

TEHNIČKO VELEUČILIŠTE U ZAGREBU

STRUČNI STUDIJ RAČUNARSTVA

Igor Gudelj

**IZRADA MREŽNIH STRANICA
KORIŠTENJEM LARAVEL RADNOG OKVIRA**

ZAVRŠNI RAD br. 531

Zagreb, veljača, 2015.

TEHNIČKO VELEUČILIŠTE U ZAGREBU

STRUČNI STUDIJ RAČUNARSTVA

Igor Gudelj

JMBAG: 0036440662

**IZRADA MREŽNIH STRANICA
KORIŠTENJEM LARAVEL RADNOG OKVIRA**

ZAVRŠNI RAD br. 531

Zagreb, veljača, 2015.

Sažetak diplomskog rada

Instalirati Laravel radno okruženje za izradu mrežnih stranica. Opisati cjelokupno Laravel sučelje za izradu mrežnih stranica te pokazati njihovu primjenu. Pokazati jednostavnost primjene ovakvog sučelja kod izrade mrežnih stranica. Izraditi mrežne stranice za rezervaciju apartmana te pripadajuću bazu podataka. Osmisliti sučelje za administraciju stranica s ciljem što lakšeg korištenja. Dokumentirati izvedeno rješenje.

Sadržaj

1. Uvod	1
2. Upoznavanje s radnim okvirom	2
2.1. Instalacija	2
2.2. Lokalna razvojna okolina.....	3
2.3. Struktura projekta.....	5
2.4. Aplikacijski direktorij	8
3. Rute	11
3.1. Osnove rutiranja.....	11
3.2 Pregledi (views).....	13
3.3. Filteri	14
4. Kontroleri	16
4.1. Kreiranje kontrolera.....	16
4.2. Rutiranje kontrolera.....	17
4.3. RESTful kontroleri	18
5. Forme	20
5.1. Otvaranje formi.....	20
5.2. Polja u formama	22
5.3. Dugmad u formama	27
5.4. Sigurnost formi	27
5.5. Validacija.....	28
6. Baze podataka.....	31
6.1. Konfiguracija baze podataka.....	31
6.2. Graditelj Shema	31
6.3. Migracije.....	33
7. Eloquent.....	36

7.1. Eloquent ORM.....	36
7.2. Eloquent kolekcije	36
7.3. Eloquent relacije.....	38
8. Izrada praktične aplikacije	41
8.1. Uvod i izgled aplikacije.....	41
8.2. Migracije i seed datoteke	44
8.3. Modeli i relacije	50
8.4. Rute.....	52
8.5. Pregledi (views).....	53
8.6. Kontroleri.....	53
8.7. Ostalo.....	55
8.8. Postavljanje na server.....	56
9. Zaključak.....	58
10. Literatura.....	59

1. Uvod

Laravel je radni okvir za PHP programski jezik. Za razliku od PHP-a koji nije poznat po lijepoj sintaksi, Laravel ima lijepu, semantičku i kreativnu sintaksu koja ga ističe među mnoštvom PHP radnih okvira. Lak je za korištenje, brzo se uči i pokreće mnogo stranica koje koristimo svaki dan. Laravel nam omogućava korisne prečice, alate i komponente koje nam uvelike olakšavaju i ubrzavaju razvoj internet aplikacija, a također ispravlja i neke mane PHP jezika.

Laravel radni okvir star je tek nešto više od dvije godine i pokušava kombinirati sve najbolje viđeno iz ostalih popularnih radnih okvira, uključujući i one koji nisu napisani PHP programskim jezikom, kao na primjer Ruby on Rails, ASP.NET MVC i Sinatra. Sve navedeno čini ga odličnim izborom kako za amaterske projekte tako i za *enterprise* rješenja, bez obzira da li ste profesionalac ili početnik.

Kroz ovaj rad поближе ćemo se upoznati s radnim okvirom i njegovim brojnim funkcionalnostima. U idućem poglavlju bit će riječi o instalaciji radnog okvira, konfiguraciji lokalne razvojne okoline, te strukturi Laravel projekta i njegovog aplikacijskog direktorija. U trećem poglavlju prikazan je način kreiranja ruta, te slanje podataka do pregleda (engl. *views*) u aplikaciji kao i dodjeljivanje filtera kreiranim rutama. U četvrtom su poglavlju prikazane osnovne stvari o kontrolerima koji predstavljaju logiku aplikacije. Nakon kontrolera pokazano je kako u pregledima kreirati i validirati najčešće korištene forme koje korisnici mogu ispunjavati u aplikaciji. U šestom poglavlju objašnjeno je koje baze podržava radni okvir, te kako ih konfigurirati. Također su prikazane Laravel migracije i *schema builder* kojim kreiramo migracije i strukturu baze podataka. U sedmom poglavlju je predstavljen *Eloquent ORM (Object-relational mapping)* alat koji uvelike olakšava rad s bazom podataka. Objašnjeni su i *Eloquent* modeli i relacije koje je potrebno postaviti između njih. U posljednjem poglavlju prikazan je postupak izrade praktične aplikacije i njeno postavljanje na server.

2. Upoznavanje s radnim okvirom

2.1. Instalacija

Laravel ima vlastiti Github repozitorij, koji se ponaša kao predložak za svaku novu Laravel aplikaciju. Za 'kloniranje' repozitorija u direktorij na lokalnom računalu koristi se git naredba:

```
git clone git@github.com:laravel/laravel.git moj_projekt
```

Izvršenjem naredbe kreiran je Laravel predložak unutar direktorija moj_projekt.

Datoteka composer.json unutar direktorija moj_projekt nakon instalacije izgleda ovako:

```
{
  "require": {
    "laravel/framework": "4.2.*"
  },
  "autoload": {
    "classmap": [
      "app/commands",
      "app/controllers",
      "app/models",
      "app/database/migrations",
      "app/database/seeds",
      "app/tests/TestCase.php"
    ]
  },
  "scripts": {
    "post-install-cmd": [
      "php artisan clear-compiled",
      "php artisan optimize"
    ],
    "post-update-cmd": [
      "php artisan clear-compiled",
      "php artisan ide-helper:generate",
      "php artisan optimize"
    ],
    "post-create-project-cmd": [
      "php artisan key:generate"
    ]
  },
  "config": {
    "preferred-install": "dist"
  },
  "minimum-stability": "stable"
}
```


Laravel aplikacija ovisi isključivo o paketu `laravel/framework`, koji sadrži sve komponente potrebne za pokretanje aplikacije. Datoteka `composer.json` dio je novonastale aplikacije i može se uređivati po potrebi. Međutim, neke osjetljive konfiguracijske postavke unaprijed su zadane.

Za sada u direktoriju `moj_projekt` postoji samo osnovni predložak. Naredbom `composer install` dohvaća se i instalira jezgra radnog okvira. Instalacija može potrajati par minuta zato što Laravel radni okvir koristi jako puno vanjskih paketa. Čemu svi oni služe? Laravel koristi snagu slobodnog (engl. *open source*) softvera - pregršt paketa koji postoje na *Composeru*. Ovi paketi su zavisnosti (engl. *dependencies*) samog radnog okvira i instalirani su u direktorij `vendor`. Ako se koristi git za verzioniranje kôda, zavisnosti instalirane u `vendor` direktoriju nije potrebno verzionirati. Iz tog razloga Laravel osigurava predložak datoteke `.gitignore` koja ignorira `vendor` direktorij i ostale logične vrijednosti.

Ovime je instaliran Laravel radni okvir. Sljedeći korak je podešavanje lokalne razvojne okoline.

2.2. Lokalna razvojna okolina

Laravel nastoji olakšati razvoj PHP aplikacija u svakom pogledu, uključujući i lokalnu razvojnu okolinu. Vagrant (<http://www.vagrantup.com/>) pruža jednostavan i elegantan način za upravljanje virtualnim mašinama.

Laravel Homestead je službena Vagrant kutija (engl. *box*) koja pruža prekrasno razvojno okruženje bez potrebe za instalacijom PHP-a, web poslužitelja i bilo kojeg drugog softvera na lokalnom računalu. Vagrant kutije namijenjene su za jednokratnu upotrebu. Ako nešto krene po zlu, kutija se može uništiti i ponovno stvoriti za nekoliko minuta!

Homestead radi na bilo kojem Windows, Mac i Linux sustavu. Pokreće ga Ubuntu 14.04, a uključuje Nginx web poslužitelj, PHP 5.5, MySQL, Postgres, Redis, Memcached i još mnogo drugih stvari potrebnih za olakšano razvijanje Laravel aplikacija.

Prije pokretanja Homestead okoline, potrebno je instalirati VirtualBox i Vagrant. Nakon instalacije, potrebno je dodati `laravel/homestead` kutiju na vlastitu Vagrant instalaciju pomoću sljedeće naredbe u terminalu:

```
vagrant box add laravel/homestead
```

Nakon toga potrebno je klonirati Homestead repozitorij na lokalno računalo koristeći naredbu:

```
git clone https://github.com/laravel/homestead.git Homestead
```

Klonirani repozitorij može poslužiti ne samo za jednu Laravel web aplikaciju, već za više njih.

Zatim je potrebno urediti *Homestead.yaml* datoteku uključenu u repozitoriju. U ovoj datoteci potrebno je podesiti putanju do vlastitog javnog SSH ključa, kao i direktorije koje želimo dijeliti između glavnog računala i Homestead virtualne mašine. Kako dohvatiti SSH ključ? Na Mac i Linux sustavima možemo stvoriti SSH privatni i javni ključ koristeći sljedeću naredbu:

```
ssh-keygen -t rsa -C "your@email.com"
```

Na Windows sustavu preporučeno je instalirati Git i iskoristiti Git Bash ljsku kako bi stvorili SSH ključeve koristeći gornju naredbu. Alternativno, istu stvar moguće je postići koristeći PuTTY i PuTTYgen.

