Sadržaj

1.1. Laravel

Laravel je besplatan PHP MVC (engl. Model-View-Controller) radni okvir (engl. framework) otvorenog koda kojeg je izradio Taylor Otwell 2011. godine te se njime značajno olakšava izrada dinamičkih web stranica. Stvoren je na temelju biblioteka već poznatog Symfony PHP radnog okvira. Zagovara princip konvencije, a ne konfiguracije (engl. convention over configuration). Za razliku od nekih Java, Python ili PHP radnih okvira koji koriste mnogo XML-a za konfiguraciju, Laravel ne zahtjeva nikakvu dodatnu konfiguraciju nakon same instalacije te je odmah spreman za rad (osim u slučaju ako programer želi nešto izričito promijeniti). Trenutna zadnja verzija je 5.2. Laravel nudi mnoštvo funkcionalnosti koje je usmjereno na pojednostavljivanje rutinskih zadataka koliko je to moguće, a s kojima se programeri stalno susreću prilikom izrade dinamičkih web stranica. Od važnijih stavki tu su Blade (služi za izradu HTML predložaka) i Eloquent ORM (engl. object-relational mapping) - napredna PHP implementacija za rad s bazom podataka gdje se svaki red tj. n-torka iz tablica tretira kao objekt.

1.1.1. LEMP i Laravel instalacija

LEMP (Linux, nginx, MySQL i PHP) stog (engl. stack) je okruženje na kojemu je moguće funkcioniranje aplikacije.

Linux je uniksoidan (engl. Unix-like) operacijski sustav koji se temelji na Linux jezgri (engl. kernel) koju je 1991. napravio Linus Torvalds. Danas se on nalazi na većini poslužitelja. Linux je operacijski sustav otvorenog koda te zbog toga postoji mnogo Linux distribucija, a neke od poznatijih su Ubuntu, Debian, Fedora, RHEL, CentOS, openSUSE i Gentoo. Za pokretanje aplikacije izrađene u sklopu ovog završnog rada korišten je 64 bitni Ubuntu 16.04 LTS (Long Term Support). LTS označava da se radi o izdanju koje će biti podržano na dulji vremenski period u odnosu na „obično“ izdanje. Ubuntu je odlučio da će svako četvrto izdanje koje se izdaje u periodu od dvije godine biti duže podržano. LTS verzije dobivaju podršku 5 godina nakon izlaska, dok „obične“ verzije imaju podršku samo 9 mjeseci nakon izlaska.

Nginx (izgovara se Engine X) je web server u LEMP stogu koji služi za posluživanje stranica korisnicima (procesira HTTP zahtjeve i vraća odgovore). On je relativno novi web server koji trenutno ima 15.6% udjela na tržištu, a odabran je umjesto Apachea (33.56% udjela) jer je brži te zahtjeva manje serverskih resursa (CPU i RAM).

Laravel 5.2 (zadnja verzija Laravela u trenutku pisanja ovog završnog rada) za svoj rad zahtjeva:

- PHP >= 5.5.9
- OpenSSL PHP ekstenziju
- PDO PHP ekstenziju
- Mbstring PHP ekstenziju
- Tokenizer PHP ekstenziju

Na početku je potrebno dodati PPA (Personal Package Archives) repozitorije kako bi mogli instalirati najnoviji PHP 7 (7.0.8) i nginx (1.10.1).

```
sudo add-apt-repository ppa:ondrej/php
```

```
sudo add-apt-repository ppa:nginx/stable
```

Instalacija je vrlo jednostavna i obavlja se preko apt-get naredbe:

```
sudo apt-get update
```

```
sudo apt-get install git unzip zip nginx php-fpm php-mysql php-cli php-mcrypt php-xml curl php-curl php-mbstring mysql-server
```

Gore navedenom naredbom se instalira PHP, nginx, MySQL te ostali potrebni paketi za rad. Zatim je potrebno promijeniti pojedine postavke u nginx i PHP konfiguraciji. Koristimo se sljedećim naredbama:

```
sudo nano /etc/nginx/sites-available/default
```

```
sudo nano /etc/php/7.0/fpm/php.ini
```

```
sudo nano /etc/php/7.0/fpm/pool.d/www.conf
```

Nano je jedan od tekst editora na Linuxu. Iz njega izlazimo kombinacijom tipki Ctrl i X te nas zatim pita želimo li spremiti učinjene promjene, pritiskom tipke „Y“ pa

enter potvrđujemo da želimo te izlazimo iz editora. U nginx default datoteci je potrebno postaviti način obrade zahtjeva te tko će procesuirati PHP, a to izgleda ovako:

```
location / {
        try_files $uri $uri/ /index.php?$query_string;
    }
location ~ \.php$ {
    include snippets/fastcgi-php.conf;
    fastcgi_pass  127.0.0.1:9000;
  }
```

Treba dodati index.php pod index te domenu ili IP adresu poslužitelja pod server_name.

Nakon izmjene nginx konfiguracije testira se njena ispravnost sljedećom naredbom:

```
sudo nginx –t
```

Ukoliko je sve u redu potrebno je restartirati nginx da učita napravljene promjene:

```
sudo systemctl reload nginx
```

ili

```
sudo systemctl restart nginx
```

Što se PHP-FPM-a tiče, u njegovoj php.ini datoteci potrebno je promijeniti samo jednu postavku. Odkomentiramo liniju cgi.fix_pathinfo=1 (makne se „;“ s početka linije) te se vrijednost postavi na 0. Navedena promjena je nužna jer se radi o vrlo nesigurnoj postavki koja govori PHP-FPM-u da izvrši PHP datoteku s najbližim nazivom ukoliko tražena PHP datoteka ne postoji.

U www.conf datoteci liniju

listen = /run/php/php7.0-fpm.sock

promijenimo u

listen = 127.0.0.1:9000

Nakon toga je potrebno restartirati PHP-FPM servis naredbom:

```
sudo systemctl restart php7.0-fpm
```

Laravel se instalira pomoću Composera. Composer je alat koji se brine o zavisnosti alata/biblioteka u PHP ekosustavu. Dozvoljava korisniku da definira koje stvari projekt zahtjeva kako bi radio te ih Composer sam pronalazi i instalira u vendor folder unutar projekta. Također nudi opciju osvježavanja postojećih paketa i biblioteka s njihovim novijim verzijama. Za svoj rad zahtjeva barem PHP 5.3.2 verziju. Na Linuxu se instalira unošenjem naredbe:

```
curl -sS https://getcomposer.org/installer | sudo php -- --install-dir=/usr/local/bin --filename=composer
```

Za Windows operativni sustav dostupna je .exe datoteka na službenim stranicama. Prije same instalacije Laravela potrebno se je pozicionirati unutar /var/www foldera naredbom:

```
cd /var/www
```

Instalacija Laravela se pokreće slijedećom naredbom:

```
sudo composer create-project laravel/laravel tvz
```

gdje je „tvz“ naziv projekta i foldera koji će biti stvoren unutar /var/www foldera.

Potom dajemo vlasništvo nad tim folderom www-data korisniku odnosno grupi (pod www-data korisnikom se vrti nginx):

```
sudo chown -R www-data:www-data /var/www/tvz
```

te dozvole za pisanje nad storage folderom (i njegovim podfolderima):

```
sudo chmod -R 775 /var/www/tvz/storage
```

Zatim je preostalo podesiti inicijalni (root) folder iz kojega će nginx servirati sadržaj:

```
sudo nano /etc/nginx/sites-available/default
```

Root nakon izmjene mora izgledati ovako:

/var/www/tvz/public;

Nakon toga potrebno je ponovno restartirati nginx kako bi prihvatio navedenu promjenu:

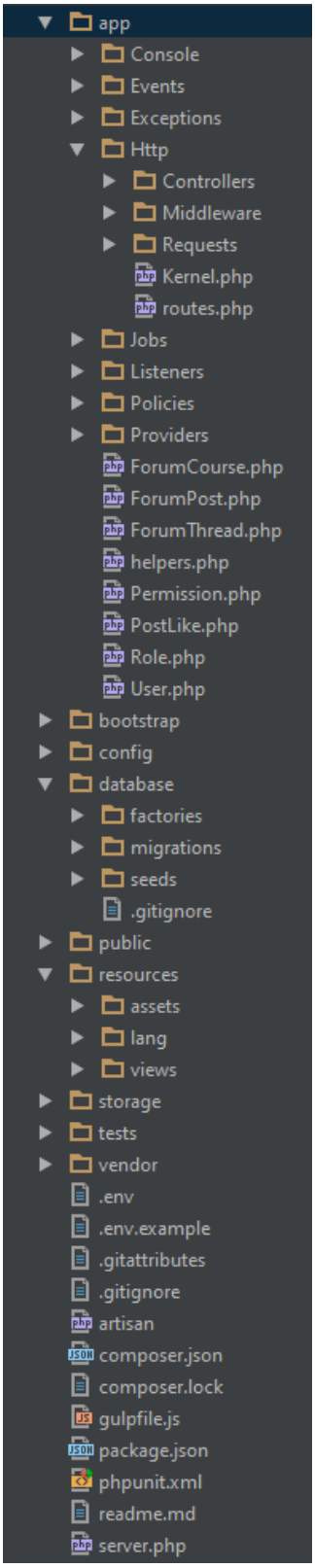
```
sudo systemctl reload nginx
```

ili

```
sudo systemctl restart nginx
```

Ovim korakom završena je instalacija, eventualno je još potrebno u Laravelovoj .env datoteci namjestiti naziv baze, korisnika i lozinku kojom se pristupa MySQL-u (DB_DATABASE, DB_USERNAME i DB_PASSWORD ključevi).

