju, a mogu se koristiti u mnogim scenarijima, uključujući **autentifikaciju korisnika, autorizaciju pristupa i prijenos sigurnih podataka**.

Kreiranje Laravel projekta

Instalirat ćemo novi Laravel paket pomoću Composera, [slično kako smo radili s monolit projektom](#).

```
composer create-project laravel/laravel algebra-blog-api
```

`algebra-blog-api` je naziv projekta.

```
INFO Preparing database.
Creating migration table ..... 120.64ms DONE
INFO Running migrations.
0001_01_01_000000_create_users_table ..... 681.73ms DONE
0001_01_01_000001_create_cache_table ..... 266.28ms DONE
0001_01_01_000002_create_jobs_table ..... 807.45ms DONE

Administrator@DESKTOP-4J5DF41 MINGW64 /i/Laravel
$ |
```

`web` middleware Grupa u principu se veže uz rute koje se nalaze u `routes/web.php`. Dok za `api` middleware grupu se veže samo jedan middleware koji se koristi. To znači da na api middleware nema `@csrf` tokena. Na `api` također nema formi (obrazaca) ni sesija, koristimo cookie. Doslovno nemamo pristup klijentu u smislu browsera, mobile-a, mi smo samo dio backend-a koji nema doticaja sa klijentom osima kada klijent pošalje zahtjev za nekim podacima.

Predavaču se ne sviđa jer nema mogućnost da prebrika autentifikacijski sistem. Kada smo otišli na neku stranicu koja je tražila da korisnik mora biti prijavljen u sistem, Laravel je po default-u kada je uključio Auth Guard.rekao preusmjeri ga na login rutu. Na API uopće nema toga. Ako netko proba doći do end pointa koji ste zaštitili, mora biti prijavljen na sistem. Netko tko će napraviti request na taj end point, mora biti prijavljen u sistem, ne želimo da Laravel kaže da korisnik nije prijavljen i redirektamo ga na login page, jer toga nemamo. Mi moramo vratiti neki odgovor a to je 401, da korisnik nije autenticiran.

Loger bilježi to jer je to reportable exception i kaže da nije pronašao rutu login. To je zato što Laravel ne handle-a, kad prebacimo na API.

Vratimo se na API projekt. Pokrenut ćemo VSC. Ne želimo instalirati Laravel Sanctum i nećemo koristiti

```
php artisan install:api
```

Ručno ćemo odraditi neke stvari jer ćemo koristiti RESTful API. Ako pogledamo rute u direktoriju `routes`, imamo `console.php` i `web.php` a nemamo `api.php` rute. `console.php` rute su za pokretanje vremenski planiranih zadataka (engl. scheduling tasks) u određeno vrijeme, za nekakve provjere zadataka, automatizaciju reporta na email i sl. Npr. svaki ponedjeljak pošalji neki report, koliko je objavljeno članaka i sl. Ako pogledamo sadržaj ove datoteke vidimo:

```
<?php

use Illuminate\Foundation\Inspiring;
use Illuminate\Support\Facades\Artisan;

Artisan::command('inspire', function () {
    $this->comment(Inspiring::quote());
})->purpose('Display an inspiring quote')->hourly();
```

Ako pogledamo tu imamo naredbu `'inspire'` i ako je pozovemo dobit ćemo nekakav `quote()` tj. citat.

```
php artisan inspire
```

```
$ php artisan inspire
“ Simplicity is an acquired taste. ”
– Katharine Gerould
```

`web.php` nam ne treba. Želimo koristiti `api.php` koi ćemo napraviti također u `routes` direktoriju.

```
<?php

use Illuminate\Support\Facades\Route;

Route::get('/', function () {
    return response()->json(['message' => 'Hello World!']);
});
```

Ovdje koristimo anonimnu funkciju kao rutu (vidi callable funkcije). U odgovoru šaljemo response JSON. JSON-u predajemo ili matricu ili kolekciju koja će se string-ificirati. JSON je običan tekst koji sa back-end-a možemo prenijeti na bilo koju drugu platformu. Podaci se **serijaliziraju** tj. pretvaraju u format koji se može lako pohraniti ili prenijeti a to je upravo JSON. Nakon serijalizacije, podaci se mogu poslati preko mreže, pohraniti u datoteku ili bazu podataka, a kasnije ponovno rekonstruirati u originalni oblik kroz proces poznat kao **deserijalizacija**. Svi programske jezice razumiju JSON.

Javna funkcija `json()` je default dužine 200. Ako odgovaramo s greškom, možemo upisati.

```
>json(['message' => 'Hello World!']);  
Illuminate\Contracts\Routing\ResponseFactory::json  
Create a new JSON response instance.  
<?php  
public function json(  
    $data = [],  
    $status = 200,  
    array $headers = [],  
    $options = 0  
) { }  
@param mixed $data  
@param int $status
```

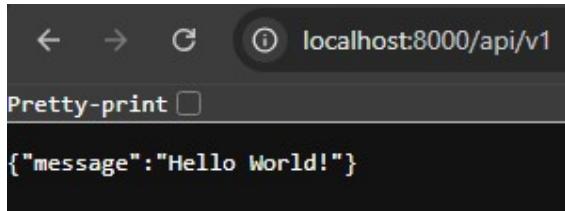
Obrisat ćemo i `web.php` s web rutama jer nam to ne treba.

Otići ćemo u `bootstrap` direktorij i doraditi `app.php`:

```
<?php  
  
use Illuminate\Foundation\Application;  
use Illuminate\Foundation\Configuration\Exceptions;  
use Illuminate\Foundation\Configuration\Middleware;  
  
return Application::configure(basePath: dirname(__DIR__))  
    ->withRouting(  
        web: __DIR__.'/../routes/web.php',  
        commands: __DIR__.'/../routes/console.php',  
        health: '/up',  
    )  
    ->withMiddleware(function (Middleware $middleware) {  
        //  
    })  
    ->withExceptions(function (Exceptions $exceptions) {  
        //  
    })->create();
```

Promijenit ćemo red `web: __DIR__.'/../routes/web.php'`, da glasi `api: __DIR__.'/../routes/api.php'`. Ispod tog reda dodat ćemo: `apiPrefix: 'api/v1'`, Najčešće ovo što je iza kose crte je ime domene ili v1 (verzija 1) ili tako nešto. Brišemo i red `health: '/up'`.

Pogledat ćemo `localhost:8000/api/v1`. Request možemo napraviti kroz browser



```
localhost:8000/api/v1
Pretty-print □
{"message": "Hello World!"}
```

Ako kažemo `localhost:8000/api/v1/asasdfafsd`:

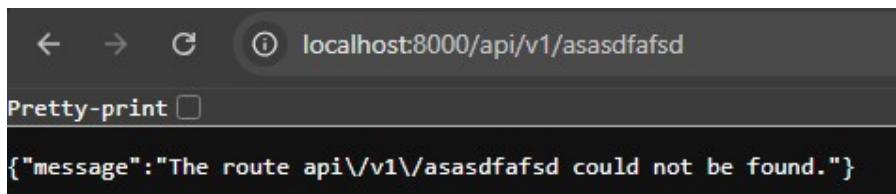


Vraća nam view, a to ne želimo. Ovo je zato što je Laravel prvenstveno fokusiran na monolit a API je došao kasnije.

U `bootstrap/app.php` u `withExceptions` želimo uhvatiti sve iznimke koje su renderable. Laravel ima podjelu iznimki u reportable i renderable. Ako pogledamo u logs vidjet ćemo da iznimka nije zapisana tj. nije reportable. I to je u redu. Ova iznimka je renderable a to znači da kad se to dogodi vratit će view.

Dakle otici ćemo u `withException` i u redu ispod dodati:

```
->withExceptions(function (Exceptions $exceptions) {
    $exceptions->renderable(function (Throwable $e) {
        return response()->json(['status' => 'ERR'], $e->getMessage());
    });
})->create();
```



```
localhost:8000/api/v1/asasdfafsd
Pretty-print □
{"message": "The route api\\v1\\asasdfafsd could not be found."}
```

Ne hvatamo pojedinu iznimku i onda na temelju nje gledamo da se dogodilo `404`, nego uhvatimo sve renderable iznimke koje se dogode i javimo 500. To nije baš pametna ideja ali ako ponovo pogledamo, nećemo više dobiti view, nego će `withExceptions` sve renderable uhvatiti, predusretni i vrati kao JSON odgovor. Tih grešaka ima dosta. Not found nije greška `500` nego `404`.

Potrebno je razviti svoj servis koji će na temelju iznimke koja se dogodila primati greške. `renderable` prima callback funkcije. Ta callback funkcija kada se dogodi exception, pošalje taj exception toj callback funkciji. Ne mora to biti exception, može biti i error. `Throwable` interface je implementiran i na exception i na error. Cilj je razviti servis kojem ćemo poslati iznimku i onda to raspisati ovisno koja iznika se dogodila. Želimo preusmjeriti kako se upravlja iznimkama, ovako je hardkodirano. To ćemo napraviti kasnije.

Mi ne znamo koji je ovdje exception. Provjerit ćemo koji je exception s `dd($e)`; iznad `return`:

```
Symfony\Component\HttpKernel\Exception\NotFoundHttpException {#205 ▾ // bootstrap/app.php:18
  #message: "The route api/v1/asasdfafsd could not be found."
  #code: 0
  #file: "I:\Laravel\laravel\framework\src\Illuminate\Routing\AbstractRouteCollection.php"
  #line: 44
  -statusCode: 404
  -headers: []
  trace: {▶}
}
```

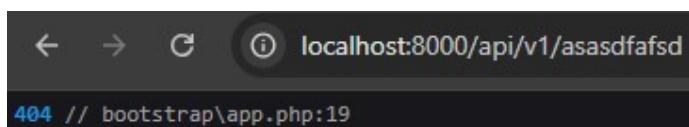
Dobijamo informaciju da se radi o `NotFoundHttpException`.

Promijenit ćemo `dd($e);` u `dd($e instanceof NotFoundHttpException);`.

```
true // bootstrap/app.php:19
```

Dobili smo `true`. Sada možemo razviti servis i usporediti ga s određenom instancom i onda će nam status biti `404`. Ako pogledamo bolje sliku, možemo izvući status kod, to je `404`.

Možemo napisati `dd($e->getStatusCode());`



Sada možemo reći:

```
->withExceptions(function (Exceptions $exceptions) {
    $exceptions->renderable(function (Throwable $e) {
        return response()->json(['message' => $e->getMessage()], $e-
    >getStatusCode());
    });
})->create();
```

A screenshot of a browser's developer tools Network tab. The request 'asasdfafsd' is highlighted in red. The details panel shows the following information:

| | |
|------------------|-----------------------------------------|
| Request URL: | http://localhost:8000/api/v1/asasdfafsd |
| Request Method: | GET |
| Status Code: | 404 Not Found |
| Remote Address: | 127.0.0.1:8000 |
| Referrer Policy: | strict-origin-when-cross-origin |

Kako nemamo poglede, nismo dužni korisniku poslati bilo kakvu poruku. Ili možemo poslati poruku koja će biti generička npr. status 0 ili status 1, status err ili status ok. Mogu biti i neke iznimke koje ne želimo da se zapisuju u log-ove.

Ako je odgovor pozitivan, moramo ga nekako strukturirati. Složit ćemo nekakav servis koji će nam biti response.

Dok god je **GET** request ne treba nam Postman, koristit ćemo ga isto kasnije.

Izgradnja API

Pokušat ćemo dići određene rute u **routes/web.php**. Krenut ćemo sa '**Login**' ali ovaj puta nemamo '**showLogin**' rutu s kojom bi prikazali nekakav view, nego ćemo imati samo rutu kojom šaljemo podatke za prijavu, rutu za registraciju, logout, nekakav admin dio. Nećemo imati dashboard dio, u **resource** dijelu nećemo imati **create** i **edit** metodu gdje se pokazuju forme za editiranje odnosno kreiranje resursa.

Pogledat ćemo kako ujediniti da odgovor koji se vraća klijentu bude ujednačen (frontend dio) ako je uspješno request obrađen i ako nije uspješno odraćen. Hoćemo to nekako ujednačiti.

O zahtjevima u MVC (engl. model-view-controller za razvoj korisničkog interface-a) ne brine view (jer ga nemamo), već JSON odgovor koji vraćamo. Dakle nemamo konkretni HTML. Za obradu zahtjeva zadužen je kontroler. Želimo da svaki kontroler koji imamo ujednačeno prati odgovor. O tome smo pričali i na monolitu. Želimo iskoristiti apstraktne kontrolere **app/Http/Controllers/Controller.php** u kome ćemo definirati zaštićenu metodu s **protected**. Zaštićenu znači da će jedino moći koristiti klase koji naslijede ovu apstraktnu klasu, a to će biti svi ostali kontroleri: ako kažemo **AuthController.php** on nasljeđuje **Controller**, ako kažemo **PostController.php** on nasljeđuje također **Controller** itd.

```
<?php

Namespace App\Http\Controllers;

Abstract class Controller
{
    //
}
```

Poanta je da ćemo nad tim kontrolerom definirati vrlo jednostavnu logiku kojom ćemo pokušati ujednačiti odgovor ako je request uspješno obrađen i u slučaju ako se dogodila neka iznimka, kako ćemo odgovoriti u slučaju iznimke. Metodu ćemo nazvati **executeServiceAction()**. Sva poslovna logika se ne dešava u kontroleru, nego u nekom servisu. Cilj je da u svakom svojem kontroleru kreiramo neku funkciju koja će u sebi pozvati servis, i tu funkciju proslijedimo **executeServiceAction()**. Dakle funkciju šaljemo funkciji. Takvu situaciju smo već imali u monolitu u **bootstrap/app.php** gdje **renderabe()** metodi u **Exceptions** šaljemo funkciju.

```
->withExceptions(function (Exceptions $exceptions) {
    $exceptions->renderable(function (Throwable $e) {
        return response()->json(['status' => 'ERR'], $e->getMessage());
    });
})->create();
```

Ta funkcija je **callable tipa**. Pojam callable u Laravelu (i PHP-u) odnosi se na bilo koji tip funkcije ili metode koja može biti pozvana. PHP podržava različite tipove callable funkcija, što omogućava fleksibilnost u pisanju koda i prosljeđivanju funkcija kao argumenata drugim funkcijama ili metodama. Inače, callable funkcija može biti: Naziv funkcije kao string — npr. `'imeFunkcije'`, matrica s klasom i metodom — npr. `[Klasa::class, 'imeMetode']`, anonimna funkcija (closure) — npr. `fn() => ...` ili `function () { ... }`, lambda funkcija (kratka forma anonimne funkcije) — npr. `fn($x) => $x * 2`. Ako imamo neku našu iznimku to neće biti obuhvaćeno s ovime.

Dakle možemo je poslati. Definirajmo neku test funkciju:

```
function test()
{
    return 'test';
}
```

Ako napišemo `test;`, PHP smatra da pozivamo konstantu. Ako napišemo `test()`; onda tek pozivamo funkciju i ona je po tipu **callable**.

Želimo spriječiti da metodi `executeServiceAction()` možemo poslati bilo koji tip podatka. Želimo omogućiti da isključivo možemo poslati funkciju tj. callable tip jer se može pozvati. Mi želimo tu funkciju pozvati da ona nešto radi. Zato ćemo napisati `executeServiceAction(callable $serviceAction)`. Ovo znači da će je pozvati ali ne zna kakva je.

```
abstract class Controller
{
    protected function executeServiceAction(callable $serviceAction)
    {
        try {
            return
        } catch (\Exception $e) {
            return $e;
        }
    }
}
```

Koristili smo **try-catch** blok, gdje se ovdje manje-više sve događa. Kada pokušamo pozvati na kontroleru neku metodu, ta metoda će pozvati u kontroleru

`app/Http/Controllers/Controller.php/executeServiceAction()` i proslijedit će joj funkciju koja poziva neki servis, tj. metodu u servisu. Kada se pozove `executeServiceAction()`, ako dođe do pucanja zbog iznimke (generirane od strane nas ili Laravela), uhvatit će ga `try-catch` blok. Ne želimo raditi samo sa exception, želimo raditi i sa error-ima i zato ćemo raditi sa `Throwable`. I `\Exception` i `\Error` implementiraju `Throwable`. Umjesto da se oslonimo na klasu, oslonit ćemo se na interface. Zato možemo reći:

```
} catch (\Throwable $e) {
```

Tako ćemo uhvatiti i `\Exception` i `\Error`. Spuštamo se na nivo apstakcije i obje klase su dobro došle.

Također kreiramo nekakav `ResponseService` u return-u. Tu pozivamo parametar `$serviceAction` ali kao fukciju, sa `O`, dakle poziva callable tip a to je funkcija. `callable` tip nam garantira da ćemo u parametru dobiti funkciju koju ćemo pokrenuti. Mi želimo tu funkciju pozvati da nešto napravi. A što će ta funkcija napraviti `$serviceAction` ne zna, on je agnostik. Kada je pozove, rezultat izvršavanja te funkcije će biti proslijeđen kao podatak `success()` metodi u `ResponseService()`. Naš servis koji se treba izvršiti će biti pozvan uz pomoć funkcije `$serviceAction()`. Servis ne smije vratiti nešto što nije implementirano. `ResponseService` će biti static, tj. metoda `success()` će biti static (zadržavati stanje od prethodnog requesta), zato što ne želimo da dižemo za svaki servis svaki puta novi objekt, želimo da radimo na nivou klase i na nivou metode. Poanta je da nećemo imati ništa što će stvarati poteškoću zato što smo zadržali nešto iz prošlog request-a. Nećemo morati ništa resetirati a tako štedimo memoriju jer za svaki request ne dižemo novu instancu ovog response servisa, nego svi rade s istom klasom i istom metodom.

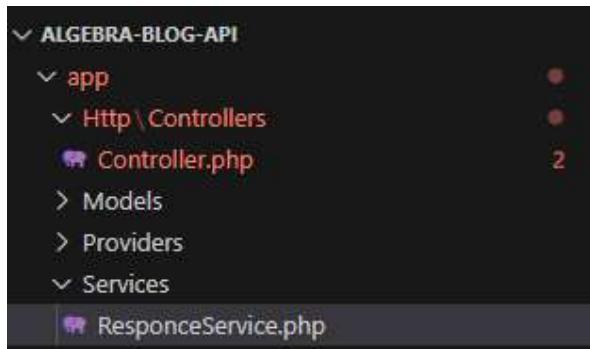
```
abstract class Controller
{
    protected function executeServiceAction(callable $serviceAction)
    {
        try {
            return ResponseService::success($serviceAction());
        } catch (\Throwable $e) {
            return ResponseService::error($e);
        }
    }
}
```

Ako pođe po zlu, vratit ćemo `ResponseService::error($e)`. To znači da naš `ResponseService` ima dvije `static` metode: `serviceAction()` i `error()`. One su i `public`, javno dosupne.

Doslovno svi kontroleri koje ćemo kreirati će kao `return` imati ono što ova metoda `executeServiceAction()` napravi. To želimo zato da unificiramo odgovor i budemo sigurni da će svaki kontroler dati isti odgovor u smislu strukture.

Kako ne bi komplikirali unutar svakog našeg kontrolera, napravili smo logiku unutar vršnog kontrolera `app\Http\Controllers\Controller.php`. Poanta try-catch bloka da ako u `$_serviceAction()`, koji će se pokrenuti putem callable naše anonimne funkcije, ovdje će se exception uhvatiti i preusmjeriti odgovor. Ako gledamo servis u kojem dođe do iznimke, taj servis neće vratiti podatke koji trebaju pristići u `success()`, nego će iznimka automatski trigerirati `catch` blok.

Za sada nemamo `ResponseService`, dakle trebamo ga kreirati. Kako nemamo direktorij `Services` u `app`, kreirat ćemo ga. U njemu ćemo napraviti `ResponseService.php`.



```
<?php

namespace App\Services;

class ResponseService
{
    public static function success($data, $status = 200)
    {
        return response()->json(['status' => 'OK', 'data' => $data], $status);
    }

    public static function error(\Throwable $e)
    {
        return response()->json(['status' => 'ERR', 'message' => $e->getMessage()],
        $e->$statusCode);
    }
}
```

Kod `$status 200`, potencijalno se ostavlja mogućnost da pošaljemo neki drugi status. Ako se vratimo na `app\Http\Controllers\Controller.php` i u metodi `executeServiceAction`, `success()` će biti po default-u, 200 The request succeeded. Mi možemo razviti da se `executeServiceAction` kada se poziva u nekom kontroleru možemo poslati status, koji će `success()` proslijediti dalje. Za sada ćemo ostaviti 200.

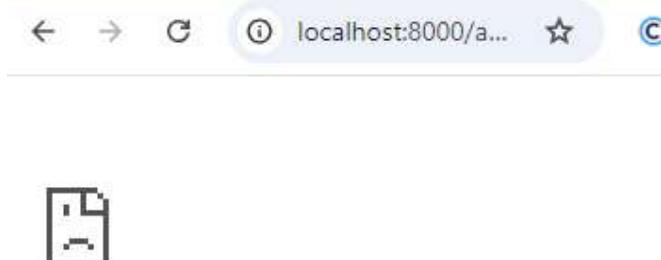
Ako pogledamo HTTP status 201 Created, to bi bio poželjan odgovor u smislu kada klijent šalje neke podatke API da se pohrane, resource da se kreira i želimo mu javiti da je prošlo u redu, tj. da smo uspješno kreirali što je tražio. To je puno bolje od generalnog 200.

Ako pogledamo HTTP statuse imamo tu još 202 Accepted. 204 No Content ćemo slati kada klijent zahtjeva da izbriše neki resource. Možemo to koristiti bilo gdje gdje ne vraćamo neki sadržaj.

Uvijek ćemo imati strukturu. Kada je zahtjev u redu, dobit će 'status' OK i ključ 'data' u kojem će biti podaci koji su vraćeni. Može biti da podataka nema ako vraćamo No content status. Kada se radi sa greškom, onda je 'status' ERR. Pošljemo eventualno 'message'. Bilo bi poželjno da frontend ove poruke ne ispisuje nego daje neke svoje poruke. Status kod izvlačimo iz exception, a ako ga nema da li će status kod biti null, ovisi od iznimke do iznimke.

\$code će uvijek biti nula u početku Exception. Ako postoji Exception koji daje 0 pogledamo što će se desiti. Iz routes/api.php morali bi poslati 0:

```
Route::get('/', function () {
    return response()->json(['message' => 'Hello World!'], 0);
});
```



Dobili smo grešku jer je negdje pukao – nešto nije u redu. Vidimo da se desila HTTP greška 500. Ako pogledamo u log datoteci, vidjet ćemo The HTTP status code „0“ is not valid.

Mogli bi u routes/api.php napisati dd(0 ?? 1);

Nullish operator (??) u PHP-u koristi se za postavljanje vrijednosti varijable samo ako je ta varijabla null. Ne provjerava da li je 0, prazan string ili false. Kako 0 nije null, operator neće preći na desnu stranu. Rezultat će zato biti 0.

```
0 // routes\api.php:7
```

Vidimo da nula nije null. Mi želimo da ako dobijemo 0 da bude 1. Ako probamo s ternarnim operatorom: dd(1 ? 1 : 400);. Prva jedinica se evaluira kao true ili false. Kako je true, izvršava se druga 1. Da je false, izvršio bi se broj iza dvotočke. Ima i jednostavniji način (i efektniji) s Elvis

operatorom `dd(1 ?: 400);`. To je skraćeni oblik ternarnog operatora. Provjerava samo jednu vrijednost i vraća je ako je truthy ili drugu ako je falsy (`false`, `0`, prazan string „“ ili `null`). Kako je 1 truthy jer nije `false`, `0`, prazan string „“ ili `null`, Elvis vraća 1 a desna strana se zanemaruje.

Želimo postaviti default value za status kod u slučaju da status kod ne dođe. Zato ćemo u `app/Services/ResponseService.php` na početku `error()` metode definirati:

```
$statusCode = $e->getStatusCode() ?: 400;
```

Cilj nam je da uvijek imamo ispravan status kod. Mi ne možemo znati ako je netko kreirao nekakav exception koji ima neispravan status kod. Ako radimo custom exception, onda imamo potencijalni problem. Ako sada slučajno završi 0 u status kodu, imat ćemo grešku 400 a aplikacija neće puknuti. Kako nam varijabla `$statusCode` ne treba nigdje, zbog jednostavnosti i čitljivosti prebacit ćemo sadržaj direktno u `return`.

```
public static function error(\Throwable $e)
{
    return response()->json(
        ['status' => 'ERR', 'message' => $e->getMessage()],
        $e->$e->getStatusCode() ?: 400
    );
}
```

Mogli smo koristiti i imenovane parametre ali onda ih moramo sve imenovati.

```
public static function error(\Throwable $e)
{
    return response()->json(
        data: ['status' => 'ERR', 'message' => $e->getMessage()],
        status: $e->$e->getStatusCode() ?: 400
    );
}
```

```

    services > ResponseService.php > Illuminate\Contracts\Routing\ResponseFactory::json
    Create a new JSON response instance.

    <?php
    public function json(
        $data = [],
        $status = 200,
        array $headers = [],
        $options = 0
    ) { }

    @param mixed $data
    @param int $status

```

The code editor shows the ResponseService.php file. A tooltip for the `Illuminate\Contracts\Routing\ResponseFactory::json` method is displayed, explaining it creates a new JSON response instance. The method signature is shown with parameters `$data`, `$status`, `$headers`, and `$options`.

Time smo riješili problem servisa.

Sada se u kontroleru `app/Http/Controllers/Controller.php` možemo pozvati na te servise sa `ResponseService`. To smo već napravili. Sada svi kontroleri nasljeđuju apstraktni kontroler i imaju pristup njegovoj metodi `executeServiceAction`.

U daljem procesu trebamo napraviti kontroler, servis. U tom servisu zapisali bi neku poslovnu logiku a u kontroleru bi se pozvali na taj servis tako da zavrap-amo taj servis u funkciju i tu funkciju pošaljemo metodi `executeServiceAction`. Ta metoda `executeServiceAction` će pokušati izvršiti taj servis pozivajući funkciju koju sm zavrap-ali naš servis. Ako nešto pukne u `try` bloku, dolazi do greške i odlazimo `catch` blok i imamo unificirani odgovor, javlja se neki error.

Autorizacija korisnika

Prvo ćemo posložiti kontroler pa servis.

U `Http\Controllers` otvorit ćemo novu datoteku `AuthController.php`.

```

<?php

namespace App\Http\Controllers;

use Illuminate\Http\Resources\Json\JsonResource;

class AuthController extends Controller
{
    public function login(): JsonResource

```

```
{
    return $this->executeServiceAction();
}
```

Odgovor će uvijek biti JSON tipa i zato ćemo u odgovoru definirati `JsonResponse` zato što `AuthController` nasljeđuje `Controller` i poziva `executeServiceAction()`. Tu se pozove iz `app/Services/ResponseAService.php` `success()` i `error()`. `success()` i `error()` vraćaju `json` odgovor. Naše metode vraćaju `JsonResponse` i to ćemo definirati:

```
<?php

namespace App\Services;

class ResponseService
{
    public static function success($data, $status = 200): JsonResponse
    {
        return response()->json(['status' => 'OK', 'data' => $data], $status);
    }

    public static function error(\Throwable $e): JsonResponse
    {
        return response()->json(
            data: ['status' => 'ERR', 'message' => $e->getMessage()],
            status: $e->$e->getStatusCode() ?: 400
        );
    }
}
```

Time dajemo legitimitet našim metodama i uvijek znamo što se dešava. Isto tako možemo reći što sve prima kao `$data` tj. što sve može primiti u `success()` metodi.

Ako pogledamo nekakav Laravel collection, ono što je nama bitno je da se podatak koji dolazi u `$data` u `app/Services/ResponseAService.php` da se može serijalizirati.

U `Http\Controllers\AuthController.php` pozvali smo metodu `login()` za koju smo sigurni da će vratiti `JsonResponse`. Sigurni smo zato što kada kažemo `return $this->executeServiceAction();`, kontroler će pozvati `ResponseService::success()` ili ako je došlo do iznimke `ResponseService::error()`. Vidjeli smo da u servisu `app/Services/ResponseService.php` isto tako na `return` tipu je `json()`.

Ono što je nama bitno je `$data`, koji ćemo poslati kao rezultat izvršavanja servisa.

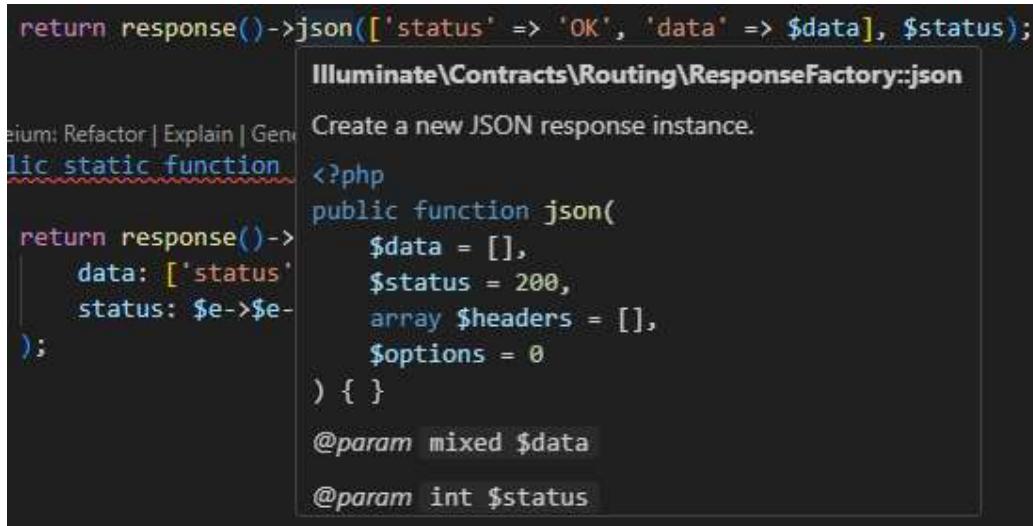
```
public function login(): JsonResponse
{
```

```

    $serviceAction = fn() => 'Login';
    return $this->executeServiceAction();
}

```

To znači da nam u kontroleru treba neka funkcija u kojoj ćemo reći `'Login'`. Servis još nemamo. Vraćamo običan string. Lako je pogriješiti ako pošaljemo neke vrijednosti a ne znamo kako će se ponašati ako uđu u `json()`. `json()` očekuje da ovaj `$data` bude matrica. Što ako koristimo kolekciju ili nešto drugo?



Sada ćemo poslati string. `fn()` je callable funkcija kojoj smo dodijelili varijablu i po tome je proslijedili metodi kao argument, gdje će se izvršiti. Direktno smo je poslali s `return $this->executeServiceAction(fn() => 'Login');`

Šaljemo funkciju koja kada se izvrši vratit će `'Login'`. Ako pogledamo u kontroleru `app/Http/Controllers/Controller.php`, u `executeServiceAction`, varijabla mora sadržavati u sebi vrijednost koju možemo pozvati, koja je callable. To je funkcija koju pozivamo, a to je `$serviceAction()`. Kada pozovemo taj `$serviceAction()`, tada `fn()` u principu, pošto je proslijeden se aktivira i zato se to naziva **callback**, kada se funkcija pokrene, tada potencijalno unutar nje naknadno pozovemo ovu funkciju callback.

Da bi smo testirali u `routes/api.php` promijenit ćemo rutu:

```
Route::get('/', AuthController::class.'@login');
```

`AuthController::class` vraća puni naziv klase npr. `"App\Http\Controllers\AuthController"`. `::class` operator daje fully qualified name (puni naziv) klase, što omogućava da Laravel zna koji kontroler treba pozvati. String `'@login'` predstavlja ime metode koja će biti pozvana unutar `AuthController` klase. Konkatenacija je napravljena s točkom `(.)` i puni naziv je `"App\Http\Controllers\AuthController@login"`.

Ako sada napravimo refresh stranice localhost:800/api/v1/ trebali bi vidjeti tekst Login iz stringa.

```
{"status": "OK", "data": "Login"}
```

Idemo probati serijalizacijsku Laravelovu kolekciju. U kontroleru [app\Http\Controllers\AuthController.php](#) u [login\(\)](#) metodu upisat ćemo:

```
public function login(): JsonResponse
{
    $serviceAction = fn() => collect([1, 2, 3]);
    return $this->executeServiceAction($serviceAction);
}
```

```
{"status": "OK", "data": [1, 2, 3]}
```

Dakle, kontroleri implementiraju [executeServiceAction\(\)](#). Svaka metoda u kontroleru će reći definiraj [\\$serviceAction](#) kao anonimnu funkciju (mogli smo i direktno poslati servis). Funkciju smo zapisali u varijablu i proslijedili varijablu [executeServiceAction](#) u baznom kontroleru Controller.php. Pošto je callable zadovoljen pozivamo funkciju [\\$serviceAction](#). Izvršava se ono što je u tijelu funkcije [collect\(\)](#).

Sada ova funkcija vraća nekakvu kolekciju i kao što vidimo radi. Matricu je pretvorio u kolekciju. Ako pogledamo [dd\(collect\(\[1, 2, 3\]\)\);](#):

```
public function login(): JsonResponse
{
    dd(collect([1, 2, 3]));
    $serviceAction = fn() => collect([1, 2, 3]);
    return $this->executeServiceAction($serviceAction);
}
```

```
Illuminate\Support\Collection {#215 ▼ // app\Http\Controllers\AuthController.php:11
  #items: array:3 [▶]
  #escapeWhenCastingToString: false
}
```

U [items](#) je matrica. JSON je primio objekt, a drugi se zove [escapeWhenCastingToString](#). Kako je JSON znao da treba izvući items je odlično pitanje, skriveno u definiciji kolekcije. Ako sada probamo pokrenuti <http://localhost:8000/api/v1/> sa definiranim stringom u [login\(\)](#):

```
public function login(): JsonResponse
{
    $serviceAction = fn() => 'Hello, world!';
    return $this->executeServiceAction($serviceAction);
}
```



This page isn't working

localhost is currently unable to handle this request.

HTTP ERROR 500

[Reload](#)

Ako pogledamo `storage/logs/laravel.log`, vidjet ćemo da se dogodio error jer string nema u potpisu interface a error nema na sebi `getStatusCode()`, jer ga nismo još napravili. Zaključak je da string nema implementiran interface za serijalizaciju. Želimo biti sigurni da dobivamo podatke koji nam trebaju a ne bilo koje podatke. Ako probamo običnu matricu, vidimo ponovo istu stvar.

```
$serviceAction = fn() => ['token' => '1234567890'];
```

Ako stavimo string, onda aplikacija ne radi:

```
$serviceAction = fn() => ['token' => '1234567890'];
```

Rješenje je da u `app/Services/ResponseService.php` `getStatusCode()` promijenimo u `getCode()`:

```
public static function error(\Throwable $e): JsonResponse
{
    return response()->json(
        data: ['status' => 'ERR', 'message' => $e->getMessage()],
        status: $e->getCode() ?: 400
    );
}
```

Na nivou interface-a imamo implementirano metodu `getCode()`. To govori da svaka klasa koja je implementirala interface, ima tu metodu. Vidimo da je `getStatusCode()` samo nad `Exception` klasom dok je nad `Error` nema.

```
{
    "status": "ERR",
    "message": "App\\Services\\ResponseService::success(): Argument #1
($data) must be of type Illuminate\\Http\\JsonResponse,
Illuminate\\Support\\Collection given, called in I:\\Laravel\\algebra-
blog-api\\app\\Http\\Controllers\\Controller.php on line 13"
}
```

Dakle po Laravelu se matrica ne može serijalizirati. Svaki puta ono što želimo je kolekcija koja će ući u success, tako da svaki puta naš servis kada treba vratiti podatak ne smije biti ništa drugo nego kolekcija.