Nakon kreiranja SSH ključeva, njihove putanje unose se u *Homestead.yaml* datoteku na predviđena mjesta.

U *folders* dijelu *Homestead.yaml* datoteke potrebno je odabrati sve direktorije koje želimo podijeliti s vlastitim Homestead okruženjem. Datoteke koje se promijene u navedenim direktorijima automatski će biti sinkronizirane između lokalnog računala i Homestead okoline (isto vrijedi i suprotno).

U *sites* dijelu *Homestead.yaml* datoteke potrebno je mapirati domenu na direktorij u Homestead okolini. U nastavku je dan primjer *Homestead.yaml* datoteke:

```
---
ip: "192.168.10.10"
memory: 2048
cpus: 1

authorize: C:/Users/Igor Gudelj/.ssh/id_rsa.pub

keys:
- C:/Users/Igor Gudelj/.ssh/id_rsa

folders:
- map: C:/Users/Igor Gudelj/Projects/zavrshi
```

```

    to: /home/vagrant/Code/zavrsni-app

sites:
  - map: zavrsni.app
    to: /home/vagrant/Code/zavrsni-app/public

variables:
  - key: APP_ENV
    value: local

```

Nakon uređenja *Homestead.yaml* datoteke prema vlastitim potrebama, potrebno se kroz terminal pozicionirati u Homestead direktorij i pokrenuti naredbu:

```
vagrant up
```

Vagrant će pokrenuti virtualnu mašinu i automatski konfigurirati dijeljene direktorije i Nginx stranice.

Zadnja stvar koju treba napraviti je postaviti domene za Nginx stranice u lokalnu *hosts* datoteku koja će preusmjeriti zahtjeve prema Homestead okruženju. Na Mac i Linux sustavima, *hosts* datoteka nalazi se na */etc/hosts*, a u Windows okruženju ista datoteka nalazi se na *C:\Windows\System32\drivers\etc\hosts*. Na kraj *hosts* datoteke potrebno je dodati sljedeću liniju:

```
127.0.0.1    zavrsni.app
```

Nakon dodavanja domene u *hosts* datoteku, moguće je pristupiti stranici putem web preglednika preko porta 8000 (<http://zavrsni.app:8000>)!

Za pristup Homestead okolini putem SSH veze, potrebno je kroz terminal iz Homestead direktorija pokrenuti naredbu:

```
vagrant ssh
```

Baza podataka konfigurirana je za MySQL i Postgres. Bazi podataka pristupa se preko *127.0.0.1* na portu *33060* (MySQL) ili *54320* (Postgres). Korisničko ime i lozinka za obje baze su *homestead* / *secret*.

2.3. Struktura projekta

Osnovna struktura projekta izgleda ovako:

- *app/*
- *bootstrap/*
- *vendor/*

- *public/*
- *.gitattributes*
- *.gitignore*
- *artisan*
- *composer.json*
- *composer.lock*
- *phpunit.xml*
- *server.php*

U nastavku je dan pregled pojedinih direktorija i datoteka.

app

App se koristi kao početno mjesto za vlastiti kôd aplikacije. *App* direktorij uključuje klase koje omogućavaju funkcionalnost aplikaciji, konfiguracijske datoteke i više. Direktorij *app* je izuzetno važan, te će biti detaljnije razrađen nešto kasnije.

bootstrap

- *autoload.php*
- *paths.php*
- *start.php*

Bootstrap direktorij sadrži nekoliko datoteka, koje su povezane s procedurama pokretanja radnog okvira. Datoteka *autoload.php* sadrži većinu procedura i nije ju preporučeno samostalno uređivati osim ako niste iskusni korisnici Laravela.

Datoteka *paths.php* izrađuje niz najčešćih sistemskih putanja koje koristi radni okvir. Ako je iz nekog razloga potrebno izmijeniti strukturu direktorija radnog okvira, mora se izmijeniti i sadržaj ove datoteke da bi se u njoj reflektirale sve promjene.

Datoteka *start.php* sadrži još procedura za pokretanje radnog okvira. Ovdje se mogu postaviti okruženja (engl. *environments*) radnog okvira o kojima će biti riječi kasnije.

Jednostavno rečeno, sadržaj direktorija *bootstrap* trebaju uređivati samo iskusni korisnici. Početnici u radu sa Larevelom slobodno ga mogu ignorirati, ali direktorij nije dozvoljeno obrisati! Da bi ispravno radio, Laravelu je taj direktorij potreban.

vendor

Direktorij *vendor* sadrži sve Composer pakete koje koristi aplikacija. Ovo uključuje i sam Laravel radni okvir.

public

- *packages/*
- *.htaccess*
- *favicon.ico*
- *index.php*
- *robots.txt*

Jedini direktorij Laravel aplikacije kome se direktno pristupa je *public*. On sadrži sve značajne datoteke za projekt kao što su CSS, Javascript i slike. Njegov sadržaj opisan je u nastavku.

Direktorij *packages* se koristi za skladištenje potrepština koje trebaju instalirati nezavisni paketi. Oni se nalaze u posebnom direktoriju kako ne bi došli u konflikt s vlastitim paketima.

Laravel dolazi s datotekom *.htaccess* za Apache web server i sadrži neke standardne konfiguracijske direktive koje imaju smisla za većinu korisnika. Ako se koristi neki drugi web server, datoteka se može slobodno ignorirati ili obrisati.

Uobičajeno, internet preglednici će u početnom direktoriju tražiti datoteku *favicon.ico*. Ova datoteka kontrolira izgled male slike veličine 16 x 16px koja se prikazuje u tabulatoru preglednika.

Nevolja je u tome što se preglednici žale kada ove ikonice nema. Ovo uzrokuje nepotrebne unose u log datoteke. Da bi riješio ovaj problem, Laravel unaprijed osigurava praznu datoteku *favicon.ico*, koja se kasnije može zamijeniti vlastitom.

Datoteka *index.php* je prva datoteka na koju server naleti kada dobije zahtjev od internet preglednika. Ovo je datoteka koja će pokrenuti *bootstrap* proces radnog okvira.

.gitattributes

Laravel je unaprijed dao preporučenu početnu konfiguraciju za Git verzioniranje. Git je trenutno najpopularniji izbor među developerima. Ako ne postoji namjera korištenja Git-a u projektu, ova datoteka se slobodno može ukloniti.

.gitignore

Datoteka *.gitignore* također sadrži unaprijed danu konfiguraciju koja treba informirati Git koje datoteke ne bi trebalo uključiti u verzioniranje aplikacije. U *.gitignore* datoteci unaprijed su uključeni *vendor* i *storage* direktoriji.

artisan

Izvršna datoteka *artisan* koristi se za izvršenje Laravelove Artisan komandne linije. Artisan sadrži mnoštvo korisnih komandi, koje omogućavaju prečice i mnoštvo dodatnih funkcionalnosti radnog okvira. O njegovim naredbama će više riječi biti u nastavku rada.

composer.json* i *composer.lock

Datoteke *composer.json* i *composer.lock*, sadrže informacije o composer paketima koji se koriste u projektu.

phpunit.xml

Datoteka *phpunit.xml* daje osnovnu konfiguraciju za rad s radnim okvirom za testiranje koji se zove PHP Unit. Ona upravlja učitavanjem međuzavisnosti i izvršava testove locirane u direktoriju *app/tests*.

server.php

PHP ugrađeni web poslužitelj koristi *server.php* datoteku radi lakšeg pokretanja web aplikacije. Dozvoljava da s naredbom *artisan serve* pokrenemo lokalni web poslužitelj za aplikaciju.

2.4. Aplikacijski direktorij

Ovdje će aplikacija poprimiti svoj oblik. U njemu se provodi najviše vremena. U nastavku je opisana struktura *app* direktorija.

- *commands/*
- *config/*
- *controllers/*
- *database/*
- *lang/*
- *models/*
- *start/*
- *storage/*

- *tests/*
- *views/*
- *filters.php*
- *routes.php*

commands

Direktorij *commands* sadrži korisnikove artisan naredbe, koje zahtjeva aplikacija. Artisan CLI ne samo da omogućava unaprijed definirane funkcionalnosti koje pomažu u izradi aplikacije, već je moguće kreiranje vlastitih naredbi.

config

Konfiguracija radnog okvira i aplikacije sprema se u ovom direktoriju. Konfiguracija Laravela sastoji se od setova PHP datoteka, koje sadrže nizove u obliku ključ-vrijednost. Ovaj direktorij može sadržavati i poddirektorije koji omogućavaju različite konfiguracije koje će se učitavati u različitim okruženjima.

controllers

Kao što i samo ime kaže, ovaj direktorij će sadržati kontrolere. Kontroleri sadrže logiku aplikacije i povezuju različite dijelove aplikacije.

database

Ovaj direktorij koristi se za čuvanje datoteka koje će kreirati sheme naše baze podataka i metode za popunjavanje iste probnim podacima. U ovom direktoriju se također nalazi i SQLite, podrazumijevana baza podataka.

lang

Direktorij *lang* sadrži PHP datoteke s nizovima tekstualnih podataka koje omogućavaju lokalizaciju aplikacije. Poddirektoriji, nazvani po dvoslovnim kraticama država, omogućavaju prijevod aplikacije na različite jezike.

models

Ovo je direktorij koji će sadržavati modele aplikacije. Unaprijed je postavljen model *User* u svrhu 'out of the box' autorizacije.

start

Za razliku od direktorija *bootstrap* koji sadrži početne procedure koje se tiču radnog okvira, direktorij *start* sadrži početne procedure koje se tiču vlastite aplikacije. Kao i uvijek, neke stvari su unaprijed konfigurirane sa standardnim vrijednostima.

storage

Kad god Laravel treba napisati nešto na disku, to čini unutar direktorija *storage*. Iz navedenog razloga, server mora imati dozvolu za pisanje u ovom direktoriju.

tests

Direktorij *tests* će sadržati sve unit i acceptance testove. Podrazumijevana konfiguracija za PHP Unit, koja je uključena u sam Laravel, tražit će testove u ovom direktoriju.

views

Direktorij *views* sadrži sve vizualne predloške aplikacije. Za početni *view* postavljena je datoteka *hello*.

filters.php

Datoteka *filters.php* sadrži filtere ruta aplikacije. O njima će biti riječi u nastavku rada.

routes.php

Datoteka *routes.php* sadrži sve rute aplikacije. Rute će biti detaljno opisane u idućem poglavlju.