1.1.2. Laravel struktura foldera



Slika 1: Struktura projekta

Laravel se sastoji od mnogo foldera i podfoldera, ali najbitnije je idućih pet: app, database, public, resources i vendor. App folder sadrži sve kontrolere i modele te se ovdje nalazi gotovo sav kod kojega programer napiše. Database folder sadrži podfolder migrations u kojem su definirane sve migracije za projekt (o njima će biti riječ kasnije). Public folder sadrži statičan sadržaj poput CSS i JS datoteka te slika. On mora biti postavljen u postavkama web servera kao inicijalni (root) folder (kao što je to objašnjeno u prethodnom poglavlju). U resources folderu se nalazi folder views u kojemu se nalaze svi pogledi. U vendor folderu se nalazi sav programski kod Laravela kao i svih drugih stvari koje je instalirao Composer.

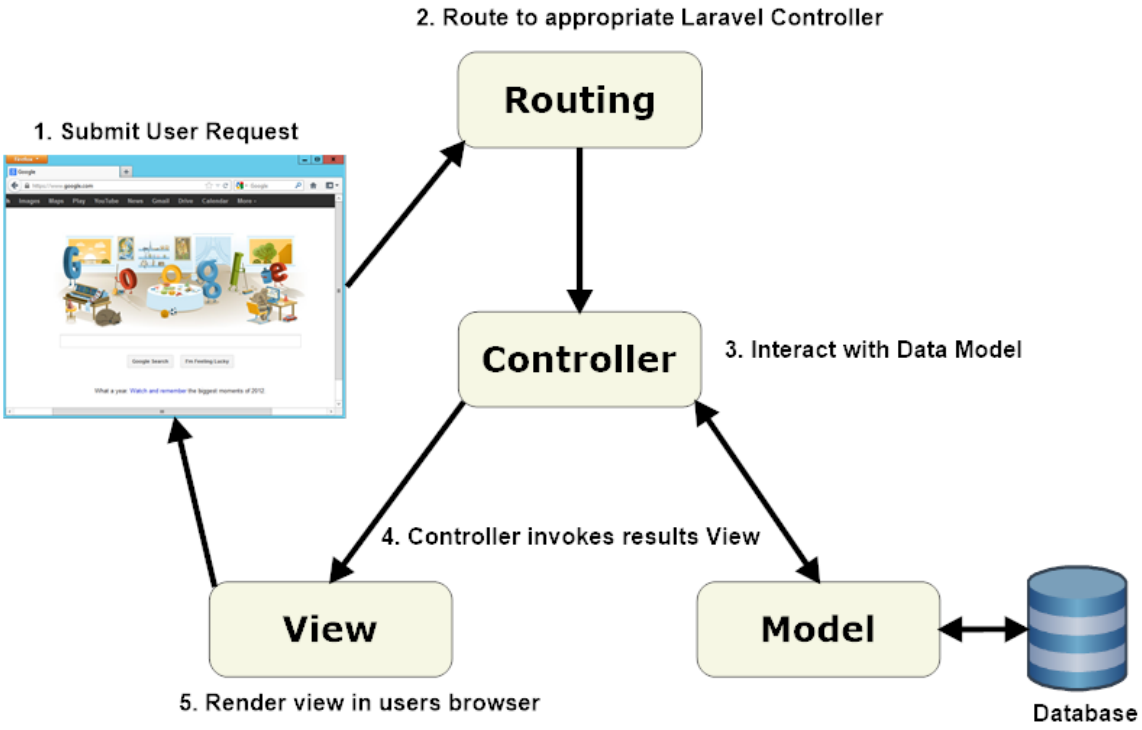
1.1.3. Model-View-Controller (MVC) arhitektura

Laravel se drži MVC arhitekture, što znači da je sve podijeljeno na tri komponente: kontrolere (engl. controllers), modele (engl. models) i poglede (engl. views). MVC arhitektura forsira separaciju između programske „logike“ i prezentacijske „logike“ povezane s korisničkim grafičkim sučeljem - GUI-em (engl. Graphical user interface). U slučaju Laravela, programska logika se sastoji od kontrolera i modela koji predstavljaju korisnike, teme, postove itd., a prezentacijska logika odnosno grafičko sučelje je u ovom slučaju web stranica u korisnikovom web pregledniku.

Model je domena oko koje je neki softver napravljen. Bazirani su na stvarima iz stvarnog svijeta, poput osobe, bankovnog računa ili proizvoda. Obično su permanentni te se najčešće pohranjuju u baze podataka. Model je više od pukih podataka, on ima opcije forsiranja pojedinih pravila na te podatke. Implementacijom tih pravila u modelu osiguravamo da ništa u našoj aplikaciji ne može invalidirati podatke. Moguće je prvo definirati sve svoje modele pa onda na temelju njih stvoriti bazu podataka ili se može stvoriti baza podataka pa na temelju nje pomoću raznih alata stvoriti sve klase modela sa svim njihovim pripadajućim atributima.

Pogled je vizualna reprezentacija modela s danim kontekstom. Pogled je jedan od mogućih odgovora korisniku, poput primjerice HTML reprezentacije forum posta. Iako može prezentirati korisniku razne načine na koje korisnik može slati podatke i zahtjeve na server, sam pogled nikada ne obrađuje te zahtjeve. Čim se pogled generira odnosno čim se on prikaže korisniku tu je kraj njegovog rada.

Kontroleri su koordinatori koji povezuju poglede i modele. Oni su ti koji obrađuju korisničke zahtjeve te im vraćaju određene odgovore (npr. pogled). Sva PHP programska logika se zapravo događa u njima jer su oni zaduženi za sva preusmjeravanja korisnika, provjeru validnosti korisničkog unosa, provjeravanje dozvola za pristup određenoj stranici i slično. Dakle, kontroleri upravljaju unesenim podacima (engl. input), a pogledi prikazuju podatke (engl. output). To u Laravelu izgleda ovako:



Slika 2: Laravel MVC arhitektura

1.1.4. Rutiranje

Kako bi se znalo koji URI (engl. Uniform Resource Identifier) otvara koju stranicu odnosno koji je kontroler tj. koja njegova metoda zadužena za obradu i vraćanje odgovora Laravel koristi tzv. rutiranje. Sve rute se nalaze u app/Http/routes.php datoteci. Primjer izgleda te datoteke:

```
<?php

Route::get('welcome', 'HomeController@index');
```

Ovo znači da pri slanju GET HTTP zahtjeva na npr. URL

http://my-page.com/welcome Laravel zna da je u ovom slučaju HomeController (tj. njegova metoda index) zadužena za obradu tog zahtjeva. Kontroleri se stvaraju s Artisan naredbom (više o Artisanu pročitajte u poglavlju 1.2.9):

```
php artisan make:controller HomeController
```

Kontroleri se nalaze u app/Http/Controllers folderu.

Primjer izgleda index metode HomeControllera:

```php
<?php

namespace App\Http\Controllers;

use App\Http\Requests;
use Illuminate\Http\Request;

class HomeController extends Controller
{
    /**
     * Show the welcome page.
     *
     * @return \Illuminate\Http\Response
     */
    public function index()
    {
        return view('welcome');
    }
}
```

Dakle u ovom slučaju sve što radi index metoda jest to da vraća pogled s nazivom welcome, a on izgleda ovako:

```
@extends('layouts.app')

@section('content')
    <div class="row">
        <div class="col-md-10 col-md-offset-1">
            <div class="panel panel-default">
                <div class="panel-heading">Dobrodošao</div>

                <div class="panel-body">
                    Početna stranica.
                </div>
```

```
        </div>
    </div>
</div>
@endsection
```

Vidimo da je odgovor obični HTML. Iz kontrolera je moguće vratiti odnosno prenijeti i dodatne podatke koji će biti ispisani u pogledu (npr. rezultat nekakvog upita na bazu podataka itd.). Primjer index metode te dio welcome pogleda gdje se ispisuju podaci preneseni iz kontrolera:

```
/**
 * Show the welcome page.
 *
 * @return \Illuminate\Http\Response
 */
public function index()
{
    $name = "Antonio";
    $number = 264;
    return view('welcome', ['name' => $name, 'number' => $number]);
}
```

```
<div class="panel-heading">Dobrodošao, {{ $name }}</div>

<div class="panel-body">
    Vaš broj je {{ $number }}
</div>
```

View funkcija kao prvi parameter prima naziv pogleda kojeg vraća, a kao drugi parametar prima asocijativno polje gdje ključ predstavlja naziv varijable koja će se ispisati u pogledu. Osim ovog načina, moguće je istu stvar napraviti i ovako:

```
return view('welcome')->with(['name' => $name, 'number' => $number]);
```

ili

```
return view('welcome', compact('name', 'number'));
```

Laravel Blade sustav za stvaranje HTML predložaka (engl. templating engine) i njegove glavne opcije poput gore vidljivih @extends i @section su obrađene u idućem poglavlju.