Ako promijenimo `['token' => '1234567890']` u `collect(['token' => '1234567890'])`

Evo konačne verzije `app/Services/ResponseService.php`:

```
<?php

namespace App\Services;

use Illuminate\Http\JsonResponse;
use JsonSerializable;

class ResponseService
{
    public static function success(JsonSerializable $data, int $status = 200): JsonResponse
    {
        return response()->json(['status' => 'OK', 'data' => $data], $status);
    }

    public static function error(\Throwable $e): JsonResponse
    {
        return response()->json(
            [
                'status' => 'ERR',
                'message' => $e->getMessage(),
                'status' => $e->getCode() ?: 400
            ],
            400
        );
    }
}
```

Sada sve radi:

```
{"status":"OK","data":{"token":"1234567890"}}
```

To znači da bi trebali kada netko napadne endpoint `'/'` u `routes/api.php` zahtjev se preusmjerava na `AuthController`, na login metodu, mi kao odgovor dobijamo json u kojem piše status OK i sadržaj tokena „1234567890“.

Riješili smo potencijalne probleme koji nastaju s nekompatibilnošću pojedinih tipova. Ako želimo omogućiti i matricu, onda ćemo napisati u servisu `app/Services/ResponseService.php` u metodi `success()`:

```
public static function success(array|JsonSerializable $data, int $status = 200): JsonResponse
```

Matricu je po default-u moguće serijalizirati.

Nad bootstrap-om u app.php imamo kontrolu nad svim exception-ima i error-ima koji su renderable i preusmjeravamo na `ResponceService::error` i šaljemo. `ResponceService::error` kaže u redu, `data` je `status` `ERR`, `message` je message, `status` je iz exception-a, izvuci kod, ako ga nema (nula je), neka je 400.

Nad kontrolerom želimo imati jedinstvenu kontrolu, kako bi objedinili odgovore prema klijentu kroz `ResponseService` (ili je `success` ili je `error`). To ćemo provući kroz `try-catch` blok. Ako u `try` nešto pukne, u `catch` se uhvati i proslijedi se u `error` u servisu koji odgovori s JSON. Da `success()` prođe, on mora proslijediti funkciju `$serviceAction()`. Kada pozove tu funkciju, poanta je da ta funkcija bude u nekom servisu koji ćemo napisati i koji će odraditi poslovnu logiku. Ono što smo rekli, taj servis mora obavezno vratiti ili matricu ili objekt koji ima implementirano `JsonSerializable` interface, ništa drugo ne može uči. Više otvorenih opcija ne treba. Ako pozovemo metodu `login()`, definiramo koji `$serviceAction`. To je neka funkcija koja će ispod haube pozvati neki servis. Taj servis koji se izvrši, vratit će kolekciju. Ta kolekcija kada uđe na `executeServiceAction()`, će se proslijediti `success()`, a `success()` kada primi te podatke, u odgovoru će ih strukturirati i uvijek će ih strukturirati.

To je neka razlika naspram monolitne aplikacije, nema pogleda. Poslovna logika je manje-više ista. Ostaje još da implementiramo JWT – servis za identifikaciju.

Kada pokrenemo `localhost:8000` dobijemo sljedeću poruku:

```
{"status": "ERR", "message": "The route \/ could not be found."}
```

To je zato jer ta ruta ne postoji. Dogodila se `NotFoundHttpException` iznimka. Dogodila se Laravelova iznimka koja je trebala iscrtati 404. Mi smo to presreli i zakomentirali. Ako napišemo u `bootstrap/app.php`:

```
->withExceptions(function (Exceptions $exceptions) {
    $exceptions->renderable(function (Throwable $e) {
        // return ResponseService::error($e);
    });
})->create();
```

Dobijemo ovakav rezultat:

404 | NOT FOUND

Ako bi u kontroleru `app/Http/Controllers/AuthController.php` izbacili neki exception:

```
class AuthController extends Controller
{
    public function login(): JsonResponse
    {
        throw new \Exception('Not implemented');
        $serviceAction = fn() => collect(['token' => '1234567890']);
        return $this->executeServiceAction($serviceAction);
    }
}
```

Dobili bi smo ovakvu poruku:

```
{"status": "ERR", "message": "Not implemented"}
```

Dakle uhvatili smo exception. Nikada nikada nije ušao u `executeServiceAction($serviceAction)`, što znači da nikada nije došao do `try-catch` bloka. Dakle nije u kontroleru. U `bootstrap/app.php` se uključio renderable i on je ovo ispisao. Sve ono što mi ne uhvatimo završit će u tom `app/app.php`:

```
->withExceptions(function (Exceptions $exceptions) {
    $exceptions->renderable(function (Throwable $e) {
        return ResponseService::error($e);
    });
});
```

Možemo pomisliti da nam ne treba naš `try-catch` blok jer će oni svejedno završiti ovdje. Razlika je u tome što Laravel neće neke stvari logirati a mi to možda želimo. Možemo pogledati što ako u `app/Http/Controllers/AuthController.php` prebacimo throw tako da sada metoda izgleda ovako:

```
class AuthController extends Controller
{
    public function login(): JsonResponse
    {
        $serviceAction = fn() => throw new \Exception('Not implemented');
        return $this->executeServiceAction($serviceAction);
    }
}
```

Sada ova iznimka postaje dio ove arrow funkcije, što znači da se ova iznimka neće pokrenuti dok ne pokrenemo funkciju. U ovu funkciju neće ući dok ne uđemo u `executeServiceAction`, tada će se pokrenuti ta funkcija. Idemo pokrenuti <http://localhost:8000/api/v1/>:

```
{"status":"ERR","message":"App\\Http\\Controllers\\AuthController::login()  
: Return value must be of type Illuminate\\Http\\JsonResponse, null  
returned"}
```

Ako kontroler `app\Http\Controllers\Controller.php` prepravimo u:

```
abstract class Controller  
{  
    protected function executeServiceAction(callable $serviceAction)  
    {  
        try {  
            return ResponseService::success($serviceAction());  
        } catch (\Throwable $e) {  
            dump('error');  
            return ResponseService::error($e);  
        }  
    }  
  
}  
  
"error" // app\Http\Controllers\Controller.php:15  
  
{"status":"ERR","message":"Not implemented"}
```

Ako `dump` i `return` stavimo pod komentare, dobit ćemo prazan ekran. To znači da je `executeServiceAction` u `app\Http\Controllers\Controller.php` vratio null.

Ako u `app\Http\Controllers\AuthController.php` vratimo red:

```
class AuthController extends Controller  
{  
    public function login()  
    {  
        throw new \Exception('Not implemented');  
        $serviceAction = fn() => throw new \Exception('Not implemented');  
        return $this->executeServiceAction($serviceAction);  
    }  
}
```



```
{"status":"ERR","message":"Not implemented"}
```

Primijetimo da smo na metodi `login` uklonili `return` tip. Može doći bilo što. Ne radi i može doći bilo što. Imamo bug.

Ako pogledamo kontroler, očekujemo da po MVC zaprima zahtjev i vrati odgovor. Ako nemamo odgovor i generiramo exception. Exception u kontroleru predstavlja nešto što se treba izrednerirati i Laravel upravo to gleda. Ako nemamo u [bootstrap/app.php](#) napisano, tj. pod komentarima je:

```
//return ResponseService::error($e);
```

Uključuje se njegov debugger view:

The screenshot shows a browser window with a debugger interface. At the top, there's a red button labeled "Exception". Below it, the text "Not implemented" is displayed. To the right, it says "GET localhost:8000" and "PHP 8.2.12 — Laravel 11.28.1". The main area shows a stack trace and the source code for `I:\Laravel\algebra-blog-api\app\Http\Controllers\AuthController.php`:

```
I:\Laravel\algebra-blog-api\app\Http\Controllers\AuthController.php :11
6
7 class AuthController extends Controller
8 {
9     public function login()
10    {
11        throw new \Exception('Not implemented');
12        $serviceAction = fn() => throw new \Exception('Not implemented');
13        return $this->executeServiceAction($serviceAction);
14    }
15 }
```

Laravel ispisuje izrendanu grešku u kontroleru jer on mora nešto vratiti a to je pogleda. Mi u Laravelu to presrećemo tako da nema pogleda.

Ako se dogodi not found greška, to neće biti zapisano u log.

Da bi smo upisali u log možemo u [bootstrap/app.php](#) napisati:

```
)>withExceptions(function (Exceptions $exceptions) {
    $exceptions->renderable(function (Throwable $e) {
        Log::critical($e->getMessage());
        return ResponseService::error($e);
    });
});
```

Ovime smo sve osim exception, gurnuli u critical. Ako ukucamo krivi link (rutu koja ne postoji), u log se zapiše critical error. Problem je što not found nije critical error. Moguće je da tako imamo previše zapisanih grešaka. Zato to ne želimo tu raditi i zato smo to prenijeli na [executeServiceAction](#) u [app/Http/Controller.php](#):

```
abstract class Controller
{
    protected function executeServiceAction(callable $serviceAction)
    {
```

```
try {
    return ResponseService::success($serviceAction);
} catch (\Throwable $e) {
    Log::critical($e->getMessage());
    return ResponseService::error($e);
}

}
```

Sada imamo razliku. Sada na 404 Not found vidimo grešku ali nema log-a. Za critical grešku se log zapisuje.

Migracije i modeli

Podešavanja

Migracije i modele ćemo prebaciti sa monolita.

Dakle migracije iz `algebra-blog-api\database\migrations` ćemo obrisati a tamo prebaciti iz `\laravel-algebra-blog\database\migrations`. Prebacit ćemo i modele iz `\laravel-algebra-blog\app\Models` u `algebra-blog-api\app\Models`.

Nećemo koristiti sesion, već ćemo koristiti JWT paket koji ćemo instalirati. On nadavlada (engl. override) `Authenticatable` odakle dolazi klasa User u `User.php`. I dalje ćemo koristiti ugrađen Laravelov servis za autentifikaciju.

Idemo instalirati Laravel JWT. To ćemo napraviti s

```
composer require tymon/jwt-auth
```

Kod generiranja secret key, mi ćemo koristiti simetrični ključ.

```
$ composer require tymon/jwt-auth
./composer.json has been updated
Running composer update tymon/jwt-auth
Loading composer repositories with package information
Updating dependencies
Lock file operations: 4 installs, 0 updates, 0 removals
- Locking lcobucci/clock (2.3.0)
- Locking lcobucci/jwt (4.0.4)
- Locking stella-maris/clock (0.1.7)
- Locking tymon/jwt-auth (2.1.1)
Writing lock file
Installing dependencies from lock file (including require-dev)
Package operations: 4 installs, 0 updates, 0 removals
- Downloading stella-maris/clock (0.1.7)
- Downloading lcobucci/clock (2.3.0)
- Downloading lcobucci/jwt (4.0.4)
- Downloading tymon/jwt-auth (2.1.1)
- Installing stella-maris/clock (0.1.7): Extracting archive
- Installing lcobucci/clock (2.3.0): Extracting archive
- Installing lcobucci/jwt (4.0.4): Extracting archive
- Installing tymon/jwt-auth (2.1.1): Extracting archive
Generating optimized autoload files
> Illuminate\Foundation\ComposerScripts::postAutoloadDump
> @php artisan package:discover --ansi

  INFO  Discovering packages.

  laravel/pail          DONE
  laravel/sail           DONE
  laravel/tinker         DONE
  nesbot/carbon          DONE
  nunomaduro/collision  DONE
  nunomaduro/termwind    DONE
  tymon/jwt-auth         DONE

81 packages you are using are looking for funding.
Use the `composer fund` command to find out more!
> @php artisan vendor:publish --tag=laravel-assets --ansi --force

  INFO  No publishable resources for tag [laravel-assets].
```

No security vulnerability advisories found.

Using version ^2.1 for tymon/jwt-auth

```
Administrator@DESKTOP-4J5DF41 MINGW64 /i/Laravel/algebra-blog-api
$ |
```

Na modelu `app/Models/User.php` izmjenit ćemo iz `class User extends Authenticatable` u:

```
class User extends Authenticatable implements JWTSubject
```

Taj `JWTSubject` interface će na tjerati da ubacimo dvije metode:

```
'App\Models\User' does not implement methods 'getJWTIdentifier',  
'getJWTCustomClaims' intelephense(P1037)
```

Codeium: Explain Problem

Tymon\JWTAuth\Contracts\JWTSubject

```
<?php  
interface JWTSubject { }  
  
View Problem (Alt+F8) No quick fixes available  
class User extends Authenticatable implements JWTSubject
```

Iz dokumentacije čemo na dno [app/Models/User.php](#) sa copy-paste prenijeti ovaj kod:

```
// Rest omitted for brevity  
  
/**  
 * Get the identifier that will be stored in the subject claim of the JWT.  
 *  
 * @return mixed  
 */  
public function getJWTIdentifier()  
{  
    return $this->getKey();  
}  
  
/**  
 * Return a key value array, containing any custom claims to be added to the  
JWT.  
 *  
 * @return array  
 */  
public function getJWTCustomClaims()  
{  
    return [];  
}
```

Sljedeći korak je publish-ati konfiguraciju. To smo mogli napraviti i kod instalacije.

```
php artisan vendor:publish --  
provider="Tymon\JWTAuth\Providers\LaravelServiceProvider"
```

```
> @php artisan vendor:publish --tag=laravel-assets --ansi --force
[INFO] No publishable resources for tag [laravel-assets].
No security vulnerability advisories found.
Using version ^2.1 for tymon/jwt-auth

Administrator@DESKTOP-4J5DF41 MINGW64 /i/Laravel/algebra-blog-api
$ php artisan vendor:publish --provider="Tymon\JWTAuth\Providers\LaravelServiceProvider"
[INFO] Publishing assets.

Copying file [I:\Laravel\algebra-blog-api\vendor\tymon\jwt-auth\config\config.php] to [I:\Laravel\algebra-blog-api\config\jwt.php] [DONE]
```

Od paketa konfiguracija je dostupna u Laravelu u config direktoriju i ako napravimo upgrade paketa, neće se promijeniti konfiguracija. U direktoriju config trebamo vidjeti `jwt.php` datoteku.

Konfiguracija se oslanja na envirement. `ttl` (default je 60) je vrijeme koliko je token valjan.

```
'ttl' => env('JWT_TTL', 60),
```

Nakon toga se s tim tokenom korisnik ne može autorizirati. Ako pogledamo kako radi session, kada korisnik napravi request, mi ga autoriziramo i onda regeneriramo session token (tj. damo mu novi token). Ponovo, kada korisnik dođe ima valjan token. Tako to radi u PHP-u. Pogledajmo sada JWT tokene. Oni imaju problem. Mi izdamo JWT token kojeg koristi web aplikacija. Prije nego što pošalje zahtjev korisnik se mora predstaviti tko je, ako to očekujemo. Imamo štićenje endpoint-e i ako korisnik želi da dobije podatke, mora se predstaviti. On zaglavlju zahtjeva pošalje taj token. Kada ga dekriptira vidimo da li ima pravo ili ne. Token se na klijentsoj strani pohranjuje najčešće u nesigurnim dijelovima: local storage, local session, cookie, koji se mogu uhvatiti i pročitati i doći do tih vrijednosti. Ovo je slično **session hijacking (otmica sesije)**, o kojoj smo već govorili.

Kada token istekne ne možemo ga samo tako osvježiti kada ga pošalje. Sistem je osmišljen tako da ima možemo odgovoriti s dva tokena: jedan token je onaj s kojim se autentificira a drugi token je refresh token. U `jwt.php` ma definiran `refresh_ttl` token, koliko dugo vrijedi taj refresh token. Session, za usporedbu, aktivan je dok je korisnik u aplikaciji i nešto radi. Čim korisnik prestane nešto raditi, token istekne. Kod JWT tokena, kada vrijeme istekne, bez obzira na aktivnost korisnika, token je istekao. Kod JWT, kako ne bi korisnika maltretirali da se ponovo prijavljuje tokom rada a originalni token je istekao, šalje se refresh token koji traži obnavljanje novog tokena. Naravno refresh token mora vrijediti i token koji je poslan mora biti nevažeći. Tada se korisniku pošalje novi token. Vremenski okvir u kojem se radi ovisi o aplikacije do aplikacije i dogovoru s frontend developerom. Token obično vrijedi sat vremena.

Problem ovoga je što u pozadini koristi karbon da bi izračunao da li `ttl` vrijeme vrijedi. Dakle uzme današnje trenutno vrijeme i nadoda 60 minuta (ako je tako definirano). Karbon radi s integer-om i `ttl` je zadan kao integer. I to je u redu. Međutim ako odlučimo vrijednost zadati preko envirement varijable `JWT_TTL`, nastaju problemi. Trebamo otici u `.env` datoteku i generirati JWT secret ako ga nemamo.

Ako zaboravimo kako to napraviti dovoljno je pokrenuti `php artisan` i vidjet ćemo help. Pod `jwt` piše `jwt:secret`. On će generirati ključ koji se koristi za kriptiranje podataka.

```
$ php artisan jwt:secret
jwt-auth secret [e3cszeQBWxF6sdaP9BuZAXnEpxDB1jLUptyyNHouMKTgfuMktAL5ywRhKjgcDVi
G] set successfully.

Administrator@DESKTOP-4J5DF41 MINGW64 /i/Laravel/algebra-blog-api
$ |
```

Postavio ga je u envirement tj. `.env` datoteku. Kao `JWT_SECRET`. Nakon toga definiramo `JWT_TTL` na vrijednost koju želimo. Problem je što su sve vrijednosti u ovoj datoteci stringovi. To uđe u 'ttl' kao string i kada se pošalje karbonu, to pukne. To je taj bug. Treba napraviti cast informacije i ispred staviti string.

```
'ttl' => (int) env('JWT_TTL', 60),
```

Time pretvorimo vrijednost iz stringa u integer. Ista priča je i za:

```
'leeway' => (int) env('JWT_LEEWAY', 0),
```

Predavač kaže da je taj bug prijavljen ali nije riješen u ovom trenutku. Prijava je iz 5. mjeseca a sada je kraj 2024. godine.

Sada ćemo podesiti `config/auth.php AUTH_GUARD` tj. isključiti ga jer ga ne želimo iskoristiti. Cijeli `AUTH_GUARD` je podešen da gleda session. Sada izgleda ovako:

```
'defaults' => [
    'guard' => env('AUTH_GUARD', 'web'),
    'passwords' => env('AUTH_PASSWORD_BROKER', 'users'),
],
```

Koristili smo middleware u monolit aplikaciji za rad s postovima u `routes/web.php`:

```
Route::group(['middleware' => AuthMiddleware::class], function () {
```

Taj middleware je customiziran u `app/Http/Middleware/AuthMiddleware` u metodi `AuthMiddleware`.

On se oslanja na auth.php i kaže ako je `'AUTH_GUARD'` web (a jeste), onda je `'driver'` `'session'`.

```
'providers' => [
    'users' => [
        'driver' => 'eloquent',
        'model' => env('AUTH_MODEL', App\Models\User::class),
    ],
],
```

Kako imamo JWT moramo sve ovo malo podesiti. Predavač preporučuje da što više koristimo enviroment varijable. U `.env` datoteku trebamo dodati:

```
AUTH_GUARD=api
```

Ako pogledamo u `.gitignore`, .env datoteka se ne prenosi na git. Možemo ukloniti red sa `.env`. Inače, ovo ne savjetuje da se napravi zbog username i password-a na produkcijsku bazu.

U `config/auth.php` u `'guards'` nemamo definiramo guard, idemo ga definirati:

```
'guards' => [
    'web' => [
        'driver' => 'session',
        'provider' => 'users',
    ],
    'api' => [
        'driver' => 'jwt',
        'provider' => 'users',
    ]
],
```

Ovo je dovoljno da radi jwt auth guard.

Ako pogledamo `storage/framework/sessions` je trenutno prazan. Tu će biti generirani tokeni. Možemo reći da token koji je već bio da se više ne koristi. Zato postoji `'blacklist_enabled'`.

Migracije

Trebamo pokrenuti migracije. Time ćemo riješiti bazu podataka.

Prije toga trebamo u .env podesiti. Treba promijeniti `DB_CONNECTION` iz `sqlite` u `mysql` i dati ime baze s `DB_DATABASE`.

```
DB_CONNECTION=mysql
DB_HOST=127.0.0.1
DB_PORT=3306
DB_DATABASE=algebra-blog-api
DB_USERNAME=root
DB_PASSWORD=
```

Postavit ćemo još:

```
APP_NAME=algebra-blog-api
SESSION_DRIVER=file
CACHE_STORE=file
```

Ako je sve OK, pokrenut ćemo migracije:

```
$ php artisan migrate

[ WARN] The SQLite database configured for this application does not exist: lar
avel.

Would you like to create it? (yes/no) [yes]
> yes

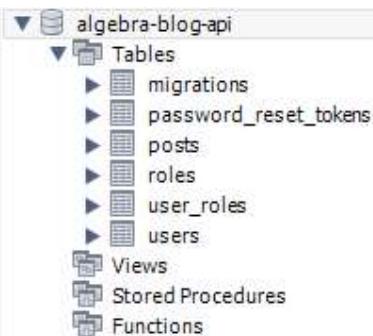
[ INFO] Preparing database.

Creating migration table ..... 185.33ms DONE

[ INFO] Running migrations.

0001_01_01_000000_create_users_table ..... 554.49ms DONE
2024_08_28_180207_create_roles_table ..... 476.74ms DONE
2024_08_28_182300_create_posts_table ..... 331.37ms DONE
2024_09_02_170653_change_posts_table ..... 157.55ms DONE
2024_09_02_171148_change_users_table ..... 148.19ms DONE

Administrator@DESKTOP-4J5DF41 MINGW64 /i/Laravel/algebra-blog-api
$ |
```



Trebat ćemo i role ali ćemo ih kasnije ručno ubaciti kasnije. Sve je sada spremno da možemo raditi s JWT. Prvi korak neće biti login nego registry u [routes/api.php](#):

```
<?php

use App\Http\Controllers\AuthController;
use Illuminate\Container\Attributes\Auth;
use Illuminate\Support\Facades\Route;

Route::post('register', AuthController::class.'@register');
```

Sljedeći korak je kako imamo rutu a trebamo kontroler. Idemo pogledati [app/Http/Controllers/AuthController.php](#) koji već imamo.

Metoda `register()` će primiti podatke iz requesta (šalje ih klijent, mobile ili desktop), očekujemo da pošalje email i password. Nećemo se baviti frontend problemom ponovnog slanja podataka. Dakle mi ćemo validirati email i provjeriti da li je jedinstven. Password ćemo provjeriti ima li 8 slova i mala i velika slova. Request ne bi željeli primiti bez validacije, nego ćemo kao i kod monolita ponovo koristiti request u kojima ćemo definirati validaciju podataka.

```
public function register(): JsonResponse
{
    $serviceAction = fn() => throw new \Exception('Not implemented');
    return $this->executeServiceAction($serviceAction);
}
```

```
php artisan make:request RegisterRequest

$ php artisan make:request RegisterRequest
[INFO] Request [I:\Laravel\algebra-blog-api\app\Http\Requests\RegisterRequest.php] created successfully.

Administrator@DESKTOP-4J5DF41 MINGW64 /i/Laravel/algebra-blog-api
```

Kreirao se [app/Http/Requests/RegisterRequest.php](#).

```
<?php

namespace App\Http\Requests;

use Illuminate\Foundation\Http\FormRequest;

class RegisterRequest extends FormRequest
{
    /**
     * Determine if the user is authorized to make this request.
     */
    public function authorize(): bool
    {
        return false;
    }

    /**
     * Get the validation rules that apply to the request.
     *
     * @return array<string,
\Illuminate\Contracts\Validation\ValidationRule|array<mixed>|string>
     */
    public function rules(): array
    {
        return [
            'name' => 'required|string',
            'email' => 'required|email|unique:users',
            'password' => 'required|confirmed',
        ];
    }
}
```

```
        //
    ];
}
```

`authorize()` smo rekli da nećemo provjeravati pa čemo ga odmah prebaciti na `true`.

```
/**
 * Determine if the user is authorized to make this request.
 */
public function authorize(): bool
{
    return true;
}

/**
 * Get the validation rules that apply to the request.
 *
 * @return array<string,
\Illuminate\Contracts\Validation\ValidationRule|array<mixed>|string>
 */
public function rules(): array
{
    return [
        'first_name' => 'required|string',
        'last_name' => 'required|string',
        'email' => 'required|email|unique:users,email',
        'password' => 'bail|required|string|confirmed|min:8',
    ];
}
```

`bail` znači kad nađe na prvu grešku, ne ide dalje.

Rolu ćemo dodijeliti naknadno ali ćemo ručno ubaciti role u tablicu s rolama.

U `AuthController.php` pomoću dependency injection-a ubacit ćemo u `register()` metodu `ReisterRequest $request`.

```
public function register(ReisterRequest $request): JsonResponse
{
    $serviceAction = fn() => throw new \Exception('Not implemented');
    return $this->executeServiceAction($serviceAction);
}
```

Iza `fn()` nam treba servis kojeg trenutno nemamo. Taj servis će biti neki `AuthServis` koji ćemo opet injectati na samu metodu ili preko konstruktora pa će nam svaki put biti dostupan u metodi (moguća oba rješenja). Predavač preferira rješenje s konstruktorom. Iznad `login()` u klasi `AuthController` stavićemo:

```
{  
    private AuthServiceInterface $authService;  
    public function __construct(AuthServiceInterface $authService)  
    {  
        $this->authService = $authService;  
    }  
}
```

Oslanjamо se na interface i potpis a ne na kompletну implementaciju, samu klasu AuthService već na neki interface. Mogu biti različite klase koje implementiraju `AuthServiceInterface`, možemo ga nazvati samo `AuthInterface`. Trebamo gledati da uvijek radimo s apstrakcijom a ne s konkretnom implementacijom. Ono što se u Laravelu koristi se ne naziva interface. Microsoft koristi u interface prefix I da razlikuje interface od klase. Laravel smjesti ugovore u direktorij `contract`. `concret` su implementacije, konkretne klase iz kojih se stvaraju objekti.

Kada se radi dependency injection u `__construct` Laravel neće znati koja je implementacija ispravna. Malo se gubi smisao, zato što kada pokušamo koristiti taj dio će neće funkcionirati ako ga ne bindamo. Laravel očekuje da mu kažemo tj. da bindamo unutar kontejera, svi servisi imaju contract koji je bindan za servis. Vidjet ćemo kako to funkcionira.

Moramo napisati `authService` interface i `$authService`. Metodu `login()` prebacit ćemo ispod `register()` metode.

```
class AuthController extends Controller  
  
{  
    private AuthServiceInterface $authService;  
    public function __construct(AuthServiceInterface $authService)  
    {  
        $this->authService = $authService;  
    }  
    /**  
     * Register a new user  
     *  
     * @param ReisterRequest $request  
     * @return JsonResponse  
     */  
  
    public function register(RegisterRequest $request): JsonResponse  
    {  
        $serviceAction = fn() => $this->authService->register($request->validated());  
    }  
}
```

```
        return $this->executeServiceAction($serviceAction);
    }

    public function login(): JsonResponse
    {
        $serviceAction = fn() => throw new \Exception('Not implemented');
        return $this->executeServiceAction($serviceAction);
    }
}
```

Očekujemo kada se instancira klasa `AuthController` koja je u `app/Http/Controllers/AuthController.php`. Instancira ga `Route routes/api.php` kada se dogodi request na URI `register`. U tom trenutku se uz pomoć dependency injection ubaci AuthServiceInterface koji nema logiku nego samo potpise. Nama treba nešto što ima implementaciju svih tih metoda. Iz toga izvlačimo `register()`. Interface će reći da AuthServiceInterface mora implementirati `register()` i `login()` metode. Idemo napraviti taj interface:

```
php artisan make:interface AuthServiceInterface
```

```
[INFO] Interface [I:\Laravel\algebra-blog-api\app\AuthServiceInterface.php] created successfully.
```

```
Administrator@DESKTOP-4J5DF41 MINGW64 /i/Laravel/algebra-blog-api
$ |
```

Vidimo da je kreirao `AuthServiceInterface.php` u direktoriju `app`. V idimo da nije ništa specijalno napravio:

```
<?php

namespace App;

interface AuthServiceInterface
{
    //
}
```

Mogli smo otvoriti novi direktorij `app/Interfaces` ili `app/Contracts`, što je možda češće u Laravelu. Ako tu želimo pratiti neku logiku, onda otvorimo `app/Contracts`. Ako će nas to buniti dovoljno je `app/Interfaces`. Preselit ćemo `AuthServiceInterface.php` u taj direktorij. Vrlo je bitno da promijenimo namespace:

```
<?php

namespace App\Interfaces;
```

```
interface AuthServiceInterface
{
    //
}
```

Ono što ćemo unutra imati:

```
<?php

namespace App\Interfaces;

interface AuthServiceInterface
{
    public function register(array $data): Enumerable;
    public function login(array $data): Enumerable;
    public function logout(): void;

}
```

Funkcija register će vratiti kolekciju, zato smo definirali `Enumerable`. Možemo tu napraviti i `refresh()` za osvježiti token. Za sada neka ostane ovako. Sada imamo apstrakciju koju možemo implementirati na bilo koju klasu i klasa će biti dužna implementirati ove metode (potpisali smo ugovor).

Kreirat ćemo servis `app/Services/AuthService.php`:

```
<?php

namespace App\Services;

use Illuminate\Support\Enumerable;

class AuthService implements AuthServiceInterface
{
    public function register(array $data): Enumerable
    {
        return collect([]);
    }

    public function login(array $data): Enumerable
    {
        return collect([]);
    }

    public function logout(): array
```

```
{  
    //  
}  
}
```

Ovo konkretno sada ne radi ništa ali imamo podatke koji će otici na servis. Ako pogledamo u `App\Http\Controllers\AuthController.php` i metodu `register` i `fn()` u njoj, taj `fn()` se trigerira ne odmah nego tek kada ga `executeServiceAction` aktivira. Do tada je ta funkcija samo definicija koja čeka da se funkcija pozove.

```
public function register(RegisterRequest $request): JsonResponse  
{  
    $serviceAction = fn() => $this->authService->register($request->validated());  
    return $this->executeServiceAction($serviceAction);  
}
```

`executeServiceAction` je taj koji trigerira funkciju koja će ispod haube pozvati servis i metodu `register` i proslijediti validirane podatke. Odgovor treba završiti u jsonu, to je prazan podatak jer je kolekcija prazna.

Prije nego što implementiramo servis idemo vidjeti: ruta - kontroler koji injecta servis. Taj servis koristimo na `register()` jer se konstruktor aktivira čim se Router instancira kontroler, `register()` zaprimi `$request` sa podacima koji su validirani. Ako nešto od validacije ne prođe, validator da neku grešku i po defaultu odgovara na neki view. To moramo modificirati da to izgeneriramo da bude JSON poruka i zašto.

Ovo potencijalno možemo testirati s RESTer-om, dodatkom za Chrome.

The screenshot shows the RESTer extension in Google Chrome. The URL is `http://jsonplaceholder.typicode.com/posts/2`. The response is a 200 OK status with the following JSON data:

```

1: {
2:   "userId": 2,
3:   "id": 12,
4:   "title": "In quibusdam tempore odit est dolorem",
5:   "body": "itaque id aut magnam\npraesentium quia et ea odit et ea volutatis et\nsapiente quia nihil amet occaecati quia id
       volutatem\nincident ea est distinctio odio"
6: }

```

Predavač kaže da je Postman praktičniji.

Ako u Postmanu probamo `localhost:8000/api/v1/register` Vidjet ćemo

The screenshot shows a Postman collection. A specific request to `localhost:8000/api/v1/register` resulted in a 400 Bad Request error. The response body is empty.

Laravel pokušava napraviti dependency injection tako da instancira klasu u objekt. Ako pogledamo, konstruktoru u `app/Http/Controllers/AuthController.php` želi instancirati objekt i zapisati ga u `$authService`. PHP ne zna kako da instancira objekt i to dojavi. Mi možemo imati više različitih implementacija i Laravel ne zna koju da instancira. Moramo dati konkretnu informaciju jer druga je stvar da ručno instanciramo `AuthController`.

Zamislimo da u `routes/api` dodamo redove:

```
Route::get('/', function () {
    $a = new AuthController();
});
```

On ovdje očekuje argument i možemo mu poslati konkretnu implementaciju:

```
Route::get('/', function () {
    $s = new AuthService();
```

```
$a = new AuthController($s);
});
```

`AuthService()` možemo instancirati a interface ne možemo. Ako to pošaljemo kontroleru, on će se uspješno instancirati. Dobit će konkretni objekt. On ne mora raditi dependency injection jer smo ovo napravili ručno. Objekt u potpisu ima ovaj interface (U `app/Services/AuthService.php` piše `class AuthService implements AuthServiceInterface`). Znači da će ga kontroler pustiti. Upravo smo to i napravili. Ako sada napišemo:

```
Route::get('/', function () {
    $s = new AuthService();
    $a = new AuthController($s);
    dd($a);
});
```

```
App\Http\Controllers\AuthController {#241 ▼ // routes\api.php:13
    -authService: App\Serv...\\AuthService {#234}
}
```

Dobili smo instancirani `AuthController` i instancirani servis `AuthService`. Ovdje to radimo ručno. U ruti:

```
Route::post('register', AuthController::class.'@register');
```

`Route::` ga instancira. Route nazna opciju da prepozna koju klasu treba instancirati jer smo u kontroleru rekli `AuthServiceInterface`. Jedno rješenje je u `__construct` staviti umjesto `AuthServiceInterface $authService` staviti `AuthService $authService`. Ako pogledamo da sada radi:

```
App\Http\Controllers\AuthController {#241 ▼ // routes\api.php:13
    -authService: App\Serv...\\AuthService {#234}
}
```

The screenshot shows a Postman test results page. At the top, there are tabs for Body, Cookies, Headers (7), and Test Results. The Headers tab has a value of '(7)'. To the right, it says '400 Bad Request' with a globe icon. Below the tabs, there are buttons for Pretty, Raw, Preview, Visualize, and JSON (which is selected). The JSON response is displayed as follows:

```
1
2     "status": "ERR",
3     "message": "The first name field is required. (and 3 more errors)"
4
```

Renderable je uhvatio exception od validacije i vidimo da je vani validation. Iako imamo greške, radi.

Predavač kaže da ne voli ovo rješenje zato da ako na više različitim mjestima koristimo neki servis i oslanjamo se na implementaciju (klasu) a sutra želimo promijeniti servis, moramo to ručno. Ako se oslonimo na apstrakciju i svuda koristimo `AuthServiceInterface` onda koristimo binding i gdje god pokušava instancirati interface to nije u stanju napraviti ali mu damo klasu koju treba instancirati.