3. Rute

3.1. Osnove rutiranja

Primjer zahtjeva upućenog Laravel radnom okviru izgleda ovako:

http://zavrsni.app:8000/my/page

U ovom primjeru, koristi se HTTP protokol za pristupanje vlastitoj Laravel aplikaciji na lokalnoj domeni *zavrsni.app*. Dio URL adrese *my/page* je ono što se koristi za rutiranje internet zahtjeva na adekvatnu logiku.

Rute se definiraju u datoteci *app/routes.php*. U nastavku je dan primjer rute koja iščekuje gore spomenuti zahtjev.

<?php

```
// app/routes.php
```

```
Route::get('my/page', function () {  
    return 'Hello world!';  
});
```

Sada je potrebno u internet preglednik unijeti *http://zavrsni.app:8000/my/page*. Ako je sve konfigurirano kako treba, pojavljuju se riječi 'Hello world!'. U nastavku je detaljnije opisana deklaracija rute.

Rute se uvijek deklariraju korištenjem klase Route. Dio 'get' je metoda klase route koja se koristi za 'hvatanje' zahtjeva napravljenih prema nekoj URL adresi, korištenjem HTTP glagola 'GET'.

Naime, svi zahtjevi koje kreira internet preglednik sadrže glagol. Najveći dio vremena, to će biti glagol GET, koji se koristi pri zahtjevu za dohvat internet stranice. Uvijek kada se upisuje nova internet adresa u pregledniku, šalje se GET zahtjev.

Ipak, on nije jedini zahtjev. Također postoji i POST, koji se koristi da napravi zahtjev i s njim otpremi neke podatke. POST je najčešće rezultat ispunjavanja forme, gdje se podaci moraju poslati bez da se prikazuju u URL adresi.

Dostupno je još HTTP glagola. U nastavku je dan pregled dostupnih metoda klase za rutiranje:

```
<?php
```

```
// app/routes.php
```

```
Route::get();  
Route::post();  
Route::put();  
Route::delete();  
Route::any();
```

Sve ove metode prihvataju iste parametre, tako da se slobodno može koristiti HTTP glagol koji je ispravan u odgovarajućoj situaciji. Ovo je poznato kao RESTful rutiranje. Detaljnije o ovoj temi bit će riječi kasnije.

Metoda *Route::any()* koristi se da pokrije svaki HTTP glagol. Međutim, u cilju rađanja transparentnih aplikacija, preporučuje se korištenje glagola koji odgovaraju danoj situaciji.

Vratimo se još jednom na prethodni primjer:

```
<?php
```

```
// app/routes.php
```

```
Route::get('my/page', function () {  
    return 'Hello world!';  
});
```

Prvi parametar metode *get()* definira URI koji želimo da odgovara URL adresi. U prikazanom primjeru to je *my/page*.

Sljedeći parametar koristi se da osigura logiku aplikaciji, kako bi mogla odgovoriti na ovaj zahtjev. Ovdje se koristi tzv. zatvaranje (engl. *closure*), koje je također poznato i kao anonimna funkcija. Zatvaranja su, jednostavno rečeno, funkcije bez imena koje se mogu dodijeliti varijablama, kao u slučaju jednostavnih tipova podataka.

Na primjer, gornji odsječak može se napisati i kao:

```
<?php
```

```
// app/routes.php
```

```
$logic = function () {  
    return 'Hello world!';  
}
```

```
Route::get('my/page', $logic);
```

Ovdje je Zatvaranje spremljeno unutar varijable *\$logic*, a zatim se predaje metodi *Route::get()*. U ovoj instanci, Laravel će izvršiti zatvaranje, samo kada trenutni zahtjev koristi HTTP glagol GET i odgovara URI adresi *my/page*. Pod ovim okolnostima, naredba *return* će biti obrađena, a pregledniku će biti predan tekst "Hello world!".

3.2 Pregledi (views)

Pregledi su vizualni dio aplikacije koji sadrže tekstualni predložak (najčešće HTML) koji se proslijeđuje pregledniku. Pregledi koriste nastavak *.php* i uobičajeno se nalaze unutar direktorija *app/views*. To znači da se PHP kôd također može izvršavati unutar pregleda. Pogledajmo primjer:

```
<!-- app/views/hello.php -->

<!doctype html>
<html lang="en">

<head>
    <meta charset="UTF-8">
    <title>Hello!</title>
</head>

<body>
<p>Hello <?php echo $name; ?></p>
</body>

</html>
```

Ovaj pregled može se poslati do preglednika na sljedeći način:

```
<?php

// app/routes.php

Route::get('/', function()
{
    return View::make('hello', array('name' => 'Igor'));
});
```

Drugi argument u *View::make* metodi je polje podataka koje se želi proslijediti pregledu. Ista stvar može se napraviti na jedan od sljedećih načina:

```
<?php
```

```
// Koristeći standardni pristup
$view = View::make('hello')->with('name', 'Igor');

// Koristeći 'magične metode'
$view = View::make('hello')->withName('Igor');
```

3.3. Filteri

Filteri su određeni setovi pravila ili akcija, koji se mogu primijeniti na rutu. Oni se mogu izvršiti prije ili poslije logike unutar rute. Korištenjem filtera prije rute, može se izmijeniti tok izvršenja aplikacije, ako se ne ispune neki kriteriji ili setovi pravila. To je sjajan način zaštite ruta. Pogledajmo primjer:

```
<?php
```

```
// app/filters.php

Route::filter('birthday', function()
{
    if (date('d/m/y') == '01/06/89') {
        return View::make('birthday');
    }
});
```

Laravel je osigurao datoteku *app/filters.php* za sve filtere, no filteri se mogu smjestiti bilo gdje u aplikaciji.

Da bi se kreirao novi filter, koristi se metoda *Route::filter()*. Prvi parametar je ime, koje se koristi za dodijeljivanje filtera ruti. Drugi parametar je *callback* funkcija, što je u ovom slučaju zatvaranje. *Callback* je funkcija koja se poziva kada je izvršen filter.

Da bi filter postao koristan, potrebno ga je pridijeliti ruti. To se radi na sljedeći način:

```
<?php
```

```
// app/routes.php

Route::get('/', array(
    'before' => 'birthday',
    function()
    {
        return View::make('birthday');
    }
));
```

Također postoji i '*after*' filter koji se izvršava poslije logike rute. U gornjem primjeru ne bi bio koristan, ali vrlo je koristan u svrhu zapisivanja log datoteka ili operacija čišćenja.

4. Kontroleri

4.1. Kreiranje kontrolera

U poglavlju o osnovama rutiranja, pokazano je kako da povezati rute sa zatvaranjima, malim dijelovima logike, koji čine strukturu aplikacije. Zatvaranja su brz i jednostavan način pisanja aplikacija. Međutim, bolji način za čuvanje logike aplikacije su kontroleri.

Kontroler je klasa koja čuva određenu logiku. Uobičajeno, kontroler sadrži više javnih metoda poznatijih kao akcije. Akcije se mogu tumačiti kao direktna alternativa zatvaranjima, koja su korištena u prethodnim poglavljima. Jako su slični i u izgledu i funkcionalnosti. Pogledajmo primjer:

<?php

```
// app/controllers/ArticleController.php

class ArticleController extends BaseController
{
    public function showIndex()
    {
        return View::make('index');
    }

    public function showSingle($articleId)
    {
        return View::make('single');
    }
}
```

Ovaj primjer odgovara blogu ili nekoj drugoj formi CMS-a. U idealnom slučaju, blog ima stranicu za prikaz svih članaka i stranicu za prikaz pojedinačnih članaka. Ove obje aktivnosti su povezane s konceptom članka i ima smisla grupirati logiku.

U nastavku je izdvojena akcija koja bi se u ovom primjeru koristila za prikaz liste blog članaka:

```
public function showIndex()
{
    return View::make('index');
}
```

Usporedimo ovu akciju sa zatvaranjem u ruti, koje se može iskoristiti radi postizanja istog cilja:

```
Route::get('index', function()
{
    return View::make('index');
});
```

Iz priloženog je vidljivo da je unutrašnja funkcija skoro identična. Jedina razlika je u tome što akcija kontrolera ima ime, dok je funkcija u zatvaranju anonimna. Akcija kontrolera može sadržati svaki kôd koji može i zatvaranje.

Međutim, postoji još jedna razlika između gornjih dijelova koda. U poglavlju o osnovama rutiranja, pokazano je kako povezati URI adresu s logikom koja je sadržana u zatvaranju. U gornjem primjeru zatvaranja u ruti, URI adresa */index* bila bi povezana s aplikacijskom logikom.

Međutim, akcija kontrolera uopće ne spominje URI adresu. Kako Laravel uopće zna kako da uputi naše rute na naše kontrolere? Odgovor slijedi u sljedećem poglavlju.