1.1.5. Blade

Blade je kao što je gore spomenuto sustav za stvaranje HTML predložaka. Uglavnom se koristi jedan glavni predložak (može ih se napraviti i više ako je potrebno) u kojeg se ovisno o tome koju je stranicu korisnik zatražio dodaju pripadne stvari. Primjer jednog takvog predloška:

```
<!DOCTYPE html>
<html>
<head>
    <meta charset="UTF-8">
    <meta http-equiv="X-UA-Compatible" content="IE=edge">
    <meta name="viewport" content="width=device-width, initial-scale=1">

    <title>X-Wind</title>

    <!-- Styles -->
    <link rel="stylesheet" href="/css/bootstrap.min.css">

@yield('stylesheets')

</head>
<body>

<div class="container">

    @yield('content')

    <div class="copyright">
        <small class="pull-left">
            X-Wind
        </small>
        <small class="pull-right">
              &copy;{{date("Y")}}  
        </small>
    </div>
</div>

<!-- JavaScripts -->
<script src="/js/jquery-2.2.4.min.js"></script>
<script src="/js/bootstrap.min.js"></script>
```

```
@yield('scripts')

</body>
</html>
```

Svi pogledi se nalaze u resources/views folderu te svi moraju imati .blade.php ekstenziju kako bi ih Laravel ispravno prepoznao kao Blade dokumente. To su zapravo PHP datoteke pa je sintaksa poziva neke PHP funkcije u Bladeu vrlo jednostavna – stave se dvije otvorene vitičaste zagrade pa zatim ide PHP funkcija i to se onda zatvori s dvije vitičaste zagrade. Primjer toga možete vidjeti u gornjem primjeru koda s PHP date funkcijom. Dvije najveće prednosti korištenja Bladea su nasljeđivanje predložaka (engl. template inheritance) i sekcije (engl. sections). To funkcionira na način da se u glavnom predlošku @yield direktivama (engl. directives) označi i da naziv mjestu na kojem će u naslijeđenom predložku biti na to mjesto umetnut nekakav sadržaj. Primjer toga imamo u gornjem predlošku gdje se nalazi više @yield direktiva s različitim imenima (stylesheets, content i scripts). Stvaranjem novog pogleda (naziva npr. home) koji u sebi ima samo slijedeće:

```
@extends('layouts.app')

@section('content')
    <div class="row">
        <div class="col-md-10 col-md-offset-1">
            <div class="panel panel-default">
                <div class="panel-heading">Početna stranica</div>

                <div class="panel-body">
                    Pozdrav, nalazite se na početnoj stranici.
                </div>
            </div>
        </div>
    </div>
@endsection
```

stvara se naslijeđeni predložak. To se postiže s @extends direktivom u koju ide naziv glavnog predloška (u ovom slučaju layouts.app). Dotacija s točkom za Blade znači da traži taj pogled u layouts folderu s nazivom app (a počinje gledati u /resources/views folderu kako je i rečeno da se tamo spremaju svi pogledati u MVC poglavlju) – dakle u ovom slučaju se radi o /resources/views/layouts/app.blade.php.

Na ovaj način dotacije s točkom može se ići unedogled kroz stablo foldera. Ovo vrijedi i kod view funkcije (kad se iz kontrolera vraća neki pogled kao odgovor što je objašnjeno u poglavlju rutiranja). Nakon @extends direktive slijedi @section direktiva koja govori Laravelu na koje mjesto odnosno u koju @yield direktivu iz glavnog predloška (@section i @yield moraju imati isti naziv) mora ubaciti sav sadržaj koji se nalazi između @section i @endsection direktiva. Za sav ispis sadržaja se inače koristi {{ $x }} sintaksa koja poziva htmlentities PHP funkciju zbog prevencije XSS napada. Ukoliko se želi ispisati sadržaj bez toga koristi se {!! $x !!} sintaksa.

Blade nudi svoje kontrolne strukture implementirane preko direktiva.

Primjer ifa:

```
@if (count($records) === 1)
    Imam jedan element.
@elseif (count($records) > 1)
    Imam više elemenata.
@else
    Nemam niti jedan element.
@endif
```

If počinje s @if, a završava s @endif.

Blade podržava sve vrste PHP petlji (pa i neke dodatne). Primjer for, foreach, forelse i while petlje:

```
@for ($i = 0; $i < 10; $i++)
    Trenutna vrijednost je {{ $i }}
@endfor

@foreach ($users as $user)
    <p>Ovo je korisnik {{ $user->id }}</p>
@endforeach

@forelse ($users as $user)
    <li>{{ $user->name }}</li>
@empty
    <p>Nema korisnika</p>
@endforelse
```

```
@while (true)
    <p>Ovo je beskonačna petlja.</p>
@endwhile
```

Laravel podržava i to da sami možete definirati svoje Blade direktive.

1.1.6. Eloquent ORM

1.1.6.1. Osnove Eloquenta

ORM (engl. Object-relational mapping) je tehnologija koja omogućava manipulaciju podataka iz baza podataka u obliku objekata. Da bi to bilo moguće potrebni su nam modeli. Svaki model je zapravo klasa koja predstavlja jednu od tablica iz baze podataka (npr. model User predstavlja tablicu „users“), a svaka instanca tog modela predstavlja jednu n-torku tj. redak iz te tablice. Laravelov ORM se naziva Eloquent. Za komunikaciju s bazom podataka koristi PDO ekstenziju što znači da se možemo spojiti na bilo koju bazu podataka (npr. MySQL ili PostgreSQL) te će Eloquent ovisno o korištenoj bazi generirati ispravne SQL upite za tu bazu podataka.

Primjer izgleda modela s osnovnim postavkama:

```
<?php

namespace App;

use Illuminate\Database\Eloquent\Model;

class User extends Model
{
    /**
     * The table associated with the model.
     *
     * @var string
     */
    protected $table = 'users';

    /**
     * The attributes that are mass assignable.
     *
     * @var array
     */
```

```
    protected $fillable = [
        'name', 'email', 'password',
    ];

    /**
     * The attributes that should be hidden for arrays.
     *
     * @var array
     */
    protected $hidden = [
        'password', 'remember_token',
    ];
}
```

Dakle, korisnički definiran model mora nasljeđivati klasu „Model“ (tj. može nasljeđivati bilo koju klasu koja je već naslijedila klasu „Model“). Nadalje Eloquent sam pretpostavi da ako se model naziva „User“ da se njegova tablica naziva „users“ (tj. doda „s“ na kraj kako bi imenica bila u množini). Ukoliko se naša tablica tako ne naziva moramo postaviti atribut $table na ispravan naziv naše tablice. To znači da je u gornjem primjeru programskog koda bespotrebna linija kojom se $table atributu pridružuje „users“ vrijednost jer bi to Eloquent znao i bez toga. Atribut $fillable služi kako bi se definirali stupci iz te tablice koji se smiju dodijeliti tijekom kreiranja nove n-torke u tablici. To služi kao zaštita od malicioznih napadača koji bi inače mogli dodati kod proslijeđivanja HTTP zahtjeva i neke neočekivane parametre i taj bi parametar mogao promijeniti stupac u bazi podataka kojeg programer nije zamislio da se s tim zahtjevom može/smije dodati. $hidden atribut definira one stupce tablice koji se ne ispisuju kada se zatraži ispis „svih“ stupaca tog modela.

Eloquent nam daje vrlo jednostavan pristup entitetima tablica za koje su stvoreni modeli. Podržava dohvaćanja (SELECT), kreiranje (INSERT), izmjene (UPDATE) i brisanja (DELETE).