Laravel po default-u daje jedan provider u [app/Provider/AppServiceProvider.php](#). Unutra su dvije metode

```
<?php

namespace App\Providers;

use Illuminate\Support\ServiceProvider;

class AppServiceProvider extends ServiceProvider
{
    /**
     * Register any application services.
     */
    public function register(): void
    {
        //
    }

    /**
     * Bootstrap any application services.
     */
    public function boot(): void
    {
        //
    }
}
```

Poanta je baza Larabela i njegov service kontejner u kojem su svi servisi. Laravel se oslanja na interfejs. On to naziva contract. Negdje je povezao contract (engl. interface) sa contret-om (engl. implementation). To su ključni koncepti koji se koriste za upravljanje ovisnostima i implementaciju dizajna modela dependency injection. Contract je interface koji definira skup metoda ili „ugovor“ koji neka klasa mora implementirati. Ugovori su apstrakcije, one ne sadrže implementaciju već samo definiraju API. Concrete je konkretna implementacija inteface-a. Concrete klasa sadrži stvarnu implementaciju metoda definiranih u contract-u. Laravel koristi dependency injection, što znači da se ovisnosti ubacuju u konstruktor klase umjesto da se instanciraju direktno.

U metodi `register()` navest ćemo koji je to contract (interface [AuthServiceInterface](#)) koji ćemo vezati s concrete (klasa [AuthService](#)).

```
public function register(): void
{
    $this->app->bind(AuthServiceInterface::class, AuthService::class);
}
```

Kada imamo neki servis koji koristimo na različitim stranama, i trebamo ubaciti novi servis trebamo tražiti gdje je to sve implementiramo. Dovoljno je dodati novi interface i sve radi. Predavač preporuča ovaj način rada.

Ako pokušamo ručno instancirati interface s ključnom riječi `new`, tj. iz interface-a stvoriti objekt:

```
Route::get('/', function () {
    $s = new AuthService();
    $a = new AuthController($s);
    dd(new AuthServiceInterface());
});
```

Dobili smo sličnu poruku prije nego što smo bindali interface:

```
{"status": "ERR", "message": "Cannot instantiate interface App\\Interfaces\\AuthServiceInterface"}
```

Nismo se oslonili na concrete klasu nego na interface a to ne zna napraviti.

JWT

Kako smo copy-paste `app/Models/Post.php`, nema `use Cviebrock\EloquentSluggable\Sluggable;` pa to trebamo riješiti. Ne znamo od kojeg je to paketa. Možemo ga ili izgoogliti. Možemo pogledati i u `composer.json` od monolit projekta:

```
"cviebrock/eloquent-sluggable": "^11.0",
```

Ovu liniju možemo kopirati i staviti na isto mjesto u našem `composer.json`. treba napraviti `composer install`. Treba obrisati `composer.lock`.

```
$ composer install
Installing dependencies from lock file (including require-dev)
Verifying lock file contents can be installed on current platform.
Nothing to install, update or remove
Generating optimized autoload files
> Illuminate\Foundation\ComposerScripts::postAutoloadDump
> @php artisan package:discover --ansi

  INFO  Discovering packages.

  cviebrock/eloquent-sluggable ..... DONE
  laravel/sail ..... DONE
  laravel/tinker ..... DONE
  nesbot/carbon ..... DONE
  nunomaduro/collision ..... DONE
  nunomaduro/termwind ..... DONE

80 packages you are using are looking for funding.
Use the 'composer fund' command to find out more!

Administrator@DESKTOP-4J5DF41 MINGW64 /i/Laravel/laravel-algebra-blog (master)
$ |
```

Mogli smo napraviti i composer require i ime paketa:

```
composer require cviebrock/eloquent-sluggable
```

Idemo vidjeti što smo napravili tj. dosadašnji flow. Ruta u `routes/api.php` 'register' ide na `AuthController` i gađa metodu `register`. Metoda `register()` u `app/Http/Controllers/AuthController.php` uz pomoć dependency injection imamo `RegisterRequest $request` koji će odraditi validaciju. Ako pogledamo `RegisterRequest`, unutra ćemo vidjeti rules za naša polja tj. attribute koji pristižu u requestu. Idemo testirati prije nego što pustimo na servis. Vidjet ćemo kako se ponaša taj request uz pomoć Postman-a.

The screenshot shows the Postman application interface. At the top, it displays the URL `http://localhost:8000/api/v1/register`. Below the URL, there's a dropdown menu set to `POST` and a large blue button labeled `Send`. Under the `Params` tab, there are sections for `Query Params` and `Body`. In the `Body` section, the `JSON` tab is selected. The response area shows a `400 Bad Request` status with a message:

```
1 "status": "ERR",
2 "message": "The first name field is required. (and 3 more errors)"
```

.

Podatke možemo slati u ovisnosti o dogovoru s frontendom. Možemo ih slati kao form-data. To su parovi ključ-value. To je ona forma koju smo slali u monolitu. Možemo ih slati kao x-www-form-urlencoded. To su podaci koji se šalju kroz URL.

This screenshot shows the `Body` tab selected in the Postman interface. Below the tab, there are several options: `none`, `form-data` (which is currently selected), `x-www-form-urlencoded`, `raw`, and `binary`.

Moguće je slati raw data, to je najčešći oblik slanja kada frontend ili klijent šalje podatke.

This screenshot shows the `raw` tab selected in the Postman interface. A dropdown menu is open, listing several options: `Text`, `JavaScript`, `JSON`, `HTML`, and `XML`.

Možemo još slati binarne podatke i GraphQL.

Mi ćemo slati raw JSON. Podaci moraju biti ograđeni vitičastim zagradama `{}`. Ključ mora biti string. U JSON ključ i value mora biti u dvostrukim navodnicima i između treba biti dvotočka.

The screenshot shows the 'Body' tab in Postman with the 'JSON' option selected. The request body contains the following JSON data:

```

1  {
2      "first_name": "Pero",
3      "last_name": "Perić",
4      "email": "test",
5      "password": 123
6  }

```

Ovime simuliramo podatke prema API. Ne zanima nas tko je klijent: mobile, desktop, web aplikacija. Podaci će ući u API i doći će kroz register request. Naravno dobit ćemo grešku:

The screenshot shows a failed API response with the following JSON data:

```

{
    "status": "ERR",
    "message": "The email field must be a valid email address. (and 2 more errors)"
}

```

The screenshot shows a successful API response after fixing the email field. The request method is POST and the URL is `http://localhost:8000/api/v1/register`. The 'Body' tab shows the JSON data:

```

{
    "first_name": "Pero",
    "last_name": "Perić",
    "email": "pero@test.com",
    "password": "12345678",
    "password_confirmation": "12345678"
}

```

The response body is:

```

{
    "status": "OK",
    "data": []
}

```

Ovo je sada u redu. U data nema ništa jer je ušao u kontroler `app\Http\Controllers\AuthController.php` tj. u anonimnu funkciju ili nekav callback gdje pozivamo servis `authService` i `register()`. Kroz argument predamo sve validirane informacije iz requesta. Tu anonimnu funkciju, njen potpis, servis se trenutno ne izvršava nego to šaljemo `executeServiceAction()` koja će reći da mora biti callable vrijednost, mora biti function ili method.

U try-catch bloku, kažemo pokušaj pokrenuti value koji se nalazi u parametru. Kako smo ga definirali kao callable znamo da ga možemo pozvati i pozovemo `.getServiceAction()`. Vrijednost toga kada se izvrši šaljemo na `ResponseService::success()`. Time modeliramo odgovor. Cilj je sa `success()` da imamo unificirani odgovor, gdje se vraća `status` i `data`. Treba završiti na servisu `app/Services/AuthService.php`. U servisu `register()` jer rekao da mora biti `Enumerable`. U `register` ćemo dodati `dd($data)`:

```
public function register(array $data): Enumerable
{
    dd($data);
    return collect([]);
}
```

Pogledajmo kako to izgleda u Postman-u:

The screenshot shows the Postman interface. The top bar has 'POST' selected and the URL 'http://localhost:8000/api/v1/register'. Below the URL, the 'Body' tab is active, showing a JSON payload:

```
{
  "first_name": "Pero",
  "last_name": "Perić",
  "email": "pero@test.com",
  "password": "12345678",
  "password_confirmation": "12345678"
}
```

The response section shows a 500 Internal Server Error with a response body:

```
array:4 [ // app\Services\AuthService.php:12
  "first_name" => "Pero"
  "last_name" => "Perić"
  "email" => "pero@test.com"
  "password" => "12345678"
]
```

Sada možemo nešto s podacima raditi. Trebamo kreirati user-a tj. zapisati ga u bazu. Moramo mu i dodijeliti rolu. Bez obzira što nemamo pogled, korisnik se mora autorizirati. Ako nema prava treba dobiti tu informaciju tj. odgovor treba biti 403 Unauthorized.

Prije nego što krenemo u proces dodjele role, trebamo napuniti tablicu s rolama. Ono što je napunilo te role bio je `database\Seeder/RoleSeeder.php`. Kopirat ćemo taj seeder iz monolita. Ako pokrenemo `php artisan db:seed` pokrenut ćemo `DatabaseSeeder.php` a ne `RoleSeeder.php` koji želimo pokrenuti. Morali bi to napraviti ručno. Umjesto pozivanja factory-ja ili pojedinačnih seedera u `database/seeders/DatabaseSeeder.php` definiramo `call` kojem možemo predati matricu svih seeder-a a to je `RoleSeeder`:

```
class DatabaseSeeder extends Seeder
{
    /**
     * Seed the application's database.
     */
    public function run(): void
    {
        $this->call([
            RoleSeeder::class
        ]);
    }
}
```

Poanta je kada ga pokrenemo da on napuni. Problem u [database/seeders/RoleSeeder.php](#) je ako veći imamo podatke a želimo nadodati novu ulogu. Postojeće uloge su možda već vezane uz user-a. Zbog stranog ključa neće raditi, pucat će. Bolje je umjesto ovoga koristiti `Role:createOrUpdate()` metodu. Ova metoda ima zadatak ako podatka nema, da ga kreira, ako podatak postoji, i razlikuje se od onoga u bazi, napraviti će update, ako se ne razlikuje ne dira. Tako ćemo uvijek imati jedinstveni ključ, taj primarni ključ koji se neće promijeniti i on neće nikada puknuti:

```
class RoleSeeder extends Seeder
{
    /**
     * Run the database seeds.
     */
    public function run(): void
    {
        Role::updateOrCreate(['name' => 'Admin']);
        Role::updateOrCreate(['name' => 'User']);
        Role::updateOrCreate(['name' => 'Author']);
    }
}
```

Predavač se ipak odlučio za rješenje s petljom:

```
class RoleSeeder extends Seeder
{
    /**
     * Run the database seeds.
     */
    public function run(): void
    {
        $roles = [
            [
                'name' => 'Admin'
            ],
            [
                'name' => 'User'
            ],
            [
                'name' => 'Author'
            ]
        ];
        foreach ($roles as $role) {
            Role::updateOrCreate($role);
        }
    }
}
```

```

        ['name' => 'Admin'],
        ['name' => 'User'],
        ['name' => 'Author'],
    ];
    foreach ($roles as $role) {
        Role::updateOrCreate($role);
    }
}
}

```

```
$ php artisan db:seed
INFO  Seeding database.
Database\Seeders\RoleSeeder ..... RUNNING
Database\Seeders\RoleSeeder ..... 226 ms DONE

Administrator@DESKTOP-4J5DF41 MINGW64 /i/Laravel/algebra-blog-api
$
```

| id | name | permissions | created_at | updated_at | deleted_at |
|------|--------|-------------|---------------------|---------------------|------------|
| 1 | Admin | NULL | 2024-10-31 11:39:56 | 2024-10-31 11:39:56 | NULL |
| 2 | User | NULL | 2024-10-31 11:39:56 | 2024-10-31 11:39:56 | NULL |
| 3 | Author | NULL | 2024-10-31 11:39:56 | 2024-10-31 11:39:56 | NULL |
| NULL | NULL | NULL | NULL | NULL | NULL |

Metoda `upsert()` je slična ali može raditi s više slogova odjednom (batch operacija), `updateOrCreate()` radi samo s jednim zapisom, `upsert` je efikasniji za veće količine podataka jer koristi jednu SQL query, `updateOrCreate` radi dvije odvojene operacije (select pa insert/update). Sa `upsert()` metodom kod bi izgledao ovako:

```

class RoleSeeder extends Seeder
{
    /**
     * Run the database seeds.
     */
    public function run(): void
    {
        Role::upsert([
            ['name' => 'Admin'],
            ['name' => 'User'],
            ['name' => 'Author']
        ], ['name'], ['name']); // 'name' je jedinstveno polje, a također se može i
        // ažurirati
    }
}

```

Ipak, ostajemo kod verzije koju je odabrao predavač, s petljom.

Ako ponovo pokrenemo taj `db:seed`, nema duplikata. Da smo ostavili `insert()` metodu, duplikati bi se pojavili. Mogli smo obrisati pa insertovati ali bi auto increment se uvećavao. Ako npr. hoćemo dodati još jednog, npr. Editor. Dovoljno ga je samo navesti:

```
class RoleSeeder extends Seeder
{
    /**
     * Run the database seeds.
     */
    public function run(): void
    {
        $roles = [
            ['name' => 'Admin'],
            ['name' => 'User'],
            ['name' => 'Author'],
            ['name' => 'Editor'],
        ];
        foreach ($roles as $role) {
            Role::updateOrCreate($role);
        }
    }
}
```

```
$ php artisan db:seed
INFO  Seeding database.
Database\Seeders\RoleSeeder ..... RUNNING
Database\Seeders\RoleSeeder ..... 69 ms DONE
```

| 1 | Admin | NULL | 2024-10-31 11:39:56 | 2024-10-31 11:39:56 | NULL |
|---|--------|------|---------------------|---------------------|------|
| 2 | User | NULL | 2024-10-31 11:39:56 | 2024-10-31 11:39:56 | NULL |
| 3 | Author | NULL | 2024-10-31 11:39:56 | 2024-10-31 11:39:56 | NULL |
| 4 | Editor | NULL | 2024-10-31 11:47:13 | 2024-10-31 11:47:13 | NULL |

Imamo novi podatak a ostali su netaknuti.

Sada ćemo se vratiti na kreiranje korisnika i dodjelu uloge.

Svi novi korisnici dobit će ulogu `User`.

Idemo u `app/Services/AuthService.php` i metodu `register()`. Uklonit ćemo `dd($data)`; i umjesto toga napisati `$user = User::create($data);`. Dakle koristimo `User` model, mass assignment `create()` a u `$data` je matrica u kojoj šaljemo informacije koje su validirane: ime, prezime i password.

Kada smo kreirali user-a, sljedeći korak je dodavanje role tom user-u. Trebamo imati objekt te role. Ako pogledamo postoji relacija u modelu `app/Models/User.php/roles()` i u

`app\Models\Role::users()`. To znači da možemo dohvatiti objekt role iz baze. Možemo napisati `$role = Role::where('name', 'user')->first();` `first()` ako nam name nije unique, ne bi trebali imati role s istim imenom.

```
public function register(array $data): Enumerable
{
    $user = User::create($data);
    $role = Role::where('name', 'user')->first();
    dd($role);
    return collect([]);
}
```

Ako stavimo `get()` dobit ćemo collection, a ako kažemo `first()` dobit ćemo objekt. `first()` može puknuti. Pogledajmo iz Postman-a

The screenshot shows a POST request to `http://localhost:8000/api/v1/register`. The request body is a JSON object:

```
1
2   "first_name": "Pero",
3   "last_name": "Perić",
4   "email": "pero@test.com",
5   "password": "12345678",
6   "password_confirmation": "12345678"
```

The response body shows the created `Role` object:

```
App\Models\Role {#1258 // app\Services\AuthService.php:16
#connection: "mysql"
#table: "roles"
#primaryKey: "id"
#keyType: "int"
#incrementing: true
#with: []
#withCount: []
#preventsLazyLoading: false
#perPage: 15
#exists: true
#wasRecentlyCreated: false
#escapeWhenCastingToString: false
#attributes: array:6 [
  "id" => 2
  "name" => "User"
  "permissions" => null
  "created_at" => "2024-10-31 11:39:56"
  "updated_at" => "2024-10-31 11:39:56"
  "deleted_at" => null
]
#original: array:6 [
  "id" => 2
  "name" => "User"
  "permissions" => null
  "created_at" => "2024-10-31 11:39:56"
  "updated_at" => "2024-10-31 11:39:56"
  "deleted_at" => null
]
#changes: []
#casts: []
#classCastCache: []
#attributeCastCache: []
#dateFormat: null
#appends: []
#dispatchesEvents: []
#observables: []
#relations: []
#touches: []
#timestamps: true
#usesUniqueIds: false
```

Kada pokušamo `dd($role->users())` rola ima relaciju ka users i tu možemo koristiti metodu `attach()`. Ona se koristi kod Many-to-Many relacija u Laravelu za dodavanje relacionih zapisa. Možemo reći `dd($role->users()->attached(1))`. U pivot tablici se zapisuju ID usera i ID role i tu će biti relacija.

Idemo vidjeti što će se desiti ako promašimo ime role:

```
$role = Role::where('name', 'Userrr')->first();
```

Predavač je htio pokazati da ako je `dd($role->users()) null`, dobijemo sljedeću grešku:

Body Cookies Headers (7) Test Results 400 Bad Request

Pretty Raw Preview Visualize

```
{"status": "ERR", "message": "Call to a member function users() on null"}
```

Ne bi trebali imati imati grešku 400, već 404 Not found. Nad Eloquentom smo mogli koristiti:

```
$role = Role::where('name', 'Userrr')->firstOrFail();
```

I dalje dobivamo grešku 400 ali drugu grešku:

Body Cookies Headers (7) Test Results 400 Bad Request

Pretty Raw Preview Visualize

```
{"status": "ERR", "message": "No query results for model [App\Models\Role]."}
```

Ovo se dešava zato što u `app/Services/ResponseService.php` u metodi `error()` imamo definirano da ako nema `getStatusCode()` tj. nema koda pretvorimo ga u 400.

Ako je sve OK (kada popravimo `Userrr` u `User`). Uhvatio je relaciju belongs to many. `create` će vratiti `User` model za popunjениm podacima, ujedno će unutra postojati i ID dodjeljen user-u.

```
Illuminate\Database\Eloquent\Relations\BelongsToMany {#988 // app\Services\AuthService.php:16
    #query: Illuminate\Builder {#245
        #query: Illuminate\Builder {#235
            +connection: Illuminate\MySQLConnection {#282 ...24}
            +grammar: Illuminate\MySQLGrammar {#283 ...5}
            +processor: Illuminate\MySQLProcessor {#288}
            +bindings: array:9 [
                "select" => []
                "from" => []
                "join" => []
                "where" => array:1 [
                    0 => 2
                ]
                "groupBy" => []
                "having" => []
                "order" => []
                "union" => []
                "unionOrder" => []
            ]
        }
        +aggregate: null
        +columns: null
        +distinct: false
        +from: "users"
        +indexHint: null
        +joins: array:1 [
            0 => Illuminate\JoinClause {#1107
                +connection: Illuminate\MySQLConnection {#282 ...24}
                +grammar: Illuminate\MySQLGrammar {#283 ...5}
                +processor: Illuminate\MySQLProcessor {#288}
                +bindings: array:9 [
                    "select" => []
                    "from" => []
                    "join" => []
                    "where" => []
                    "groupBy" => []
                    "having" => []
                    "order" => []
                    "union" => []
                    "unionOrder" => []
                ]
            }
        ]
        +aggregate: null
    }
}
```

Bez problema možemo reći `$role->users()->attach($user);`:

```
{
    $user = User::create($data);
    $role = Role::where('name', 'User')->firstOrFail();
    $role->users()->attach($user);
    return collect([]);
}
```

Sada možemo izvući ID. Sljedeći korak bi bila prijava korisnika. Imamo kreiranje User-a, dohvati rolu User, dodijeli tu rolu user-u. Nakon toga ćemo prijaviti korisnika. Prijavit ćemo ga s `Auth::login($user);`

Ono što je razlika sada da login koji pozovemo nad Auth servisom, u biti nam vrati token. To znači da ovdje trebamo dobiti JWT token koji možemo vratiti.

```

public function register(array $data): Enumerable
{
    $user = User::create($data);
    $role = Role::where('name', 'User')->firstOrFail();
    $role->users()->attach($user);

    $token = Auth::login($user);

    return collect([
        "token" => $token,
    ]);
}

```

POST <http://localhost:8000/api/v1/register> Send

Params Auth Headers (8) Body **JSON** Pre-req. Tests Settings Cookies Beautify

```

1 {
2     "first_name": "Pero",
3     "last_name": "Perić",
4     "email": "pero@test.com",
5     "password": "12345678",
6     "password_confirmation": "12345678"
7 }

```

Body Cookies Headers (7) Test Results 200 OK 1100 ms 580 B Save Response

Pretty Raw Preview Visualize

```
{"status": "OK", "data": {"token": "eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9.eyJpc3MiOiJodHRwOi8vbG9jYWxob3N0OjgwMDAvYX_vTzInUzrwjhOvBz-eE"}}
```

Dobili smo token koji klijent može koristiti za autorizaciju (vrijedi 60 minuta). Može napraviti request prema zaštićenom dijelu autorizacije. Proces je jednostavan i sličan procesu kod monolita u [app/Services/AuthService.php](#) gdje smo definirali drugačije rolu. Proces je obrnut ali sličan. I u jednom i u drugom slučaju imamo `void` koji se vraća. Kod JWT se on zakačio na postojeći Auth servis.

Zgodna informacija je znati koliko dugo vrijede ti tokeni. Predavač želi napraviti testni endpoint koji bi zaštitio sa guardom Auth API. On će štiti taj endpoint ako korisnik nije prijavljen, ne može pristupiti. To znači da token mora biti u request-u, preciznije u header-u u request-u, u autorizaciji. Vidjet ćemo što je bearer token (tip autentikacijskog tokena koji se koristi u HTTP zahtjevima) koji se koristi na našoj strani da bi se iz tog tokena izvukle informacije o korisniku i provjerilo da li je svo točno i valja li taj token i na temelju toga pustio korisnik.

Kreirat ćemo endpoint u [routes/api.php](#) i dodati rutu test:

```
Route::get('/test', function () {
    return 'Hello World';
})->middleware('auth:api');
```

`auth` je ime middleware-a koji provjerava je li korisnik autenticiran. `:api` je ime guard-a (sve nakon dvotočke). Guard određuje način autentikacije (kako se provjerava identitet korisnika). Laravel podržava različite guard-e: `'web'`, `'api'`, `'sanctum'`, itd. Svaki guard može imati različitu konfiguraciju (sessions, tokens, itd.) Guard `'api'` obično koristi token-based autentikaciju. Ova ruta će: prvo provjeriti je li zahtjev autenticiran koristeći `'api'` guard. Ako nije, vratit će 401 Unauthorized. Ako jest, izvršit će funkciju i vratiti 'Hello World'. `AUTH GUARD=api` u .env datoteci definira defaultni guard koji će Laravel koristiti za autentikaciju. U konfiguraciji `config/auth` `'api'` znači da je `driver jwt` a `provider users`. Ako pokušamo prići bez da smo prijavljeni (nismo poslali token) bićemo odbijeni.

Gađamo rutu, bez da smo poslali token.

The screenshot shows a POSTMAN interface. The URL is `http://localhost:8000/api/v1/test`. The method is `GET`. The `Body` tab is selected, showing `JSON` as the format. The response body is:

```

1
2   "status": "ERR",
3   "message": "The route api/v1/test could not be found."
4

```

The status code is `400 Bad Request`.

Odgovor nije 401 nego 400 Bad Request. Postoje iznimke koje su reportable i iznimke koje su rendereble. Ovo je iznimka koja je reporable, dakle bit će zapisana u `logs/laravel.log`. Mi ne želimo da se ovo upisuje u log svaki puta kada neko ko nije autentificiran proba otvoriti endpoint. Ugasit ćemo da Laravel reporta exception. U `bootstrap/app.php` izmjenit ćemo:

```

->withExceptions(function (Exceptions $exceptions) {
    $exceptions->dontReport([
        RouteNotFoundException::class,
    ]);
}

```

`RouteNotFoundException` smo izvukli iz log-a. On ne kaže ne ti si neautentificiran i preusmjeri na login. Želimo da ovo završi kao 401 a ne 400. Zato želimo provjeriti da li taj `error()` završi kao renderable error. To ćemo pogledati u `app/Services/ResponseService.php/error()` da li dođe do ovdje. To ćemo jednostavno tako da na početku ubacimo `dd($e);`:

```
Symfony\Component\HttpKernel\Exception\NotFoundHttpException {#233 // app\Services\ResponseService.php:18
#message: "The route api/v1/test could not be found."
#code: 0
#file: "I:\Lara_\laravel\framework\src\Illuminate\Routing\AbstractRouteCollection.php"
#line: 44
->statusCode: 404
->headers: []
trace: [
    I:\Lara_\laravel\framework\src\Illuminate\Routing\AbstractRouteCollection.php:44 {
        Illuminate\AbstractRouteCollection->handleMatchedRoute(Request $request, $route) ...
        >
        > throw new NotFoundHttpException(sprintf(
        >     'The route %s could not be found.',
    )
    I:\Lara_\laravel\framework\src\Illuminate\Routing\RouteCollection.php:163 {
        Illuminate\RouteCollection->match(Request $request) ...
        >
        >     return $this->handleMatchedRoute($request, $route);
        >
    arguments: [
        $request: Illuminate\Request {#43 ...}
        $route: null
    ]
}
I:\Lara_\laravel\framework\src\Illuminate\Routing\Router.php:763 {
    Illuminate\Router->findRoute($request) ...
    >
    > $this->current = $route = $this->rout...
    >
    arguments: [
        $request: Illuminate\Request {#43 ...}
    ]
}
I:\Lara_\laravel\framework\src\Illuminate\Routing\Router.php:750 {
    Illuminate\Router->dispatchToRoute(Request $request) ...
    >
    > {
    >     return $this->runRoute($request, $this->findRoute($request));
    >
    arguments: [
        $request: Illuminate\Request {#43 ...}
    ]
}
```

Dobili smo `$e`. code je 0. Mi bio mogli reći ako je `$e` instanca od `RouteNotFoundException` onda generiraj kod, ne 400, nego 401.

```
public static function error(\Throwable $e): JsonResponse
{
    return response()->json(
        data: ['status' => 'ERR', 'message' => $e->getMessage()],
        status: self::getStatusCode($e),
    );
}

private static function getStatusCode (\Throwable $e): int
{
    if ($e instanceof RouteNotFoundException) {
        return 401;
    }

    return $e->getCode() ?: 400;
}
```

Prima bilo koju iznimku (Throwable). Vraća JSON odgovor s dvije informacije: status: 'ERR' (oznaka greške), message: poruka iz iznimke. Status kod se određuje pomoću `getStatusCode` metode. `getStatusCode` metoda određuje HTTP status kod za odgovor. Ako je iznimka tipa `RouteNotFoundException`, vraća 401 (Unauthorized). Inače vraća kod iz iznimke, a ako kod nije postavljen, vraća 400 (Bad Request). Ovo je error handler koji standardizira format error odgovora u API-ju.

GET http://localhost:8000/api/v1/test/

Params Auth Headers (8) Body Pre-req. Tests Settings

Query Params

| | Key | Value |
|--|-----|-------|
| | Key | Value |

Body Cookies Headers (7) Test Results

401 Unauthorized

Pretty Raw Preview Visualize

```
{"status": "ERR", "message": "Route [login] not defined."}
```

Promijenio se kod greške. Pokušali smo doći na endpoint ali nismo autorizirani.

Ako probamo napasti rutu koje nema:

HTTP http://localhost:8000/api/v1/test/asdasdaf

GET http://localhost:8000/api/v1/test/asdasdaf

Params Auth Headers (8) Body Pre-req. Tests Settings

Query Params

| | Key | Value |
|--|-----|-------|
| | Key | Value |

Body Cookies Headers (7) Test Results

400 Bad Request 491

Pretty Raw Preview Visualize

```
{"status": "ERR", "message": "The route api\\v1\\test\\asdasdaf could not be found."}
```

Idemo gurnuti teken i vidjeti da li će nas pustiti ako ga gurnemo u requestu. Uzećemo token koji smo gore definirali. Kopirat ćemo taj token iz **POST** i prebaciti se na **GET** request. Idemo na Authorization. To je moguće napraviti i na nivou kolekcije (moramo biti registrirani na Postman-u da bi mogli raditi kolekcije) i definirati kao Auth Type, Bearer Token i ubaciti taj naš token. U mom slučaju:

```
eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9.eyJpc3Mi0iJodHRw0i8vbG9jYWxob3N0OjgwMDAvYXBpL3YxL3J1Z1zdGVyIiwiaWF0IjoxNzMwNDU1OTczLCJleHAiOjE3MzA0NTc3NzMzMt5iZiI6MTczMDQ1NTk3MywianRpIjoiT Eg4UGJ1QWxEbW9Nd1dibyIsInN1YiI6IjQ iLCJwcnYi0iIyM2JkNWM40TQ5ZjYwMGFkYjM5ZTcwMwMOMDA4NzJkYjhNTk3NmY3In0.agGO_MV16ckc7si1_DVPs0-_vTtZ1nUzrwjh0vBz-eE
```

Često puta imamo različite uloge i različita prava u requestu pa ne možemo na taj način raditi. Predavač je ipak ubacio u sam endpoint ubacio token i pritisnuo dugme Send:

The screenshot shows the Postman interface with a GET request to `http://localhost:8000/api/v1/test/`. The 'Auth' tab is selected, and 'Bearer Token' is chosen as the type. A token value is pasted into the input field. The response shows a 200 OK status with the body 'Hello World'.

Ako malo promijenimo token (npr. dodamo jedno slovo) ili ako token prestane važiti, bit ćemo odbijeni:

```
{"status": "ERR", "message": "Route [login] not defined."}
```

Ono što bi još korisniku potencijalno mogli vratiti je osim **\$token**-a, je refresh token što možemo potencijalno koristiti za refresh kada glavni token istekne i koliko dugo traje (njegov time to live).

Htjeli bi ne ugnijezditi ovako kako smo do sada napravili:

```
return collect([
    "token" => $token,
]);
```

već kao response za register i login metode u `app/Services/AuthService.php`.

```
return $this->respondWithToken($token, $user);
```

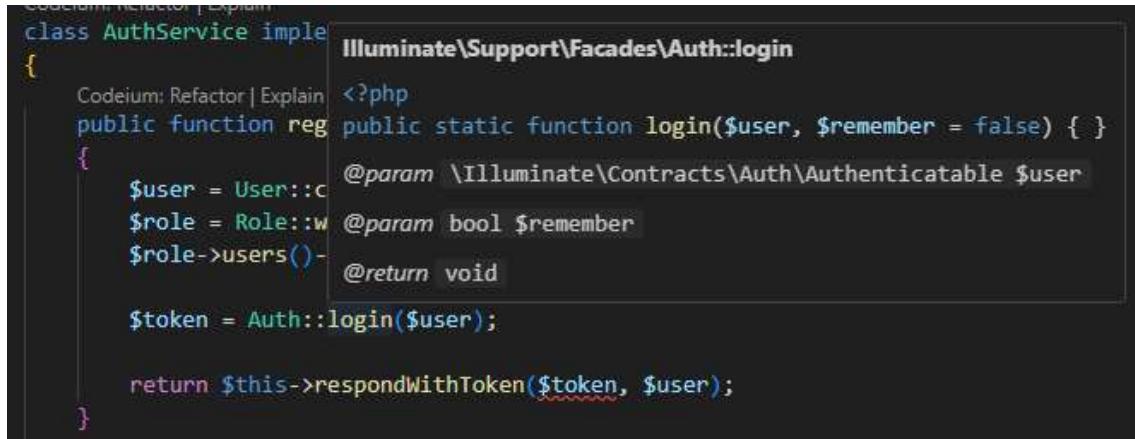
Idemo napisati `respondWithToken()`:

```
public function register(array $data): Enumerable
{
    $user = User::create($data);
    $role = Role::where('name', 'User')->firstOrFail();
    $role->users()->attach($user);

    $token = Auth::login($user);

    return $this->respondWithTokens($token, $user);
}

private function respondWithTokens(string $token, User $user): Enumerable
{
    return collect([
        'access_token' => $token,
        'token_type' => 'bearer',
        'expires_in' => Auth::factory()->getTTL(),
        'user' => $user
    ]);
}
```



`$token` je podvučen crvenim i dojavljuje grešku jer IntelliSense prepoznaje da `login()` očekuje `void` a mi smo u `respondWithToken` rekli da očekujemo `string`. Visual Studio Code ne vidi da će da je JWT promijenio funkcionalnost.

Idemo pogledati flow. Treba nam ruta. U `routes/api` ubacit ćemo:

```
Route::post('login', AuthController::class.'@login');
```

Ne štimmo ove rute nikakvim middleware-om, nema potrebe za `@CSRF` tokenom jer nema forme (bilo tko može poslati request kroz Postman), nema session gdje je CSRF token zapisan. Ono što ćemo napraviti je kao i kod `register()`, treba nam request na AuthController, isto tako na metodu `login()` u `app/Http/Controllers/AuthController.php` trebamo injectati request. Ono što možemo koristiti je Artisan za generiranje ili copy/paste što je iz VSC jednostavnije. Iskopirat ćemo `app/Http/Request/RegisterRequest.php` i promijeniti mu ime u `LoginRequest.php`. Unutra ćemo promijeniti ime klase s `RegisterRequest` u `LoginRequest`:

```
class LoginRequest extends FormRequest
```

Treba još promijeniti `rules()`. Uklanjamo `first_name` i `last_name`.

```
public function rules(): array
{
    return [
        'email' => 'required',
        'password' => 'required',
    ];
}
```

Imamo kontroler `app/Http/Controllers/AuthController.php` u koji trebamo injectati taj request. Tu ćemo dadati u `login()` metodu `LoginRequest $request`:

```
public function login(LoginRequest $request): JsonResponse
{
    $serviceAction = fn() => throw new \Exception('Not implemented');
    return $this->executeServiceAction($serviceAction);
}
```

Kao i kod `register()`, pozvali smo servis `login()` jer šaljemo samo validirane podatke:

```
public function login(LoginRequest $request): JsonResponse
{
    $serviceAction = fn() => $this->authService->login($request->validated());
    return $this->executeServiceAction($serviceAction);
}
```

Ako ne želimo validirati neki podatak, a želimo se osloniti na `validated()` metodu, ako nemamo u našem requestu neki rules za neki podatak koji dolazi u zahtjev, validated ga nikada neće prikazati. Što god se poslalo u requestu, će dati samo atribute definirane u `app/Http/Request/LoginRequest.php` u metodi `rules()` a to su `email` i `password`. Ako imate još neki podatak koji pristigne u requestu, možemo ga definirati kao nullable jer nam ne treba i ne mora postojati. Npr. možemo definirati '`'test'`:

```
public function rules(): array
{
    return [
        'email' => 'required',
        'password' => 'required',
        'test' => 'nullable',
    ];
}
```

Ako ne pošaljemo taj test on je nullable a ako ga pošaljemo dobićemo tu vrijednost u validated.

Imamo kontroler koji bi trebao validirane podatke poslati servisu. Servis bi trebao reći da ima podatke. Da to provjerimo što imamo u `app/Services/AuthService.php` u metodu `Login()` dodat ćemo `dd($data);` na početku metode:

```
public function login(array $data): Enumerable
{
    dd($data);
    return collect([]);
}
```

Testirat ćemo što imamo. Duplicirali smo register point i promjenili ga u `http://localhost:8000/api/v1/login`, Authorization uklonili a u Body->raw->Json ostavili smo:

```
{
    "email": "pero@test.com",
    "password": "12345678"
}
```

```
array:2 [ // app\Services\AuthService.php:28
  "email" => "pero@test.com"
  "password" => "12345678"
]
```

Došlo je do servisa i tu su podaci. Bitno je samo da pošaljemo email i password.