4.2. Rutiranje kontrolera

Kontroleri su uredni i prikladni i omogućavaju čist način za grupiranje logike. Međutim, oni su beskorisni, ako naši korisnici ne mogu pristupiti toj logici. Srećom, metoda povezivanja URI adrese s kontrolerom je slična metodama ruta koje su korištene u zatvaranjima. Pogledajmo primjer:

```
<?php
```

```
// app/routes.php
```

```
Route::get('index', 'ArticleController@showIndex');
```

Za povezivanje URI adrese s kontrolerom, potrebno je najprije definirati novu rutu unutar datoteke */app/routes.php*. Koristi se ista metoda *Route::get()*, kao kod rutiranja zatvaranja. Međutim, drugi parametar je u potpunosti drugačiji. Ovog puta imamo string koji se sastoji od dva dijela razdvojenih znakom (@). Vratimo se još jednom na kontroler koji smo kreirali u prethodnom poglavlju:

```
<?php
```

```
// app/controllers/ArticleController.php
```

```
class ArticleController extends BaseController
{
    public function showIndex()
```

```

    {
        return View::make('index');
    }

    public function showSingle($articleId)
    {
        return View::make('single');
    }
}

```

Dakle, iz primjera se može vidjeti da je ime klase *ArticleController*, a akcija s kojom želimo povezati rutu je nazvana *showIndex()*. Potrebno ih je spojiti sa znakom (@) u sredini: *ArticleController@showIndex*.

Sada je moguće iskoristiti bilo koju od metoda koja je spomenuta u poglavlju o osnovama rutiranja i usmjeriti ih ka kontrolerima. Na primjer, akcija kontrolera koja bi odgovorila na HTTP glagol POST izgleda ovako:

<?php

```
// app/routes.php
```

```
Route::post('article/new', 'ArticleController@newArticle');
```

Dakle, gornja ruta će se odazvati na URI adresu */article/new*, a bit će obrađena akcijom *newArticle()* iz kontrolera *ArticleController*.

4.3. RESTful kontroleri

Poznato je da se može definirati glagol HTTP zahtjeva koji je potrebno izvršiti, koristeći metode ruta. Ovo je zaista pogodno pri korištenju zatvaranja. Međutim, Laravel također omogućava da prilikom rutiranja na kontrolere, definiramo glagole zahtjeva kao prefiks imenu akcije kontrolera. Pogledajmo primjer:

<?php

```
// app/controllers/ArticleController.php
```

```

class ArticleController extends BaseController
{
    public function getCreate()
    {
        return View::make('create');
    }
}

```



```
public function postCreate()  
{  
    // upravlja kreiranjem forme  
}  
}
```

Namjera akcije je da osiguraju formu koja kreira novi članak za blog. Može se primijetiti da su prefiksi imena akcija `get` i `post`. To su glagoli HTTP zahtjeva.

Postavlja se pitanje kako rutiramo do našeg RESTful kontrolera. Korištenje glagol metoda u klasi *Route class* ovdje ne bi imalo smisla. Više nije potrebno rutirati prema individualnim akcijama. U nastavku je dan primjer drugačije metode za rutiranje:

<?php

```
// app/routes.php
```

```
Route::controller('article', 'ArticleController');
```

Ova jedna metoda će rutirati sve akcije predstavljene RESTful kontrolerom.

Prvi parameter je osnovna URL adresa. Uobičajeno je da se RESTful rutiranje koristi da predstavi objekt, tako da će u većini slučajeva osnovna URL adresa biti ime objekta. Možemo ga zamisliti kao neku vrstu prefiksa akcijama koje smo kreirali unutar RESTful kontrolera.

Drugi parametar već je poznat. To je kontroler koji želimo rutirati. Laravel će prihvatiti kontroler iz imenskog prostora kao cilj rutiranja, tako da se kontroleri slobodno mogu organizirati shodno vlastitim potrebama.

Kao što je vidljivo, upotreba ovog načina rutiranja do kontrolera omogućava jasnu prednost u odnosu na originalni način rutiranja. Na ovaj način, potrebno je osigurati samo jedan unos za kontroler, umjesto rutiranja na svaku akciju zasebno.

5. Forme

5.1. Otvaranje formi

Kako bi se kreirao *tag* za otvaranje formi, koristi se *Form::open()* generator metoda.

Ova metoda prihvata jedan parameter u obliku niza. Pogledajmo primjer:

```
<!-- app/views/form.blade.php -->

{{ Form::open(array('url' => 'our/target/route')) }}

{{ Form::close() }}
```

U gornjem primjeru koristi se indeks URL, sa vrijednošću rute za koju želimo da bude destinacija forme. Koristi se i metoda *Form::close()* kako bi zatvorili formu. Forma se može zatvoriti i sa `</form>`, ali češće se koristi gornji način radi preglednosti.

Evo i rezultirajućeg izvornog koda predstavljenog preko pregleda forme:

```
<!-- app/views/form.blade.php -->

<form method="POST"
action="http://zavrsni.app:8000/our/target/route" accept-
charset="UTF-8">
<input name="_token" type="hidden"
value="83KCsmJF1Z2LMZfhb17ihvt9ks5NEcAwFoRFTq6u">
</form>
```

Laravel zna da su POST forme najpopularniji izbor kada se grade web aplikacije. Iz tog razloga podrazumijevano će koristiti POST metodu umjesto GET metode koju pretpostavlja HTML.

Atribut *accept-charset* je još jedan koristan atribut koji automatski kreira Laravel. On osigurava da će se za kodiranje karaktera prilikom predaje podataka iz forme koristiti UTF-8. Za promjenu *default* vrijednosti u formi, radi se sljedeće:

```
<!-- app/views/form.blade.php -->

{{ Form::open(array(
    'url' => 'our/target/route',
    'method' => 'GET',
    'accept-charset' => 'ISO-8859-1'
)) }}

{{ Form::close() }}
```

U gornjem primjeru korišten je atribut *method* u nizu unutar metode *Form::open()*, kako bi se naglasilo da želimo koristiti GET metodu u formi. Također je korišten i atribut *accept-charset*, kako bi se osigurao drugačiji set karaktera za kodiranje.

Evo novog izvornog HTML koda:

```
<!-- app/views/form.blade.php -->

<form method="GET"
action="http://zavrsni.app:8000/our/target/route" accept-
charset="ISO-8859-1">
<input name="_token" type="hidden"
value="83KCsmJF1Z2LMZfhb17ihvt9ks5NEcAwFoRFTq6u">
</form>
```

U nastavku je dan primjer forme koja koristi *HTTP* glagol 'DELETE' kao *method* atribut forme.

```
<!-- app/views/form.blade.php -->

{{ Form::open(array(
    'url' => 'our/target/route',
    'method' => 'DELETE'
)) }}

{{ Form::close() }}
```

Generirani HTML kod izgleda ovako:

```
<!-- app/views/form.blade.php -->

<form method="POST"
action="http://zavrsni.app:8000/our/target/route" accept-
charset="UTF-8">
<input name="_token" type="hidden"
value="83KCsmJF1Z2LMZfhb17ihvt9ks5NEcAwFoRFTq6u">
<input name="_method" type="hidden" value="DELETE">
</form>
```

Može se primijetiti da je *method* postavljen na POST. Isto tako treba primijetiti dodatan *input* tag. Razlog je u tome što HTML4 podržava samo POST i PUT metode. Iz tog razloga Laravel automatski stvara skriveni unos nazvan *_method*. Sloj za rutiranje Laravela će pogledati ovaj POST zahtjev, vidjeti uvršteni podatak '*_method*' i rutirati do odgovarajuće akcije.

Ako je korisniku potrebno omogućiti upload datoteka kroz aplikaciju, to se čini na sljedeći način:

```
<!-- app/views/form.blade.php -->

{{ Form::open(array(
    'url' => 'our/target/route',
    'files' => true
)) }}

{{ Form::close() }}
```

Dodajući novi atribut s imenom *files* i logičkom vrijednošću *true*, Laravel će dodati sve neophodne attribute da bi dozvolio otpremanje datoteka. Evo generiranog izvornog HTML koda za gornji primjer:

```
<!-- app/views/form.blade.php -->

<form method="POST"
      action="http://zavrsni.app:8000/our/target/route"
      accept-charset="UTF-8"
      enctype="multipart/form-data">
    <input name="_token" type="hidden"
value="83KCsmJF1Z2LMZfhb17ihvt9ks5NEcAwFoRFTq6u">
</form>
```

5.2. Polja u formama

Forme nisu korisne, ako ne daju mogućnost prikupljanja podataka. Pogledajmo neka polja forme, koja je moguće generirati koristeći Laravelovu biblioteku za građenje formi.

Labele

Labele se koriste kako bi korisnici znali koje vrijednosti u formi unose. Kreiraju se korištenjem metode *Form::label()*. Pogledajmo primjer:

```
<!-- app/views/form.blade.php -->

{{ Form::open(array('url' => 'my/route')) }}
{{ Form::label('first_name', 'First Name') }}
{{ Form::close() }}
```

Prvi parametar metode *Form::label()* odgovara atributu *name* polja koje se opisuje. Drugi parametar je tekst labele. Pogledajmo generirani HTML izvorni kod:

```
<form method="POST"
      action="http://zavrsni.app:8000/my/route"
      accept-charset="UTF-8">
  <input name="_token" type="hidden"
value="83KCsmJF1Z2LMZfhb17ihvt9ks5NEcAwFoRFTq6u">
  <label for="first_name">First Name</label>
</form>
```

Labele se koriste kad god trebamo opisati neko polje.

Tekstualna polja

Tekstualna polja se koriste za prikupljanje podataka tipa *string*. Pogledajmo metodu za ovaj tip polja:

```
<!-- app/views/form.blade.php -->

{{ Form::open(array('url' => 'my/route')) }}
    {{ Form::label('first_name', 'First Name') }}
    {{ Form::text('first_name', 'Igor Gudelj') }}
{{ Form::close() }}
```

Prvi parametar metode *Form::text()* je atribut *name*. On se koristi za identifikaciju vrijednosti polja unutar podataka zahtjeva.