Primjer Eloquent sintakse:

```
// primjeri dohvaćanja n-torki

$users = User::all(); // dohvaćanje svih korisnika sa svim stupcima osim onih definiranih u
$hidden atributu
$user = User::find(1); // dohvaćanje korisnika s ID-om 1
$user = User::find([1, 2, 3]); // dohvaćanje korisnika s ID-ovima 1, 2 i 3
$user = User::findOrFail(1); // ukoliko ne postoji korisnik s ID-om 1 baca
```

```
ModelNotFoundException iznimku
$user = User::where('email', 'apauletic@tvz.hr')->first(); // dohvaćanje korisnika s apauletic@tvz.hr emailom
$user = User::where('posts', '>', 50)->first(); // dohvaćanje prvog korisnika koji ima više od 50 postova
$count = User::where('posts', '>', 50)->count(); // dohvaćanje broja korisnika koji imaju više od 50 postova
$max = Flight::where('active', 1)->max('price'); // dohvaćanje najviše cijene od svih aktivnih letova

// primjeri unosa n-torki

// 1. način unosa nove n-torke
$user = new User;
$user->name = 'Antonio';
$user->email = 'apauletic@tvz.hr';
$user->save();

// 2. način unosa nove n-torke (ovdje treba pripaziti na $fillable atribut iz modela)
$user = User::create(['name' => 'Antonio', 'email' => 'apauletic@tvz.hr']);

// primjeri izmjene (engl. update) n-torki

// korisniku s ID-om 1 se ime postavlja na Toni
$user = User::find(1);
$user->name = 'Toni';
$user->save();

// korisnicima koji imaju rolu Admin i preko 50 postova se ime postavlja na Antonio
User::where('role', 'Admin')
    ->where('posts', '>', 50)
    ->update(['name' => 'Antonio']);

// primjeri brisanja n-torki iz tablice

// brisanje n-torke s ID-om 1
$user = User::find(1);
$user->delete();

// također briše n-torku s ID-om 1
User::destroy(1);

// brisanje n-torki s ID-ovima 1, 2 i 3
User::destroy([1, 2, 3]);
```

Vrijedi istaknuti kako where funkcija može primiti dva ili tri parametra. Kada prima dva parametra, prvi parametar je naziv stupca iz tablice, a drugi je vrijednost te se

pretpostavlja da želite ispitati gdje vam je taj uvjet jednakoj toj vrijednosti tj. koristi se operator „=“. Ukoliko želite ispitati gdje su vrijednosti primjerice manje ili veće onda drugi parametar postaje sam operator (npr. „<“, „>“, „<=“, „>=“, „!=“ itd.), a treći parametar postaje zadana vrijednost. Dakle izostavljanjem operatora kao drugog parametra automatski se pretpostavlja korištenje operatora „=“.

Za razliku od agregatnih funkcija (u gornjim primjerima count i max) koji vraćaju skalarne vrijednosti, ostale funkcije vraćaju instancu tog modela ako se radi o jednoj n-torci ili kolekciju (objekt Illuminate\Database\Eloquent\Collection klase) instanci tog modela ako se radi o više n-torci kao rezultatu upita.

1.1.6.2. Laravel „Collection“ klasa

Laravel kolekcije su izrazitno korisna stvar jer se zapravo radi o klasi za rad s poljima za koje su implementirane razne vrlo korisne metode kako bi nam olakšao rad.

Najjednostavniji primjeri rada s kolekcijama:

```
$collection = collect([1, 2, 3]); // stvaranje kolekcije iz običnog polja

// stvaranje kolekcije i izračunavanje prosječnog broja stranica
$collection = collect([
    ['name' => 'Easy Laravel 5 – Jason Gilmore', 'pages' => 300],
    ['name' => 'Laravel Up & Running – Matt Stauffer', 'pages' => 284],
]);

$collection->avg('pages'); // 292

// stvaranje kolekcije i provjera da li postoji određena stvar u polju
$collection = collect(['name' => 'Laravel 5.1 Beauty', 'price' => 20]);
$collection->contains('Laravel 5.1 Beauty'); // true
$collection->contains('Learning Laravel 5'); // false

// brojanje koliko polje ima elemenata
$collection = collect([1, 2, 3, 4]);
$collection->count(); // 4

// ispis razlike između dvije kolekcije
$collection = collect([1, 2, 3, 4, 5]);
$diff = $collection->diff([2, 4, 6, 8]);
$diff->all(); // [1, 3, 5]

// metoda filter filtrira sadržaj kolekcije po zadanom uvjetu (u ovom slučaju vraća sve
elemente veće od 2)
```

```php
$collection = collect([1, 2, 3, 4]);
$filtered = $collection->filter(function ($value, $key) {
    return $value > 2;
});
$filtered->all(); // [3, 4]

// metoda first daje prvi element kolekcije koji zadovoljava uvjet (u ovom slučaju vraća 3)
collect([1, 2, 3, 4])->first(function ($key, $value) {
    return $value > 2;
});

// metoda get vraća vrijednost pohranjenu na traženom ključu (ako taj ključ ne postoji vraća null)
$collection = collect(['name' => 'taylor', 'framework' => 'laravel']);
$value = $collection->get('name'); // taylor
$value = $collection->get('email'); // null

// pluck metoda vraća sve vrijednosti iz kolekcije za traženi ključ
$collection = collect([
    ['product_id' => 'prod-100', 'name' => 'Desk'],
    ['product_id' => 'prod-200', 'name' => 'Chair'],
]);
$plucked = $collection->pluck('name');
$plucked->all(); // ['Desk', 'Chair']

// search metoda ispituje da li se u kolekciji nalazi tražena vrijednost te vraća njezin ključ (ili false ako ne postoji)
$collection = collect([2, 4, 6, 8]);
$collection->search(4); // 1
$collection->search(5); // false

/* map metoda iterira kroz kolekciju, proslijeđuje vrijednost anonimnoj funkciji koja tu vrijednost
može modificirati te ju vratiti nazad (i na taj način stvara novu kolekciju) */
$collection = collect([1, 2, 3, 4, 5]);
$multiplied = $collection->map(function ($item, $key) {
    return $item * 2;
});
$multiplied->all(); // [2, 4, 6, 8, 10]
```

Sve ove metode se mogu ulančano pozivati (engl. method chaining). Ovo je samo dio dostupnih metoda koje nudi Laravel „Collection“ klasa, a sve ostale su također izvrsno dokumentirane na Laravel službenoj stranici.

1.1.6.3. Eloquent odnosi između modela

Ukoliko želimo primjerice dobiti za ispis nekog korisnika sa svim njegovim forum postovima koristeći Eloquent, moramo prvo definirati odnos/relaciju (engl. relationships) između modela „User“ i modela „ForumPost“. U ovom slučaju, jedan korisnik može imati više postova pa trebamo definirati 1:N relaciju (engl. "one-to-many" relationship“). To se radi tako da se u model „User“ doda slijedeća metoda:

```
/**
 * Get all posts of the user.
 */
public function posts()
{
    return $this->hasMany(ForumPost::class);
}
```

Eloquent će na ovaj način pretpostaviti da je strani ključ (engl. foreign key) na kojeg se veže user_id (dakle na naziv modela koji je vlasnik doda „_id“). Ukoliko to nije slučaj kao drugi parametar hasMany prima strani ključ te tablice. Eloquent također pretpostavlja da je naziv lokalnog ključa na kojeg se spaja na modelu vlasnika „id“, ako to nije slučaj može se kao treći parametar predati i njegova vrijednost.

Primjer korištenja za ispis:

```
// nađi sve forum postove korisnika s ID-om 1
$posts = User::find(1)->posts;

foreach ($posts as $post) {
    // ispis
}
```

Eloquent nudi i mogućnost filtracije s npr. where metodom:

```
// nađi prvi forum post korisnika s ID-om 1 kojemu je naziv posta jednak TVZ
$post = User::find(1)->posts()->where('title', 'TVZ')->first();
```

Ovdje je važno napomenuti da za razliku od prvog primjera ispisa, ovdje se posts poziva kao metoda kako bi se dobila Laravel Query Builder instanca na koju se onda

mogu pozivati sve njene metode ulančavanjem kao npr. u gornjem primjeru where i first. Sve te metode su izvrsno dokumentirane na Laravel službenoj stranici.

Može se definirati i inverzna relacija. Tako npr. svaki forum post ima svog (jednog) autora. To se u „ForumPost“ modelu definira s belongsTo metodom:

```
/**
 * Get the author of the post.
 */
public function author()
{
    return $this->belongsTo(User::class);
}
```

Ispis izgleda ovako:

```
// ispis imena korisnika koji je napisao forum post koji ima ID 1
$post = ForumPost::find(1);
echo $post->author->name;
```

Eloquent podržava 1:1 (engl. one to one) relacije (hasOne i belongsTo metode), 1:N (engl. one to many) relacije (hasMany i belongsTo metode), M:N (engl. many to many) relacije (belongsToMany metoda) te također podržava i polimorfne relacije.

1.1.6.4. N + 1 problem

Kada pozivate Eloquent relacije kao atribut, relacija je lijeno učitana (engl. lazy loaded). To znači da se relacijski podaci nisu zapravo učitali sve dok niste pristupili tom atributu. Eloquent može također i željno učitati (engl. eager loading) relacije. To riješava N + 1 problem. Ovaj problem možemo ilustrirati na primjeru s gore već definiranim odnosima između „User“ i „ForumPost“ modela.