Proces prijave korisnika je jednostavan. Možemo vratiti potencijalno neki error ili exception ako korisnik ne bude autentificiran. Staviti ćemo attempt() koji vraća true ili false. Nama treba user da ga možemo prijaviti da možemo dobiti token.

```
public function login(array $data): Enumerable
{
    dd(Auth::attempt($data));
    if (!Auth::attempt($data)) {
        return collect([]);
    }
}
```

```
        }
        return collect([]);
    }
```

Dobivamo nazad upravo token.

```
"eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9eyJpc3Mi0iJodHRwOi8vbG9jYWxob3N0OjgwMDAvYXBpL3YxL2xzZ2luIiwiaiW
0IjoxNzMwNTA0NzA0LCJ1eHAiOiJE3MzA1MDY1MDQsIm5iZiI6MTczMDUwNDcwNCwianRpIjoiTnNsRkc5QRDTmdqZVZQWCIsInN
1YiI6IjyILCJwcnYi0IiyM2JkNN4OTQ5ZjYwNGFkYjM5ZTcwMWM0MDA4NzJkYjhNTk3NmY3In0.Bi2H3STIIgkRMfBionClDSE
sc-1bv4wTNpHK73cWE8
" // app\Services\AuthService.php:28
```

Napisat ćemo presretanje iznimke:

```
public function login(array $data): Enumerable
{
    if (!Auth::attempt($data)) {
        throw new \Exception('Invalid credentials', 401);
    }
    return collect([]);
}
```

The screenshot shows a Postman interface with the following details:

- Method:** POST
- URL:** <http://localhost:8000/api/v1/login>
- Headers:** (8)
- Body:** (JSON)

```
{
  "email": "pero@test.com",
  "password": "12345678"
}
```
- Status:** 200 OK

```
{"status":"OK", "data":[]}
```

The screenshot shows a Postman interface with a POST request to `http://localhost:8000/api/v1/login`. The Body tab is selected, showing a JSON payload:

```
1 "email": "pero1@test.com",
2 "password": "12345678"
```

The response status is `401 Unauthorized`.

```
{"status": "ERR", "message": "Invalid credentials"}
```

Dakle ako unesmo krivi mail ili password ne pušta nas dalje. Dakle ako uspješno prijavimo korisnika, tada bi trebali imati `$token` i zato možemo vratiti:

```
public function login(array $data): Enumerable
{
    if (!$token = Auth::attempt($data)) {
        throw new \Exception('Invalid credentials', 401);
    }
    return $this->respondWithTokens($token, Auth::user());
}
```

Kako ovo radi, `User $user` iz `respondWithTokens` ni ne treba.

POST http://localhost:8000/api/v1/login

Params Auth Headers (8) Body Pre-req. Tests Settings Cookies

raw JSON Beautify

```
1 "email": "pero@test.com",
2 "password": "12345678"
```

Body 200 OK 813 ms 960 B Save Response

Pretty Raw Preview Visualize

```
{"status": "OK", "data": {"access_token": "eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9.eyJpc3MiOiJodHRwOi8vbG9jYWxob3N0ZWx1ZnxsM3cvsmz9VmCzqY"}}
```

Ako uzmemo taj token i zamjenimo s onim koji je istekao u prozoru dobijemo:

GET http://localhost:8000/api/v1/test/

Params Auth Headers (7) Body Pre-req. Tests Settings

Type Bearer Token Token eyJ0eXAiOiJKV1Qi...

The authorization header will be automatically stored with end-to-end encryption locally using Vault.

Body 200 OK 638 ms 240 B

Pretty Raw Preview Visualize HTML

```
1 Hello World
```

Idemo vidjeti kako uz pomoć refresh_tokena zatražiti novi token. Trenutno imamo podešeno da token vrijedi 30 minuta. Kada korisniku istekne token, ne može pristupiti sadržaju. Što je vremenski okvir manji, tada je dužan obnoviti vremenski period. Kada je token invalidiran, dakle ne vrijedi. Predavač preporuča da refresh_token bude max. 24 sata a token 30 do 60 minuta.

Klijent može zatražiti novi token uz `refresh_token`. Zadržavamo postojeći `refresh_token` a zadužujemo novi `access_token`. Na nekoj ruti trebamo zaprimiti token i onda na temelju tog tokena pokušati generirati novi token, dodijeliti taj novi token user-u i vratiti taj novi token kao odgovor. Token treba raditi i biti funkcionalan.

U `routes/api.php` dodat ćemo red:

```
Route::post('refresh', AuthController::class.'@refresh');
```

U `app/Http/Controllers/AuthController.php` dodat ćemo `refresh()` metodu:

Trebamo imati neki request gdje ćemo prihvati `refresh()` metodu.

```
public function refresh(): JsonResponse
{
    $serviceAction = fn() => $this->authService->refresh();
    return $this->executeServiceAction($serviceAction);
}
```

Refresh token neće slati kroz header, nego će ga slati kroz tijelo.

Ako neko ukrade `refresh_token` je nezgodna stvar. Pitanje je gdje će držati korisnik tokene. Mi možemo kontrolirati serversku stranu ali ne i klijentsku. Naravno bilo bi najbolje da je token pohranjen u storage i prethodno kriptiran.

Uzećemo `LoginRequest.php` iskopirati ga i promijeniti mu ime u `app/Http/Requests/RefreshRequest.php`. Treba provo promijeniti ime klase iz `LoginRequest` u `RefreshRequest`. U metodi `rules()` promijenit ćemo što vraćamo s `return`:

```
public function rules(): array
{
    return [
        'refresh_token' => 'required|string',
    ];
}
```

Očekujemo da `refresh_token` bude uvijek i da bude string. Time imamo neku validaciju. Ili je moguće kriptiranje na serveskoj strani, poslati kriptirani token tamo, gdje ga on dekriptira i koristi. Mostoje mehanizmi kako dodatno zaštiti token.

Sada imamo request `app/Http/Requests/RefreshRequest.php` i imamo kontroler `app/Http/Controllers/AuthController.php`, a u kontroleru trebamo koristiti taj `RefreshRequest $request`.

```
public function refresh(RefreshRequest $request): JsonResponse
{
    $serviceAction = fn() => $this->authService->refresh($request->validated());
    return $this->executeServiceAction($serviceAction);
}
```

Ista stvar u refreshed metodi dodajemo `$request->validated()`.

U `app/Services/AuthService.php` dodajemo `refresh()` metodu:

```
public function refresh(array $data): Enumerable
{
    return $this->respondWithTokens(Auth::refresh(), Auth::user());
}
```

Mi više u `Auth` nemamo user-a jer je glavni token prestao vrijediti, što znači da mi nemamo uopće autentificiranog trenutnog usera. Mi bi trebali ponovo autentificirati user-a. Kada pogledamo attempt i login upravo to rade, autentificiraju user-a i dobijemo nekakav token. Mi želimo dobiti token. U `Login()` smo koristili `$user` i dobili smo token a u `attempt()` smo slali username i password, pa smo dobili `token`. Kada pogledamo u `Auth` nemamo usera jer je on odjavljen. Tražimo novi token jer je stari prestao vrijediti i mi nemamo pristup Auth user-u.

U `routes/api.php` dodat ćemo `dd()`:

```
Route::get('/test', function () {
    dd(FacadesAuth::user());
    return 'Hello World';
})->middleware('auth:api');
```

Dobit ćemo `null`.

Ako ubacimo novi token, koji vrijedi dobit ćemo user-a:

```
App\Models\User {#1056 // routes\api.php:18
    #connection: "mysql"
    #table: "users"
    #primaryKey: "id"
    #keyType: "int"
    +incrementing: true
    #with: []
    #withCount: []
    +preventsLazyLoading: false
    #perPage: 15
    +exists: true
    +wasRecentlyCreated: false
    #escapeWhenCastingToString: false
    #attributes: array:11 [
        "id" => 6
        "first_name" => "Pero"
        "last_name" => "Perić"
        "email" => "pero@test.com"
        "email_verified_at" => null
        "password" => "$2y$12$FAweTANvSzqVJQeR40TooO8acdqowu2ZnuI.Pa5OJiiqj3PJrhe6"
        "remember_token" => null
        "created_at" => "2024-11-01 20:39:07"
        "updated_at" => "2024-11-01 20:39:07"
    ]}
```

U refresh() trenutku mi nemamo user-a. Mi želimo vidjeti da li naš token uopće valja. `Auth::refresh()` ne zna kome treba dodijeliti token. Ako na jwt dokumentaciji pogledamo [refresh\(\) dokumentaciju](#), i tamo nema riječi o user-u. Piše da proslijedimo prvi parametar kako bi se toke „zauvijek“ stavio na crnu listu a drugi parametar će resetirati zahtjev za novim tokenom. Dodat ćemo novi redu metodu u `app/Services/AuthService.php/refresh()`.

```
public function refresh(array $data): Enumerable
{
    $newToken = JWTAuth::setToken($data['refresh_token'])->refresh();
    $user = JWTAuth::authenticate($newToken);
    return $this->respondWithTokens($newToken, $user);
}
```

Sada ćemo dobiti novi token. Da bi taj novi token vezali za usera, dodali smo još jedan red. Time dajemo token i taj authenticate na osnovu novog tokena znat će o kojem se user-u radi.

Pogledajmo što se događa ako u `.env` datoteci smanjimo `JWT_TTL=1`. Želimo iz Postmana prvo korisnika logirati i time mu dodijeliti `access_token`.

<http://localhost:8000/api/v1/login>:

The screenshot shows the Postman application interface. At the top, it displays the URL <http://localhost:8000/api/v1/login>. Below the URL, there are buttons for 'POST' and 'Send'. The 'Body' tab is selected, showing a JSON payload:

```
1 {  
2   "email": "pero@test.com",  
3   "password": "12345678"  
4 }
```

Below the body, the response details are shown: 200 OK, 1418 ms, 959 B. The response itself is a JSON object:1 {
2 "status": "OK",
3 "data": {
4 "access_token": "eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9.
eyJpc3Mi0iJodHRwOi8vbG9jYWxob3N0OjgwMDAvYXBpL3YxL2xvZ2luIiwiaWF0IjoxNzMwN
jQzODgwLCJleHAiOjE3MzA2NDM5NDAsIm5iZiI6MTczMDY0Mzg4MCwianRpIjoidzRxcVkzU2
MyODFucWhidSIisInN1YiI6IjYiLCJwcnYiOiIyM2JkNWM40TQ5ZjYwMGFkYjM5ZTcwMWM0MDA
4NzJkYjdhNTk3NmY3In0.CQsxY30v9meiYwhS2ITjVBBvBwjmerhDOXUjhfN-Qkc",
5 "token_type": "bearer",
6 "expires_in": 1,
7 "refresh_token": "eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9.
eyJpc3Mi0iJodHRwOi8vbG9jYWxob3N0OjgwMDAvYXBpL3YxL2xvZ2luIiwiaWF0IjoxNzMwN
jQzODgwLCJleHAiOjE3MzA2NDM5NDAsIm5iZiI6MTczMDY0Mzg4MCwianRpIjoiZUdieU1TY0
xDaVdvU040QSIisInN1YiI6IjYiLCJwcnYiOiIyM2JkNWM40TQ5ZjYwMGFkYjM5ZTcwMWM0MDA
4NzJkYjdhNTk3NmY3In0.CQsxY30v9meiYwhS2ITjVBBvBwjmerhDOXUjhfN-Qkc",
8 }
9 }

Ako nakon toga pogledamo <http://localhost:8000/api/v1/test/> bit će OK. Trebamo unijeti token.

Ako malo pričekamo, token je istekao:

GET <http://localhost:8000/api/v1/test/>

Params Auth Headers (7) Body Pre-req Tests Settings

Type Bearer Token Token eyJ0eXAiOiJKV1QiLCJhbG...

The authorization header will be automatically stored with end-to-end encryption locally using Vault.

Body 401 Unauthorized 455 ms 286 B Save

Pretty Raw Preview Visualize

```
{"status": "ERR", "message": "Route [login] not defined."}
```

Nakon toga u Body ćemo unijeti:

```
{  
    "refresh_token": null  
}
```

I pozvati link <http://localhost:8000/api/v1/refresh>.

Javi se validator s porukom `The refresh token filed is required.`

Ako zalijepimo refresh_token iza u tijelo:

```
{  
    "refresh_token":  
"eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9.eyJpc3Mi0iJodHRwOi8vbG9jYWxob3N0OjgwMDAvYXBpL  
3YxL2xvZ2luIiwiaWF0IjoxNzNjQzODgwLCJleHAiOjE3MzA2NDM5NDAsIm5iZiI6MTczMDY0Mzg4MCwi  
anRpIjoizUdieU1TY0xDaVdvU040QSIisInN1YiI6IjYiLCJwcnYi0iIyM2JkNWM4OTQ5ZjYwMGFkYjM5ZTc  
wMWMMOMDA4NzJkYjdhNTk3NmY3In0.OJ6nc4sbYDfJKvK1HK4z0LBsgAeMiQwtygEHV0_bukc"  
}
```

Dobijemo opet grašku. Greška je `Token could not be parsed from the request.`

Malo ćemo izmijeniti kod u `refresh()` mtdoi:

```
public function refresh(array $data): Enumerable  
{  
    $refreshToken = $data['refresh_token'];
```

```
dd($refreshToken);

$newToken = JWTAuth::setToken()->refresh();
$user = JWTAuth::authenticate($newToken);
return $this->respondWithTokens($newToken, $user);
}
```

Sada vidimo `refresh_token`:

```
"eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9eyJpc3MiOiJodHRwOi8vbG9jYWxob3N0OjgwMDAvYXBpL3YxL2xvZ2luIiwiaWF
0IjoxNzMwNjQzODgwLCJ1eHAiOiEzMzA2NDM5NDAsIm5iZiI6MTczMDY0Mzg4MCwianRpIjoizUdieU1TY0xDaVdvU040QSI
1YiI6IjYiLCJwcnYiOiIyM2JkNW40TQ5ZjYwMGFkYjM5ZTcwMjM0MDA4NzJkYjhNTk3NmY3In0.0J6nc4sbYDfJKvK1HK4z0LB
sgAeMiQwtygEHV0_bukc
" // app\Services\AuthService.php:40
```

Greška je u tome da se ne može parsirati token, dakle u `setToken()`. Problem je u stvari u `respondWithTokens`. I redu s `'refresh_token' => Auth::refresh()`,

Promijenit ćemo metodu :

```
private function respondWithTokens(string $token, User $user): Enumerable
{
    return collect([
        'access_token' => $token,
        'token_type' => 'bearer',
        'expires_in' => Auth::factory()->getTTL(),
        'refresh_token' => JWTAuth::claims('is_refresh_token' => true)-
>fromUser($user),
    ]);
}
```

Privatna metoda prima dva parametra `$token` i `$user`. Vraćamo `Enumerable` kolekciju. Kreiramo `refresh_token` i dodajemo zahtjev (engl. claim) `is_refresh_token => true`, generira se iz User modela, služi za dobivanje novog access tokena kada stari istekne.

Ako probamo otvoriti `http://localhost:8000/api/v1/test` i sa `token` i `refresh_token` ćemo uspjeti. Pogledajmo što se dešava ispod haube. U Laravelu kada se dogodi request, na neki endpoint, idemo na neku rutu, idemo na `/test` rutu. Ta ruta je zaštićena s auth:api. Request prolazi prvo kroz ugrađeni middleware čija je zadaća da autorizira korisnika preko tokena. Tamo nema mehanizma koji će razlučiti koji je to token – ako je valjan pušta ga. Mi na neki način to moramo proširiti i provjeriti što se nalazi u tom tokenu. Kreirat ćemo vlastiti middleware i iz tog tokena izvučemo informaciju ako postoji. Ako je `is_refresh_token` jednak `true`, onda se radi o refresh_token. Ako je neko gurnuo taj token, ne može se user s tim tokenom prijaviti već samo za dobivanje novog tokena. Mi u našem

middleware trebamo to spriječiti da iz header-a izvučemo informaciju tokena, i da onda iz tog tokena izvučemo informaciju da li je `refresh_token`.

Napraviti ćemo `VerifyAccessToken` middleware.

```
php artisan make:middleware VerifyAccessToken
```

```
$ php artisan make:middleware VerifyAccessToken
[INFO] Middleware [I:\Laravel\algebra-blog-api\app\Http\Middleware\VerifyAccessToken.php] created successfully.
```

Kreirao se `app/Http/Middleware/VerifyAccessToken.php`:

```
<?php

namespace App\Http\Middleware;

use Closure;
use Illuminate\Http\Request;
use Symfony\Component\HttpFoundation\Response;

class VerifyAccessToken
{
    /**
     * Handle an incoming request.
     *
     * @param \Closure(\Illuminate\Http\Request):
     * (\Symfony\Component\HttpFoundation\Response) $next
     */
    public function handle(Request $request, Closure $next): Response
    {
        return $next($request);
    }
}
```

Ovdje možemo vidjeti što se događa s requestom, dodat ćemo u handle:

```
public function handle(Request $request, Closure $next): Response
{
    dump($request->header('Authorization'));
    return $next($request);
}
```

Na requestu možemo doći do header-a.

Prvo trebamo middleware vezati za našu rutu. Prvo ćemo ga registrirati da ga možemo pozvati. Da bi to mogli moramo otići u `bootstrap/app.php`: gdje možemo kreirati uz pomoć middleware paramtera. Možemo nalijepiti i promijeniti da ide kroz naš Auth. Mi to ne želimo (pregaziti ga), jer on radi provjeru ispravnosti tokena. Mi želimo samo raditi provjeru tokena kojeg smo dobili u authorization header-u da nije `refresh_token` jer ne želimo dopustiti provjeru s njim.

```
->withMiddleware(function (Middleware $middleware) {
    $middleware->alias([
        'auth.verify.token' =>
\App\Http\Middleware\VerifyAccessToken::class,
    ]);
})
```

Tek sada idemo u rute i grupirat ćemo rute koje želimo zaštiti sa middleware-om (to smo već prije radili). Sve svoje rute vežemo uz `middleware` ili više `middleware`-a. Treba nam još jedan `middleware`, a to je `auth:api`.

```
Route::group(['middleware' => ['auth:api', 'auth.verify.token']], function () {
    Route::get('user', AuthController::class.'@user');
    Route::post('logout', AuthController::class.'@logout');
});
```

Trebamo ubaciti i ovu rutu u tu grupu:

```
Route::get('/test', function () {
    dd(FacadesAuth::user());
    return 'Hello World';
})->middleware('auth:api');
```

Sada kod izgleda ovako:

```
Route::group(['middleware' => ['auth:api', 'auth.verify.token']], function () {
    Route::get('/test', function () {
        return 'Hello World';
    });
});
```

Sve rute koje smo željeli smo zaštitali s middleware.

Trebamo pokrenuti iz Postman-a, <http://localhost:8000/api/v1/login> da bi smo dobili novi token (ako stari ne radi), te nakon toga uzeti taj važeći token i otici u <http://localhost:8000/api/v1/test>. Ako sada pokrenemo aplikaciju:

```
"  
Bearer eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9.eyJpc3MiOiJodHRwOi8vbG9jYmxob3N0OjgwMDAvYXBpL3YxL2xvZ21u  
IiwiaWF0IjoxNzMWzUzMjcwLC1eHaiOjE3MzA3NTMzMzAsIm5iZii6MTczMDc1MzI3MCwianRpIjoiOGdCdndRa2w1dTBWdnUw  
UyIsInN1YII6IjYiLCJwcnYiOiIyM2JkNWM4OTQ5ZjYwMGFkYjM5ZTcwMWM0MDA4NzJkYjhNTk3NmY3In0.38MjQ6SyFQH4UR7T  
LoBcAdYhRjNjuqUXReTYm25LuRg  
" // app\Http\Middleware\VerifyAccessToken.php:18
```

Hello World

Sada je token valjan i otišao je u naš `app\Http\Middleware\VerifyAccessToken.php` i u `$request->header('Authorization')`. Kao odgovor dobili smo token koji je poslan u autorizaciju. Razlika je što u odgovoru uvijek piše na početku Bearer. JWT već ima mogućnost da automatski dođemo do payload-a. Payload koji nama treba je onaj `is_refresh_token`. Prvo ćemo izmjeniti `app\Http\Middleware\VerifyAccessToken.php`, izbacit ćemo taj dump i umjesto njega napisati:

```
$user = JWTAuth::parseToken()->authenticate();
```

Napravili smo to preko JWT. `authenticate()` nam služi da provjerimo da li je token ispravan i da li se korisnik uspješno autentificira. Možemo i bez toga. Reći ćemo samo:

```
$user = JWTAuth::parseToken();  
dump($token);
```

Sada osim tokena, vidimo cijeli objekt. A u objektu su informacije iz onoga `is_refresh_token`. Promijenit ćemo malo metodu:

```
public function handle(Request $request, Closure $next): Response  
{  
    $user = JWTAuth::parseToken();  
    $refresh = $token->getClaim('is_refresh_token');  
    dump($user);  
    return $next($request);  
}
```

Pogledajmo što se desilo, vidimo `null` jer smo koristili `access_token` i u njemu ne postoji payload `is_refresh_token`.

```
null // app\Http\Middleware\VerifyAccessToken.php:21
```

Hello World

Ako se prijavimo sa `refresh_token`om dobili smo `true`. Imamo razliku u tokenima i razlikujemo radi li se o `refresh_token`. Ako se korisnik proba prijaviti sa `refresh_token` reći ćemo:

`refresh_token` se treba koristiti isključivo za dobivanje novog token-a a ne za prijavu.

```
public function handle(Request $request, Closure $next): Response
{
    try {
        $token = JWTAuth::parseToken();
        $refresh = $token->getClaim('is_refresh_token');
        if ($refresh) {
            return ResponseService::error(new \Exception('Invalid token', 401));
        }
    } catch (\Exception $e) {
        return ResponseService::error(new \Exception('Unauthorized', 401));
    }

    return $next($request);
}
```

Sada bi trebalo biti u redu.

Dobili smo grešku `Invalid token`, 401. Unauthorized, zato što smo bacili iznimku koju kada pogledamo `ResponseService` prima `\Throwable` parametar (ili exception ili error, mi smo koristili exception). Drugi način je refaktorirati `ResponseService` i reći da uz proslijedjeni exception i osloniti se na to da odredimo koja je instanca i vratimo 400 ili 401.

U `app/Services/AuthService.php` promijenit ćemo metodu `respondWithTokens()`:

```
private function respondWithTokens(string $token, User $user): Enumerable
{
    return collect([
        'access_token' => $token,
        'token_type' => 'bearer',
        'expires_in' => Auth::factory()->getTTL(),
        'refresh_token' => Auth::claims(['is_refresh_token' => true])->refresh(),
    ]);
}
```

Sada ako na ruti `http://localhost:8000/api/v1/test` probamo `access_token` prolazi a `refresh_token` ne prolazi.

Ako pričekamo da `access_token` istekne, ne radi ni `refresh_token`, što je greška u kodu. Dakle vratit ćemo kako je bilo.

```

private function respondWithTokens(string $token, User $user): Enumerable
{
    return collect([
        'access_token' => $token,
        'token_type' => 'bearer',
        'expires_in' => Auth::factory()->getTTL(),
        'refresh_token' => JWTAuth::claims(['is_refresh_token' => true])->fromUser($user),
    ]);
}

```

Sada ako na ruti <http://localhost:8000/api/v1/test> probamo `access_token` prolazi a `refresh_token` ne prolazi. Ako probamo na <http://localhost:8000/api/v1/refresh> ruti (naravno trebamo zadati `refresh_token` u Body):

The screenshot shows a Postman interface with the following details:

- Method:** POST
- URL:** http://localhost:8000/api/v1/refresh
- Body:** JSON (selected)
- Body Content:**

```

1 "refresh_token": "eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9.
2     eyJpc3Mi0iJodHRw0i8vbG9jYWxob3N0OjgwMDAvYXBpL3YxL2xvZ2luIiwiWF0IjoxNzMwODkzNjY1LCJleH
3     Ai0jE3MzA40TM3MjUsIm5iZiI6MTczMDg5MzY2NSwianRpIjoiTmhPeFhSREROSXd5cHBJSISInN1YiI6IjYi
4     LCJwcnYi0iIyM2JkNWm40TQ5ZjYwMGFkYjM5ZTcwMWM0MDA4NzJkYjhNTk3NmY3IiwiiaXNfcnvMcmvzaF90b2
5     tlbiI6dHJ1ZX0.8MLw41UDXw9AUSSqr2Jv708bMe_ixUMR2WoNzsruF"

```

Također dobijemo odgovor i novi `access_token` u njemu. `refresh_token` naravno ostaje stari:

The screenshot shows a JSON response with the following structure:

```

{
    "status": "OK",
    "data": {
        "access_token": "eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9.
            eyJpc3Mi0iJodHRw0i8vbG9jYWxob3N0OjgwMDAvYXBpL3YxL3JlZnJlc2giLCJpYXQiOjE3MzA5MDMxMTQ
            sImV4cCI6MTczMDkwMzIzMswibmJmIjoxNzMwOTAzMTC1LCJqdGkiOiJnTGJrc2U5M1hLb2kzYVBaIiwic3
            ViIjoiNiIsInBydiI6IjIzYmQ1Yzg5NDlmNjAwYWRiMzllNzAxYzQwMDg3MmRin2E10Tc2ZjciLCJpc19yZ
            WZyZXNoX3Rva2VuIjp0cnVlfQ.BoaBuuFNy3EyAZ_PyP8c7a9UXy6pQ3uMy7SvFN6FKuA",
        "token_type": "bearer",
        "expires_in": 1,
        "refresh_token": "eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9.
            eyJpc3Mi0iJodHRw0i8vbG9jYWxob3N0OjgwMDAvYXBpL3YxL3JlZnJlc2giLCJpYXQiOjE3MzA5MDMxNzU
            sImV4cCI6MTczMDkwMzIzMswibmJmIjoxNzMwOTAzMTC1LCJqdGkiOiJz0XVPQmZnbWZyYlpaY21TIiwic3
            ViIjoiNiIsInBydiI6IjIzYmQ1Yzg5NDlmNjAwYWRiMzllNzAxYzQwMDg3MmRin2E10Tc2ZjciLCJpc19yZ
            WZyZXNoX3Rva2VuIjp0cnVlfQ.KVD2@q0WDupEJv0fRwIiJLBh70wiAYeNu07rntrtXQM"
    }
}

```

Međutim, ako sada probamo s tim novim tokenom pristupiti <http://localhost:8000/api/v1/test>, javi nam Invalid token. Nešto nije u redu. Puklo je u `app\Http\Middleware\VerifyAccessToken.php` u `handle()` metodi gdje bacamo `$refresh`. Ubacit ćemo `dump()` ispred da vidimo što se dešava:

```
public function handle(Request $request, Closure $next): Response
{
    try {
        $token = JWTAuth::parseToken();
        $refresh = $token->getClaim('is_refresh_token');
        dump($refresh);
        if ($refresh) {
            return ResponseService::error(new \Exception('Invalid token', 401));
        }
    } catch (\Exception $e) {
        return ResponseService::error(new \Exception('Unauthorized', 401));
    }

    return $next($request);
}
```

Ovo bi značilo da nismo dobili novi `access_token`, ostao je payload `is _refresh_token` zato što smo iz njega `$refreshToken` kreirali novi token:

```
true // app\Http\Middleware\VerifyAccessToken.php:24
{"status":"ERR","message":"Invalid token"}
```

Morat ćemo u `app\Services\AuthService.php` u `refresh()` metodi promijeniti provjeru iz `true` u `false`:

```
public function refresh(array $data): Enumerable
{
    $refreshToken = $data['refresh_token'];

    $newToken = JWTAuth::claims(['is_refresh_token' => false])->setToken($refreshToken)->refresh();
    $user = JWTAuth::authenticate($newToken);

    return $this->respondWithTokens($newToken, $user);
}
```

Objasnit ćemo ovu metodu još jednom. Metoda `refresh()` je javna (`public`) jer joj pristupamo van klase u kojoj je definirana. Argument koji prihvaća je parametar `$data` koji je tipa `array`. Matrica sadrži podatke potrebne za obnovu tokena, među njima i `refresh_token`. Ova metoda vraća instancu tipa `Enumerable`. U Laravelu `Enumerable` može predstavljati kolekciju podataka koju se može iterirati,

što znači da će rezultat sadržavati više vrijednosti, na primjer nove tokene i korisničke podatke. Metoda koristi `$data` kako bi dohvatile vrijednost `refresh_token`. Metoda `claims` omogućava dodavanje dodatnih podataka (claimova) unutar tokena. Ovdje se dodaje `is_refresh_token` s vrijednošću `false`. `setToken($refreshToken)` postavlja `refresh_token` kao aktivni token s kojim će se raditi. Pozivom `refresh()` metoda obnavlja token. Generira novi JWT (JSON Web Token) koristeći `refresh_token`, kako bi korisnik dobio novi pristupni token. Metoda `authenticate` koristi JWTAuth za autentifikaciju korisnika temeljem novo-stvorenog tokena (`$newToken`). Metoda vraća rezultat pozivanjem `respondWithTokens` koji je unutar iste klase. Ova metoda prima novi token i korisnički objekt te ih pakira u strukturu koja odgovara tipu `Enumerable`. Vraćamo kolekciju podataka koja uključuje novi token (`$newToken`) i podatke o korisniku (`$user`).

Klijent (web aplikacija) radi request sa tokenom. API vrati 401 Unauthorized.

The screenshot shows a browser's developer tools Network tab. A single request is listed with a status of 401 Unauthorized. The response body is a JSON object: {"status": "ERR", "message": "Invalid token"}.

Klijentska aplikacija treba na to reagirati na sljedeći način: Imamo `refresh_token` koji smo dobili prilikom prijave u sistem, uz pomoć njega trebamo ga poslati na refresh point <http://localhost:8000/api/v1/test> i vidjeti što ćemo dobiti. Ako je taj `refresh_token` nevažeći (istekao ili nije dobar), neće ga dobiti. Ako je taj `refresh_token` OK, dobit ćemo novi `access_token` i `refresh_token`. Pitanje je treba li nam uvijek novi `refresh_token`. U tom trenutku nam i dalje vrijedi stari `refresh_token`. Dakle trebamo invalidirati taj token i staviti na crnu listu.

`Logout()` metoda

Ako korisnik nije prijavljen u sistem, ne bi trebao moći pristupiti ovoj metodu. Zato ćemo ga staviti u `routes/api.php`. Ova ruta treba biti `GET` – ne šaljemo nikakav podatak.

```
Route::get('logout', AuthController::class.'@logout');
```

U `app/Http/Controllers/AuthController.php` dodat ćemo metodu `Logout()` na kraju klase:

```
public function logout(): JsonResponse
{
    $serviceAction = fn() => $this->authService->logout();
    return $this->executeServiceAction($serviceAction);
}
```

Ova metoda vraća `JsonResponse` kao odgovor. Ovdje je definirana anonimna funkcija `fn()` koja će pozvati `Logout()` metodu iz `AuthService` klase. Ova anonimna funkcija se dodjeljuje varijabli `$serviceAction`. Poziva se `executeServiceAction()` metoda, koja prima `$serviceAction` kao argument.

Ova `executeServiceAction()` metoda vjerojatno sadrži logiku za izvršavanje `$serviceAction` anonimne funkcije i vraćanje odgovarajućeg `JsonResponse` objekta.

Ukratko, ova metoda kontrolera: Definira anonimnu funkciju koja će pozvati `Logout()` metodu iz `AuthService` klase. Prosljeđuje tu anonimnu funkciju u `executeServiceAction()` metodi.

`executeServiceAction()` metoda izvršava anonimnu funkciju i vraća odgovarajući `JsonResponse` objekt.

Ovakav pristup se često koristi kako bi se odvojila logika kontrolera od poslovne logike servisa. Kontroler samo poziva odgovarajući servis, a servis implementira poslovnu logiku.

U `app/Interfaces/AuthServiceInterface.php` nadopunit ćemo interface:

```
<?php

namespace App\Interfaces;

use Illuminate\Support\Enumerable;

interface AuthServiceInterface
{
    public function register(array $data): Enumerable;
    public function login(array $data): Enumerable;
    public function refresh(array $data): Enumerable;
    public function logout(): array;
}
```

Idemo na `app/Services/AuthService.php` i na metodu `Logout()`:

```
public function logout(): void
{
    Auth::logout(true);

    return ['message' => 'Successfully logged out'];
}
```

Mogli bi smo ovdje i napisati kada izdajemo nove tokene da ovdje stvari invalidiramo ali mi to nećemo sada radi. U `.env` datoteci ćemo podići `JWT_TTL=10`

Dakle prijavimo korisnika na `http://localhost:8000/api/v1/login` i iskopiramo `access_token`. Zatom idemo na `http://localhost:8000/api/v1/test` i vidjet ćemo Hello Word. Zatim s istim tokenom idemo na `http://localhost:8000/api/v1/logout` i odjavimo korisnika. Ako sada probamo otvoriti `http://localhost:8000/api/v1/test` ne ide.

Umjesto login dijela i prijavljivanja korisnika u sistem, mogli bi koristiti dvofaktorsku autentifikaciju. Dakle mogli smo na email poslati neki kod i korisnik ga npr. mora unijeti.

<https://itnext.io/laravel-free-two-factor-authentication-5a4be723dfa7>

Event-i u projektu

Eventi su objašnjeni detaljno u poglavlju niže.

Idemo kreirati event (događaj) za naš projekt.

```
php artisan make:event RegisterEvent
```

```
$ php artisan make:event RegisterEvent
[INFO] Event [I:\Laravel\algebra-blog-api\app\Events\RegisterEvent.php] created successfully.
```

Ova naredba kreira novu klasu `RegisterEvent.php` u direktoriju `app/Events`, koja može sadržavati podatke koje želite proslijediti slušateljima kada se event emitira.

Može biti jedan event i više lisenera koji osluškuju. Nadalje, kreirat ćemo lisener:

```
php artisan make:listener RegisterListener --event=RegisterEvent
```

```
$ php artisan make:listener RegisterListener --event=RegisterEvent
[INFO] Listener [I:\Laravel\algebra-blog-api\app\Listeners\RegisterListener.php] created successfully.
```

Ako ga i ne vežemo s `RegisterEvent`, možemo to kasnije napraviti u kodu. Trebamo vidjeti dva nova direktorija: `app/Events` i `app/Listeners`. U `app/Events` nalazi se `RegisterEvent.php`. U njemu se nalaze `__construct()` i `broadcastOn()` namjenjen emitovanju na različitim kanalima. U `app/Listeners` je `RegisterListener.php`.

Postoji opcija queue kada ima puno listenera i eventa, ne želimo da sve počne od jednom raditi. Onda se smjesti u queue i rješava se jedan po jedan. Postoje sistemi za to kao što je [Kafka](#) i [RabbitMQ](#). Moguće je imati kontrolu u redu slijeda što se događa.