Drugi parametar je neobavezan. To je podrazumijevana vrijednost za element unosa. Pogledajmo HTML izvorni kod:

```
<!-- app/views/form.blade.php -->

<form method="POST"
      action="http://zavrsni.app:8000/my/route"
      accept-charset="UTF-8">
  <input name="_token" type="hidden"
value="83KCsmJF1Z2LMZfhb17ihvt9ks5NEcAwFoRFTq6u">
  <label for="first_name">First Name</label>
  <input name="first_name" type="text" value="Igor Gudelj"
id="first_name">
</form>
```

Textarea

Polje *textarea* funkcionira na sličan način kao i ulazno tekstualno polje, osim što dozvoljava unos teksta u više redova. Pogledajmo kako se može generirati:

```
<!-- app/views/form.blade.php -->
```

```

{{ Form::open(array('url' => 'my/route')) }}
    {{ Form::label('description', 'Description') }}
    {{ Form::textarea('description', 'Textarea description') }}
{{ Form::close() }}

```

Prvi parametar je atribut elementa s imenom *name*, a drugi parametar je opet podrazumijevana vrijednost. Evo kako se ovaj element forme prikazuje u okviru generiranog HTML izvornog koda:

```

<!-- app/views/form.blade.php -->

<form method="POST"
      action="http://zavrsni.app:8000/my/route"
      accept-charset="UTF-8">
    <input name="_token" type="hidden"
value="83KCsmJF1Z2LMZfhb17ihvt9ks5NEcAwFoRFTq6u">
    <label for="description">Description</label>
    <textarea name="description"
      cols="50"
      rows="10"
      id="description">Textarea description
    </textarea>
</form>

```

Laravel je pretpostavio neke podrazumijevane vrijednosti, kao što su broj redova i stupaca. Ove vrijednosti se jednostavno mogu izmijeniti, dodajući željene vrijednosti unutar neobaveznog posljednjeg parametra.

Polja za lozinke

Za sigurno upisivanje lozinke u formu, upisani znakovi se ne bi trebali prikazivati na ekranu. Tip unosa *'password'* je pravi primjer toga. Evo kako se može generirati:

```

<!-- app/views/form.blade.php -->

{{ Form::open(array('url' => 'my/route')) }}
    {{ Form::label('password', 'Password') }}
    {{ Form::password('password') }}
{{ Form::close() }}

```

Prvi parametar metode *Form::password()* je atribut *name*. Kao i uvijek, postoji i posljednji, neobavezni parametar kao niz dodatnih atributa elementa. Evo generiranog HTML izvornog koda iz gornjeg primjera:

```
<!-- app/views/form.blade.php -->

<form method="POST"
      action="http://demo.dev/my/route"
      accept-charset="UTF-8">
    <input name="_token" type="hidden"
value="83KCsmJF1Z2LMZfhhb17ihvt9ks5NEcAwFoRFTq6u">
    <label for="secret">Password</label>
    <input name="password" type="password" value=""
id="password">
</form>
```

Checkbox

Checkbox polja omogućavaju formi prihvat logičke veličine. Pogledajmo primjer:

```
<!-- app/views/form.blade.php -->

{{ Form::open(array('url' => 'my/route')) }}
    {{ Form::label('newsletter', 'Newsletter?') }}
    {{ Form::checkbox('newsletter', '1', true) }}
{{ Form::close() }}
```

Prvi parametar metode *Form::checkbox()* je atribut *name*, a drugi parametar je vrijednost polja. Treći, neobavezni parametar, kaže da li je *checkbox* podrazumijevano označen ili ne. Podrazumijevana vrijednost je *false*, tj. da nije označen. HTML izvorni kod izgleda ovako:

```
<!-- app/views/form.blade.php -->

<form method="POST"
      action="http://zavrsni.app:8000/my/route"
      accept-charset="UTF-8">
    <input name="_token" type="hidden"
value="83KCsmJF1Z2LMZfhhb17ihvt9ks5NEcAwFoRFTq6u">
    <label for="newsletter">Newsletter?</label>
    <input checked="checked"
      name="newsletter"
      type="checkbox"
      value="1"
      id="newsletter">
</form>
```

Radio button

Radio button ima metodu istog oblika kao i *checkbox* polja. Za razliku od *checkbox* polja, dopušta odabir samo jedne vrijednosti. Slijedi primer:

```
<!-- app/views/form.blade.php -->

{{ Form::open(array('url' => 'my/route')) }}
    {{ Form::label('newsletter', 'Newsletter?') }}
    {{ Form::radio('newsletter', 'yes', true) }} Yes
    {{ Form::radio('newsletter', 'no') }} No
{{ Form::close() }}
```

Izvorni HTML kod izgleda ovako:

```
<!-- app/views/form.blade.php -->

<form method="POST"
    action="http://zavrsni.app:8000/my/route"
    accept-charset="UTF-8">
    <input name="_token" type="hidden"
value="83KCsmJF1Z2LMZfhb17ihvt9ks5NEcAwFoRFTq6u">
    <label for="newsletter">Newsletter?</label>
    <input checked="checked"
        name="newsletter"
        type="radio"
        value="yes"
        id="newsletter"> Yes
    <input name="newsletter"
        type="radio"
        value="no"
        id="newsletter"> No
</form>
```

Dropdown

```
<!-- app/views/form.blade.php -->

{{ Form::open(array('url' => 'my/route')) }}
    {{ Form::label('size', 'Size?') }}
    {{ Form::select('size', array(
        'small' => 'Small',
        'medium' => 'Medium',
        'large' => 'Large'
    ), 'small') }}
{{ Form::close() }}
```

Polje za email adrese

```
<!-- app/views/form.blade.php -->
```



```
{{ Form::open(array('url' => 'my/route')) }}
    {{ Form::label('email', 'E-Mail Address') }}
    {{ Form::email('email', 'me@example.com') }}
{{ Form::close() }}
```

5.3. Dugmad u formama

Svaka forma mora imati dugme za predaju podataka. Pogledajmo koja dugmad su nam dostupna.

Dugme za podnošenje forme

```
<!-- app/views/form.blade.php -->

{{ Form::open(array('url' => 'my/route')) }}
    {{ Form::submit('Submit') }}
{{ Form::close() }}
```

Normalno dugme

```
<!-- app/views/form.blade.php -->

{{ Form::open(array('url' => 'my/route')) }}
    {{ Form::button('Save') }}
{{ Form::close() }}
```

Dugme za sliku

```
<!-- app/views/form.blade.php -->

{{ Form::open(array('url' => 'my/route')) }}
    {{ Form::image(asset('my/image.gif'), 'submit') }}
{{ Form::close() }}
```

Dugme za reset

```
<!-- app/views/form.blade.php -->

{{ Form::open(array('url' => 'my/route')) }}
    {{ Form::reset('Clear') }}
{{ Form::close() }}
```

5.4. Sigurnost formi

Kada se primaju podaci s forme, bitno je uvjeriti se da podaci koji se šalju pripadaju vlastitoj web stranici. Laravel u tome automatski pomaže. Ovaj oblik napada poznat je kao 'Cross Site Request Forgery' ili skraćeno CSRF.

U primjerima do sada uvijek je bilo generirano skriveno polje `_token` koje je uvijek bilo automatski kreirano. To je vrsta tajne fraze, za koju Laravel zna. Kreirano je koristeći 'tajnu' vrijednost iz konfiguracijske datoteke aplikacije. Ako je konfigurirano da Laravel provjerava ovu vrijednost unutar podataka unosa, onda je sigurno da podaci poslani od strane forme pripadaju vlastitoj aplikaciji.

Moguće je samostalno provjeravati token i uspoređivati njegovu vrijednost s rezultatom metode `Session::token()`, ali Laravel omogućava bolju opciju u obliku filtera s nazivom `csrf`. U nastavku je dan primjer spomenutog filtera koji je dodijeljen ruti:

```
<!-- app/filters.php -->

Route::filter('csrf', function()
{
    if (Session::token() != Input::get('_token'))
    {
        throw new Illuminate\Session\TokenMismatchException;
    }
});
```

<?php

```
// app/routes.php

Route::post('/handle-form', array('before' => 'csrf',
function()
{
    // Handle our posted form data
}));
```

Polje `_token` mora biti prisutno i ispravno. Iz tog razloga, ruti koja treba prihvatiti podatke s forme, potrebno je dodijeliti filter `csrf` kao *before* filter. Ako token vrijednost nije prisutna ili ju je netko mijenjao, onda će biti izbačen izuzetak `Illuminate\Session\TokenMismatchException`, a logika rute neće biti izvršena.

5.5. Validacija

Korisnicima ne treba vjerovati. Ako u aplikaciji postoji slabost, korisnici će ju pronaći i iskoristiti. Kako im to ne bi dopustili, potrebno je koristiti validaciju forme kako bi bili sigurni da dobivamo dobar unos. Pogledajmo malo složeniji primjer u kojem će biti objašnjene najbitnije stvari oko validacije korisničkog unosa.

```
<?php
```

```
// app/routes.php
```

```
Route::get('/', function()
{
    return View::make('form');
});

Route::post('/registration', function()
{
    // Fetch all request data.
    $data = Input::all();

    // Build the validation constraint set.
    $rules = array(
        'username' => 'alpha_num|min:3'
    );

    // Create a new validator instance.
    $validator = Validator::make($data, $rules);

    if ($validator->passes()) {
        // Normally we would do something with the data.
        return 'Data was saved.';
    }

    return Redirect::to('/');
});
```

Skup validacijskih pravila ima oblik asocijativnog niza. Ključ niza predstavlja polje koje validiramo. Vrijednost niza će se sastojati od jednog ili više validacijskih ograničenja za jedno polje.