Primjer dohvaćanja svih forum postova i njihovih autora:

```
$posts = ForumPost::all();

foreach ($posts as $post) {
    echo $post->author->name;
}
```

Ovdje će se izvršiti jedan SQL upit (engl. query) da bi se dohvatili svi forum postovi i onda još po jedan upit za svaki forum post kako bi se dohvatio njegov autor.

To znači da ako imamo 50 forum postova da će se ovdje izvršiti 51 upit prema bazi podataka: 1 za dohvatiti sve forum postove i 50 dodatnih upita za dohvatiti autora za svaki pojedini post. Sa željnim učitavanjem možemo svesti broj upita na samo 2. To se radi ovako:

```
$posts = ForumPost::with('author')->get();

foreach ($posts as $post) {
    echo $post->author->name;
}
```

Dakle, koristi se with metoda koja kao parametar prima točan naziv metode iz modela kojom smo definirali odnos prema drugom modelu. U našem slučaju se radi o author metodi pa je u with predano „author“. Na ovaj se način izvršavaju samo 2 upita prema bazi:

```
SELECT * FROM forum_posts;

SELECT * FROM users WHERE id IN (1, 2, 3, 4, 5, ...);
```

Moguće je željno učitati i više relacija, a to se radi na način da se with metodi predaju dodatni argumenti:

```
$posts = ForumPost::with('author', 'some_other_relationship')->get();
```

Eloquent pruža i mogućnost željnog učitavanja ugnježđenih relacija pomoću dotacije s točkom. Primjer za željno učitavanje svih autora postova i sve kontakte autora posta:

```
$posts = ForumPost::with('author.contacts')->get();
```

Laravel pruža i mogućnost željnog učitavanja s dodavanjem uvjeta. Npr. željno učitavanje svih korisnika sa svim njihovim postovima koji u svom naslovu sadrže riječ TVZ.

```
$users = User::with(['posts' => function ($query) {
    $query->where('title', 'like', '%TVZ%');
}])->get();
```

1.1.7. Query Builder

Bitno je za naglasiti da za rad s bazom podataka osim opcije rada s Eloquentom i modelima postoji i Laravelov tzv. Query Builder. Primjer dohvaćanja svih korisnika:

```
// Query Builder način
$users = DB::table('users')->get();
// Eloquent način
$users = User::all();
```

Ovo su dakle dva različita način za dobiti istu stvar. Važno je napomenuti da za razliku od Eloquenta koji vraća objekt „Collection“ klase, Query Builder vraća obično PHP polje. Već je najavljeno da će od Laravel verzije 5.3 i Query Builder vraćati objekt „Collection“ klase.

Query Builder ne radi s modelima pa samim time nije moguće definirati relacije između tih modela, nego se koristi JOIN sintaksa za spajanje tablica.

```
// dohvaćanje svih korisnika (koji imaju barem 1 post) i njihovih postova
$users = DB::table('users')
        ->join('forum_posts', 'users.id', '=', 'forum_posts.user_id')
        ->select('users.*', 'forum_posts.title', 'forum_posts.content')
        ->get();
```

Sve Laravel Query Builder metode (a koje se kako je već rečeno mogu koristiti i s Eloquentom) su odlično dokumentirane na Laravel službenim stranicama.

Bitno je za napomenuti kako Query Builder i Eloquent koriste PDO pripremljene naredbe (engl. prepared statements) tako da je Laravel aplikacija zaštičena od SQL Injection napada.

1.1.8. Migracije

Migracije su mehanizan u Laravelu koji služi kako bi svi članovi tima koji rade na aplikaciji u svakom trenutku imali istu strukturu baze podataka. Migracija se stvara Artisan naredbom (o njemu će više biti napisano u idućem poglavlju):

```
php artisan make:migration create_forum_posts_table
```

Pokretanjem ove naredbe stvorit će se migracija u database/migrations folderu. Svaka migracija na početku svog naziva ima vremenski žig (engl. timestamp) kako bi Laravel znao kojim redosljedom da izvršava migriranje.

Primjer migracije:

```php
<?php

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

class CreateForumPostsTable extends Migration
{
    /**
     * Run the migrations.
     *
     * @return void
     */
    public function up()
    {
        // prvi parametar je naziv tablice
        Schema::create('forum_posts', function (Blueprint $table) {
            // stupac id - autoinkrementirajući primarni ključ
            $table->increments('id');
            // stupac author_user_id - pozitivni INT(10) uz stvaranje osnovnog indeksa
            $table->integer('author_user_id')->unsigned()->index();
            // definiranje author_user_id kao strani ključ koji se referencira na id stupac users tablice i pri izmnjenjivanju ili brisanju iz users tablice i on se mijenja/briše
            $table->foreign('author_user_id')->references('id')->on('users')->onUpdate('cascade')->onDelete('cascade');
            // stupac title tipa VARCHAR(255), kao drugi parametar prima dužinu, podrazumijevana vrijednost je 255
            $table->string('title');
            // stupac content tipa TEXT
            $table->text('content');
            // stvara stupce created_at i updated_at tipa TIMESTAMP koje postavlja Eloquent kod stvaranja odnosno izmjenjivanja
            $table->timestamps();
        });
    }

    /**
     * Reverse the migrations.
     *
     * @return void
     */
    public function down()
```

```
    {
        // dropa cijelu tablicu
        Schema::drop('forum_posts');
    }
}
```

U up metodi se definira što se radi kod pokretanja migracije, a u down metodi što se radi kad se migracija želi poništiti odnosno vratiti na prethodno stanje (engl. roll back).

Migriranje se pokreće Artisan naredbom:

```
php artisan migrate
```

Ukoliko se želi poništiti samo zadnje migriranje koristi se rollback naredba:

```
php artisan migrate:rollback
```

Reset naredba poništava sve već učinjene migracije:

```
php artisan migrate:reset
```

Ukoliko se želi poništiti sve učinjene migracije te nanovo sve migrirati koristi se refresh naredba:

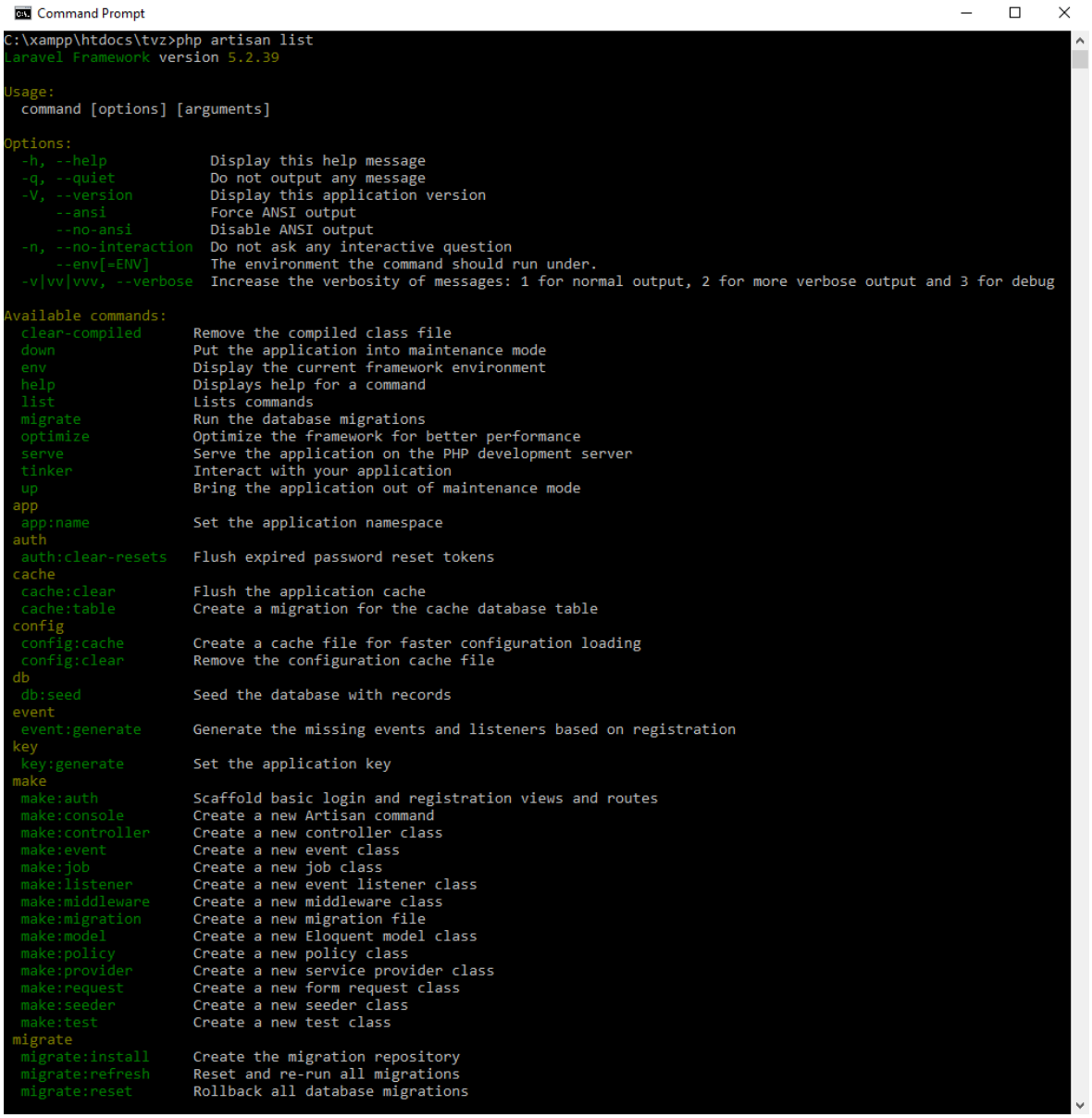
```
php artisan migrate:refresh
```

1.1.9. Artisan

Artisan je naziv konzolnog alata koji dolazi s Laravelom. Ima mnogo naredbi s kojima se ubrzava proces izrade aplikacije (npr. izrada migracija, modela, kontrolera, događaja, migriranje baze podataka itd.). Kako bi se koristio Artisan mora se biti pozicioniran u Laravel folderu. Za ispis svih naredbi postoji naredba:

```
php artisan list
```