`app/Events/RegisterEvent.php` izgledao ovako:

```
<?php

namespace App\Events;

use Illuminate\Broadcasting\Channel;
use Illuminate\Broadcasting\InteractsWithSockets;
use Illuminate\Broadcasting\PresenceChannel;
use Illuminate\Broadcasting\PrivateChannel;
use Illuminate\Contracts\Broadcasting\ShouldBroadcast;
use Illuminate\Foundation\Events\Dispatchable;
use Illuminate\Queue\SerializesModels;

class RegisterEvent
{
```

```

use Dispatchable, InteractsWithSockets, SerializesModels;

/**
 * Create a new event instance.
 */
public function __construct()
{
    //
}

/**
 * Get the channels the event should broadcast on.
 *
 * @return array<int, \Illuminate\Broadcasting\Channel>
 */
public function broadcastOn(): array
{
    return [
        new PrivateChannel('channel-name'),
    ];
}

```

Želimo promijeniti `RegisterEvent` tako da hendlamo user-a. Kada se event dispatcha, možemo eventu poslati User-a. To radimo u servisu `app/Services/AuthService.php`. U `register()` metodi prije nego što napravimo `return`, trebamo pozvati event i kažemo `dispatch()` i pošaljemo objekt `$user`:

```
RegisterEvent::dispatch($user);
```

Kada dispatchamo event, aktivirat ćemo listener zato što `dispatch()` instancira event objekt. User trebamo negde zapisati (neće biti automatski dostupan). Ako pogledamo `app/Listeners/RegisterListener.php`:

```

<?php

namespace App\Listeners;

use App\Events\RegisterEvent;
use Illuminate\Contracts\Queue\ShouldQueue;
use Illuminate\Queue\InteractsWithQueue;

class RegisterListener
{

```

```
/**  
 * Create the event listener.  
 */  
public function __construct()  
{  
    //  
}  
  
/**  
 * Handle the event.  
 */  
public function handle(RegisterEvent $event): void  
{  
    //  
}
```

U metodu `handle` možemo dodati `dump`:

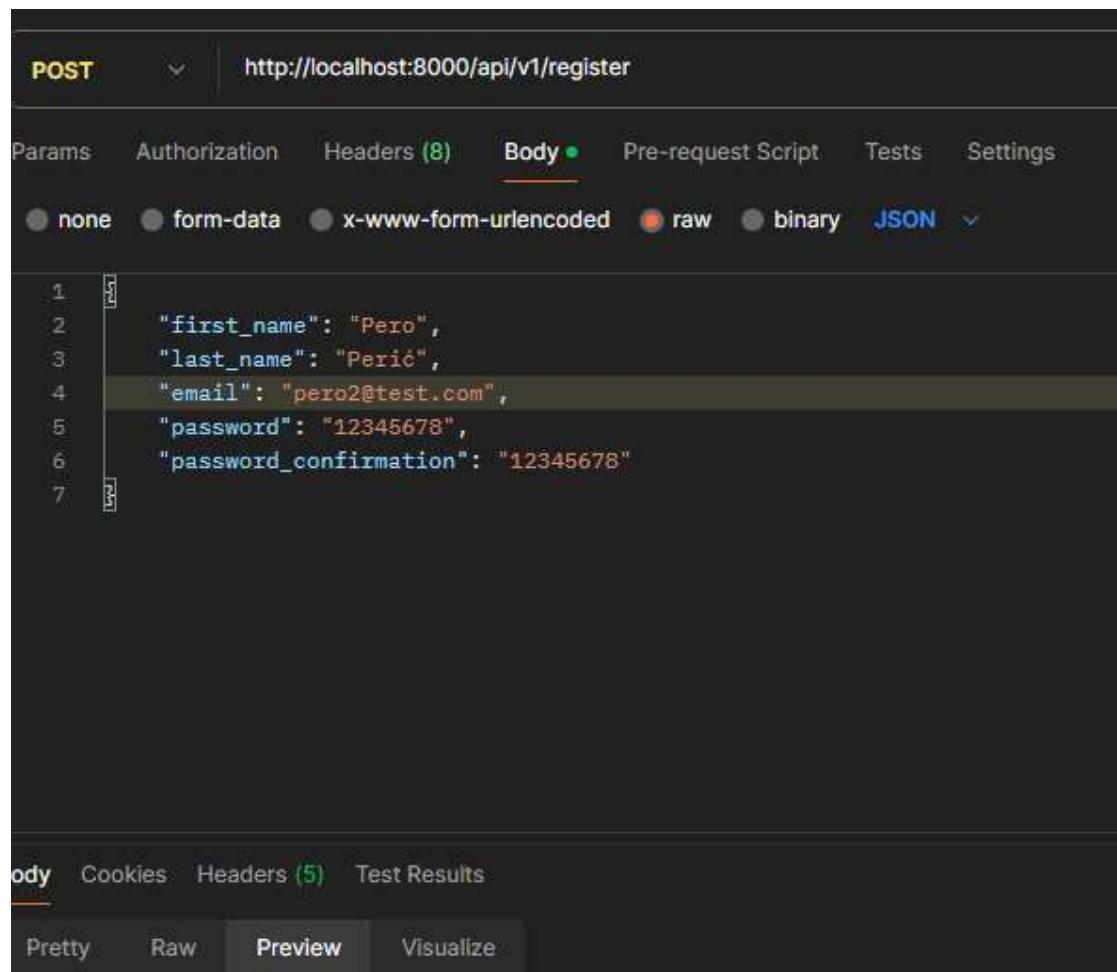
```
public function handle(RegisterEvent $event): void  
{  
    dump($event->user);  
}
```

On ne vidi user ali `dispatch()` je proslijedio tog User u konstruktoru. `__construct` ga ne vidi pa ga možemo proslijediti u `app/Events/RegisterEvent.php`

```
public User $user;  
  
/**  
 * Create a new event instance.  
 */  
public function __construct(User $user)  
{  
    $this->user = $user;  
}
```

Sada imamo svojstvo u `RegisterEvent.php` koje će nam biti dostupno. Poanta je da će `$user` pristići prilikom dispatch-a. Sada možemo pokrenuti proces slanja nekakvog email-a.

Pogledajmo što će se dogoditi prilikom registracije korisnika kada otvorimo: <http://localhost:8000/api/v1/register>:



POST <http://localhost:8000/api/v1/register>

Params Authorization Headers (8) Body Pre-request Script Tests Settings

Body (8)

none form-data x-www-form-urlencoded raw binary JSON

```
1
2     "first_name": "Pero",
3     "last_name": "Perić",
4     "email": "pero2@test.com",
5     "password": "12345678",
6     "password_confirmation": "12345678"
7
```

Body Cookies Headers (5) Test Results

Pretty Raw Preview Visualize

```
App\Models\User {#291 // app\Listeners\RegisterListener.php:24
  #connection: "mysql"
  #table: "users"
  #primaryKey: "id"
  #keyType: "int"
  +incrementing: true
  #with: []
  #withCount: []
  +preventsLazyLoading: false
  #perPage: 15
  +exists: true
  +wasRecentlyCreated: true
  #escapeWhenCastingToString: false
  #attributes: array:7 [
    "first_name" => "Pero"
    "last_name" => "Perić"
    "email" => "pero2@test.com"
    "password" => "$2y$12$vDhfCGDCxFtI74GnC9uDaunfJGXlwFapVtoFgJEymyoJ6Eb/BGDjG"
    "updated_at" => "2024-11-09 00:47:22"
    "created_at" => "2024-11-09 00:47:22"
    "id" => 8
  ]
```

Prošao je kroz listener, dobili smo tog user-a i ispod vidimo odgovor koji se dogodio (klasičan odgovor koji smo imali i prije). Sada kada imamo usera, možemo poslati mail korisniku iz [app/Listeners/RegisterListener.php](#) i metode `handle()`:

Za slanje mailova Laravel nudi `Mailable` klase. Ove klase pohranjene su u [app/Mail](#) direktoriju. Direktorij će se pojaviti kada generiramo svoju prvu mailable klasu (u našem slučaju `WelcomeMail`) pomoću Artisana:

```
php artisan make:mail WelcomeMail
```

```
$ php artisan make:mail WelcomeMail
[INFO] Mailable [I:\Laravel\algebra-blog-api\app\Mail>WelcomeMail.php] created
successfully.
```

Dakle kreirali smo [app/Mail/WelcomeMail.php](#):

```
<?php

namespace App\Mail;

use Illuminate\Bus\Queueable;
use Illuminate\Contracts\Queue\ShouldQueue;
use Illuminate\Mail\Mailable;
use Illuminate\Mail\Mailables\Content;
use Illuminate\Mail\Mailables\Envelope;
use Illuminate\Queue\SerializesModels;

class WelcomeMail extends Mailable
{
    use Queueable, SerializesModels;

    /**
     * Create a new message instance.
     */
    public function __construct()
    {
        //
    }

    /**
     * Get the message envelope.
     */
    public function envelope(): Envelope
    {
```

```
        return new Envelope(
            subject: 'Welcome Mail',
        );

    }

    /**
     * Get the message content definition.
     */
    public function content(): Content
    {
        return new Content(
            view: 'view.name',
        );
    }

    /**
     * Get the attachments for the message.
     *
     * @return array<int, \Illuminate\Mail\Mailables\Attachment>
     */
    public function attachments(): array
    {
        return [];
    }
}
```

Možemo vidjeti da za Content koristi Blade. Kontent iz mail-a nije ništa drugo nego HTML. Mi iz `RegisterListener.php` i `handle()` šaljemo object usera:

```
public function handle(RegisterEvent $event): void
{
    // Send an email to the user
    Mail::to($event->user->email)->send(new \App\Mail\WelcomeMail($event-
>user));
}
```

Tu šaljemo object user-a.

U konstruktoru ćemo dohvatiti user-a. U `app/Mail/WelcomeMail.php`:

```
public User $user;

/**
 * Create a new message instance.

```

```
 */
public function __construct()
{
    $this->user = $user;
}
```

```
public function content(): Content
{
    return new Content(
        view: 'mail.welcome',
    );
}
```

Možemo definirati da je public i automatski je dostupno sa `with`. `public` je moguće reći i unutar `__construct()`. U svom Blade-u tj. view imat ćemo pristup user objektu. Iz tog user objekta moći ćemo doći do informacija (`first_name`, `last_name`) ako želimo personifikaciju poruke.

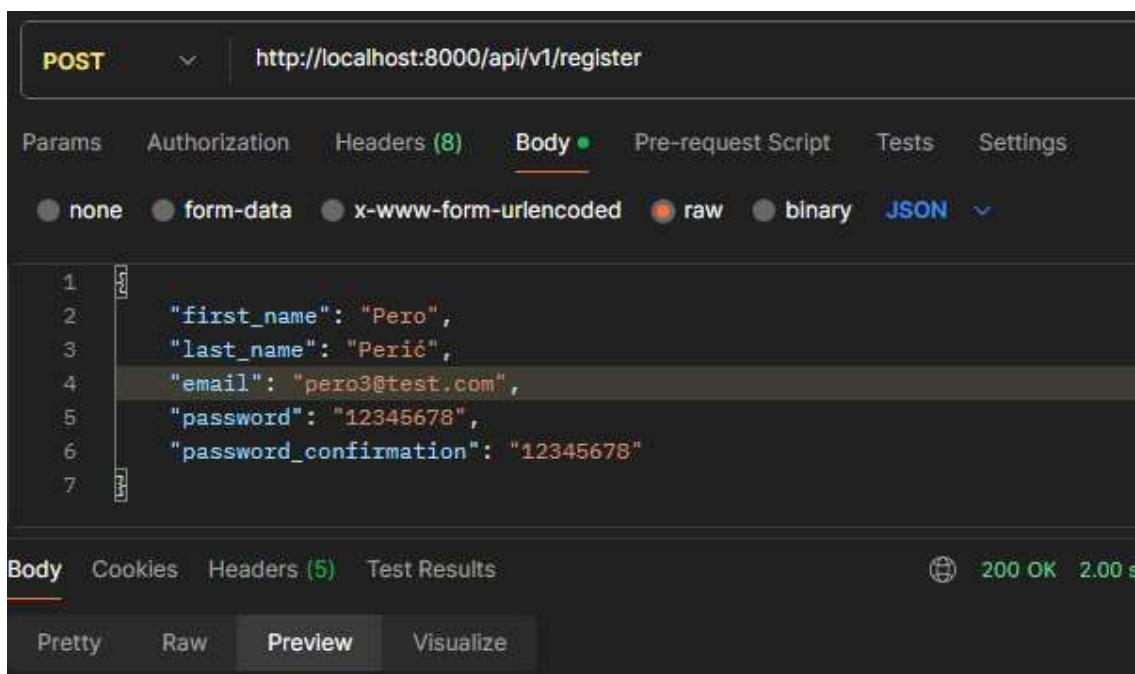
U `resource/views` otvorit ćemo novi poddirektorij `mail` i u njemu pravimo novi `welcome.blade.php`,

```
<h1>Welcome{{ $user->first_name }}{{ $user->last_name }}</h1>
<p>Lorem ipsum dolor, sit amet consectetur adipisicing elit. Autem necessitatibus odio,
partiatur, sapiente assumenda vel optio eos, dolorem debitis voluptates libero a!
Provident
laboriosam quas aperiam. Hic iusto odit omnis.</p>
```

Idemo pogledati flow. Imamo event koji će se pokrenuti kada se korisnik registrira u `app/Services/AuthService.php`. Dispatch-a se taj event u kojem ćemo dobiti usera koji se registrirao tj. objekt tog usere. Kada se taj event pokrene tj. kada se trigerira, dobit će obavijest listener (njegova zadaća je da sluša kada se taj listener dogodi), a dogodi se tako da dispatcher kreira objekt a kada se to dogodi aktivirat će se u listeneru handle metoda koja prima `$event` objekt. U tom trenutku mi kažemo `Mail::to` i iz `$event`-a dodemo do `user` objekta a iz `user` objekta do `email`-a. Moramo biti sigurni da su to sve stvari iz našeg modela `app/Models/User/User` (definirane u `$fillable` a to su u našem slučaju `first_name`, `last_name`, `email`, `password`, `phone`). Nakon toga kažemo send i damo mailable klasi `WelcomeMail`. Poanta je da toj mailable klasi proslijedimo usera iz `$event` jer će ta klasa kreirati mail sa subject-om, od koga je poslano i nekakav bot. Moguće je promijenti tko šalje ili će se Laravel osloniti na konfiguraciju.

Po default-u svi testovi za mail rade se u logu. Ako želimo da pošaljemo stvarni mail moramo uključiti `smtp`. U `.env` datoteci trema uključiti `smtp`, i ostale stvari oko mail-a. U `config/mail.php` već je postavljena konfiguracija koja čita iz `.env`. Ako želimo koristiti Gmail, još uvijek je moguće koristiti SMTP iako zvanično više ne radi ali moguće je koristiti alternative (Mailtrap, Brevo, Mailgun, SendGrid). `from` je podešen kao `hello@example.com` i `from` kao `Example`. To možemo promijeniti sa varijablama.

Mi bi trebali u [storage/logs/laravel.log](#) dobiti mail kada ga registriramo.



The screenshot shows a POST request to `http://localhost:8000/api/v1/register`. The Body tab is selected, showing a JSON payload:

```
1  "first_name": "Pero",
2  "last_name": "Perić",
3  "email": "pero3@test.com",
4  "password": "12345678",
5  "password_confirmation": "12345678"
```

The response tab shows a 200 OK status with a response time of 2.00s. The response body is displayed as:

```
App\Models\User {#291 // app\Listeners\RegisterListener.php:24
  #connection: "mysql"
  #table: "users"
  #primaryKey: "id"
  #keyType: "int"
  +incrementing: true
  #with: []
  #withCount: []
  +preventsLazyLoading: false
  #perPage: 15
  +exists: true
```

U `laravel.log` trebao bi biti sadržaj mail-a. Većina sistema će označiti ovakav mail kao spam zato što nije upaljen name zapis nad domenom, MX zapis kojim se validira i daje certifikat da je ispravna adresa a ne spam. Gdje god su promocije tj. reklame ne završi u spam-u. Mailovi namješteni po firmama su rigorozniji od Gmaila. Certifikat od strane servera obično prouste mail i ne završe u spamu.

```
public function content(): Content
{
    return new Content(
        view: 'mail.welcome',
        with: ['user' => $this->user],
    );
}
```

```
storage > logs > mail.log
1 [2024-11-10 00:51:47] local.DEBUG: From: algebra-blog-api <hello@example.com>
2 To: pero3@test.com
3 Subject: Welcome Mail
4 MIME-Version: 1.0
5 Date: Sun, 10 Nov 2024 00:51:47 +0000
6 Message-ID: <9fbbf8dd9095dc58284a8f3f8aa52bea@example.com>
7 Content-Type: text/html; charset=utf-8
8 Content-Transfer-Encoding: quoted-printable
9
10 <h1>Welcome Pero Perić</h1>
11 <p>Lorem ipsum dolor, sit amet consectetur adipisicing elit. Autem necessitatibus odio,
12 partiatur, sapiente assumenda vel optio eos, dolorem debitis voluptates libero a! Provident
13 laboriosam quas aperiam. Hic iusto odit omnis.</p>
```

Ako želimo da se logging izdvoji u `mail.log` (dakle u drugu log datoteku) napisat ćemo u `config/logging.php`:

```
'channels' => [
    // ... ostali channels ...

    'mail' => [
        'driver' => 'single',
        'path' => storage_path('logs/mail.log'),
        'level' => 'debug',
    ],
],
```

Event

U Laravelu, event-i (događaji) omogućavaju odvajanje različitih dijelova aplikacije kako bi reagirali na specifične akcije ili promjene. Oni su korisni za obavljanje operacija koje trebaju biti pokrenute kao odgovor na određeni događaj, kao što su stvaranje korisničkog računa, ažuriranje profila ili slanje e-mail obavijesti.

Kako rade event-i u Laravelu?

Event-i rade na principu emitiranja i slušanja (**dispatching i listening**):

- Emitiranje događaja:** Kada se neka akcija dogodi, kao što je stvaranje novog zapisa u bazi, može se "emitirati" odgovarajući event.
- Slušanje događaja:** Slušači (*listeners*) čekaju na specifične event-e i reagiraju kada se oni emitiraju. Na primjer, nakon kreiranja korisnika, slušatelj može poslati e-mail dobrodošlice.

Struktura event-a u Laravelu

Laravel ima ugrađeni EventServiceProvider, gdje se registriraju svi event-i i njihovi slušatelji.

Kreiranje event-a

Da biste kreirali novi event, možete koristiti Artisan naredbu:

```
php artisan make:event ImeEvent-a
```

Ova naredba kreira novu klasu u direktoriju `app/Events`, koja može sadržavati podatke koje želite proslijediti slušateljima kada se event emitira.

Primjer jednostavnog event-a `UserRegistered`:

```
<?php

namespace App\Events;

use App\Models\User;

class UserRegistered
{
    public $user;

    public function __construct(User $user)
    {
        $this->user = $user;
    }
}
```

Ovdje `UserRegistered` event sadrži podatke o korisniku koji je upravo registriran.

Kreiranje listener-a (slušatelja)

Listener je komponenta koja reagira kada se određeni event emitira. Možete ga generirati pomoću:

```
php artisan make:listener ImeListener-a
```

Ova naredba stvara listener u `app/Listeners`.

Primjer listener-a `SendWelcomeEmail` za `UserRegistered` event:

```
<?php

namespace App\Listeners;

use App\Events\UserRegistered;
use Illuminate\Support\Facades\Mail;
use App\Mail\WelcomeEmail;

class SendWelcomeEmail
```

```
{  
    public function handle(UserRegistered $event)  
    {  
        Mail::to($event->user->email)->send(new WelcomeEmail($event->user));  
    }  
}
```

Ovdje `SendWelcomeEmail` **listener** koristi podatke iz `UserRegistered` event-a da pošalje e-mail dobrodošlice novom korisniku.

Registracija event-a i listener-a

Event-i i njihovi listener-i registriraju se u `EventServiceProvider` klasi, koja se nalazi u `app/Providers`:

```
protected $listen = [  
    UserRegistered::class => [  
        SendWelcomeEmail::class,  
    ],  
];
```

Ovaj kod govori Laravelu da kad se emitira `UserRegistered` event, treba pokrenuti `SendWelcomeEmail` listener.

Emitiranje event-a

Da biste emitirali event, koristite `event()` helper funkciju ili Event facade:

```
event(new UserRegistered($user));
```

Prednosti korištenja event-a

- **Decoupling (Razdvajanje koda):** Event-i omogućavaju da kod bude modularan i lakše održiv.
- **Ponovljena upotreba koda:** Jeden event može imati više listener-a, što omogućava da isti event pokrene različite akcije.
- **Jednostavna skalabilnost:** Ako aplikacija treba dodatne funkcionalnosti vezane uz neki događaj, jednostavno je dodati nove listener-e.

Laravel Events i Queues

Listener-i se često postavljaju da se izvrše u pozadini koristeći queue (redove) za poboljšanje performansi aplikacije. Dodavanje listener-a na queue postiže se jednostavno implementacijom `ShouldQueue` interface-a na listener klasu:

```
use Illuminate\Contracts\Queue\ShouldQueue;  
  
class SendWelcomeEmail implements ShouldQueue  
{
```

```
// Kod listener-a  
}
```

Livewire

Livewire je dodatak (library) za Laravel framework koji omogućava kreiranje dinamičkih, interaktivnih korisničkih interface-a koristeći samo PHP. Dizajniran je za jednostavnije povezivanje front-end i back-end logike, bez potrebe za dodatnom JavaScript radnom okolinom kao što je Vue.js ili React.

Kako funkcionira Livewire?

Livewire koristi AJAX zahtjeve kako bi ažurirao korisnički interface na temelju promjena na serveru. Evo kako izgleda osnovni radni tok s Livewireom:

Kreiranje komponente: Svaka Livewire komponenta se sastoji od PHP klase i Blade pogledu, koji zajedno omogućavaju server-side obradu logike (PHP) i renderiranje interface-a (Blade).

Reaktivnost: Kada korisnik izvrši neku radnju u interacije-u, kao što je unos teksta u formu ili klik na dugme, Livewire automatski šalje AJAX zahtjev na server. Tamo se ažurira stanje komponente i zatim se vraća novi HTML, koji se dinamički prikazuje na stranici.

Bez JavaScript-a: Sve interakcije, promjene i ažuriranja mogu se raditi isključivo s PHP-om i Bladeom. Livewire se brine o komunikaciji između servera i klijenta koristeći vlastiti JavaScript kod u pozadini.

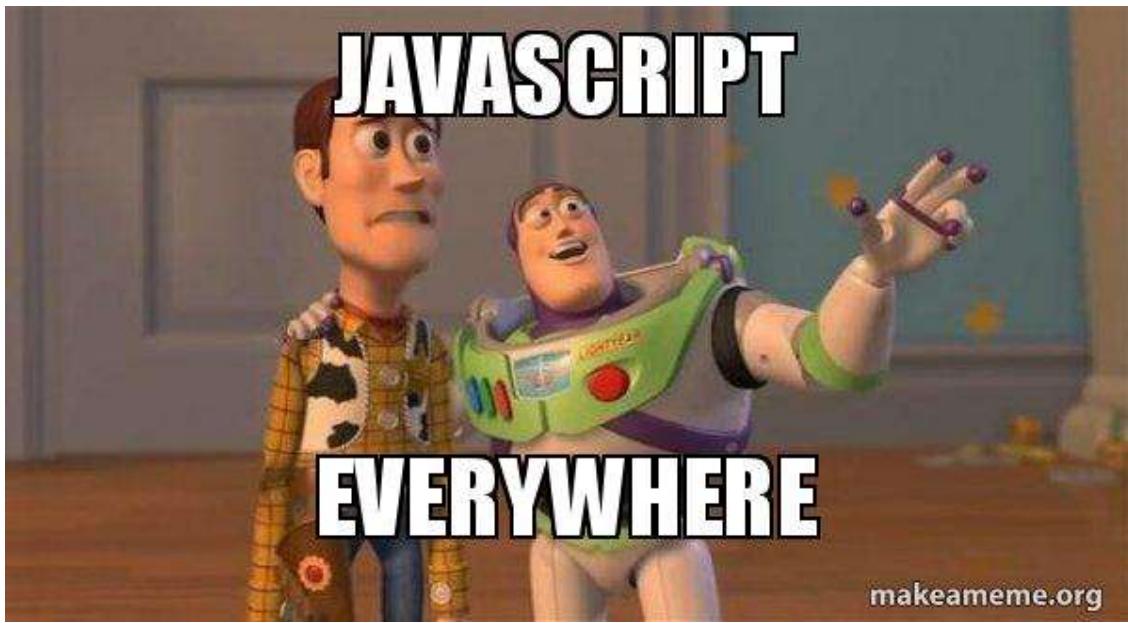
Bundling asset

U Laravelu, **bundling asset-a** podrazumijeva proces kombiniranja i optimiziranja statičkih datoteka kao što su JavaScript, CSS, slika, i fontova radi poboljšanja performansi web aplikacije. Laravel koristi Vite kao alat za bundling i upravljanje assetima.

Node.js

Da bi Vite radio, moramo imati instaliran **Node.js** na kompjuteru. Node.js je runtime environment. JavaScript se izvršava u browser-u, browser ima engine kojim interpretira JavaScript. PHP se interpretira na Apache serveru ili nekom drugom (npr. nginx, Caddy) i možemo ga instalirati gdje želimo za razliku od engina za JavaScript. Postoje više JavaScript engine-a koji se danas koriste: V8 od Google-a, SpiderMonkey od Mozille, JavaScriptCore od Apple-a, Chakra od Microsoft-a i Hermes od Meta. V8 i SpiderMonkey koriste Just-In-Time (JIT) kompajliranje, koje konvertira JavaScript u strojni kod za brzo izvršavanje. Najbrži je V8.

Instalacija samostalnog JavaScript engine-a, omogućava izvršavanje JavaScript koda izvan browsera, na serveru ili drugim okruženjima. U browserima, JavaScript enginei dolaze ugrađeni (V8 u Chromeu, SpiderMonkey u Firefoxu itd.), ali samostalna instalacija omogućava da JavaScript može raditi na serveru, što omogućava izgradnju backend aplikacija, API-ja i drugih server-side servisa korištenjem JavaScripta (primjerice s Node.js). Također omogućava razvojne alate, kao što su transpilatori (npr. Babel) ili bundle-alati (npr. Webpack ili Vite), gdje je potrebno obraditi JavaScript kôd izvan browsera.



Vite koristi Node.js okruženje za instalaciju i upravljanje JavaScript paketima, kao i za izvršavanje `npm` naredbi koje pokreću Vite-ov razvojni server i kreiraju optimizirane asete za produkciju.

Zašto je Node.js potreban?

Upravljanje paketima: Node.js dolazi s npm (Node Package Manager), koji je potreban za instalaciju svih JavaScript ovisnosti koje Vite koristi. Na primjer, kada pokrenete `npm install`, npm instalira sve potrebne biblioteke navedene u datoteci `package.json`.

Izvršavanje razvojnih naredbi: Node.js pokreće razvojne naredbe kao što su `npm run dev` i `npm run build`, koje pokreću Vite-ov server za razvoj i stvaraju produkcijske bundle.

Moderni bundling alati: Vite i mnogi drugi alati za bundling (kao što su Webpack i Rollup) temelje se na Node.js ekosistemu. Webpack i Rollup su JavaScript alati za bundling i optimizaciju assets-a (datoteka kao što su JavaScript, CSS, slike) koje aplikacija koristi. Njihova osnovna svrha je objediniti sve potrebne resurse u manji broj datoteka radi bolje performanse i organizacije. Webpack je bolji izbor za kompleksne projekte, posebno za velike aplikacije s mnogim vrstama datoteka i naprednim zahtjevima za optimizaciju. Rollup je optimalan za biblioteke i projekte koji trebaju manje konfiguracije, gdje je važna jednostavnost i čisti, optimizirani output.

Kako instalirati Node.js?

Skinite [Node.js](#) s službene stranice i instalirajte ga prema uputama za vaš operativni sistem.

Nakon instalacije, provjerite je li instaliran pokretanjem sljedeće naredbe u terminalu:

```
node -v  
npm -v
```

Ove naredbe trebale bi prikazati verzije Node.js i npm-a, što znači da su uspješno instalirani.

Nakon što imate [Node.js](#), možete instalirati Vite-ove ovisnosti s [npm install](#) i koristiti Vite u Laravelu za razvoj i produkciju asseta.

[Što je Vite?](#)

Vite je alat za moderni razvoj front-enda koji omogućava brzi razvoj i bundling, posebno dizajniran da optimizira vrijeme učitavanja tokom razvoja i produkcije. Laravel je integrirao Vite kako bi zamijenio Laravel Mix, olakšavajući rad s modernim JavaScript i CSS modulima.

Vite omogućava:

- **Brzo učitavanje u razvoju:** Koristi ESM (ECMAScript Modules), zbog čega se samo korišteni moduli učitavaju i bundlaju.
- **Brzi hot module replacement (HMR):** Promjene u datotekama reflektiraju se gotovo odmah, poboljšavajući brzinu razvoja.
- **Optimizirani bundling za produkciju:** Vite automatski minimizira datoteke i upravlja njima radi brzeg učitavanja stranica u produkciji.

[Kako Vite radi u Laravelu?](#)

Laravel dolazi s pred-konfiguriranim Vite postavkama, koje su definirane u datoteci [vite.config.js](#). Vite omogućava da se frontend datoteke poput CSS-a i JavaScript-a učitavaju i automatski ažuriraju.

Evo osnovnog toka rada za korištenje Vite-a u Laravelu:

Instalacija ovisnosti: Laravelova instalacija obično već uključuje potrebne Vite ovisnosti, ali ako želite ručno dodati Vite, možete instalirati potrebne pakete:

```
npm install
```

Pokretanje lokalnog servera za razvoj: Kada želite pokrenuti aplikaciju za razvoj s Vite-om, pokrenite naredbu:

```
npm run dev
```

Ova naredba pokreće Vite u razvojnim postavkama i omogućava hot-reloading (HMR). Vite se digne na <http://localhost:5173> a Laravel na <http://localhost>. Naravno moramo dići server sa [php artisan serve](#). Vite će injectati CSS i JavaScript u Laravel.

Korištenje asseta u Blade pogledu: Umjesto korištenja standardnih `<script>` ili `<link>` oznaka, koristite `@vite` direktivu u vašem Blade pogledu, kao u primjeru:

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>My Laravel App</title>
```

```
@vite(['resources/css/app.css', 'resources/js/app.js'])  
</head>  
<body>  
    <h1>Welcome to my Laravel App</h1>  
</body>  
</html>
```

Kompajliranje za produkciju: Kada je aplikacija spremna za produkciju, koristite naredbu:

```
npm run build
```

Ova naredba stvara optimizirane, minimizirane datoteke koje su spremne za produkciju.

Što radi vite.config.js?

Datoteka vite.config.js koristi se za konfiguriranje Vite-a. Laravel dolazi s osnovnim postavkama koje možete proširiti prema potrebi. U osnovi, ova konfiguracija definira putanje do asseta, integraciju s Laravelom i dodatne postavke za specifične projekte.

Uključivanje .css datoteke u layout.blade.php

Povezivanje s Vite

U `resources/css` napravimo apoteku `app.css`:

```
body{  
    background-color: #a1c8f0;  
    margin: 20px;  
}
```

Taj app.css mora biti uključen u `vite.config.js`. Ako želimo koristiti ovaj CSS u svome pogledu (Blade-u), onda u `resources/views/layout.blade.php` ćemo dodati ispred `</head>` taga:

```
@vite(['resources/css/app.css', 'resources/js/app.js'])
```

Ručno povezivanje .css datoteke

Ako nemamo instaliran Vite, možemo u `public` direktoriju otvoriti `css` direktorij i u njega staviti naš `app.css`. Pred kraj `</head>` taga trebamo upisati:

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

Ovo je način kako da to napravimo pješke, bez Vite.

Kada bi htjeli isto ručno ubaciti JavaScript, ubacili bi `public/js/app.js` i u njemu:

```
console.log('Hello from app.js');
```

Dобра пракса је да се JavaScript пoveže на крају `</body>` тага:

```
<script src="{{ asset('js/app.js') }}"></script>
```

Postoje три начина како се JavaScript учијава: стандардно учијавање `<script>` тагом у `<head>` секцији – броузер одмах учијава и изврши скрипту, учијавање с `defer` атрибутом (блокирајући начин) – омогућава да се JavaScript учијава у позадини док се HTML парсер наставља са својом обрадом. Скрипта ће се извршити тек након што је цијела HTML структура учијана, али ће задржати редослед извршења ако постоји више `defer` скрипти. Трећи начин је `async` атрибутом – он омогућава паралелно учијавање и извршење скрипте чим је преузета, без чекања на остale `async` скрипте или потпуно учијавање HTML-а. Ово убрзава учијавање странице јер се скрипте учијавају не зависно једна о другој, али редослед извршења није загарантiran.

Скрипту smo ставили на крај па нас nije брига што се заустави даљне парсирање.

У конзоли ћемо добити `Hello from app.js`.

Laravel Dusk

Laravel Dusk је алат за end-to-end (E2E) тестирање апликација, развијен унутар Laravel екосистема. Користи се за аутоматизирено тестирање корисниčког интерфејса (UI) како би се осигурало да све функционалности апликације раде исправно с корисниčким странама, симулирајући стварне интеракције корисника с веб страницом. Dusk омогућава тестирање понашања апликације кроз интеракцију с елементима као што су форме, линкови, дугмад, и други UI елементи, чиме помаже у детекцији бугова и осигурува стабилност апликације.

Glavna svojstva Laravel Duska

Симулација корисниčких интеракција: Омогућава симулацију корисниčких акција попут клика, уноса текста, навигације кроз странице и промјере приказа садрžaja.

Без потребе за екстеријарним browser driverima: Laravel Dusk користи `ChromeDriver` с уграденим headless начином рада, па нема потребе за руčним постављањем browser driver-a, што олакшава постављање и тестирање.

Једнотаван запис и reproducija testova: Омогућава снимање и поновно покрећање тестова, чиме се убрзава процес тестирања.

Pojednostavljenio kreiranje testova: Пружи једнотаван API који омогућава креирање тестова на читак и јасан начин користећи методе као што су `visit`, `click`, `type`, `assertSee`, и друге.

Подршка за тестирање аутентификације: Dusk омогућава тестирање апликација које захтјевaju аутентификацију корисника, што олакшава тестирање сценарија са пријавом корисника.

Kako koristiti Laravel Dusk?

Instalacija: Dusk се инсталира у Laravel пројект помоћу Composer-a.

```
composer require --dev laravel/dusk  
php artisan dusk:install
```

Kreiranje testova: Nakon instalacije, testovi se mogu kreirati pomoću Artisan komande:

```
php artisan dusk:make TestIme
```

Pisanje testa: Test se piše u novokreiranoj datoteci unutar direktorija `tests/Browser`, koristeći Dusk metode za definiranje koraka koje treba izvršiti i provjere koje treba obaviti.