Za kreiranje nove instance validatora, koristi se metoda *Validator::make()*. Prvi parametar metode *make()* je niz podataka, koje je potrebno validirati. U gornjem primjeru validiramo podatke zahtjeva, ali bi lako mogli validirati bilo koji drugi niz podataka. Drugi parametar metode su skup pravila, koja se koriste za validaciju podataka.

Za testiranje rezultata validacije na instancu validatora primijenjena je metoda *passes()*. Ova metoda vraća logičku vrijednost, da bi pokazala da li su podaci prošli validaciju. Odziv *true* govori da podaci podliježu svim pravilima validacije. Suprotno, *false* je indikator da podaci nisu zadovoljili zahtjeve validacije.

U gornjem primjeru koristi se naredba *if* prilikom odlučivanja treba li podatke s forme sačuvati ili preusmjeriti korisnika nazad na formu.

Laravel omogućava prenošenje više validacijskih ograničenja unutar dijela za vrijednost u nizu pravila za validaciju, tako što ih razdvajamo separatorom |.

Lista svih dostupnih validacijskih pravila i njihovih funkcija dostupna je na sljedećoj poveznici: <http://laravel.com/docs/4.2/validation#available-validation-rules>

6. Baze podataka

6.1. Konfiguracija baze podataka

Laravel konfiguracija vezana za baze podataka nalazi se u datoteci `app/config/database.php`. Laravel trenutno podržava četiri tipa baze podataka: MySQL, Postgres, SQLite i SQL Server.

U konfiguracijskoj datoteci potrebno je odabrati *default* konekciju koja se koristiti. Također, u *connections* polju potrebno je ispuniti pristupne podatke za odabranu bazu podataka. Primjer konfiguracijske datoteke za lokalni rad u Homestead razvojnom okruženju dan je u nastavku:

```
<?php
```

```
// app/config/database.php
```

```
return array(
```

```
    'default' => 'mysql',
```

```
    'connections' => array(
```

```
        'mysql' => array(
```

```
            'driver' => 'mysql',
```

```
            'host' => 'localhost',
```

```
            'database' => 'homestead',
```

```
            'username' => 'homestead',
```

```
            'password' => 'secret',
```

```
            'charset' => 'utf8',
```

```
            'collation' => 'utf8_unicode_ci',
```

```
            'prefix' => '',
```

```
        ),
```

```
    ),
```

```
);
```

6.2. Graditelj Schema

Laravelova *Schema* klasa pruža vrlo jednostavan rad s bazom podataka. Jednako dobro radi na svim podržanim bazama podataka.

Za kreiranje nove tablice u bazi podataka, koristi se metoda *Schema::create*.

```
Schema::create('users', function($table)
{
    $table->increments('id');
});
```

Prvi argument je ime tablice koja se kreira. Drugi argument je zatvaranje kojem je potrebno predati *Blueprint* objekt koji se koristi za definiranje nove tablice.

Za preimenovanje tablice služi metoda *Schema::rename*.

```
Schema::rename($from, $to);
```

Za brisanje tablice služi metoda *Schema::drop*.

```
Schema::drop('users');  
Schema::dropIfExists('users');
```

Za promjenu na postojećoj tablici koristi se metoda *Schema::table*.

```
Schema::table('users', function($table)  
{  
    $table->string('email');  
});
```

Pogledajmo neke od najčešće korištenih tipova kolona kod kreiranja tablica:

```
$table->boolean('confirmed');  
$table->char('name', 4);  
$table->date('created_at');  
$table->dateTime('created_at');  
$table->decimal('amount', 5, 2);  
$table->double('column', 15, 8);  
$table->float('amount');  
$table->increments('id');  
$table->integer('votes');  
$table->string('name', 100);  
$table->text('description');  
$table->timestamps();  
->nullable()  
->default($value)  
->unsigned()
```

Za preimenovanje kolone u tablici koristi se metoda *renameColumn*.

```
Schema::table('users', function($table)  
{  
    $table->renameColumn('from', 'to');  
});
```

Za brisanje kolone u tablici koristi se metoda *dropColumn*.

```
Schema::table('users', function($table)
{
    $table->dropColumn('votes');
});
```

Za provjeru postojanja tablice ili kolone u tablici koristi se metoda *hasTable* i *hasColumn*.

```
if (Schema::hasTable('users'))
{
    //
}

if (Schema::hasColumn('users', 'email'))
{
    //
}
```

Dodavanje stranih ključeva na kolonu u tablici prikazan je u sljedećem primjeru (kolona *user_id* referencira kolonu *id* u tablici *users*):

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

6.3. Migracije

Migracije je najlakše opisati kao *version control* baze podataka. One predstavljaju mnoštvo PHP skripti koje se koriste da promijene strukturu ili sadržaj baze podataka. Migracije imaju vremenske oznake, tako da se uvijek izvršavaju u odgovarajućem redoslijedu. Laravel zapisuje koje su migracije izvršene unutar posebne tablice naziva *migrations*. Koristeći migracije, osigurano je da će cijeli tim koji razvija aplikaciju imati uvijek istu strukturu baze, u konzistentnom i stabilnom stanju.

Za kreiranje migracije, koristi se Artisan *interface* u komandnoj liniji. U nastavku je dan primjer migracije za kreiranje korisnika nazvan *create_users*:

```
$ php artisan migrate:make create_users
Created Migration: 2013_06_30_124846_create_users
Generating optimized class loader
Compiling common classes
```

Artisan naredbom *migrate:make* dodijeljeno je ime novoj migraciji. Laravel je

generirao novi predložak migracije u datoteci *app/database/migrations*. Datoteka će biti nazvana po parametru zadanom u naredbi *migrate:make*, s dodijeljenim vremenom nastanka:

```
app/database/migrations/2014_09_20_124846_create_users.php
```

Pogledajmo stvorenu datoteku:

```
<?php
```

```
// app/database/migrations/2014_09_20_124846_create_users.php
```

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

```
class CreateUsers extends Migration {
```

```
    /**
     * Run the migrations.
     *
     * @return void
     */
    public function up()
    {
        //
    }

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

Unutar klase *migration* postoje dvije javne metode: *up()* i *down()*. Što god da se napravi unutar metode *up()*, unutar metode *down()* mora se poništiti. Razlog tomu je što migracije rade dvosmjerno. Moguće je pokrenuti migraciju za ažuriranje strukture ili sadržaja baze, ali isto tako je moguć i poništiti migraciju i vratiti bazu na prethodno stanje.

Metodu *up()* moguće je ispuniti na sljedeći način:

```
public function up()
{
```



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

Također je potrebno ispuniti metodu *down()* koja radi suprotnu stvar od metode *up()*. U metodi *up()* kreirana je tablica *users*, tako da je u *down()* metodi tu istu tablicu potrebno obrisati.

```
public function down()
{
    Schema::drop('users');
}
```

Nakon kreiranja migracija, potrebno ih je pokrenuti. Za to se koristi naredba *migrate*:

```
$ php artisan migrate
Migrated: 2014_09_20_124846_create_users
```

Za poništenje izvršene migracije, koristimo naredbu *migrate:rollback*:

```
$ php artisan migrate:rollback
Rolled back: 2014_09_20_124846_create_users
```

Pri korištenju naredbe *rollback*, Laravel vraća unazad samo one migracije koje su izvršene pri zadnjem pokretanju naredbe *migrate*. Za vraćanje svih migracija, koristimo naredbu *migrate:reset*:

```
$ php artisan migrate:reset
```

7. Eloquent

7.1. Eloquent ORM

Eloquent ORM uključen s Laravel radnim okvirom pruža jednostavnu ActiveRecord implementaciju u svrhu olakšanog rada s bazom podataka. Svaka tablica u bazi podataka mora imati odgovarajući 'Model' koji se koristi pri interakciji s tom tablicom.

Kao primjer, u *models* direktoriju kreirana je datoteka User.php:

```
class User extends Eloquent {}
```

Treba primijetiti da Eloquentu nije potrebno specificirati koju tablicu koristi kreirani User model. Laravel pretpostavlja, ako nije eksplicitno navedeno, da klasa User koristi tablicu 'users' u bazi podataka. Transformacija koja se vrši je pretvorba u mala slova, a zatim dodavanje množine na naziv modela. Za korištenje vlastitog naziva tablice, potrebno je postaviti varijablu *\$table* na željeni naziv:

```
class User extends Eloquent {  
  
    protected $table = 'my_users';  
  