To izgleda ovako:

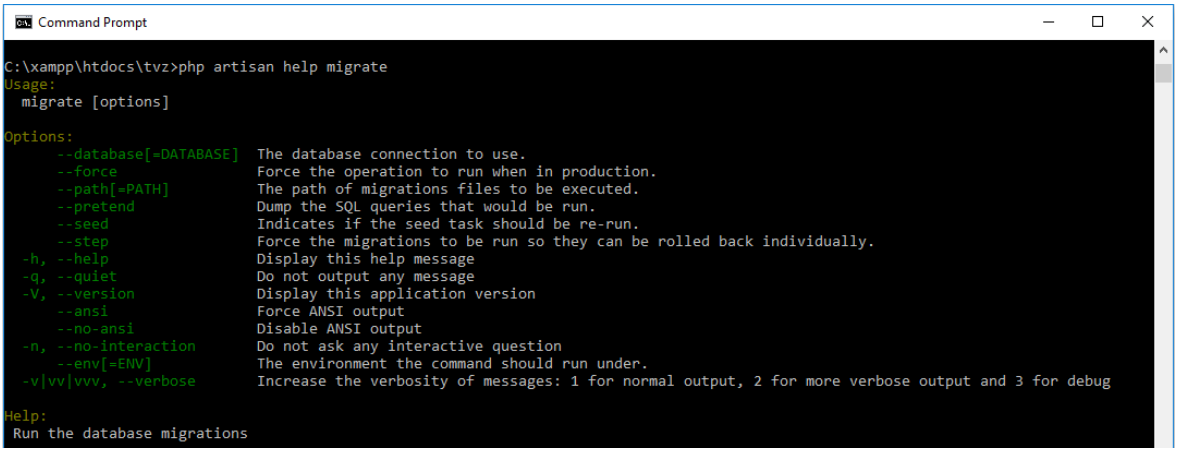


Slika 3: Laravel Artisan prikaz ispisa dijela naredbi

Svaka naredba ima ekran za pomoć gdje je opisana naredba sa svim njenim mogućim argumentima i opcijama. Kako bi se vidio taj ekran za pojedinu naredbu, postoji naredba:

```
php artisan help migrate
```

gdje je „migrate“ naziv naredbe za koju želimo vidjeti taj ekran (u ovom slučaju se dakle radi o „migrate“ naredbi za migriranje baze podataka).



Slika 4: Laravel Artisan prikaz ekrana pomoći za migrate naredbu

Laravel pruža mogućnost korisniku da implementira nove Artisan naredbe. Artisan naredbe se osim iz konzole mogu pozvati i iz same aplikacije.

1.1.10. Middleware

Laravelov međusoftver (engl. middleware) je mehanizam filtriranja HTTP zahtjeva. Više njih dolazi sa samim Laravelom, poput međusoftvera za autentifikaciju, održavanje, CSRF zaštitu itd. Svi se oni nalaze u app/Http/Middleware folderu. Novi međusoftver se stvara s Artisan naredbom:

```
php artisan make:middleware AgeMiddleware
```

gdje je „AgeMiddleware“ naziv međusoftvera. Izvršavanjem ove naredbe se stvara AgeMiddleware klasa u app/Http/Middleware folderu. Primjer AgeMiddlewarea:

```php
<?php

namespace App\Http\Middleware;

use Closure;

class AgeMiddleware
{
    /**
     * Handle an incoming request.
     *
     * @param  \Illuminate\Http\Request  $request
     * @param  \Closure  $next
     * @return mixed
```

```
     */
    public function handle($request, Closure $next)
    {
        if ($request->input('age') <= 17) {
            return redirect('home');
        }

        return $next($request);
    }
}
```

Svaki međusoftver ima handle metodu u kojoj se definira što on radi. U ovom slučaju se dopušta pristup ruti samo ako je predan „age“ koji je veći od 17. Inače se korisnik preusmjerava na „home“ URI. Ako je „age“ veći od 17, onda se taj zahtjev proslijeđuje dalje u aplikaciju (da ga može obraditi/provjeriti idući međusoftver ako postoji). Dakle to je sustav slojeva gdje HTTP zahtjev mora uspješno proći kroz sve slojeve kako bi Laravel dopustio da se uopće dođe do same aplikacije i njenog odgovora.

Moguće je definirati međusoftver koji se pokreće prije ili nakon obrađivanja zahtjeva od same aplikacije. Primjer međusoftver handle metode koja se prvo izvrši pa tek onda eventualno dođe do obrađivanja zahtjeva:

```
public function handle($request, Closure $next)
{
    // izvrši kod

    return $next($request);
}
```

Ukoliko se želi prvo obraditi zahtjev pa tek onda izvršiti kod samog međusoftvera definira se handle metoda na idući način:

```
public function handle($request, Closure $next)
{
    $response = $next($request);

    // izvrši kod

    return $response;
}
```

Kako bi Laravel prepoznao definirani međusoftver mora ga se registrirati. Moguće ga je registrirati kao globalni međusoftver koji će se izvršavati prilikom svakog HTTP zahtjeva. To se radi na način da se putanja do tog međusoftvera doda u $middleware atribut u app/Http/Kernel.php klasi. Ovako to izgleda na početku:

```
/**
 * The application's global HTTP middleware stack.
 *
 * These middleware are run during every request to your application.
 *
 * @var array
 */
protected $middleware = [
    \Illuminate\Foundation\Http\Middleware\CheckForMaintenanceMode::class,
];
```

Ako se želi definirati međusoftver koji će se izvršavati samo na pojedinim rutama, potrebno mu je prvo dodijeliti kratko ime tj. ključ u $routeMiddleware atributu u app/Http/Kernel.php. To za npr. AgeMiddleware izgleda ovako:

```
/**
 * The application's route middleware.
 *
 * These middleware may be assigned to groups or used individually.
 *
 * @var array
 */
protected $routeMiddleware = [
    'auth' => \App\Http\Middleware\Authenticate::class,
    'auth.basic' => \Illuminate\Auth\Middleware\AuthenticateWithBasicAuth::class,
    'can' => \Illuminate\Foundation\Http\Middleware\Authorize::class,
    'guest' => \App\Http\Middleware\RedirectIfAuthenticated::class,
    'throttle' => \Illuminate\Routing\Middleware\ThrottleRequests::class,
    'age' => \App\Http\Middleware\AgeMiddleware::class,
];
```

Pridružuje ga se pojedinoj ruti na idući način:

```
Route::get('index', ['middleware' => 'age', function () {
    //
}]);
```

Na svaku rutu je moguće dodati više međusoftvera:

```
Route::get('index', ['middleware' => ['auth', 'age'], function () {
    //
}]);
```

Primjer kako to radi, s time da handle metoda sada izgleda ovako:

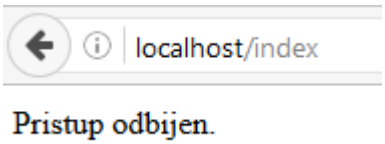
```
/**
 * Handle an incoming request.
 *
 * @param  \Illuminate\Http\Request  $request
 * @param  \Closure  $next
 * @return mixed
 */
public function handle($request, Closure $next)
{
    if ($request->input('age') <= 17) {
        return response('Pristup odbijen.', 403);
    }

    return $next($request);
}
```

Umjesto redirektiranja korisnika na „home“ URI sada mu se vraća odgovor „Pristup odbijen“ s HTTP statusnim kodom 403 – zabranjen pristup (engl. forbidden).