```
public function testExample()
{
    $this->browse(function ($browser) {
        $browser->visit('/login')
            ->type('email', 'user@example.com')
            ->type('password', 'password')
            ->press('Login')
            ->assertPathIs('/home');

    });
}
```

Pokretanje testova: Nakon definiranja testova, oni se mogu pokrenuti naredbom:

```
php artisan dusk
```

Kada koristiti Laravel Dusk?

Laravel Dusk je posebno koristan za:

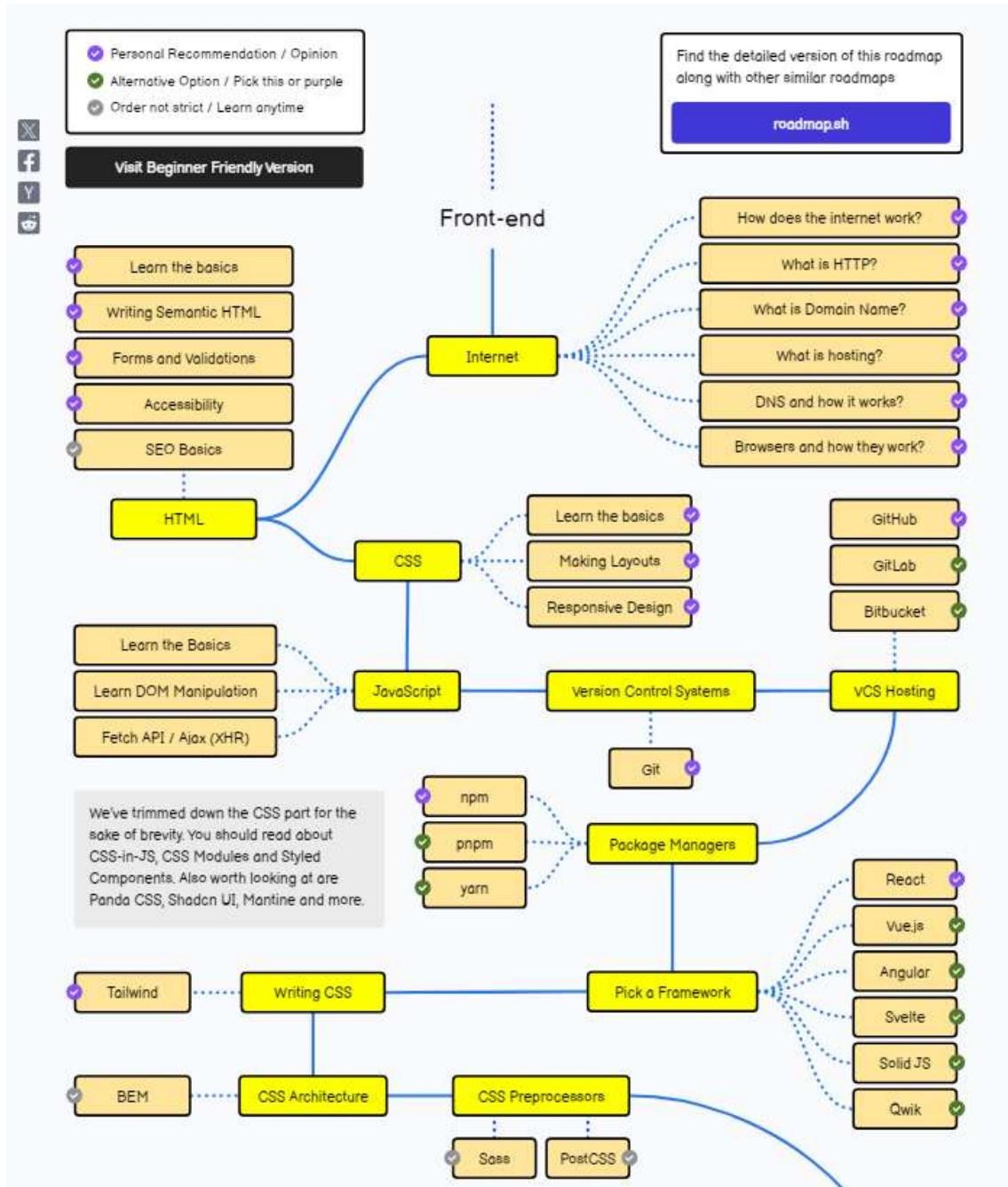
- Testiranje korisničkog interface-a i interakcija: Provjera ispravnog prikaza i rada elemenata interface-a.
- **Testiranje autentifikacije i autorizacije:** Uvjeravanje da sustav ispravno provodi prijavu i pristupne kontrole.
- **End-to-end testiranje kompletnih korisničkih scenarija:** Provjera da aplikacija funkcionira ispravno od početka do kraja korisničkog toka.

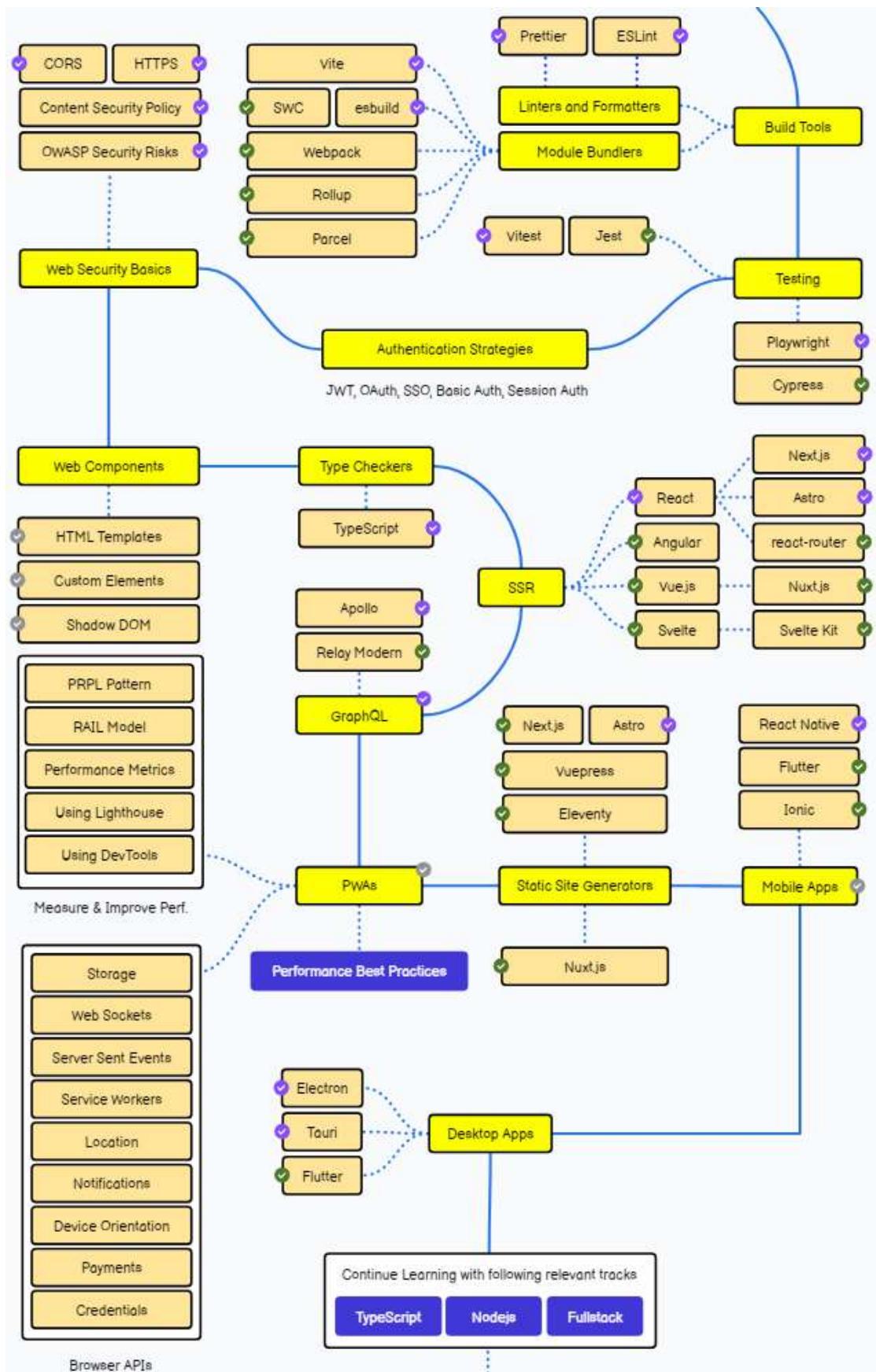
Laravel Dusk olakšava automatizaciju testiranja, što omogućava bržu identifikaciju problema u aplikaciji i osigurava dosljednost u funkcionalnostima kroz različite verzije i nadogradnje.

Ostalo

Ako želimo da aplikacija bude javno dostupna, potrebna nam je domena.

Frontend Developer





[roadmap.sh/frontend](#)

Kriptografija

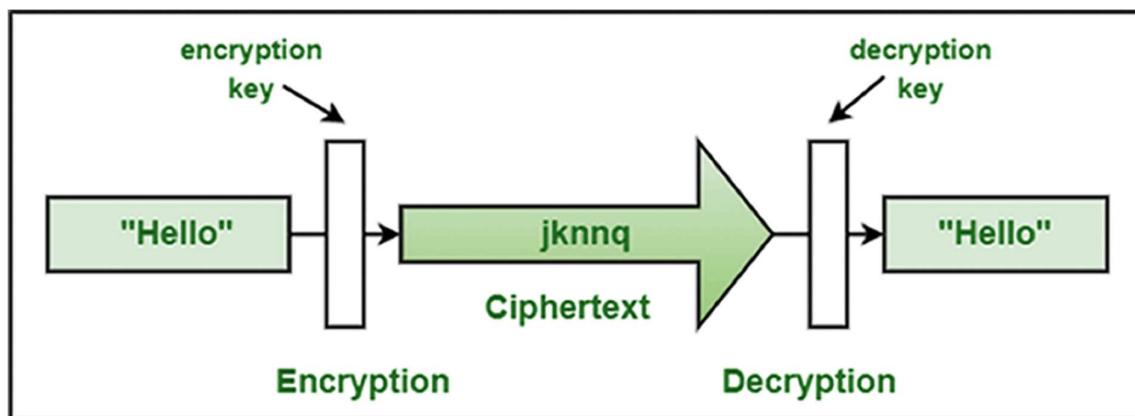
Nekada je kriptografija bila sinonim za enkripciju – konverziju informacija iz „normalnog“, čitljivog oblika, u nerazumljivi (kriptirani) niz znakova. Samo su sugovornici znali na koji način se informacija pretvara iz čitljivog u kriptirani oblik i obrnuto, time sprječavajući nepoželjne strane, javnost ili maliciozne entitete odgometnuti sadržaj tajne komunikacije.

Naravno, ta osnovna funkcionalnost ostala je i danas, ali su se stvari i značajno proširile – područje kriptografije se kontinuirano bavi razvojem i analizom protokola za njene brojne funkcionalnosti.

Moderna kriptografija danas obuhvaća poveći spektar disciplina kao što su matematika, računalne i komunikacijske znanosti, pa čak i elektrotehnika i fizika. Kriptografski sustavi danas koriste računski kompleksne algoritme i velike kriptografske ključeve koji su izuzetno teški za praktične pokušaje direktnih napada.

Čemu nam služi kriptografija? Vrlo konkretno, ona nam omogućuje:

- Povjerljivost
- Integritet
- Autentifikaciju
- Neporecivost



Simetrična i asimetrična kriptografija

Simetrična kriptografija koristi jedan ključ za kriptiranje i dekriptiranje podataka. Taj ključ je tajan i treba biti poznat samo stranama koje žele u tajnosti komunicirati. Zbog toga, ti sugovornici na neki se način prethodno moraju dogovoriti oko zajedničkog ključa.

Kako to izvesti pomoću nesigurnog kanala kao što je internet? To pitanje predstavlja jedan od značajnih problema simetrične kriptografije. Srećom, istraživači su našli rješenja - algoritme koji služe za efikasnu i sigurnu razmjenu (odnosno načine generiranja) tajnih ključeva.

Što se tiče algoritama za simetričnu enkripciju, dobro je znati koji se algoritmi više NE preporučuju jer se smatraju „razbijenima“: DES, 3DES, RC2, RC4. Preporučuje se izbjegavati sisteme koji koriste ove algoritme.

Algoritmi za simetričnu enkripciju koji se u ovome trenutku preporučuju su primjerice AES (varijanta Rijndael algoritma koju je podržala vlada SAD-a), te Serpent, TwoFish (nasljednik Blowfisha, oboje autora Brucea Schneiera) i Camellia. Laravel nativno podržava AES šifriranje pomoću paketa [openssl](#). Laravel nativno koristi AES (Advanced Encryption Standard) za kriptiranje, podržavajući ključne dužine od 128 i 256 bita. Laravelov Crypt facade koristi AES-256-CBC način šifriranja, što znači da nije potrebna dodatna konfiguracija. Serpent, TwoFish i Camellia algoritmi nisu direktno podržani u Laravelu niti su dio [openssl](#) ekstenzije, što znači da su potrebne dodatne PHP ekstenzije ili biblioteke. Biblioteka [phpseclib](#) omogućava podršku za Serpent i TwoFish, kao i mnoge druge simetrične algoritme. Ova biblioteka koristi PHP implementaciju i može se lako integrirati u Laravel. OpenSSL podržava Camellia algoritam, ali Laravel nativno ne omogućava rad s njim. Korištenje ovog algoritma može se postići kroz [openssl_encrypt](#) i [openssl_decrypt](#) funkcije.

Simetrična kriptografija vrlo je brza, govori se čak o performansama koje pod određenim uvjetima mogu biti i do 10.000 puta brže od asimetrične kriptografije. Također, zbog pozadinske matematičke priče, moguće su i hardverske implementacije za dodatnu efikasnost.

Asimetrična kriptografija nastoji riješiti neke probleme simetrične kriptografije. U asimetričnom sistemu svaki korisnik mora imati dva ključa – jedan javni, koji je dostupan svima na internetu, te drugi tajni, koji je poznat samo korisniku. Zanimljiva je sljedeća činjenica: ako se poruka kriptira javnim ključem, onda se može dekriptirati povezanim tajnim ključem, ali i obrnuto – poruka kriptirana tajnim ključem, može se dekriptirati povezanim javnim ključem. Moguće su obje opcije, i obje opcije imaju svoju važnu funkcionalnost.

Ako Sanja kriptira poruku Brankovim javnim ključem (koji Sanja lako nabavi na internetu), samo Branko može dekriptirati tu poruku svojim tajnim ključem – jer samo on ima taj tajni ključ. Dakle, imamo tajnost. S druge strane, ako Sanja kriptira svoju poruku svojim tajnim ključem i pošalje Branku, nemamo baš neku tajnost, jer svatko tko uhvati tu poruku, može je dekriptirati Sanjinim javnim ključem. Međutim, tko god dekriptira poruku Sanjinim javnim ključem, može biti siguran u jedno – da je tu poruku definitivno kriptirala Sanja, jer nitko drugi to nije mogao pod pretpostavkom da samo ona ima taj tajni ključ.

U tom slučaju imamo – autentifikaciju. Ovakav sistem ima brojne prednosti – jednostavnija distribucija ključeva, omogućuje se integritet, autentifikacija i neporecivost, ključevi se mogu jednostavno ukinuti ili ponovno generirati ako bi došlo do neke vrste kompromitacije, itd.

Međutim, asimetrična kriptografija ima jednu tešku manu – spora je. Također, ključevi su ogromni – npr. simetrični ključevi mogu biti veličine 256 bita (AES 256), a asimetrični 2048 i više bita (npr. RSA 2048, RSA 4096). Zbog toga se u brojnim scenarijima koji zahtijevaju prijenos većih količina podataka koristi hibridna kriptografija: prvo se asimetričnom kriptografijom uspostavlja konekcija i „razmjena“ simetričnog tajnog ključa; nakon toga se komunikacija odvija kriptiranjem poruka simetričnim ključevima.

Najpoznatiji algoritmi asimetrične kriptografije su RSA (Rivest–Shamir–Adleman), ElGamal te EC (Elliptic Curve). Glavni problem s RSA je taj što svoju sigurnost temelji na matematičkom problemu faktorizacije integera, za koji trenutno ne postoji efikasno rješenje (polynomial-time), ali – ovo je bitno – nije dokazano da takvo rješenje ne postoji „somewhere out there“! Što znači da, ako neki matematički genij ili sretnik pronađe rješenje za efikasnu faktorizaciju, RSA i svi sustavi kritično bazirani na tom algoritmu naći će se u velikim problemima. Zbog toga, ali i pitanja efikasnosti i kvantne kriptografije, sve se više odvija prijelaz na E.C. Laravel nativno ne održava direktno ove algoritme. Naravno, možemo ih koristiti kroz dodatne PHP biblioteke kao što su [phpseclib](#) ili OpenSSL. Za podršku

ElGamal ne znam u ovom trenutku. Eliptičke krivulje se često koriste za ECDSA (Elliptic Curve Digital Signature Algorithm) za digitalne potpise i ECDH (Elliptic Curve Diffie-Hellman) za razmjenu ključeva. Moguće je također koristiti phpseclib ili OpenSSL za ovo.

<https://www.vidilab.com/teme/softverska-tema/6156-criptografija>

WYSIWYG editori

[TinyMCE](#)

TinyMCE je popularan WYSIWYG (What You See Is What You Get) editor, koji omogućava korisnicima da pišu i formatiraju tekst unutar web aplikacija na način sličan editorima teksta kao što je Microsoft Word. TinyMCE pretvara standardno HTML `textarea` polje u bogat editor koji podržava napredne funkcije kao što je formatiranja teksta, umetanja slika, tablica, linkova, promjene fontova, te mnoge druge opcije.



Osnovna svojstva TinyMCE-a

Tekstualno formatiranje: Podržava osnovno oblikovanje teksta (podebljano, kurziv, podcrtano), promjenu boje, poravnanje, umetanje popisa, citata itd.

Multimedijiški sadržaj: Korisnici mogu umetati slike, videozapise i druge medijske datoteke izravno u tekst.

HTML uređivanje: Omogućava uređivanje sirovog HTML koda za napredne korisnike koji žele ručno prilagoditi sadržaj.

Plug-in sistem: TinyMCE ima bogat izbor dodataka (plug-ins) koji omogućuju proširenje funkcionalnosti, kao što su spellchecker, integracija s CMS-ovima (npr. WordPress), podrška za emotikone i još mnogo toga.

Responzivan dizajn: Editor se prilagođava različitim veličinama ekrana, pa se može koristiti na računalima, tabletima i mobilnim uređajima.

Integracija s web aplikacijama: Može se lako integrirati u različite web framework-e kao što je Laravel, React, Vue, Angular, itd.

Primjer upotrebe

U web aplikaciji, umjesto običnog `textarea` polja za unos teksta, pomoću TinyMCE-a korisnik dobiva napredni editor za uređivanje sadržaja blog postova, stranica, ili komentara, čime se olakšava rad i pruža bolje korisničko iskustvo.

TinyMCE je jedan od najčešće korištenih WYSIWYG editora zahvaljujući svojoj fleksibilnosti, prilagodljivosti i lakoći korištenja.

Konkurenca

TinyMCE ima nekoliko konkurentnih WYSIWYG editora koji nude slične funkcionalnosti za uređivanje teksta unutar web aplikacija. Evo nekih od najpopularnijih alternativa:

[CKEditor](#)

CKEditor je vrlo popularan WYSIWYG editor koji nudi napredne funkcionalnosti, uključujući bogato formatiranje teksta, umetanje



medijskih datoteka, podršku za prilagodljive plug-inove, te mogućnost uređivanja dokumenata kao što su Microsoft Word datoteke.

Prednosti:

Podrška za collaborative editing (istovremeno uređivanje više korisnika).

Velik izbor dodataka.

Izuzetno prilagodljiv s podrškom za različite platforme.

Ima integraciju s alatima kao što su Google Docs i Dropbox.

Ostali

Quill je moderan WYSIWYG editor otvorenog koda koji se fokusira na jednostavnost, responzivnost i brzinu. Ima lagano korisnički interface i lako se integrira u moderne web aplikacije.



Froala je premium WYSIWYG editor s naglaskom na brzinu, jednostavnost korištenja i bogate mogućnosti uređivanja. Podržava responzivan dizajn, uređivanje multimedije i različite integracije.

Summernote je lagani WYSIWYG editor s jednostavnim interface-om. Njegova glavna prednost je lakoća integracije i minimalni set funkcionalnosti, što ga čini prikladnim za projekte koji ne trebaju napredne opcije.

Slate.js je framework za kreiranje potpuno prilagodljivih rich text editora. Za razliku od TinyMCE-a i CKEditor-a, Slate.js je više alat za izradu editora nego gotov editor.

Trix je open-source WYSIWYG editor razvijen od strane Basecampa. Fokusira se na jednostavnost i osnovno uređivanje teksta. Iako nema toliko opcija kao TinyMCE ili CKEditor, idealan je za aplikacije koje traže minimalistički pristup.

Redactor je premium WYSIWYG editor koji je poznat po svojoj brzini i jednostavnom korisničkom sučelju. Usmjeren je prema pružanju čistog i lako prilagodljivog uređivačkog iskustva.

DataTables

DataTables je open-source jQuery plugin koji omogućava dodavanje interaktivnih i naprednih funkcionalnosti tablicama u HTML-u. Pomaže pri prikazu i radu s velikim količinama podataka na pregledan način, te omogućava korisnicima jednostavno pretraživanje, sortiranje, filtriranje i paginaciju podataka unutar HTML tablica. DataTables je vrlo popularan alat za rad s tablicama zbog jednostavne implementacije i fleksibilnosti.

10 unosa po stranici

Pretraživanje:

| Ime | Položaj | Ured | Napredak | Datum početka | Plaća |
|----------------|-------------------------------|---------------|--------------------------------------|---------------|---------------------|
| Airi Satou | Računovođa | Tokio | <div style="width: 100%;">100%</div> | 2008-11-28 | 162.700,00 dolara |
| Angelica Ramos | Glavni izvršni direktor (CEO) | London | <div style="width: 95%;">95%</div> | 2009-10-09 | 1.200.000,00 dolara |
| Ashton Cox | Mlađi tehnički autor | San Francisco | <div style="width: 80%;">80%</div> | 2009-01-12 | 86.000,00 dolara |
| Bradley Greer | Softverski inženjer | London | <div style="width: 70%;">70%</div> | 2012-10-13 | 132.000,00 dolara |

Ključne karakteristike DataTables-a

Pretraživanje: Omogućava dinamično pretraživanje unutar tablica.

Sortiranje: Stupci u tablici mogu biti sortirani klikom na zaglavlje stupca.

Paginacija: Automatsko dodavanje paginacije kako bi se olakšala navigacija kroz veće setove podataka.

Filtriranje: Omogućava korisnicima filtriranje podataka na temelju različitih kriterijusa.

Ajax integracija: Može se koristiti za dohvaćanje podataka putem Ajax zahtjeva, čineći ga idealnim za prikazivanje velikih datasetova bez učitavanja cijele stranice. Ajax (Asynchronous JavaScript and XML) je tehnika u web programiranju koja omogućava aplikacijama da komuniciraju s serverom u pozadini, asinkrono, bez potrebe za ponovnim učitavanjem cijele web stranice. Korištenjem Ajaxa, podaci se mogu slati i primati s servera u realnom vremenu, a da se korisničko iskustvo ne prekida.

Responsive: Tablice se automatski prilagođavaju različitim veličinama ekrana (mobilni, tableti, računari).

Podrška za ekstenzije: DataTables se može proširiti s dodatnim funkcionalnostima putem ekstenzija kao što su:

Buttons: Za dodavanje dugmadi za izvoz u Excel, CSV, PDF, printanje, itd.

Editor: Za kreiranje, uređivanje i brisanje redaka direktno iz tablice.

Responsive: Pomaže da tablice budu prilagodljive za različite uređaje.

Kako radi?

DataTables transformira standardne HTML tablice u interaktivne i funkcionalne komponente putem JavaScript-a. Njegova osnovna struktura koristi jQuery za manipulaciju DOM-om (Document Object Model) kako bi dodaо svojstva kao što je paginacija, pretraživanje i sortiranje.

Povezivanje DataTables i Laravel-a

Prvo instalirajte paket za Laravel DataTables koristeći Composer. Ovaj paket olakšava integraciju s DataTables-om, uključujući server-side obradu podataka.

```
composer require yajra/laravel-datatables-oracle
```

Od verzije 5.5, Laravel automatski registrira pružatelje usluga pa nije potrebno dodati Service Provider u vaš `config/app.php` datoteku

Ako želimo koristiti facade, možemo ga registrirati u `config/app.php`:

```
'DataTables' => Yajra\DataTables\Facades\DataTable::class,
```

Zatim objavimo konfiguracijske datoteke (opcionalno):

```
php artisan vendor:publish --tag=datatables
```

Sljedeći korak je kreiranje rute koja će obrađivati zahtjeve za dohvaćanje podataka. Na primjer, kreirat ćemo rutu u `web.php`:

```
Route::get('/posts/datatables', [PostController::class, 'getPosts'])->name('posts.datatables');
```

U vašem kontroleru, možete koristiti `DataTables` facade kako biste omogućili server-side obradu podataka. Na primjer, u `PostController`-u, dodajte metodu `getPosts`:

```
namespace App\Http\Controllers;

use App\Models\Post;
use Yajra\DataTables\Facades\DataTable;

class PostController extends Controller
{
    public function getPosts()
    {
        $posts = Post::select(['id', 'title', 'created_at', 'updated_at']);

        return DataTables::of($posts)
            ->addColumn('action', function ($post) {
                return 'Edit';
            })
            ->make(true);
    }
}
```

Ovdje dohvaćamo podatke iz modela Post. Koristimo metodu `DataTables::of()` za server-side obradu podataka. Dodajemo kolonu "action" koja omogućava uređivanje posta.

U vašem Blade pogledu, dodajte HTML tablicu i JavaScript kod za inicijalizaciju DataTables-a. Na primjer, u datoteci `resources/views/posts/index.blade.php`:

```

@extends('layout')

@section('content')


# Posts



| ID | Title | Created At | Updated At | Action |
|----|-------|------------|------------|--------|
|----|-------|------------|------------|--------|



<script
src="https://cdn.datatables.net/1.10.21/js/jquery.dataTables.min.js"></script>
<script>
$(document).ready(function() {
    $('#posts-table').DataTable({
        processing: true,
        serverSide: true,
        ajax: '{{ route('posts.datatables') }}',
        columns: [
            { data: 'id', name: 'id' },
            { data: 'title', name: 'title' },
            { data: 'created_at', name: 'created_at' },
            { data: 'updated_at', name: 'updated_at' },
            { data: 'action', name: 'action', orderable: false, searchable: false }
        ]
    });
});
</script>
@endsection

```

Definiramo tablicu s HTML oznakom `<table>`, a DataTables automatski upravlja sadržajem putem Ajax poziva. Inicijaliziramo DataTables koristeći jQuery. Koristimo server-side način rada, gdje podaci dolaze putem Ajax zahtjeva prema ruti `/posts/datatables`.

Sada, kada otvorite stranicu koja koristi ovaj pogled, trebali biste vidjeti interaktivnu tablicu s mogućnošću sortiranja, paginacije i pretraživanja, a podaci će dolaziti dinamički putem Ajaxa.

Korištenje CSS-a u Laravel-u

Predavač umjesto Tailwind-a preporuča korištenje [Bootstrap-a](#). Kaže da je ovo najbrži način za doći do rezultata s CSS-om, ali naravno ne i najbolji. Dovoljno ga je uključiti s linkom a on nudi različite komponente koje su naravno bolje od golog HTML. Admin interface (backoffice) ne moraju imati neki specijalni dizajn ali ovo pomaže da na nešto liči. Treba primjetiti da pod published-om redovi imaju detalje kada su napravili a to nam ne odgovara pa ćemo napisati:

```
<td>{{ $post->created_at->format('d.m.Y. H:i') }}</td>
```

Podatak `created_at` je carbon podatak. Carbon je moćan paket koji radi s datumima i vremenima. Mogli smo napisati i na ovaj način:

```
<td>{{ $post->created_at->diffForHumans() }}</td>
```

Onda bi nam ispisivao prije koliko tjedana je post napravljen, naravno na engleskom.

Što je Webpack?

Webpack je alat za bundling JavaScript datoteka (i drugih resursa kao što je CSS, slike, fontovi) u jednu ili više optimiziranih datoteka za web aplikacije. U osnovi, pomaže u organizaciji i optimizaciji resursa za brže učitavanje stranica. U Laravelu, Webpack se koristi putem alata [Laravel Mix](#), koji olakšava rad s njim, skrivanjem kompleksnosti iza jednostavne sintakse.

Što je Sass i a što SCSS?

Sass (Syntactically Awesome Stylesheets) je preprocesor za CSS, što znači da proširuje CSS s dodatnim funkcionalnostima. Dvije su osnovne sintakse koje Sass podržava:

- **Sass** sintaksa je starija i koristi uvlake i praznine, bez vitičastih zagrada `{}` i točaka sa zarezom `,`;
- **SCSS (Sassy CSS)** sintaksa je novija i kompatibilna s običnim CSS-om. Koristi zagrade i točke sa zarezom, pa ako znate pisati CSS, SCSS je odmah razumljiv.

I Sass i SCSS se kompajliraju u standardni CSS koji browseri podržavaju.

Sass (Syntactically Awesome Stylesheets) je preprocesor za CSS koji omogućava pisanje CSS-a na napredniji način. Nudi svojstva kao što su variable, ugnjeđivanja pravila, miješanja (mixins), nasleđivanja i funkcija, što olakšava upravljanje velikim CSS datotekama i njihovu ponovnu upotrebu. Kod napisan u Sass-u se kompajlira u standardni CSS, koji browseri mogu interpretirati.

Što se tiče Larabela, Sass se ne veže direktno s njim kao neki ugrađeni dio frameworka, ali se Sass često koristi u Laravel projektima za stiliziranje korisničkog interface-a. Laravel dolazi s [Laravel Mix](#)-om, alatom temeljenim na Webpacku, koji olakšava postavljanje i kompilaciju Sass (ili SCSS) datoteka. Na taj način možete integrirati Sass u Laravel projekt jednostavno kroz Mix konfiguraciju.

Kako koristiti Bootstrap s Laravelom, a kako Sass?

Bootstrap je popularan frontend framework koji olakšava stvaranje responzivnih, dobro stiliziranih web stranica. Najčešće se koristi za raspored elemenata na stranici i osnovne stilove.

Evo kako koristiti oba alata s Laravelom:

1. **Bootstrap:** U Laravel Mix možete uključiti Bootstrap i njegov stil dodajući ga u `webpack.mix.js`:

```
mix.js('resources/js/app.js', 'public/js')
    .sass('resources/sass/app.scss', 'public/css');
```

Ovdje `app.scss` može uključivati Bootstrap:

```
@import '~bootstrap/scss/bootstrap';
```

Drugi način je da u kreiramo `index.html` u root (u slučaju Laravela to stavljamo u `resources/views/layout.blade.php`) i uključimo:

```
<meta name="viewport" content="width=device-width, initial-scale=1">
```

Time će ispravno regirati na mobilnim uređajima. Treba još uključiti Bootstrapov CSS i JS, tako da postavimo `<link>` tag u `<head>` (iza `<title>` taga) za naš CSS i `<script>` tag u naš Javascript.