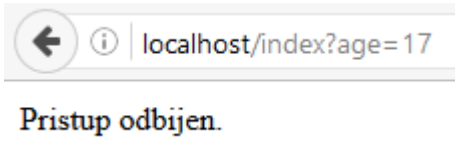
Ruta izgleda ovako:

```
Route::get('index', ['middleware' => 'age', function () {
    return "Uspjeh";
}]);
```

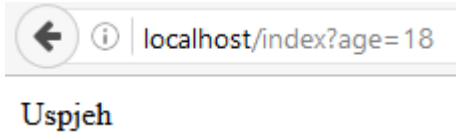
Dakle, u slučaju da zahtjev prođe kroz međusoftver korisniku se na ekran ispisuje „Uspjeh“, a u suprotnom mu se ispisuje „Pristup odbijen“.



Slika 5: Primjer kada uopće nije predan „age“ parametar



Slika 6: Primjer kada je age=17



Slika 7: Primjer kada je age=18

Laravel 5.2 donosi kao novost međusoftverske grupe. To znači da se više međusoftvera može definirati unutar jedne grupe te onda umjesto pisanja svih njih posebno kod dodjeljivanja ruti, dovoljno je napisati samo ime grupe. Te međusoftverske grupe su definirane u app/Http/Kernel.php pod atributom $middlewareGroups. U Laravelu 5.2 to izgleda ovako:

```
/**
 * The application's route middleware groups.
 *
 * @var array
 */
protected $middlewareGroups = [
    'web' => [
        \App\Http\Middleware\EncryptCookies::class,
        \Illuminate\Cookie\Middleware\AddQueuedCookiesToResponse::class,
        \Illuminate\Session\Middleware\StartSession::class,
        \Illuminate\View\Middleware\ShareErrorsFromSession::class,
        \App\Http\Middleware\VerifyCsrfToken::class,
    ],

    'api' => [
        'throttle:60,1',
    ],
];
```

Dakle, postoje dvije definirane grupe međusoftvera, a to su web i api. U web grupu pripadaju međusoftveri za rad s kolačićima (engl. cookies), sesijama, CSRF zaštitu itd. Api grupa sadržava samo throttle međusoftver koji (s trenutnim

postavkama) dozvoljava 60 pokušaja po minuti nakon čega uskraćuje pristup na jednu minutu.

Laravel automatski stavlja web međusoftversku grupu na rute iz app/Http/routes.php datoteke.

Osim pridruživanja međusoftvera na pojedine rute moguće ih je dodijeliti u konstruktoru kontrolera pa bi bio pozvan kod svih metoda tog kontrolera (moguće je specificirati i ako ih hoćemo dodijeliti samo na neke specifične metode):

```
/**
 *
 * @return void
 */
public function __construct()
{
    // međusoftver auth se pridružuje svim metodama
    $this->middleware('auth');

    // na ovaj način se pridružuje samo index i create metodama
    $this->middleware('auth', ['only' => [
        'index',
        'create',
    ]]);

    // na ovaj način se pridružuje svim metodama u kontroleru osim store metodi
    $this->middleware('auth', ['except' => [
        'store'
    ]]);
}
```

1.1.11. Događaji

Laravel pruža vrlo jednostavnu mogućnost da se može pretplatiti i slušati kada će se dogoditi neki događaj (engl. event) u aplikaciji. „Event“ klase se nalaze u app/Event, a njihovi slušatelji (engl. listeners) se nalaze u app/Listeners folderu.

Događaji se registriraju u EventServiceProvider klasi koja se nalazi u app/Providers folderu (treba ih upisat pod $listen atribut). Primjer:

```
/**
 * The event listener mappings for the application.
 *
 * @var array
```

```
 */
protected $listen = [
    'App\Events\ThreadWasOpenedEvent' => [
        'App\Listeners\ThreadWasOpenedEventListener',
    ],
];
```

Nakon toga se pokrene Artisan naredba:

```
php artisan event:generate
```

Ova naredba generira sve (nove) događaje/slušatelje koji su navedeni u $listen atributu (već generirane ne dira).

Primjer događaja:

```
<?php

namespace App\Events;

use App\Events\Event;
use Illuminate\Queue\SerializesModels;
use Illuminate\Contracts\Broadcasting\ShouldBroadcast;
use App\ForumThread;

class ThreadWasOpenedEvent extends Event
{
    use SerializesModels;

    public $thread;

    /**
     * Create a new event instance.
     *
     * @param  ForumThread  $thread
     * @return void
     */
    public function __construct(ForumThread $thread)
    {
        $this->thread = $thread;
    }

    /**
     * Get the channels the event should be broadcast on.
     *
     * @return array
     */
```

```
    public function broadcastOn()
    {
        return [];
    }
}
```

Ova klasa događaja ne sadržava nikakvu logiku, to je jednostavno kontejner (engl. container) za ForumThread objekt kojeg je korisnik otvorio. Primjer slušatelja:

```
<?php

namespace App\Listeners;

use App\Events\ThreadWasOpenedEvent;
use Illuminate\Queue\InteractsWithQueue;
use Illuminate\Contracts\Queue\ShouldQueue;

class ThreadWasOpenedEventListener
{
    /**
     * Create the event listener.
     *
     * @return void
     */
    public function __construct()
    {
        //
    }

    /**
     * Handle the event.
     *
     * @param  ThreadWasOpenedEvent  $event
     * @return void
     */
    public function handle(ThreadWasOpenedEvent $event)
    {
        // ovdje se može pristupiti forum threadu kojeg je korisnik otvorio koristeći $event->thread...
    }
}
```

U ovoj handle metodi se implementira sva logika koju je potrebno izvršiti kao odgovor na događaj, npr. povećati broj pogleda za tu temu u bazi podataka. Pokretanje događaja se vrši preko fire metode (iz npr. neke metode iz kontrolera):

```
Event::fire(new ThreadWasOpenedEvent($thread));
```

Moguće ga je pokrenuti i s globalnom pomoćnom (engl. helper) funkcijom event:

```
event(new ThreadWasOpenedEvent($thread));
```

1.1.12. Rad s formama

Primjer za spremanje novog forum posta u bazu – prvo je potrebno definirati rute:

```
// prikaz pogleda s formom za stvaranje novog forum posta
Route::get('post/create', 'ForumPostController@create');

// spremanje forum posta u bazu
Route::post('newPost', 'ForumPostController@store');
```

Izgled create metode:

```
/**
 * Show the form to create a new forum post.
 *
 * @return Response
 */
public function create()
{
    // vraćanje pogleda koji sadrži formu za kreiranje novog forum posta
    return view('post.create');
}
```

Napomena: unutar te forme mora biti generiran input s tokenom zbog CSRF zaštite – u Bladeu se samo unutar forme napiše {{ csrf_field() }} kako bi se to dobilo.

Početni izgled store metode:

```
/**
 * Store a new forum post.
 *
 * @param  Request  $request
 * @return Response
 */
public function store(Request $request)
{
    // ovdje ide validacija i spremanje forum posta u bazu
}
```

Store metoda prima objekt Illuminate\Http\Request klase koji sadržava sve podatke koji su poslani preko forme. Primjer s osnovnom validacijom:

```php
/**
 * Store a new forum post.
 *
 * @param  Request  $request
 * @return Response
 */
public function store(Request $request)
{
    $validator = Validator::make($request->all(), [
        'title' => 'required|unique:forum_posts|max:255',
        'body' => 'required',
    ]);

    if ($validator->fails()) {
        return redirect('post/create')
            ->withErrors($validator)
            ->withInput();
    }

    // validacija je prošla, spremi forum post u bazu...
}
```

Prvi argument koji je predan make metodi su podaci koje treba validirati. To je u ovom slučaju $request->all() što predstavlja sve podatke koji su poslani kroz formu. Drugi argument su validacijska pravila za pojedine unose. U ovom slučaju je za „title“ postavljeno da je obavezan, da mora biti unikatan u tablici „forum_posts“ te da smije imati najviše 255 znakova, dok je za „body“ postavljeno samo da je obavezan. Ukoliko validacija ne prođe korisnika se vraća na prethodni pogled s formom gdje je unosio te podatke. Zbog withInput metode je korisniku već sve popunjeno onako kako je on to popunio odnosno poslao na server. WithErrors metoda postavlja greške za stavke koje nisu prošle validaciju u sesiju kako bi onda mogle biti ispisane korisniku u pogledu. WithErrors metoda prihvaća instancu validatora, MessageBag instancu ili PHP polje kao parametar. Primjer izgleda pogleda koji ispisuje te greške ako one postoje:

```php
<!-- /resources/views/post/create.blade.php -->

<h1>Create Forum Post</h1>
```

```
@if (count($errors) > 0)
    <div class="alert alert-danger">
        <ul>
            @foreach ($errors->all() as $error)
                <li>{{ $error }}</li>
            @endforeach
        </ul>
    </div>
@endif

<!-- Create Post Form -->
```

Razlog zašto je $errors varijabla dostupna u pogledu je Illuminate\View\Middleware\ShareErrorsFromSession međusoftver koji je dio web grupe. On povezuje greške u pogled (u $errors varijablu) koje u sesiju sprema withErrors metoda.