```
<!doctype html>
<html lang="en">
    <head>
        <meta charset="utf-8">
        <meta name="viewport" content="width=device-width, initial-scale=1">
        <title>Bootstrap demo</title>
        <link
            href="https://cdn.jsdelivr.net/npm/bootstrap@5.2.3/dist/css/bootstrap.min.css"
            rel="stylesheet"
            integrity="sha384-rbsA2VBKQhgwxH7pPCaAq046Mgn0M80zW1RWuH61DGLwZJEdK2Kadq2F9CUG65"
            crossorigin="anonymous">
    </head>
    <body>
        <h1>Zdravo, svijete!</h1>
        <script
            src="https://cdn.jsdelivr.net/npm/bootstrap@5.2.3/dist/js/bootstrap.bundle.min.js"
            integrity="sha384-kenU1KFdBIE4zVF0s0G1M5b4hcpxyD9F7jL+jjXkk+Q2h455rYXK/7HAuoJl+0I4"
            crossorigin="anonymous"></script>
    </body>
```

Kod Laravela obično ne trebamo `<script>` tag.

2. **Sass:** Sass možemo koristiti za stiliziranje prilagođenih dijelova aplikacije. Bootstrap se često koristi za grid sistem i osnovne stilove formi, dok ostatak CSS-a prilagođavaš koristeći Sass, što ti omogućava naprednije i jednostavnije stiliziranje.

Primjer korištenja Sass-a s Laravel Mix-om u `webpack.mix.js` datoteci:

```
mix.sass('resources/sass/app.scss', 'public/css');
```

Ova konfiguracija omogućava da Sass datoteke iz `resources/sass/` direktorija budu kompajlirane u standardni CSS i smještene u `public/css/` direktorij, spremne za upotrebu u aplikaciji.

Dakle, iako Sass nije specifičan dio Laravel-a, može se jednostavno koristiti u Laravel projektima uz pomoć Laravel Mix-a.

Što je grid?

Grid je raspored elemenata na stranici u stupce i redove. Bootstrap dolazi s grid sistemom koji vam omogućava lako postavljanje elemenata u različite stupce, bez potrebe za ručnim definiranjem pozicija. To je posebno korisno za izradu responzivnih web stranica koje se dobro prikazuju na različitim veličinama ekrana.

Primjer jednostavnog Bootstrap grida:

```
<div class="container">
  <div class="row">
    <div class="col-md-6">Lijevi stupac</div>
    <div class="col-md-6">Desni stupac</div>
  </div>
</div>
```

Za što se koristi Bootstrap, a za što Sass?

- **Bootstrap** nam olakšava stvaranje mreže elemenata (grid), stiliziranje formi, dugmadi, navigacija i drugih osnovnih komponenti.
- **Sass/SCSS** koristiš za stilizaciju koja je specifična za tvoj projekt, omogućavajući nam korištenje varijabli, funkcija i miješanja kako bi CSS bio modularniji i održiviji.

Primjer upotrebe Bootstrap-a za grid i forme, dok ostatak stilova pišemo u Sass-u, ima smisla jer Bootstrap nudi osnovne funkcionalnosti (kao što je responzivnost⁸), dok prilagođene stilove izrađujemo u Sass-u.

Postman

Postman je popularan alat za testiranje API-ja, a u kombinaciji s Laravelom može biti vrlo koristan za testiranje vaših RESTful ili GraphQL API endpointa bez potrebe za frontendom. Postman omogućava

⁸ Responzivnost u web dizajnu odnosi se na sposobnost web stranice da se prilagodi različitim veličinama ekrana i uređajima. To znači da će web stranica izgledati i funkcionirati dobro na različitim uređajima, uključujući desktop računala, tablete i mobilne telefone, bez potrebe za zumiranjem ili horizontalnim pomicanjem.

Responzivni dizajn koristi CSS media queries, fleksibilne mreže i slike kako bi osigurao da se elementi na stranici automatski prilagođavaju ovisno o veličini prozora browser-a. Cilj je pružiti optimalno korisničko iskustvo na bilo kojem uređaju.

slanje HTTP zahtjeva (**GET**, **POST**, **PUT**, **DELETE**) prema vašem Laravel API-ju i prikazuje odgovore u jednostavnom interface-u.

Postman i Laravel: Osnovne stvari

Ne trebate ništa instalirati u Laravelu posebno za Postman. Postman radi s HTTP zahtjevima i Laravel automatski prihvata te zahtjeve putem svojih route-a. Sve što vam treba je da imate Postman aplikaciju na svom kompjuteru.

Instalacija Postmana:

- Preuzmite i instalirajte Postman s njihove službene web stranice.
- Postman je dostupan na Windows, macOS i Linux.

Kako koristiti Postman s Laravelom

1. Pokrenite vaš Laravel server

Prije nego što testirate svoj Laravel API, trebate pokrenuti lokalni server koristeći Artisan:

```
php artisan serve
```

To će pokrenuti vaš Laravel projekt na lokalnom hostu, npr. <http://127.0.0.1:8000>.

2. Postavljanje Postman zahtjeva

Postman omogućuje slanje HTTP zahtjeva prema vašem Laravelu. Evo koraka kako to koristiti:

- Otvorite Postman.
- Upišite URL vašeg API-ja. Na primjer, ako imate rutu u Laravelu za prikaz svih postova:

```
GET http://127.0.0.1:8000/api/posts
```

- Odaberite **HTTP metodu** s lijeve strane (**GET**, **POST**, **PUT**, **DELETE**, itd.).

Ako trebate poslati **POST** zahtjev (npr. za stvaranje posta), odaberite **POST** i postavite polja koja trebate (obično unutar **Body** sekcije).

Odaberite **raw** i JSON kao tip podataka u Body sekciji, a zatim unesite JSON podatke:

```
{
    "title": "Novi post",
    "content": "Ovo je sadržaj posta"
}
```

- Kliknite **Send** i vidjet ćete odgovor vašeg API-ja.

3. Postavljanje zaglavlja (headers)

Laravel zahtijeva specifična zaglavla za određene zahtjeve. Na primjer:

- Za JSON zahtjeve često ćete morati postaviti zaglavlje:

```
Content-Type: application/json
```

- Ako radite s autorizacijom (npr. JWT token), trebate poslati i Authorization zaglavlje:

```
Authorization: Bearer <vaš_token>
```

4. Korištenje API-a s autentifikacijom

Ako vaš Laravel API koristi autentifikaciju, kao što je **JWT (JSON Web Token)** ili **Laravel Sanctum**, morat ćete koristiti autentifikacijske tokene za slanje autoriziranih zahtjeva.

Evo kako to možete učiniti u Postmanu:

- Prvo napravite zahtjev prema API endpointu za prijavu, npr.:

```
POST http://127.0.0.1:8000/api/login
```

Pošaljite potrebne podatke (kao što su email i lozinka) putem **Body** sekcije.

Ako je autentifikacija uspješna, Laravel će vratiti **token**.

U sljedećim zahtjevima, koristite **Authorization** zaglavlje i dodajte token:

```
Authorization: Bearer <vaš_token>
```

5. Testiranje validacije podataka

Kada radite s Postmanom, lako možete testirati različite scenarije, kao što su neispravni podaci ili obavezna polja u koja nisu uneseni podaci. Na primjer, ako imate formu za kreiranje posta i ne pošaljete naslov, Laravel će vratiti grešku koju možete pregledati u Postmanu.

6. Pohrana zahtjeva i kreiranje kolekcija

Postman vam omogućuje da kreirate **kolekcije zahtjeva** koje možete pohraniti i ponovo koristiti.

Možete postaviti okolinu (environment) s različitim varijablama, kao što je baza URL-a (<http://127.0.0.1:8000>), tokena, ili API ključeva, i dinamički ih koristiti unutar svojih zahtjeva.

Povezivanje Laravel API-ja i Postmana

Laravel Sanctum / Passport: Ako radite s API autentifikacijom, možete koristiti Laravel Sanctum ili Passport kako bi omogućili korisnicima da autentificiraju svoje zahtjeve koristeći token. Postman može biti koristan alat za testiranje tih autentifikacijskih procesa.

API rute: Definirajte svoje rute unutar [routes/api.php](#) i testirajte ih direktno putem Postmana.

Na primjer, definiranje rute u [api.php](#):

```
Route::get('posts', [PostController::class, 'index']);  
Route::post('posts', [PostController::class, 'store']);
```

Tada te rute (definirane kao callable tip, gdje se koristi kao pokazivač na metodu index unutar [PostController](#) klase) možete testirati u Postmanu.

Cloudflare

Cloudflare je mrežna firma koja nudi niz usluga za optimizaciju performansi, sigurnost i zaštitu web stranica i aplikacija. Njihove usluge mogu značajno poboljšati performanse i sigurnost Laravel aplikacija, iako Cloudflare nije specifično vezan za Laravel, već za bilo koje web tehnologije i platforme koje koriste HTTP/HTTPS protokole.

Glavne usluge koje Cloudflare nudi

CDN (Content Delivery Network)

Cloudflare koristi globalnu mrežu poslužitelja kako bi statički sadržaj (npr. slike, CSS, JS datoteke) učinio dostupnim iz servera koji su bliži korisnicima.

Poboljšava performanse Laravel aplikacija tako što smanjuje kašnjenja i ubrzava učitavanje stranica.

DDoS zaštita

Cloudflare štiti aplikacije od DDoS napada (napada distribuiranom uskraćivanju usluge), koji mogu preopteretiti server velikim brojem lažnih zahtjeva.

Ovo je korisno za Laravel aplikacije koje mogu biti meta takvih napada, jer štiti njihovu dostupnost.

SSL certifikati i HTTPS

Cloudflare nudi besplatne SSL certifikate putem Full SSL (End-to-End Encryption) opcije.

Možete postaviti SSL certifikat na Cloudflare i prisiliti da vaša Laravel aplikacija koristi HTTPS bez potrebe da sami upravljate SSL certifikatima na svom serveru.

WAF (Web Application Firewall)

Cloudflare nudi Web Application Firewall koji može otkriti i blokirati štetne zahtjeve prema vašoj aplikaciji, štiteći Laravel od naprednih prijetnji, SQL injekcija, XSS napada i drugih.

WAF pravila mogu se prilagoditi za specifične potrebe aplikacije.

Cache i optimizacija sadržaja

Cloudflare kešira statički sadržaj i koristi minifikaciju (smanjivanje veličine CSS, JavaScript i HTML datoteka) za poboljšanje brzine učitavanja aplikacije.

Laravel aplikacije mogu imati koristi od bržeg vremena učitavanja uz pravilno postavljanje keširanja.

Boot Management

Cloudflare može pomoći u identifikaciji i blokiranju neželjenog bot prometa koji može usporiti ili napasti Laravel aplikaciju.

Ova usluga može smanjiti opterećenje na vašem serveru filtriranjem zlonamjernih zahtjeva.

DNS usluge

Cloudflare nudi brze i sigurne DNS usluge, što omogućava brže i sigurnije povezivanje korisnika s vašom Laravel aplikacijom.

Time se smanjuje vrijeme odgovora vašeg DNS-a i poboljšava ukupna dostupnost aplikacije.

Argo Smart Routing

Ova tehnologija koristi Cloudflare-ovu mrežu kako bi optimizirala putovanje podataka između korisnika i servera, smanjujući kašnjenje.

Laravel aplikacije s globalnim korisnicima mogu značajno ubrzati dostavu sadržaja zahvaljujući pametnom rutiranju podataka.

Jenkins

Jenkins je popularni open-source alat za kontinuiranu integraciju (CI) i kontinuiranu isporuku (CD) koji pomaže timovima u automatizaciji procesa izgradnje, testiranja i postavljanja softverskih projekata. Koristi se kako bi se osiguralo da promjene koda, kad se pošalju u repozitorij, prođu kroz automatizirane procese i testiranja, čime se smanjuje mogućnost pogrešaka i povećava brzina razvoja.

Jenkins se može koristiti s Laravelom za automatizaciju CI/CD (kontinuirane integracije i isporuke) procesa, čime se omogućuje brži i pouzdaniji razvoj i postavljanje Laravel aplikacija. Korištenjem Jenkinsa s Laravelom, možete automatizirati razne zadatke kao što su preuzimanja koda iz repozitorija, pokretanja testova, generiranja buildova i automatskog postavljanja aplikacije na server.

Evo kako Jenkins može biti koristan za Laravel projekte:

Automatizacija izgradnje (build)

Jenkins može automatski povući promjene iz Git repozitorija nakon svakog "push-a" ili kada se otvori "pull request". Nakon preuzimanja koda, Jenkins može pokrenuti Laravelove build korake, uključujući kompajliranje CSS i JavaScript datoteka koristeći alat kao što je **Laravel Mix** (koji koristi Webpack).

Koraci mogu uključivati instalaciju Composer paketa, instalaciju NPM paketa i pokretanje build procesa.

Automatizirano testiranje

Jenkins može pokretati PHPUnit testove unutar Laravel projekta kako bi osigurao da sav kod radi ispravno prije nego što se postavi na produkciju.

Laravel podržava i Feature i Unit testove, koje Jenkins može pokretati automatski nakon svakog commit-a. U slučaju neuspješnih testova, Jenkins može poslati upozorenje putem e-pošte ili integracija sa sistemima kao što je Slack-a.

Automatizacija migracija i seedanja baze

Kada Jenkins izvrši proces izgradnje i testiranja, možete postaviti zadatke koji pokreću Laravel migracije kako bi ažurirali bazu podataka te pokrenuti seedere ako je potrebno.

Postavljanje aplikacije (Deployment)

Jenkins može automatski postaviti Laravel aplikaciju na razvojne ili produkcijske servere (npr. DigitalOcean, AWS, Heroku) nakon što testovi uspješno prođu.

Ova automatizacija omogućuje brži i sigurniji tok isporuke aplikacije.

Integracija s Dockerom

Laravel se često koristi unutar Docker kontejnera, a Jenkins nudi odličnu podršku za Docker. Jenkins može automatizirati izgradnju Docker slike i pokretanje kontejnera, što je korisno za lokalno testiranje i produkcijsko okruženje.

Konfiguriranje Jenkins Pipelines za Laravel

Jenkins Pipelines omogućuju definiranje kompletnih CI/CD cjevovoda pomoću skripte (Jenkinsfile), gdje možete definirati Laravel specifične korake kao što su:

- Povlačenje repozitorija
- Instalacija Composer ovisnosti
- Pokretanje testova (PHPUnit, Dusk za end-to-end testove)
- Build procesa (Laravel Mix)
- Deployment na server

Osnovni koraci za konfiguraciju Jenkins pipeline-a za Laravel

Instalirajte Jenkins i kreirajte novi posao (Job) za Laravel projekt.

Konfigurirajte izvor koda – povežite Jenkins s Git repozitorijem u kojem se nalazi Laravel kod.

Postavite build skriptu koja uključuje:

Instaliranje Composer ovisnosti (`composer install`)

Instaliranje NPM paketa (`npm install`)

Pokretanje Laravel testova (`php artisan test`)

Pokretanje Laravel migracija (`php artisan migrate`)

Postavljanje automatskog deploymenta na server nakon uspješnog builda i testiranja.

Korištenje Jenkinsa s Laravelom može značajno ubrzati i poboljšati kvalitetu isporuke vaše aplikacije, minimizirajući ljudske greške i osiguravajući da se aplikacija postavlja u konzistentnom stanju.

[Firebase](#)

Firebase je platforma koju je razvio Google i pruža širok spektar backend servisa i alata za izradu web i mobilnih aplikacija. Namijenjena je programerima koji žele brzu integraciju rješenja za autentifikaciju korisnika, pohranu podataka, obavijesti u stvarnom vremenu, analitiku i još mnogo toga. Nudi gotove alate za razvoj, testiranje, distribuciju i održavanje aplikacija bez potrebe za vlastitim serverima.

Ključne funkcionalnosti Firebase-a

[Firebase Authentication](#)

Omogućava jednostavno dodavanje prijave/autentifikacije korisnika u aplikaciju pomoću različitih metoda (e-mail, Google, Facebook, itd.).

[Firebase Realtime Database](#)

Omogućava pohranu podataka u stvarnom vremenu, gdje se podaci sinkroniziraju između korisnika u stvarnom vremenu, idealno za aplikacije kao što je chat aplikacija.

[Cloud Firestore](#)

Napredna NoSQL baza podataka koja podržava fleksibilno i skalabilno pohranjivanje podataka i sinkronizaciju podataka u stvarnom vremenu.

[Firebase Hosting](#)

Pruža hosting za statičke web stranice i aplikacije, uključujući automatsku SSL enkripciju.

[Firebase Cloud Messaging \(FCM\)](#)

Alat za slanje obavijesti (push notifications) na web ili mobilne aplikacije.

[Firebase Analytics](#)

Pruža detaljnu analitiku i podatke o ponašanju korisnika, što je korisno za optimizaciju aplikacija.

Firebase Cloud Functions

Serverless funkcije koje omogućuju izvršavanje koda kao odgovor na događaje, bez potrebe za upravljanjem vlastitim serverima.

Firebase i Laravel

Laravel i Firebase mogu surađivati na način da iskoriste svoje prednosti kako bi stvorili robusne aplikacije koje kombiniraju **snagu Laravelovog backenda i Firebase-ovih servisa u oblaku**.

1. Firebase Authentication s Laravelom

Možete koristiti **Firebase Authentication** kao servis za prijavu korisnika (npr. kroz Google račun), a Laravel za vođenje backenda. Firebase će upravljati autentifikacijom korisnika, a Laravel će koristiti te podatke kako bi omogućio pristup resursima, API-ima i kontrolu korisničkih prava.

2. Firebase Realtime Database ili Firestore kao zamjena za tradicionalne baze podataka:

Laravel aplikacija može koristiti **Firebase Realtime Database** ili **Cloud Firestore** kao bazu podataka umjesto tradicionalnih relationalnih baza podataka (npr. MySQL). Ovo je osobito korisno za aplikacije koje trebaju sinkronizaciju u stvarnom vremenu (npr. chat aplikacije ili kolaborativne alate).

Laravelova fleksibilna struktura omogućava jednostavnu integraciju s Firestore-om putem dostupnih SDK-ova ili REST API-ja.

3. Firebase Cloud Messaging (FCM) za obavijesti

Ako koristite Laravel za backend, možete koristiti **Firebase Cloud Messaging (FCM)** za slanje obavijesti korisnicima vaših mobilnih aplikacija (Android/iOS) ili web aplikacija. Laravel može poslati zahtjev FCM-u da obavijesti korisnike o novom sadržaju, porukama ili bilo kojem događaju unutar aplikacije.

4. Firebase Hosting za frontend aplikacije:

Laravel može upravljati API backendom, dok se **Firebase Hosting** može koristiti za hosting frontend aplikacija (npr. Vue.js ili React frontend). To omogućava da se frontend aplikacija brzo postavlja na Firebase-u, dok Laravel brine o svim poslovnim logikama i API-ima.

5. Firebase Analytics i Laravel:

Ako vaša aplikacija ima komponentu praćenja korisnika i njihovog ponašanja, možete koristiti Firebase Analytics za praćenje događaja u aplikaciji, dok Laravel može dodatno obrađivati podatke ili integrirati s vašim poslovnim sistemima.

6. Serverless funkcije s Cloud Functions:

Firebase Cloud Functions mogu se koristiti zajedno s Laravelom za izvršavanje specifičnih zadataka bez potrebe za održavanjem vlastitih servera. Na primjer, Laravel može pokrenuti Cloud Function koja šalje obavijest ili obrađuje podatke kada se događaj dogodi.

Digital Ocean usluge

DigitalOcean je popularan pružatelj cloud usluga koji nudi infrastrukturu i platformu za hosting web aplikacija, uključujući Laravel i PHP aplikacije. Evo što DigitalOcean nudi u kontekstu Laravel i PHP aplikacija:

1. Droplets (Virtualni serveri)

Droplets su virtualni privatni serveri na kojima možete instalirati PHP, Laravel, i druge potrebne komponente kako biste hostali svoju aplikaciju. DigitalOcean nudi nekoliko unaprijed konfiguiriranih slika operativnih sistema koje su spremne za instalaciju PHP-a i Laravel-a.

Ključne karakteristike:

- Brza implementacija virtualnih servera s različitim opcijama hardvera (CPU, RAM, pohrana).
- Podrška za popularne Linux distribucije kao što su Ubuntu-a, CentOS-a i Debian-a.
- Dodatne mogućnosti kao što su automatsko sigurnosno kopiranje (backups) i snapshoti.
- Fleksibilne opcije skaliranja resursa.

2. One-Click Laravel Deployment

DigitalOcean nudi **one-click** instalaciju za **Laravel**. Ova opcija omogućava brzu postavku Laravel aplikacije na Dropletu, bez potrebe za ručnim konfiguriranjem servera.

Što uključuje:

- Laravel okolinu s unaprijed instaliranim PHP-om, MySQL/MariaDB-om, Apache/Nginx-om, Composerom, i potrebnim PHP ekstenzijama.
- Automatsko podešavanje sigurnosnih mjera kao što su firewalls i SSL certifikati (uz Let's Encrypt).
- Brza implementacija, idealna za razvojne timove koji žele brzo započeti s radom na Laravel aplikaciji.

3. Managed Databases

DigitalOcean nudi **Managed Databases** za PostgreSQL, MySQL i Redis, što olakšava upravljanje bazama podataka potrebnim za Laravel aplikacije. Umjesto da ručno konfigurirate i upravljate bazom podataka, Managed Databases nude automatske sigurnosne kopije, skaliranje i jednostavnu integraciju s aplikacijom.

Ključne karakteristike:

- Automatska sigurnosna kopiranja i povrat podataka.
- Integracija sa DigitalOcean-ovim Dropletima putem privatne mreže.
- Jednostavno skaliranje baze podataka.

4. App Platform

DigitalOcean App Platform je platforma kao usluga (PaaS) koja omogućava jednostavno hostanje, izgradnju i skaliranje aplikacija bez potrebe za upravljanjem serverima. To je idealna opcija za Laravel aplikacije jer pojednostavljuje proces postavljanja aplikacije.

Ključne karakteristike:

- Automatsko izgrađivanje i deploy aplikacija direktno iz vašeg GitHub ili GitLab repozitorija.
- Podrška za horizontalno i vertikalno skaliranje bez intervencije u infrastrukturu.
- Jednostavno dodavanje drugih servisa kao što su baze podataka ili Redis-a.
- SSL certifikati i firewall-ovi automatski uključeni.

5. Block Storage i Spaces

Block Storage nudi skalabilno rješenje za pohranu podataka, što može biti korisno za Laravel aplikacije koje trebaju dodatni prostor za pohranu datoteka.

Spaces je servis za Object Storage, sličan Amazon S3-u, koji omogućava pohranu statičkog sadržaja kao što su slike, videi i druge datoteke koje Laravel aplikacije često koriste.

Ključne karakteristike:

- Jednostavno skaliranje kapaciteta pohrane.
- Podrška za hostanje statičkih datoteka i CDN za bržu isporuku sadržaja.

6. Automatski Backupi i Snapshoti

DigitalOcean omogućava automatsko sigurnosno kopiranje vaših Dropleta. To je ključno za Laravel aplikacije koje često mijenjaju podatke i trebaju sigurnosne kopije u slučaju gubitka podataka.

Ključne karakteristike:

- Jednostavan proces povrata podataka iz backupa.
- Snapshot opcija omogućava brzo vraćanje Dropleta u prethodno stanje.

7. Monitoring i Alati za Sigurnost

DigitalOcean nudi alate za praćenje performansi servera, uključujući CPU, RAM, pohranu, i mrežne statistike.

Ugrađeni **Firewall-ovi** omogućuju postavljanje sigurnosnih pravila za kontrolu pristupa serveru.

Ključne karakteristike:

- Puna kontrola nad sigurnosnim postavkama.
- Alarmi i notifikacije za korištenje resursa.

8. Dokumentacija i Podrška

DigitalOcean nudi opsežnu dokumentaciju za postavljanje i konfiguriranje Laravel aplikacija, uključujući vodiče korak-po-korak za početnike.

Aktivna zajednica koja pruža podršku i dijeli rješenja.

DigitalOcean je pouzdan pružatelj cloud usluga koji omogućuje jednostavno i skalabilno hostanje **Laravel** i PHP aplikacija. Sa svojim One-Click instalacijama, Managed Databases, i App Platform-om, olakšava proces postavljanja i upravljanja aplikacija bez previše administrativnih komplikacija. Idealna je platforma za developere koji traže fleksibilnost, pouzdanost i lako upravljanje infrastrukturom.

[WebAssembly \(Wasm\)](#)

WebAssembly (skraćeno Wasm) je binarni format koji omogućava izvođenje programskog koda u web browser-ima gotovo nativnom brzinom. Razvijen je kako bi se prevazišla ograničenja JavaScript-a u smislu performansi i efikasnosti, posebno za aplikacije koje zahtijevaju visoko intenzivne računalne resurse, kao što su igre, uređivanje videa, ili simulacije.

Ključne karakteristike WebAssembly-ja

Binarni format: WebAssembly koristi binarni format koji je kompaktan, efikasan za mrežni prijenos i brz za parsiranje.

Visoke performanse: Kod napisan u programskim jezicima kao što su C, C++, Rust ili Go može se kompajlirati u WebAssembly i izvoditi u web browser-ima blizu nativnih brzina, daleko definitivno brže nego što je to moguće sa JavaScript-om.

Portabilnost: WebAssembly kod može raditi na različitim platformama bez modifikacija, jer je dizajniran da bude portabilan.

Sigurnost: Radi u sigurnom okruženju, sandboxu, unutar web preglednika, što znači da ima isti sigurnosni model kao i JavaScript, ograničavajući pristup osjetljivim resursima sistema.

Interoperabilnost s JavaScript-om: WebAssembly može lako komunicirati s JavaScript-om, omogućujući razvoj aplikacija koje kombiniraju performanse WebAssembly-ja i fleksibilnost JavaScript-a.

Kako se koristi

Programski kod, napisan na jezicima kao što su C, C++, Rust, kompajlira se u WebAssembly modul. Taj modul se zatim može učitati i pozivati iz JavaScript-a unutar web aplikacije.

Prednosti WebAssembly-ja

Poboljšane performanse: Pogodan za aplikacije kao što su igare, editori slike i videa, matematičke simulacije, i druge aplikacije koje zahtijevaju puno računalnih resursa.

Manja potrošnja memorije: Zbog svoje kompaktne veličine i načina na koji upravlja memorijom.

Višejezičnost: Podržava različite jezike, za razliku od JavaScript-a koji je jedini izvorno podržani jezik za web browser-e.

WebAssembly je ključan korak prema omogućavanju složenih aplikacija u web browserima, otvarajući vrata za razvoj visokoperformantnih aplikacija na webu.

Programiranje WebAssembly-ja

Ako želite programirati aplikacije koje se kompajliraju u WebAssembly modul, ali tražite nešto što se relativno lako uči, Rust i AssemblyScript su izvrsne opcije. Ove tehnologije nude dobar balans između jednostavnosti i performansi. Evo nekoliko jezika koji se često koriste za stvaranje WebAssembly modula:

Rust

Rust je sve popularniji jezik zbog svog naglaska na sigurnost i performanse, a lako se kompajlira u WebAssembly.

Zašto Rust?

- Nudi visoke performanse kao C ili C++, ali bez problema s memorijskom sigurnošću.
- Ima bogatu zajednicu i dokumentaciju.
- Alati za rad s WebAssembly-jem, kao što je **wasm-bindgen** i **cargo-web**, olakšavaju integraciju Rust-a u web projekte.

Koliko je težak za učenje?

- Rust ima strmu krivulju učenja zbog upravljanja vlasništvom memorije, ali je vrlo robustan kada se jednom savlada. Osnove su razumljive, i Rust pruža odlične alate za rad s WebAssembly-jem.

AssemblyScript

AssemblyScript je superset JavaScript-a, sličan TypeScript-u, koji je lak za učenje ako već poznajete JavaScript. Omogućava ti pisanje TypeScript-like koda koji se prevodi u WebAssembly.

Zašto AssemblyScript?

Ako već znate JavaScript ili TypeScript, lako ćete se prilagoditi.

AssemblyScript vam omogućava da koristite poznat jezik i jednostavno stvarate WebAssembly module bez potrebe za učenjem novih paradigmatičnih koncepta.

Iako ne postoji centralizirana kompanija koja posjeduje AssemblyScript, OpenJS Foundation podržava razvoj ovog projekta. OpenJS Foundation je organizacija koja podržava različite projekte u vezi s JavaScript ekosistemom, uključujući popularne alate kao što su Node.js i jQuery.

AssemblyScript je dizajniran da se koristi s WebAssembly-em i omogući jednostavan način za JavaScript/TypeScript programere da iskoriste prednosti WebAssembly-ja.

Koliko je težak za učenje?

Vrlo jednostavan za početak ako već radite s JavaScript-om ili TypeScript-om, ali ima ograničenja u odnosu na jače tipizirane jezike kao što je Rust.

C / C++

C i C++ su tradicionalno najbrži jezici i često se koriste za stvaranje visoko performatnih aplikacija. Mogu se lako kompajlirati u WebAssembly pomoću **Emscripten** toolchain-a.

Zašto C/C++?

- Imaju dugu povijest i veliki ekosistem, uz maksimalne performanse.
- Idealni su za intenzivne aplikacije kao što su igare i obrada videa.

Koliko su teški za učenje?

- C i C++ su snažni, ali teški za početnike zbog složenih koncepata, upravljanja memorijom i manje sigurnosti u odnosu na jezike kao što je Rust.

Laravel Forge

Laravel Forge je alat za upravljanje serverima namijenjen programerima koji koriste Laravel framework. Omogućava brzo i jednostavno postavljanje, konfiguriranje i upravljanje PHP aplikacijama na različitim serverima. Svrha Forge-a je da automatizira mnoge zadatke upravljanja serverima kako bi programeri mogli usmjeriti više vremena na razvoj aplikacija, a manje na server administraciju.

Ovaj paket nije besplatan i dodatno se instalira. Postoje 3 različita paketa s različitim cijenama.

Ključne funkcionalnosti Laravel Forge-a

Podešavanje servera: Forge vam omogućava jednostavno postavljanje i konfiguraciju servera na cloud platformama kao što su:

- **DigitalOcean**
- **Linode**
- **AWS (Amazon Web Services)**
- **Vultr**
- **Hetzner**
- i mnoge druge.

SSL certifikati: Laravel Forge automatizira izdavanje i postavljanje **SSL certifikata** putem **Let's Encrypt**. Na taj način osigurava HTTPS vezu za vaše aplikacije.

Automatsko postavljanje aplikacija: Omogućava jednostavno povezivanje s vašim Git rezervorijima (GitHub, Bitbucket, GitLab), nakon čega možete postaviti aplikacije na servere jednim klikom.

Upravljanje bazama podataka: Forge omogućava instalaciju i upravljanje različitim bazama podataka, kao što su MySQL, PostgreSQL i Redis. Također olakšava kreiranje korisnika i baza podataka.

Upravljanje pozadinskim procesima: Omogućava jednostavno pokretanje i upravljanje queue workerima i pozadinskim procesima kako bi vaša aplikacija funkcionirala neometano.

Automatsko ažuriranje sigurnosnih zakrpa: Forge može automatski instalirati sigurnosne zakrpe za server, osiguravajući sigurnost vašeg sistema.

Nadzor i obavijesti: Ugrađena funkcionalnost nadzora servera omogućava da pratite performanse vašeg servera i aplikacija. Možete postaviti obavijesti o statusu servera, crash reportovima i slično.

Cron Jobs: Laravel Forge omogućava jednostavno postavljanje cron jobs zadataka za vašu aplikaciju, što je korisno za automatske poslove koji se trebaju obavljati u određenim vremenskim intervalima (npr. svakodnevne ili tjedne provjere, automatske email obavijesti itd.).

Zero Downtime Deployment: Korištenjem Forge-a možete osigurati **zero downtime deployment**, što znači da će vaša aplikacija ostati dostupna korisnicima i za vrijeme procesa postavljanja nove verzije.

Kako Laravel Forge funkcionira

Prvi korak je povezivanje vašeg Forge računa s cloud providerom (npr. DigitalOcean). Forge će za vas automatski instalirati softver potreban za rad PHP aplikacija (npr. Nginx, MySQL, PHP, itd.).

Drugi korak je kloniranje Git rezervorija koji sadrži vašu aplikaciju te postavljanje istog na server.

Treći korak je upravljanje vašim aplikacijama, uključujući SSL certifikate, pozadinske procese i još mnogo toga putem jednostavnog web interfejsa-a.

Prednosti Laravel Forge-a

Ušteda vremena: Automatizacija procesa postavljanja i upravljanja serverom smanjuje vrijeme potrebno za ručne konfiguracije.

Jednostavnost korištenja: Nije potrebno imati dubinsko znanje o administraciji servera kako biste upravljali serverima za vaše Laravel aplikacije.

Sigurnost: Redovno održavanje servera, automatska instalacija sigurnosnih zakrpa i jednostavno postavljanje SSL certifikata doprinose sigurnosti aplikacija.

Skalabilnost: Omogućava jednostavno dodavanje novih servera i aplikacija kako vaša aplikacija raste.

Laravel Forge je idealno rješenje za one koji žele brzo postaviti i upravljati serverima za PHP aplikacije, bez gubljenja vremena na komplikirane konfiguracije.

Podman i Docker

Podman i Docker su dva popularna alata za upravljanje kontejnerima, ali se razlikuju u arhitekturi, sigurnosti i načinu na koji rade. Podman je potpuno besplatan i open-source. Docker Desktop je besplatan za osobnu upotrebu i za male firme. Plaćaju se Docker Hub, Docker Pro, Team i Business planovi. Evo usporedbe Docker-a i Podman-a po ključnim karakteristikama:

Arhitektura

Docker koristi klijentsko-server model. Docker **daemon** je središnji proces koji pokreće i upravlja kontejnerima. Docker daemon zahtijeva pokretanje s privilegijama superkorisnika (root), što može predstavljati sigurnosni rizik.

Podman ne koristi daemon. To je alat bez servera (daemonless), što znači da svaki kontejner radi izravno s korisnikovim procesima. Podman ne zahtijeva pokretanje s root privilegijama, omogućujući **rootless** kontejnere, što ga čini sigurnijim izborom.

Sigurnost

Docker zahtijeva da se **Docker daemon pokreće kao root**, što otvara potencijalne sigurnosne ranjivosti. Ako dođe do problema s Dockerom, napadači mogu potencijalno dobiti root pristup cijelom sistemu.

Podman omogućava **pokretanje kontejnera bez root privilegija (tzv. rootless containers)**, što smanjuje mogućnost sigurnosnih prijetnji. Time se smanjuje napadačka površina, jer korisnici pokreću kontejnere pod svojim korisničkim računima, bez direktnog pristupa superkorisniku.

Kompatibilnost

Docker je stariji alat i široko prihvaćen u industriji. Ima veći ekosistem, više alata, dodataka i podrške, te je standard u mnogim firmama.

Podman je kompatibilan s Docker CLI-jem, što znači da se **Docker image-i i Docker file-ovi** mogu koristiti i s Podmanom. Možete zamjeniti `docker` s `podman` u mnogim slučajevima, bez promjena u sintaksi ili komandama.

Upravljanje kontejnerima i image-ima

Docker koristi komande kao što su `docker run`, `docker ps`, `docker build`, dok sve to radi unutar jednog Docker daemon-a.

Podman koristi gotovo iste komande (`podman run`, `podman ps`, `podman build`), ali radi bez daemon procesa, što znači da svaki kontejner funkcioniра kao zaseban proces.

Orkestracija

Docker koristi **Docker Swarm** za orkestraciju, no Kubernetes je postao industrijski standard za orkestraciju kontejnera. Docker sada podržava Kubernetes putem **Docker Desktop-a**.

Podman ima ugrađenu podršku za **Kubernetes**, omogućavajući jednostavnu integraciju s njim. Podman može kreirati YAML datoteke koje su kompatibilne s Kubernetesom, a također koristi alat **podman-compose** kao alternativu za **docker-compose**.

Systemd integracija

Docker nije izravno integriran s **systemd**, alatom za upravljanje servisima u Linux sistemima. Docker se pokreće kao daemon, a za integraciju s systemd-om potreban je dodatni rad.

Podman se lako integrira s **systemd**. Svaki kontejner može se automatski generirati i pokretati kao **systemd servis**, što olakšava upravljanje kontejnerima na Linux serverima.

Rootless vs Root

Docker ima root pristup po zadanim postavkama, što ga čini potencijalno nesigurnim u određenim okruženjima. Docker nudi mogućnost pokretanja rootless kontejnera, ali ta funkcionalnost nije bila prvotno dizajnirana.

Podman je osmišljen od početka da bude **rootless**. Svi korisnici mogu upravljati kontejnerima bez superkorisničkih privilegija, što je korisno u produkcijskim okruženjima gdje je sigurnost ključna.

Ekosistem

Docker ima veliki ekosistem, uključujući **Docker Hub** za spremanje i dijeljenje image-a, **Docker Compose** za upravljanje višekontejnerskim aplikacijama i **Docker Desktop** za lokalni razvoj.

Podman ima slične funkcionalnosti, ali podrška za neke alate nije tako široko rasprostranjena kao za Docker. Podman koristi **Buildah** za izgradnju image-a i **Skopeo** za manipulaciju kontejnerskim image-ima.

Zaključak:

Docker je standardni alat za kontejnerizaciju, s velikim ekosistemom i podrškom, ali s nekim sigurnosnim nedostacima zbog potrebe za root pristupom.

Podman nudi sličnu funkcionalnost, ali bez daemona i s boljom sigurnosnom arhitekturom, omogućujući rootless kontejnere. Idealan je za okruženja u kojima je sigurnost ključna, a posebno za Linux sisteme.

Oba alata imaju svoje prednosti, a izbor između njih ovisi o specifičnim potrebama vaše aplikacije i infrastrukture.

Pojam kontejnerizacije

Kontejnerizacija je proces pakiranja aplikacije i njenih zavisnosti (npr. biblioteke, konfiguracijske datoteke, runtime okruženja) u jedan **kontejner**. Kontejner je lagano, izolirano okruženje koje sadrži sve što je potrebno da aplikacija radi neovisno o sistemu na kojem se pokreće.

Glavna prednost kontejnerizacije je:

- **Portabilnost:** Kontejneri se mogu pokretati na različitim platformama (razvojna mašina, serveri, cloud) bez potrebe za promjenama u kodu ili podešavanjima.
- **Izolacija:** Svaka aplikacija u kontejneru je izolirana od drugih aplikacija i od osnovnog sistema, što omogućava bolju sigurnost i upravljaljivost.
- **Efikasnost:** Kontejneri dijele kernel operativnog sistema, pa troše manje resursa od virtualnih mašina.

Najpoznatiji alati za kontejnerizaciju su **Docker** i **Podman**. Docker je popularizirao kontejnerizaciju u IT industriji.

Pojam orkestracije

Orkestracija je proces automatizacije upravljanja kontejnerima u produkciji. Kada imate više kontejnera koji čine jednu aplikaciju (npr. web server, baza podataka, servis za autentifikaciju), morate osigurati da svi kontejneri pravilno komuniciraju, skaliraju se prema potrebi i budu otporni na kvarove.

Orkestracijski alati pomažu u tome upravljujući kontejnerima automatski, a najpoznatiji alat za orkestraciju je **Kubernetes**. Orkestracija omogućava:

- **Automatsko skaliranje:** Dodavanje ili uklanjanje kontejnera prema trenutnom opterećenju aplikacije.
- **Otkrivanje i komunikacija između servisa:** Kontejneri međusobno komuniciraju bez potrebe za ručnim konfiguriranjem.
- **Load balancing:** Distribuira promet između različitih instanci aplikacije.
- **Opstanak u slučaju greške:** Ako jedan kontejner prestane raditi, orkestracijski alat može automatski pokrenuti novi.

Agilno upravljanje projektima – najbolje prakse i metodologije

Faze upravljanja projektom

Bez obzira na opseg, svaki projekt treba slijediti slijed radnji koje treba kontrolirati i upravljati. Karakterističan **proces upravljanja projektima** uključuje sljedeće faze:

1. Inicijacija
2. Planiranje
3. Izvršenje
4. Praćenje performansi
5. Projekt zatvoren

Korištene kao putokaz za postizanje specifičnih zadataka, ove faze definiraju životni ciklus upravljanja projektom.

Ipak, ova struktura je previše općenita. Projekt obično ima niz unutarnjih faza unutar svake faze. Mogu se tako razlikovati ovisno o opsegu posla, timu, industriji i samom projektu.

U pokušaju pronalaženja univerzalnog pristupa upravljanju bilo kojim projektom, stručnjaci su razvili brojne PM (engl. project management) tehnike i metodologije.

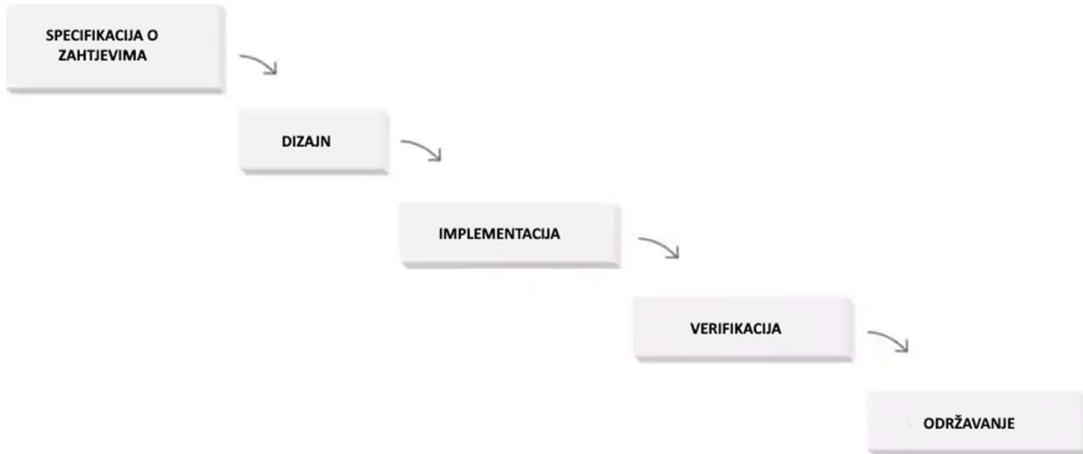
Tradicionalna metodologija upravljanja projektima

Na temelju gore opisane klasične redne okoline (engl. framework), tradicionalne metodologije imaju korak po korak pristup izvršenju projekta. Dakle, projekt prolazi kroz inicijaciju, planiranje, izvođenje i praćenje sve do zatvaranja u uzastopnim fazama.

Ovaj pristup koji se često naziva **linearnim**, uključuje brojne unutrašnje faze koje su sekvenčne i izvršavaju se kronološkim redom. Primjenjeno najčešće u građevinskoj ili proizvodnoj industriji, gdje su potrebne male ili nikakve promjene u svakoj fazi, tradicionalno upravljanje projektima također je našlo svoju primjenu u softverskom inženjerstvu.

Poznat kao **model vodopada**, to je dominantna metodologija razvoja softvera od ranih 1970-ih, kada ju je službeno opisao Winston W. Royce : "Postoje dva bitna koraka zajednička svim razvojima računalnih programa, bez obzira na veličinu ili složenost. Prvo je korak analize, zatim korak kodiranja... Ova vrsta vrlo jednostavnog koncepta implementacije zapravo je sve što je potrebno ako je napor dovoljno mali i ako konačnim proizvodom trebaju upravljati oni koji su ga izradili – kao što se obično radi s računalnim programima za internu upotrebu."

Model vodopada



Model vodopada ima snažan naglasak na planiranju i razvoju specifikacija, što oduzima do **40 posto vremena i proračuna projekta**. Drugo osnovno načelo ovog pristupa je strogi redoslijed faza projekta. Nova faza projekta ne počinje dok se prethodna ne završi.

Metoda dobro funkcioniра за jasno definirane projekte s jednom isporukom i fiksnim rokom. Vodopad pristup zahtijeva temeljito planiranje, opsežnu projektnu dokumentaciju i strogu kontrolu nad procesom razvoja. U teoriji, to bi trebalo dovesti do isporuke na vrijeme, prema proračunu, niskih projektnih rizika i predvidljivih konačnih rezultata.

Međutim, kada se primjeni na stvarni proces softverskog inženjeringu, metoda vodopada obično je spora, skupa i nefleksibilna zbog brojnih ograničenja. U mnogim slučajevima, njegova nesposobnost da prilagodi proizvod zahtjevima tržišta koji se razvijaju često rezultira velikim rasipanjem resursa i konačnim neuspjehom projekta.

Što je agilno upravljanje projektima

Povijest Agilea može se pratiti unatrag do 1957. godine : U to su vrijeme Bernie Dimsdale, John von Neumann, Herb Jacobs i Gerald Weinberg koristili tehnike inkrementalnog razvoja (koje su sada poznate kao Agile), izrađujući softver za IBM i Motorolu. Ne znajući kako klasificirati pristup koji su prakticirali, jasno su shvatili da se na mnogo načina razlikuje od vodopada.

Suvremenii Agile službeno je predstavljen 2001. godine kada se grupa od 17 stručnjaka za razvoj softvera sastala kako bi razgovarala o alternativnim metodologijama upravljanja projektima. Imajući jasnu viziju fleksibilnog, laganog i timski orientiranog pristupa razvoju softvera, mapirali su to u [Manifestu za agilni razvoj softvera](#).

Usmjeren na "otkrivanje boljih načina razvoja softvera," Manifest jasno specificira Agile temelje:

Strana 341 od 353

“ Kroz ovaj rad, došli smo do vrijednosti:
pojedinaca i interakcija preko procesa i alata
Radnog softvera preko sveobuhvatne dokumentacije
Suradnje korisnika preko pregovora o ugovoru
Odgovor na promjene preko slijedećenja plana.”

Dopunjena s [dvanaest principa agilnog softvera](#), filozofija je postala univerzalan i učinkovit novi način upravljanja projektima.

Agilni pristup i proces

Agilne metodologije imaju iterativni pristup razvoju softvera. Za razliku od jednostavnog linearног modela vodopada, agilni projekti sastoje se od niza manjih ciklusa. Svaki od njih je projekt u malom: sastoji se od faza dizajna, implementacije, testiranja i implementacije unutar unaprijed definiranog opsega posla.

Na kraju svakog ciklusa isporučuje se proizvod koji se potencijalno može isporučiti. Stoga se sa svakom iteracijom proizvodu dodaju nove značajke, što rezultira postupnim rastom projekta. Sa značajkama koje su potvrđene u ranoj fazi razvoja, šanse za isporuku potencijalno neuspjelog proizvoda znatno su manje.

Agilni koraci razvoja softvera

End-to-end Agile tok upravljanja projektima sastoji se od pet različitih faza koje uglavnom odgovaraju općim fazama upravljanja projektima koje smo gore spomenuli.

Zamišljanje ili faza inicijacije. Prva faza unutar agilne metodologije upravljanja projektima odnosi se na prepoznavanje potreba krajnjih kupaca, postavljanje poslovnih ciljeva i ocrtavanje željenih rezultata. Voditelj projekta identificira prave dionike i dodjeljuje uloge cijelom timu.

Faza nagađanja ili planiranja. Faza ima dva glavna cilja: rastavljanje projekta na prekretnice i postavljanje vremenskih rokova. Da biste postigli prvi cilj, potrebno vam je barem opće razumijevanje [funkcionalnih zahtjeva projekta](#). Agile tim daje prioritet značajkama i procjenjuje koliko će dugo trajati njihov razvoj. Faza rezultira izradom plana izvršenja koji će se, za razliku od scenarija vodopada, dodatno prilagođavati promjenama.

Faza istraživanja. Uključuje istraživanje različitih načina za rješavanje [zahtjeva projekta](#) dok ostaje unutar vremenskih i proračunskih ograničenja. Nakon što se odluči za najbolju opciju, tim dodaje dio korisničkih priča u plan ponavljanja i nastavlja s njihovim razvojem i testiranjem. Faza istraživanja ide paralelno s četvrtom fazom prilagodbe budući da tim razmatra povratne informacije korisnika i uči iz prethodnog iskustva.

Faza prilagođavanja. Ova faza je jedinstvena za Agile razvoj softvera. Omogućuje timu pregled rezultata prethodnih iteracija, procjenu postojeće situacije, prikupljanje povratnih informacija korisnika i provjeru izvedbe u odnosu na plan izvršenja. Zatim možete prilagoditi svoje planove i pristupe u skladu s tim, uvodeći sve potrebne promjene i nove zahtjeve ako postoje.

Faza zatvaranja. U završnoj fazi, tim osigurava da je projekt dovršen i da ispunjava sve ažurirane zahtjeve. Najbolja praksa ovdje je rasprava o pogreškama koje su se dogodile tokom projekta i područjima za poboljšanja kako bi se donosile bolje odluke u budućnosti.

Agile najbolje prakse

Sažmimo najbolje prakse kojih se Agile drži.

Fleksibilnost: Opseg rada može se mijenjati u skladu s novim zahtjevima.

Raščlanjivanje rada: Projekt se sastoji od malih ciklusa.

Vrijednost timskog rada: Članovi tima blisko surađuju i imaju jasnu viziju svojih odgovornosti.

Iterativna poboljšanja: postoji česta ponovna procjena obavljenog rada unutar ciklusa kako bi se konačni proizvod učinio boljim.

Suradnja s klijentom: Kupac je usko uključen u razvoj i može promijeniti zahtjeve ili prihvati prijedloge tima.

Agilni okviri i metodologije upravljanja projektima

Agilnost je krovni pojam za veliki izbor metodologija i tehnika koje dijeli gore opisana načela i vrijednosti. Svaki od njih ima svoja područja uporabe i karakteristične značajke. Najpopularniji okviri i prakse su Scrum, Kanban, Hybrid, Lean, Bimodal, XP i Crystal. Prije nego što o njima detaljnije raspravljamo, proučimo njihove ključne značajke.

Scrum

Scrum je okvir za upravljanje projektima i razvojem softvera koji se temelji na principima Agilnog pristupa. Scrum omogućava timovima da rade na složenim projektima u kratkim, iterativnim ciklusima, zvanim sprintovima, obično u trajanju od dva do četiri tjedna. Cilj je često isporučivati funkcionalne dijelove proizvoda i kontinuirano poboljšavati proces rada.

Ključni elementi Scruma

Timovi

- **Scrum tim** se sastoji od **Product Ownera, Scrum Mastera i Razvojnog tima**.
- **Product Owner** upravlja prioritetima projekta i definira što treba biti izgrađeno.
- **Scrum Master** je facilitator, osigurava da se Scrum procesi pravilno provode i uklanja prepreke s kojima se tim suočava.
- **Razvojni tim** je zadužen za stvaranje funkcionalnog proizvoda ili njegovog dijela.

Sprint

Sprint je vremenski okvir (najčešće 2-4 tjedna) u kojem tim treba dovršiti određeni skup zadataka (user stories).

Na kraju svakog sprinta tim prezentira funkcionalan dio proizvoda koji može biti potencijalno isporučiv.

Scrum događaji

- Sprint planning: Planiranje sprinta gdje tim i Product Owner dogovaraju koje zadatke će obraditi u tom sprintu.
- Daily Scrum: Kratki, dnevni sastanak (obično 15 minuta) na kojem svaki član tima izvještava o napretku, preprekama i planovima za taj dan.
- Sprint review: Pregled rezultata sprinta gdje se pokazuje ono što je dovršeno i diskutuju potencijalne promjene.
- Sprint retrospective: Sastanak nakon završetka sprinta za analizu što je dobro funkcionalo, a što treba poboljšati u narednim sprintovima.

Backlog (neizvršen rad, neobavljen posao, radni ostatak)

- **Product backlog:** Lista svih funkcionalnosti koje treba razviti za proizvod, prioritizirana od strane Product Ownera.
- **Sprint backlog:** Skup zadataka koji su izabrani za razvoj tokom jednog sprinta.

Prednosti Scruma

- **Fleksibilnost:** Scrum omogućava prilagodbe tijekom razvoja projekta na temelju povratnih informacija.
- **Transparentnost:** Kroz redovite sastanke i recenzije sprintova, svi članovi tima su u toku s napretkom.
- **Brza isporuka:** Fokus na male, ali funkcionalne dijelove omogućava da se novi dijelovi proizvoda isporučuju brže.

Scrum je idealan za projekte gdje se zahtjevi mogu mijenjati, a timovi trebaju brzo reagirati na promjene ili povratne informacije korisnika.

Scrum i Laravel

Scrum i Laravel mogu se kombinirati kako bi se olakšao razvoj softverskih aplikacija na efikasan način, koristeći Scrum kao metodologiju upravljanja projektima, a Laravel kao PHP framework za razvoj aplikacija.

Evo kako se Scrum može primijeniti u razvoju aplikacija s Laravelom:

Planiranje (Sprint Planning)

Product Owner definira **Product backlog** (popis zadataka) s funkcionalnostima koje aplikacija treba imati. Na primjer, funkcionalnosti kao što su autentifikacija korisnika, upravljanje postovima, integracija s bazom podataka, i druge karakteristike aplikacije razvijene u Laravelu.

Tim koristi **Sprint planning** kako bi odlučio koje će funkcionalnosti iz Product backloga raditi u narednom sprintu (obično 2-4 tjedna).

Razvoj tokom sprinta

Laravel pruža alate i resurse koji olakšavaju izradu tih funkcionalnosti. Razvojni tim može koristiti Laravel-ove ugrađene značajke kao što su **ORM (Eloquent)** za rad s bazom podataka, **Blade templates** za kreiranje korisničkog sučelja, i middleware za sigurnosne provjere.

Svaki član tima može preuzeti određene user stories (zadaci) i raditi na razvoju specifičnih dijelova aplikacije u Laravelu.

Daily Scrum

Svakodnevno, tim održava kratke sastanke (15 minuta) kako bi se osiguralo da svi razumiju što se radi i s kojim izazovima se suočavaju. Na primjer, članovi tima mogu izvještavati o problemima s bazom podataka, rutama ili API integracijama u Laravelu.

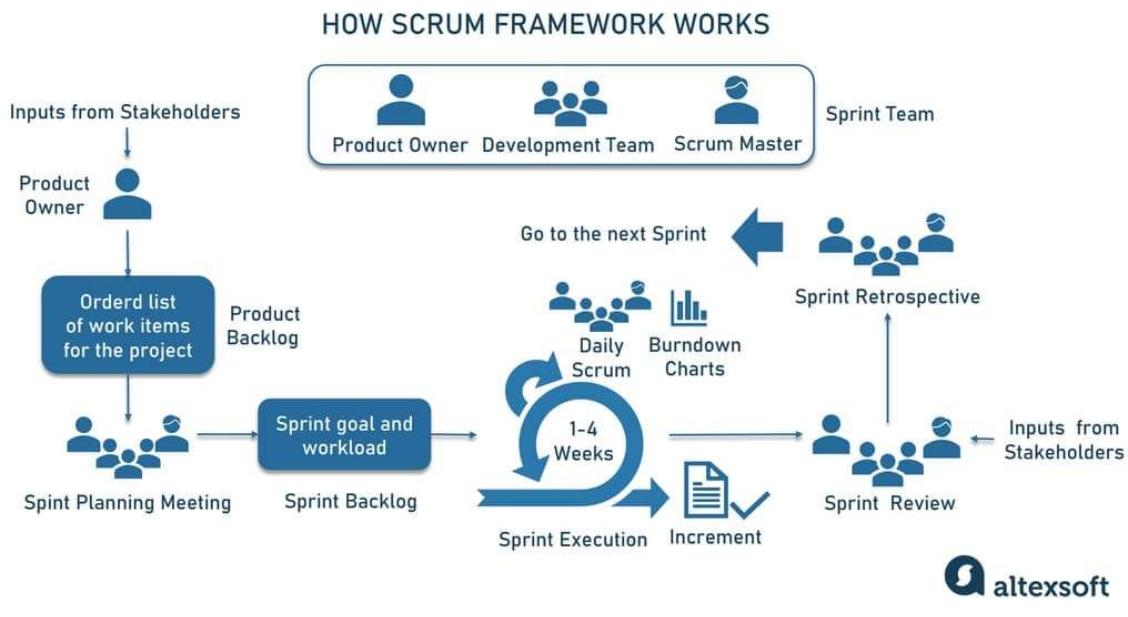
Sprint Review

Na kraju sprinta, tim prezentira ono što je izgrađeno u Laravelu. Na primjer, može se prezentirati novi modul za korisničku registraciju ili nadzorna ploča za administratore, koja je u potpunosti funkcionalna i spremna za testiranje.

Product Owner daje povratne informacije i odlučuje da li funkcionalnosti odgovaraju poslovnim potrebama.

Sprint Retrospective

Nakon svakog sprinta, tim analizira što je dobro prošlo u razvoju aplikacije s Laravelom, a što treba poboljšati. Na primjer, mogu primjetiti da bi bilo korisno unaprijediti procese testiranja, rad s rutama ili automatizaciju deploya korištenjem alata kao što su **Laravel Forge** ili **Envoyer**.



Kanban

Kanban je metoda koja se fokusira na vizualizaciju radnog toka, upravljanje radnim opterećenjem i poboljšanje efikasnosti procesa.

Kanban Board



Ključni elementi

Vizualna ploča: Kanban ploča koristi se za praćenje statusa zadataka kroz različite faze, poput "To Do" (Za napraviti), "In Progress" (U radu) i "Done" (Završeno). Ploča može biti fizička (s post-it bilješkama) ili digitalna (koristeći softverske alate).

Ograničenje rada u toku (WIP - Work In Progress): Postavljaju se granice za broj zadataka koji se mogu istovremeno raditi u svakoj fazi kako bi se spriječila preopterećenost tima. Ovo pomaže u održavanju fokusiranosti i poboljšanju kvalitete rada.

Kontinuirano poboljšanje: Kanban podstiče timove da stalno analiziraju i poboljšavaju svoje procese, koristeći povratne informacije i metrike za identifikaciju područja za optimizaciju.

Prednosti

- **Fleksibilnost:** Timovi mogu brzo reagirati na promjene prioriteta i zahtjeva.
- **Jasna vizualizacija:** Ploča pruža jasnu sliku napretka i statusa projekta.
- **Povećana efikasnost:** Ograničavanje WIP-a poboljšava fokus i produktivnost tima.

Hybrid

Hibridni pristup kombinira elemente različitih agilnih metodologija (npr. Scrum i Kanban) kako bi se prilagodio specifičnim potrebama projekta i tima.

Ključni elementi

Kombinacija pristupa: Tim može koristiti Scrum za planiranje iteracija i definiranje ciljeva, dok istovremeno primjenjuje Kanban za upravljanje radnim tokom i vizualizaciju zadataka.

Fleksibilnost u primjeni: Hibridni pristup omogućava timovima da izaberu najbolje prakse iz različitih metodologija kako bi stvorili vlastiti okvir rada.

Prednosti

Prilagodljivost: Timovi mogu prilagoditi svoj pristup na temelju vrsta projekata, veličine tima i potrebnih resursa.

Optimalno korištenje resursa: Mogućnost miješanja različitih pristupa omogućava efikasnije upravljanje resursima.

Lean

Lean metodologija usredotočuje se na smanjenje otpada, optimizaciju procesa i povećanje vrijednosti za korisnika.

Ključni elementi

Identifikacija vrijednosti: Važno je definirati što korisnik smatra vrijednim i usredotočiti se na aktivnosti koje dodaju vrijednost.

Eliminacija otpada: Lean identificira i eliminira aktivnosti koje ne dodaju vrijednost (npr. višak inventara, čekanje, nepotrebni koraci u procesu).

Poboljšanje toka rada: Fokus na kontinuirano poboljšanje procesa radi optimizacije vremena i resursa.

Prednosti

Smanjenje troškova: Eliminacija otpada dovodi do smanjenja troškova.

Brža isporuka: Optimizirani procesi omogućuju bržu isporuku proizvoda i usluga.

Bimodal

Bimodalni pristup kombinira dva različita načina rada: jedan za stabilne, predvidljive projekte (mod 1) i drugi za inovativne, brze projekte (mod 2).

Ključni elementi

Mod 1: Ovaj način rada fokusira se na stabilnost i efikasnost, koristeći tradicionalne metode upravljanja projektima.

Mod 2: Ovaj način rada fokusira se na fleksibilnost, inovaciju i eksperimentiranje, koristeći agilne metodologije.

Prednosti

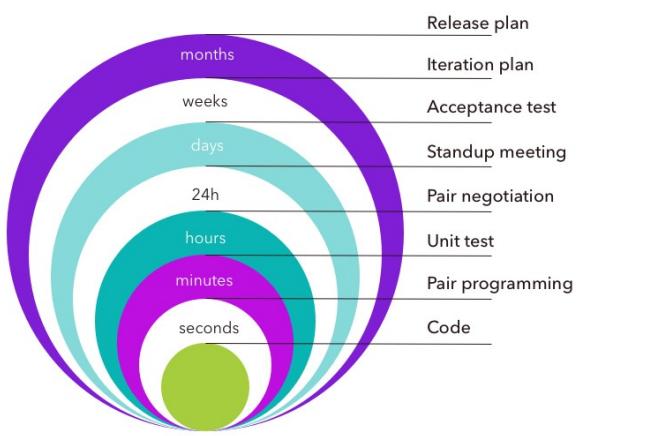
Prilagodljivost tržištu: Organizacije mogu brže odgovoriti na promjene u tržištu i zahtjevima korisnika.

Uravnoteženje stabilnosti i inovacije: Omogućava organizacijama da zadrže stabilne operacije dok istražuju nove mogućnosti.

Extreme Programming (XP)

Extreme Programming (XP) je agilna metodologija koja se fokusira na poboljšanje kvalitete softverskog razvoja i sposobnosti prilagodbe promjenama.

XP Feedback Loops



Source: <http://www.extremeprogramming.org>

objectstyle

Ključni elementi

Iteracije: Kratke iteracije omogućuju brzu povratnu informaciju i omogućuju timu da brzo reagira na promjene zahtjeva.

Automatizirano testiranje: Automatizirano testiranje je standardna praksa, što pomaže u održavanju kvalitete i smanjenju grešaka.

Parno programiranje: Dva programera rade zajedno na istom kodu, što poboljšava kvalitetu i olakšava razmjenu znanja.

Prednosti

Povećana kvaliteta: Kontinuirano testiranje i povratne informacije poboljšavaju kvalitetu softvera.

Brža prilagodba: Kratke iteracije omogućuju timovima da se brzo prilagode promjenama u zahtjevima.

Crystal

Crystal je skup metodologija koje se prilagođavaju specifičnim potrebama tima i projekta, s naglaskom na ljudske aspekte razvoja.

Ključni elementi

Prilagodljivost: Svaki projekt odabire vlastiti "Crystal" pristup ovisno o veličini tima, složenosti i zahtjevima projekta (npr. Crystal Clear, Crystal Yellow, Crystal Orange).

Naglasak na ljudima: Crystal stavlja naglasak na interakciju među članovima tima, komunikaciju i međusobno povjerenje.

Prednosti

Fleksibilnost: Omogućuje timovima da pronađu pristup koji najbolje odgovara njihovim potrebama i radnom okruženju.

Povećana produktivnost: Naglasak na ljudskim aspektima može poboljšati suradnju i komunikaciju unutar tima.

Automatizacija i Continuous Integration

Laravel se može lako integrirati u **Continuous Integration (CI)** pipeline. Korištenjem alata kao što su **GitLab CI/CD**, **Jenkins**, ili **Travis CI**, možete automatski testirati i implementirati Laravel aplikaciju nakon svakog sprinta.

Agilno podešavanje funkcionalnosti

Laravel pruža fleksibilnost da se funkcionalnosti mijenjaju ili proširuju, što je u skladu sa Scrumovom filozofijom prilagodljivosti. Ako Product Owner promijeni prioritete, Laravel olakšava brze promjene unutar koda, dodavanje novih funkcionalnosti ili izmjenu postojećih modula.

CI/CD

CI/CD u kontekstu Laravela odnosi se na praksu **kontinuirane integracije (CI)** i **kontinuirane isporuke/deploymenta (CD)** u procesu razvoja softvera. Ove metode automatiziraju dijelove procesa razvoja, testiranja i isporuke koda, osiguravajući brži i pouzdaniji tijek rada.

CI (Continuous Integration) kod Laravela

Kontinuirana integracija podrazumijeva proces automatskog testiranja i integriranja promjena koda u zajednički repozitorij više puta dnevno. U kontekstu Laravela, to znači:

Automatizacija testiranja

- Svaki put kad se izvrši promjena u Laravel projektu, CI alat (npr. Jenkins, GitHub Actions, GitLab CI, Travis CI) automatski pokreće **unit testove**, **feature testove**, ili druge vrste testova (npr. PHP Unit testovi).
- Laravel nudi podršku za testiranje kroz ugrađene alate kao što su **PHPUnit** za unit testove i Dusk za end-to-end testove.

Integracija novih promjena

- Nakon što developeri pošalju novi kod (npr. putem pull requestova ili pushanja na granu), CI sistem automatski preuzima kod, pokreće testove, i javlja da li su promjene uspješne ili su testovi pali.
- Time se omogućuje da se brzo otkriju i isprave greške, a kod ostaje stabilan.

CD (Continuous Delivery/Deployment) u Laravela

Kontinuirana isporuka i deployment omogućuju da se uspješno testirani kod automatski pripremi za implementaciju na produkcijski server. U kontekstu Laravela, CD se može odnositi na:

Kontinuirana isporuka (Continuous Delivery)

Kod koji je prošao sve CI testove može se automatski pripremiti za manualni deployment na produkciju. To znači da se verzija koda uvijek može implementirati bez potrebe za dodatnim testiranjem ili manualnim pripremama.

U Laravel projektima, ovo uključuje automatsku izradu buildova, pripremu baza podataka, instalaciju potrebnih Composer paketa, itd.

Kontinuirani deployment (Continuous Deployment)

Kod se automatski implementira na produkciju nakon uspješno provedenih testova, bez potrebe za manualnom intervencijom.

Alati kao što su **Laravel Forge-a** ili **Envoyer-a** mogu olakšati CD procese u Laravel projektima, omogućujući deployment bez prekida rada aplikacije.

[Alati za CI/CD kod Laravela](#)

[GitHub Actions](#)

Popularan alat za CI/CD s direktnom integracijom s GitHubom. Može se koristiti za pokretanje testova, lintersa i automatski deployment nakon svakog pushanja koda.

[GitLab CI](#)

GitLab nudi ugrađeni CI/CD sistem koji omogućava testiranje, buildanje i deployment Laravel aplikacija izravno s GitLab repositorija.

[Jenkins](#)

Alat za CI koji se može prilagoditi za razne CI/CD zadatke i koristi se za automatizaciju procesa testiranja i deploymenta Laravel aplikacija.

[Laravel Forge i Envoyer](#)

Alati specifično prilagođeni Laravelu koji olakšavaju proces deploymenta aplikacija na servere i orkestraciju procesa bez downtime.

[Prednosti CI/CD u Laravelu](#)

Brža dostava novih funkcionalnosti.

Manje grešaka u produkciji zbog rigoroznijeg testiranja.

Automatizacija procesa testiranja i deploymenta, što smanjuje manualne greške.

Poboljšana produktivnost tima jer developeri mogu fokusirati više vremena na razvoj umjesto na ručno testiranje ili deployment.

CI/CD pomaže u uspostavljanju glatkog i pouzdanog toka rada, posebno za Laravel aplikacije, osiguravajući da se promjene brže i sigurnije implementiraju na produkciju.

Mapiranje fabule (engl. Story Mapping)

Mapiranje fabule (engl. Story Mapping) je tehnika koju je razvio Jeff Patton tokom svoje duge prakse kao Agile product owner/scrum master. Mapiranje fabule (engl. Story Mapping) dobilo je svoje ime jer pomaže vizualno mapirati korisničku fabulu i druge neobavljenе poslove. Stavke su raspoređene u dvije dimenzije: vertikalna označava prioritet, dok horizontalna predstavlja korake koje korisnik poduzima da izvrši radnje u sistemu (putovanje korisnika). Mapiranje fabule koju je opisao Patton uključuju takve strukturne elemente kao što su:

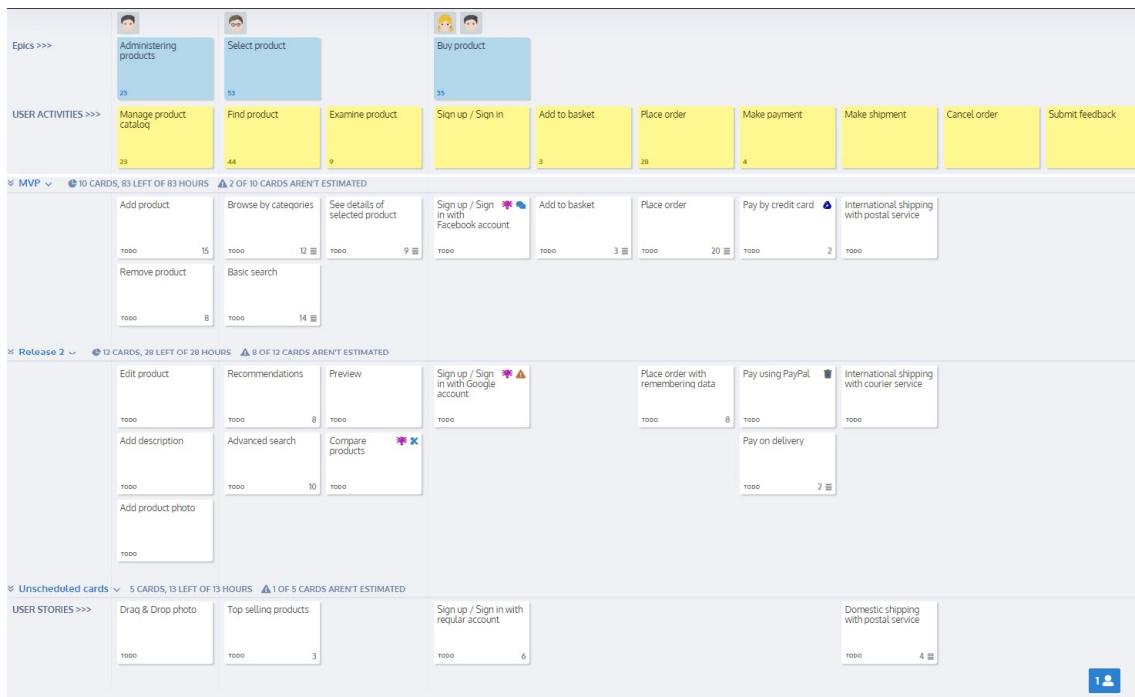
Osnovna mreža (engl. Backbone) je osnova mape. Sastoji se od epova ili tema koje opisuju ukupne aktivnosti korisnika u sistemu, npr. *proizvode za pretraživanje*. Epovi su raspoređeni horizontalnim redom jer predstavljaju korake koje korisnik poduzima tokom interakcije s proizvodom, što je u osnovi jednostavan prikaz *korisničkog putovanja*.

Fabule (priče, engl. Stories). Za razliku od ploše strukture neobavljenog posla, korisničke fabule su raspoređene u vertikalnu i horizontalnu dimenziju. Korisničke fabule grupirane su po odgovarajućim epovima, opisujući specifične zadatke koje korisnik može zahtijevati. Ako ep opisuje fazu pretraživanja, može uključivati fabule kao što su osnovno pretraživanje, filtriranje proizvoda, napredno pretraživanje itd. Kada se fabulama da vertikalni prioritet, mogu se podijeliti na izdanja.

Osobe korisnika (engl. User personas) su izmišljeni prikazi ljudi koji će koristiti proizvod/izvoditi korake opisane u korisničkim fabulama. Korisnici koje su izradili UX stručnjaci nakon intervjuja s korisnicima, daju opise o tome tko su korisnici i kako bi mogli međusobno djelovati na proizvod. Na mapi fabule (engl. story map), osobe su vezane za namjenske epove u kojima će sudjelovati.

Lijepo je imati ideje. Kako bi se prikazala cijelovita slika, mapa fabule (engl. story map) može također uključivati blokove kao što su ideje ili svojstva koja bi bilo lijepo imati. Oni će imati na umu korisničke fabule koje još nisu potrebne ili nisu navedene u početnim zahtjevima, ali ipak dodaju vrijednost proizvodu.

Pristup mapiranju također potiče produktivan razgovor unutar razvojnog tima. Iako njome može upravljati vlasnik proizvoda (PO – engl. Product Owner) ili voditelj proizvoda (engl. Product Manager), sama mapa se kreira u radionici mapiranja fabule kao zajednički napor. Tokom radionice, članovi tima mogu postavljati pitanja poput "zašto nam treba ova fabula?" ili "zašto su ove fabule važnije od onih?" To jamči međusobno razumijevanje razvojnih ciljeva i konteksta svake priče. Nadalje, mapa pomaže prikazati sve dodatne pojedinosti o proizvodu i prenijeti te informacije cijelom timu.



Jednostavna Mapa fabule (engl. Story Map) za komercijalni proizvod. Izvor: storiesonboard.com

Danas postoji mnogo digitalnih proizvoda koji pomažu timovima bilo kao dokument koji se može dijeliti na webu ili kao ugrađeno CRM rješenje:

- FeatureMap.co
- Trello
- Jira
- CardBoardIt.com
- Miro.com

Aimeos

Aimeos je open-source ecommerce platforma razvijena za integraciju s PHP frameworkovima, prvenstveno s Laravelom i Symfonyjem. Nudi fleksibilan i skalabilan sistem za izgradnju webshopova i naprednih ecommerce rješenja. Aimeos je poznat po svojoj prilagodljivosti i funkcijama koje omogućavaju kreiranje ecommerce rješenja koja mogu pokriti različite poslovne potrebe, od malih webshopova do kompleksnih enterprise sistema.

Glavna svojstva Aimeos

Multikanalna podrška: Omogućava prodaju na više kanala, npr. online, u trgovinama, mobilnim aplikacijama i na društvenim mrežama.

Multi-vendor i multi-site podrška: Omogućava kreiranje platformi gdje više različitih prodavača može imati svoje trgovine unutar jedne aplikacije.

Fleksibilnost u dizajnu: Dolazi s modularnim dizajnom koji omogućava prilagođavanje funkcionalnosti, kao što su košarica, proces plaćanja i profil korisnika.

Podrška za više valuta i jezika: Aimeos nudi integraciju za globalno poslovanje, omogućujući korisnicima korištenje različitih valuta i jezika.

SEO optimizacija: Sadrži alate za optimizaciju pretraživača (SEO), kao što su prilagodljive URL strukture i meta podaci.

Visoke performanse: Optimiziran je za rad s velikim brojem proizvoda (milijuni proizvoda i narudžbi) zahvaljujući podršci za caching i drugim tehnikama za optimizaciju performansi.

Kako se koristi Aimeos?

Aimeos se može integrirati u Laravel projekte pomoću Aimeos Laravel paketa ili u Symfony projekte pomoću Aimeos Symfony paketa. Instalacija uključuje nekoliko koraka, poput konfiguracije baza podataka, kreiranja potrebnih migracija i postavljanja modula za frontend i backend.

Primjer instalacije u Laravelu:

```
composer require aimeos/aimeos-laravel
php artisan aimeos:setup
php artisan migrate
php artisan serve
```