Laravel ima svoje podrazumijevane greške koje ispisuje u slučaju da neki dio validacije ne prođe. To se također može definirati da se ispisuje nešto drugo:

```
/**
 * Store a new forum post.
 *
 * @param  Request  $request
 * @return Response
 */
public function store(Request $request)
{
    $input = $request->all();

    // definiranje pravila
    $rules = [
        'title' => 'required|unique:forum_posts|max:255',
        'body' => 'required',
    ];

    // definiranje koja će poruka biti ispisana ako neko od tih pravila ne prođe validaciju
    $messages = [
        'title.required' => 'The post title is required.',
        'title.unique' => 'This post title already exists.',
        'title.max' => 'The post title is too long. Max length is 255 characters.',
        'body.required' => 'The forum post body is required.',
    ];
```

```
    $validator = Validator::make($input, $rules, $messages);

    if ($validator->fails()) {
        return redirect('post/create')
            ->withErrors($validator)
            ->withInput();
    }

    // validacija je prošla, spremi forum post u bazu...
}
```

Ovaj put se još posebno definiraju pravila (koja make metoda prima kao drugi parametar) te poruke koje će biti ispisane ako neko od tih pravila ne prođe validaciju (make metoda to prima kao treći parametar).

Ako HTTP zahtjev sadrži ugnježdene (engl. nested) parametre, može ih se specificirati sa sintaksom s točkom (engl. "dot" syntax):

```
$rules = [
    'title' => 'required|unique:forum_posts|max:255',
    'body' => 'required',
    'author.name' => 'required',
    'author.description' => 'required|min:30',
];
```

Novost u Laravelu 5.2 je i olakšana validacija kod forme s poljima (engl. form array validation).

```
$rules = [
    // email mora biti ispravno formatiran te mora biti unikatan u tablici users
    'author.*.email' => 'email|unique:users',
    // first_name je obavezan za svakog autora, kao i last_name
    'author.*.first_name' => 'required_with:author.*.last_name',
];
```

Na isti način se koristi znak „*“ kod specificiranja koja će biti poruka ako neko od tih pravila ne prođe validaciju.

Ukoliko se radi o AJAX zahtjevu te ako ne prođe validacija, Laravel generira JSON odgovor s HTTP 422 statusnim kodom koji sadržava sve validacijske greške.

Kod složenijih validacija se mogu napraviti tzv. „form request“ klase. To su korisnički definirane klase koje nasljeđuju „Request“ klasu te sadrže validacijsku logiku. Kako bi ih kreirali koristi se make:request Artisan naredba:

```
php artisan make:request StoreForumPostRequest
```

StoreForumPostRequest klasa će biti generirana u app/Http/Requests folderu. Tu postoji rules metoda gdje se definiraju validacijska pravila:

```php
/**
 * Get the validation rules that apply to the request.
 *
 * @return array
 */
public function rules()
{
    return [
        'title' => 'required|unique:forum_posts|max:255',
        'body' => 'required',
    ];
}
```

Sve što se treba napraviti da se koristi ova klasa za validaciju je u store metodi iz kontrolera staviti da umjesto Request prima StoreForumPostRequest objekt:

```php
/**
 * Store a new forum post.
 *
 * @param  StoreForumPostRequest  $request
 * @return Response
 */
public function store(StoreForumPostRequest $request)
{
    // samim ulaskom u tijelo ove metode znači da je validacija prošla
}
```

Artisan osim rules metode generira i authorize metodu. Ona provjerava da li je korisnik koji je autentificiran (ulogiran) ovlašten (autoriziran) da mijenja sadržaj sa svojim zahtjevom, npr. kada korisnik hoće promijeniti sadržaj svog forum posta – to smije biti dozvoljeno samo njemu i još eventualno forumskim administratorima. Primjer rute za izmjenu forum posta te authorize metode:

```php
Route::post('post/{post}');
```

```php
/**
 * Determine if the user is authorized to make this request.
 *
 * @return bool
 */
public function authorize()
{
    $postId = $this->route('post');

    return ForumPost::where('id', $postId)
        ->where('user_id', Auth::id())->exists();
}
```

Route metoda u ovom primjeru dohvaća sadržaj {post} parametra iz rute. Npr. za rutu /post/1 bi $postId poprimio vrijednost 1. Authorize metoda vraća true (istinu) ako forum post pripada ulogiranom korisniku (njegov id se dobiva s Auth::id()) što znači da je korisnik autoriziran za ovu izmjenu, u suprotnom vraća false (laž) što znači da korisnik nije autoriziran napraviti ovu izmjenu. U slučaju da authorize metoda vrati false tj. laž, klijentu će automatski biti poslan odgovor s HTTP 403 statusnim kodom i store metoda iz kontrolera se neće izvršiti. Ukoliko se planira autorizacijska logika napisati negdje drugdje onda je potrebno u authorize metodu postaviti da vraća true tj. istinu.

1.1.13. Rad s kešom

Laravel nudi jednak API za rad s raznim sistemima za rad s kešom (engl. cache). S kojim sistemom se želi raditi se definira u .env datoteci (ključ CACHE_DRIVER). Podrazumijevana vrijednost mu je file, ali on još može biti database, memcached i redis. Vrijednost se u keš stavlja pod nekim ključem te joj se daje vrijeme u minutama koliko će biti zapamćena:

```php
Cache::put('ključ', 'vrijednost', $minute);
```

Umjesto na koliko minuta će biti zapamćena vrijednost, moguće je kao treći parametar predati DateTime instancu koja predstavlja točno vrijeme do kada će vrijednost biti zapamćena.

Ukoliko je potrebno dodati vrijednost pod ključ samo u slučaju da on već ne postoji koristi se add metoda (ako je dodana nova vrijednost vraća true, inače vraća false):

```
Cache::add('ključ', 'vrijednost', $minute);
```

Vrijednost je moguće spremiti u keš zauvijek:

```
Cache::forever('ključ', 'vrijednost');
```

Tako spremljene vrijednosti se brišu iz keša pozivom forget metode:

```
Cache::forget('ključ');
```

Ukoliko želimo obrisati sve što se nalazi u kešu koristimo flush metodu:

```
Cache::flush();
```

Vrijednost se iz keša dobiva s get metodom koja prima kao parametar naziv ključa pod kojim je spremljena:

```
$vrijednost = Cache::get('ključ');
```

Ukoliko ne postoji ta vrijednost odnosno ključ, get metoda vraća null.

Želimo li dohvatiti vrijednost iz keša, ali i spremiti neku vrijednost ako tražena vrijednost/ključ ne postoje koristi se remember funkcija:

```
$vrijednost = Cache::remember('korisnici', $minute, function() {
    return User::all();
});
```

U ovom slučaju, ako ne postoji vrijednost odnosno ključ „korisnici“, bit će vraćen rezultat User:all() (svi korisnici iz baze podataka) te će to biti i zapisano pod ključ „korisnici“ na onoliko minuta koliko to definiramo. Ukoliko želimo spremiti tu vrijednost zauvijek koristimo metodu rememberForever koja ne prima parametar za minute, a sve ostalo je jednako.

Ako se želi dohvatiti neka vrijednost te ju odmah i obrisati iz keša koristi se pull metoda:

```
$vrijednost = Cache::pull('ključ');
```

Kao i kod get metode, ako traženi ključ ne postoji bit će vraćena null vrijednost.

Ako se kao sistem za keširanje koristi memcached ili Redis, onda na raspolaganju imamo i opciju keš tagova (engl. cache tags). Oni nam dozvoljavaju da označimo povezane stvari s imenom (što kasnije omogućava da ih s flush metodom možemo sve odjednom obrisati, a da pritom ne obrišemo sav keš).

Postavljanje keš tagova se vrši na idući način:

```
// pod tagove ljudi i pjevaci stavi ključ Ivan s vrijednosti $ivan na duljinu od $minute
Cache::tags(['ljudi', 'pjevaci'])->put('Ivan', $ivan, $minute);

// pod tagove ljudi i autori stavi ključ Ana s vrijednosti $ana na duljinu od $minute
Cache::tags(['ljudi', 'autori'])->put('Ana', $ana, $minute);
```

Osim put, može se koristiti i forever i add metode.

Za dohvaćanje tagiranih keš elemenata treba se predati jednak popis tagova u tags metodu kao i kod unosa:

```
$ivan = Cache::tags(['ljudi', 'pjevaci'])->get('Ivan');
$ana = Cache::tags(['ljudi', 'autori'])->get('Ana');
```

Mogu se pobrisati svi elementi koji su dodijeljeni pojedinom tagu ili listi tagova. Na primjer:

```
// briše sve elemente koji su tagirani s bilo kojim od navedenih tagova - i Ivan i Ana bi bili obrisani
Cache::tags(['ljudi', 'autori'])->flush();
```

```
// briše samo one elemente koji su tagirani kao autori što znači da bi Ana bila obrisana, a Ivan bi ostao
Cache::tags('autori')->flush();
```