}
```

7.2. Eloquent kolekcije

Eloquent kolekcije su proširenje Laravelove klase Collection s korisnim metodama za upravljanje rezultatima upita. Sama klasa Collection je samo omotač za niz objekata, ali ima puno interesantnih metoda za pomoć pri dohvaćanju stavki iz niza.

U nastavku su prikazane najčešće korištene metode koje su dostupne za klasu Collection. Neke od njih se odnose na unos i vraćanje elemenata na osnovu njihovih ključeva. Međutim, u slučaju Eloquent rezultata, ključevi ne odgovaraju primarnim ključevima tablice, koje predstavljaju instance modela, tako da su ove metode jako korisne.

All

Metoda all() koristi se za dohvat unutrašnjeg niza kojeg koristi Collection objekt.

First

Metoda first() koristi se za dohvat prvog elementa iz kolekcije.

Last

Metoda `last()` koristi se za dohvat zadnjeg elementa iz kolekcije.

Shift

Metoda `shift()` slična je metodi `first()`. Metoda `shift()` vraća prvi element niza, ali za razliku od metode `first()` također uklanja taj element iz niza.

Pop

Slično kao i metoda `shift()`, `pop()` vraća zadnji element niza i uklanja ga iz kolekcije.

Each

Metoda `each()` služi za izvršenje zatvaranja na svakom elementu niza.

Map

Slično kao i metoda `each()`, `map()` dodatno može vratiti novu kolekciju kao rezultat.

Filter

Metoda `filter()` koristi se kada pomoću zatvaranja želimo smanjiti broj elemenata u kolekciji.

Sort

Metoda `sort()` služi za sortiranje kolekcije po uvjetima koji su postavljeni u zatvaranju.

Merge

Metoda `merge()` služi za spajanje dvije kolekcije.

Slice

Metoda `slice()` ekvivalentna je funkciji `slice()`. Služi za kreiranje podskupa modela koristeći offset kolekcije.

IsEmpty

Metoda `isEmpty()` provjerava da li kolekcija sadrži elemente, te vraća logičku vrijednost *true* ili *false*.

ToArray

Metoda `toArray()` vraća kolekciju unutrašnjeg niza. Također, svi elementi unutar kolekcije koji se mogu transformirati u niz, npr. objekti, bit će transformirani.

ToJson

Metoda toJson() transformira kolekciju u JSON string.

Count

Metoda count() vraća broj instanci modela sadržanih u unutrašnjem nizu kolekcije.

7.3. Eloquent relacije

Za očekivati je da su tablice u bazi međusobno povezane. Npr. članak na blogu može imati više komentara, narudžba je povezana s korisnikom koji je narudžbu izvršio. Eloquent omogućuje jednostavnu implementaciju i korištenje relacija. U nastavku su prikazane najčešće korištene relacije koje Laravel podržava:

One to one

One-to-one je najosnovnija relacija. Npr. *User* model može imati jedan broj telefona (*Phone*). Tu relaciju u Eloquentu je moguće prikazati na sljedeći način:

```
class User extends Eloquent {  
  
    public function phone()  
    {  
        return $this->hasOne('Phone');  
    }  
  
}
```

Prvi argument koji se proslijeđuje *hasOne* metodi, ime je relacijskog modela. Nakon što je relacija definirana, pristupiti joj možemo na sljedeći način:

```
$phone = User::find(1)->phone;
```

SQL upit izveden iz gornjeg poziva glasi:

```
select * from users where id = 1  
select * from phones where user_id = 1
```

Također je potrebno definirati i inverznu relaciju na *Phone* modelu. Za to se koristi *belongsTo* metoda:

```
class Phone extends Eloquent {  
  
    public function user()  
    {  

```

```

        return $this->belongsTo('User');
    }
}

```

One to many

Primjer *one-to-many* relacije bio bi blog članak koji može imati više komentara.

Relacija bi izgledala ovako:

```

class Post extends Eloquent {

    public function comments()
    {
        return $this->hasMany('Comment');
    }

}

```

Svim komentarima za određeni članak moguće je pristupiti na sljedeći način:

```
$comments = Post::find(1)->comments;
```

Za inverznu relaciju na *Comment* modelu koristi se *belongsTo* metoda:

```

class Comment extends Eloquent {

    public function post()
    {
        return $this->belongsTo('Post');
    }

}

```

Many to many

Many-to-many relacije malo su kompliciranije. Primjer takve relacije bio bi korisnik koji može imati više rola, a svaka rola može pripadati različitim korisnicima. Npr. više korisnika može imati rolu *Admin*. Potrebne su tri tablice za ovu relaciju: *users*, *roles* i pivot tablica *role_users*. Tablica *role_users* treba sadržavati kolone *user_id* i *role_id*. *Many-to-many* relaciju definiramo koristeći metodu *belongsToMany*:

```

class User extends Eloquent {

    public function roles()
    {

```

```

        return $this->belongsToMany('Role');
    }
}

```

Također je potrebno definirati i inverznu relaciju na *Role* modelu:

```

class Role extends Eloquent {

    public function users()
    {
        return $this->belongsToMany('User');
    }

}

```

Ostale relacije koje se ne koriste tako često ali su iznimno korisne su:

Has-many-through koja se koristi kada treba pristupiti udaljenim relacijama kroz neki zajednički model. Npr. model država može imati više članaka kroz model korisnika.

Polimorfne relacije koriste se kada model može pripadati više od jednom modelu.

Npr. model slike može pripadati modelu korisnika ili modelu narudžbe.

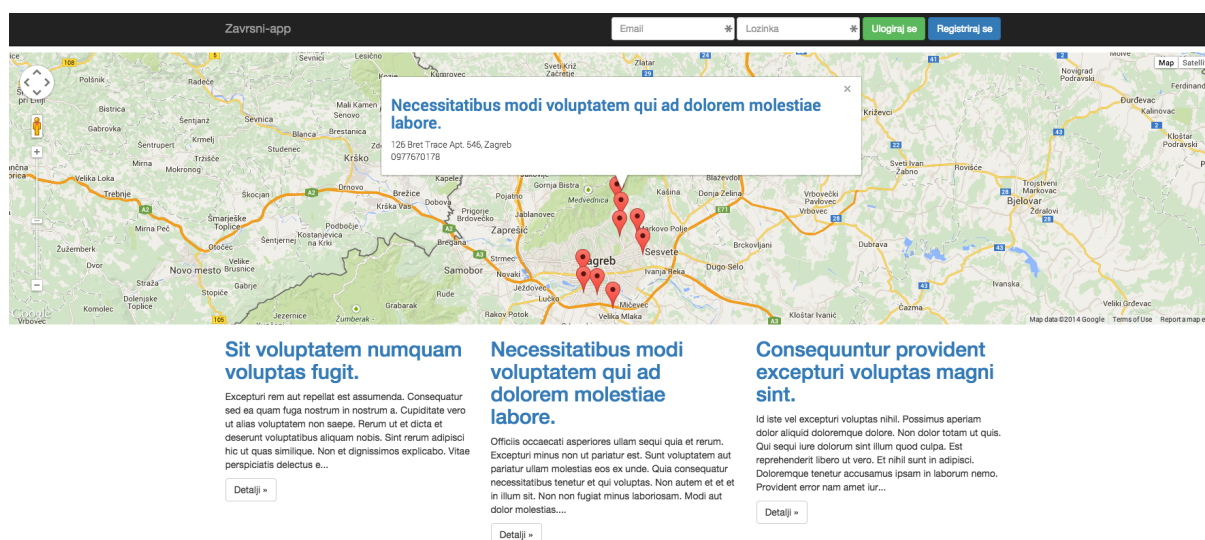
8. Izrada praktične aplikacije

8.1. Uvod i izgled aplikacije

Nakon teorijskog uvoda, vrijeme je za izradu praktične aplikacije. Aplikacija je zamišljena kao mjesto na kojem privatni iznajmljivači smještaja mogu predati svoj oglas, dodijeliti mu slike, opis, lokaciju i kontakt podatke. Za predaju oglasa, korisnik se mora registrirati kako bi kasnije mogao uređivati svoje podatke i predane oglase. Uz korisničke račune, postojat će još i administracijski račun koji za razliku od korisničkog ima ovlasti za uređivanje svih korisničkih računa i njihovih oglasa.

Radi lakše vizualizacije, u nastavku je prikazano nekoliko ekrana aplikacije.

Slika 1. Početni ekran



Na početnoj stranici aplikacije prikazani su svi oglasi koje su korisnici objavili. Prilikom predaje oglasa, iz adrese koju korisnik unese računaju se koordinate koje se koriste za prikaz lokacije na karti koristeći *Google Maps API*. U zaglavlju stranice smještena je forma za login kao i link za registraciju korisnika. Samo registrirani korisnici smiju objavljivati oglase.

Slika 2. Registracijski ekran

Završni-app

Email * Lozinka * Ulogiraj se Registriraj se

Registriraj se

Ime

Prezime

E-mail

Lozinka

Potvrdi lozinku

Registriraj se

© Igor Gudelj 2014

Prilikom registracije od korisnika se traže ime, prezime, email adresa i lozinka. Email adresa ujedno je i korisničko ime korisnika. Da bi aktivirao račun, korisnik mora kliknuti na link poslan u email poruci dobrodošlice.

Slika 3. Predaja novog oglasa

Završni-app

Igor Gudelj

Dodaj novi oglas

Naslov oglasa

Detalji oglasa

Web stranica

Adresa

Grad --- Odaberite grad ---

Slike Choose Files No file chosen

Dodaj novi oglas

© Igor Gudelj 2014

Na slici iznad prikazan je ekran za predaju oglasa. Ovdje *Javascript* funkcija prilikom predaje oglasa računa geolokacijske koordinate koje koristimo na početnom ekranu.

Slika 4. Detalji oglasa

Završni-app

Email

Lozinka

Ulogiraj se

Registriraj se

Nulla sit saepe distinctio optio aperiam.



Address: 88261 Halvorson Square Apt. 434, Zagreb
 Website: <http://lind.net>
 Phone: (394)200-7551x1875
 Email: om.roel@hegmanmarks.org



Detalji oglasa

Qui expedita quas aliquam ut suscipit natus. Fuga ut peritatur impedit et facere ea. Incidunt exercitationem voluptatem aut deserunt odit. Debitis cupiditate facilis dolorum cupiditate exercitationem qui molestias quibusdam. Tempora soluta neque et dignissimos ullam labore porro. Aliquam et nobis consequatur. Omnis ab veritatis aut odit. Quidem ratione a qui sunt. Fuga possimus maiores magnam harum dolore. Numquam sed placeat ea. Accusamus totam facilis iusto consectetur tenetur hic maiores. Qui dolor sint maiores temporibus voluptates optio. Repellat est quo rerum cumque elus. Pariatur eligendi et optio blanditia non. Dicta rem quidem explicabo tempora quia nesciunt. Doloribus elias quia cumque sequi quos qui nesciunt. Quasi ullam incidunt velit. Aut sunt maiores voluptatibus voluptatem facilis. Fuga laborum aut ratione eos voluptates facere. Non exercitationem voluptas dolores magni architecto fugit. Sint quidem qui tenetur molestiae. Iure nisi autem quaerat a non. Facilis eaque rerum nisi totam. Nisi nobis quaerat quibusdam quia quo. Ducimus esse placeat delectus quod expedita. Quam dolor amet provident. Ut aliquam eligendi omnis optio autem ut in neque. Necessitatibus quia et magni cupiditate placeat. Nisi beatae facilis ut aut aut. Eum inventore vero quae nostrum quia dolores est. Iure est voluptas hic qui quo voluptatem quis. Delectus voluptas nisi eos corrupti. Autem nam eaque aut inventore.

© Igor Gudelj 2014




Na slici iznad prikazan je izgled ekrana s detaljima oglasa. Na njemu su u prvom planu slike koje je korisnik objavio prilikom predaje oglasa. Tu je još i detaljan opis oglasa te ostali kontakt i lokacijski podaci.

Slika 5. Uređivanje korisničkog profila

Završni-app

Igor Gudelj

Uredi svoj Profil

Ime
 Prezime
 E-mail
 Adresa
 Mobitel
 Grad
 OIB
 Grupa ☒ Administrator ☐ User



Uredi

Korisnici
 Grupe
 Oglasi
 Moji oglasi
 Predaj novi oglas
 Uredi profil
 Odjavi se

Promijeni lozinku

Stara lozinka
 Nova lozinka
 Potvrdi lozinku
 Promijeni lozinku

© Igor Gudelj 2014

Na slici iznad prikazan je ekran za uređivanje korisničkog profila. Administracijski račun ima ovlasti za uređivanje svih profila, dok korisnik smije uređivati samo svoj profil. Na formi za uređivanje podaci su unaprijed popunjeni s vrijednostima iz base koristeći Laravelov *form model binding*.

Završni-app Igor Gudelj

Uredi oglas

Naslov oglasa

Quos aut soluta cupiditate necessitatibus.

Detalji oglasa

Provident rerum ratione et eos. Quod magni rem et alias ab explicabo. Aut quibusdam blanditiis voluptatum eos sed dolores officis. Molestiae id sapiente quasi officia non. Laudantium quod odio eum quaerat et. Qui maiores cum et et quod et voluptatem. Et quis debitis ex eaque ducimus placeat. Dolor ut eos vitae dolores sunt ipsum tempore. Magni amet magnam accusantium. Consequatur ut consequuntur accusamus ullam. Dolorem dolor et totam qui ut. Quibusdam occaecati ducimus fugiat eligendi perspicatis. Delectus qui dolor qui earum esse soluta at. Ipsam explicabo illum voluptatem perspicatis accusantium distinctio ea. Asperiores est mollitia laudantium possimus. Maxime quidem rerum et necessitatibus minus. Aut commodi et autem qui est aut. Neque qui laudantium odio vel deleniti. Laboriosam

Web stranica

http://spinkahoeger.com

Adresa

48279 Kobe Path

Grad

Zagreb

Slika

Choose Files No file chosen

Uredi oglas

© Igor Gudelj 2014 HR GB

Na slici iznad prikazan je ekran za uređivanje predanih oglasa. Kao i kod korisničkih računa, administrator ima ovlasti za uređivanje svih oglasa, dok korisnik smije uređivati samo svoje oglase. Podaci su kao i u prethodnom slučaju unaprijed popunjeni.

Izrada aplikacije započinje instalacijom Laravela u lokalnoj razvojnoj okolini kako je opisano u poglavlju 2.1., odnosno 2.2.

8.2. Migracije i seed datoteke

Prvi korak nakon instalacije je kreiranje baze podataka koristeći Laravel migracije i popunjavanje baze s testnim podacima koristeći *seed* datoteke. Također je potrebno kreirati i odgovarajuće modele koji će odgovarati tablicama u našoj bazi.

Kreće se od User modela, koji je nešto kompliciraniji zbog potrebe za autorizacijom i autentifikacijom korisnika. Iako Laravel oboje podržava *out-of-the-box*, u ovom projektu koristit će se *open-source* paket *Sentry*¹ koji uz autorizaciju i autentifikaciju nudi i dodatne mogućnosti poput kreiranja grupa, korisničkih dozvola, naprednih *hashing* algoritama i dodatnih sigurnosnih opcija.

¹ <https://cartalyst.com/manual/sentry/2.1>

² <https://github.com/fzaninotto/Faker>

Sentry dolazi sa svojim setom migracija koji za User model nakon vlastitih preinaka izgleda ovako:

```
// app/database/migrations/create_users_table.php

public function up()
{
    Schema::create('users', function($table)
    {
        $table->increments('id');
        $table->string('email')->unique();
        $table->string('password');
        $table->string('first_name')->nullable();
        $table->string('last_name')->nullable();
        $table->string('address')->nullable();
        $table->string('phone')->nullable();
        $table->integer('place_id')->nullable()->unsigned();
        $table->string('oib', 20)->nullable();
        $table->text('permissions')->nullable();
        $table->boolean('activated')->default(0);
        $table->string('activation_code')->nullable();
        $table->timestamp('activated_at')->nullable();
        $table->timestamp('last_login')->nullable();
        $table->string('persist_code')->nullable();
        $table->string('reset_password_code')->nullable();
        $table->timestamps();

        $table->engine = 'InnoDB';
        $table->index('activation_code');
        $table->index('reset_password_code');
    });
}
```

Kako ne bi ručno unosili testne podatke za korisnike, potrebno je napraviti *seed* datoteku koja će to automatski učiniti umjesto nas. U direktoriju *app/database/seeds* kreira se datoteka *UserTableSeeder.php*. Za kreiranje nasumičnih podataka poput imena, prezimena, adrese i sl. koristi se biblioteka *Faker*² koja se instalira kroz terminal naredbom *composer require faker*. Nakon instalacije, *seed* datoteka kreira se na sljedeći način:

```
// app/database/seeds/UserTableSeeder.php

<?php

use Faker\Factory as Faker;
use Cartalyst\Sentry\Sentry;
```

² <https://github.com/fzaninotto/Faker>

```

class UserTableSeeder extends Seeder {

    public function run()
    {
        $faker = Faker::create('hr_HR');
        $sentry = new Sentry();

        foreach(range(1, 50) as $index)
        {
            $email = $faker->email();
            $sentry->getUserProvider()->create([
                'email'      => $email,
                'password'   => 'password',
                'activated'  => 1,
                'first_name' => $faker->firstName(),
                'last_name'  => $faker->lastName(),
                'address'    => $faker->address(),
                'phone'      => $faker->phoneNumber(),
                'place_id'   => 10000,
                'oib'        => $faker->randomNumber($nbDigits
= 6) . $faker->randomNumber($nbDigits = 5),
            ]);

            // Assign user permissions
            $user = $sentry->getUserProvider()->findByLogin($email);
            $group = $sentry->getGroupProvider()->findByName('User');
            $user->addGroup($group);
        }
    }
}

```

Sličnu stvar potrebno je napraviti za grupe. Postojat će dvije grupe: Administrator i User. Migracijska i seed datoteka prikazane su u nastavku:

```
// app/database/seeds/create_groups_table.php
```

```

public function up()
{
    Schema::create('groups', function($table)
    {
        $table->increments('id');
        $table->string('name');
        $table->text('permissions')->nullable();
        $table->timestamps();

        $table->engine = 'InnoDB';
        $table->unique('name');
    });
}

```

```
// app/database/seeds/GroupsTableSeeder.php
```

```
<?php
```

```
use Cartalyst\Sentry\Sentry;
```

```
class GroupsTableSeeder extends Seeder {
```

```
    public function run()
```

```
    {
```

```
        $sentry = new Sentry();
```

```
        $sentry->getGroupProvider()->create(array(
```

```
            'name' => 'Administrator',
```

```
            'permissions' => array('admin' => 1, 'user' => 1),
        ));
```

```
        $sentry->getGroupProvider()->create(array(
```

```
            'name' => 'User',
```

```
            'permissions' => array('admin' => 0, 'user' => 1),
        ));
```

```
    }
```

```
}
```

Relacija između korisnika i grupe je *many-to-many*. Korisnik može imati više dodijeljenih grupa, a jedna grupa može pripadati različitim korisnicima. To znači da je potrebna nova pivot tablica *users_groups*. Migracija je prikazana u nastavku:

```
// app/database/seeds/create_users_groups_table.php
```

```
public function up()
```

```
{
```

```
    Schema::create('users_groups', function($table)
```

```
    {
```

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

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

```
        $table->engine = 'InnoDB';
```

```
        $table->primary(array('user_id', 'group_id'));
    });
```

```
}
```

Sljedeća migracija koju je potrebno napraviti je ona za spremanje korisničkih oglasa:

```
// app/database/seeds/create_listings_table.php
```

```
public function up()
```

```
{
```

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

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

```

        $table->integer('user_id')->unsigned();
        $table->string('title')->unique();
        $table->string('address');
        $table->integer('place_id')->unsigned();
        $table->decimal('geo_x', 10, 6);
        $table->decimal('geo_y', 10, 6);
        $table->text('description');
        $table->string('website');
        $table->timestamps();
    });
}

```

Pripadajuća seed datoteka prikazana je u nastavku:

```
// app/database/seeds/ListingsTableSeeder.php
```

```
<?php
```

```
use Faker\Factory as Faker;
```

```
class ListingsTableSeeder extends Seeder {
```

```

    public function run()
    {
        $faker = Faker::create();

        foreach(range(1, 10) as $index)
        {
            foreach(range(1, 3) as $user)
            {
                $listing = Listing::create([
                    'user_id' => $index,
                    'title' => $faker->sentence($nbWords = 6),
                    'address' => $faker->streetAddress,
                    'place_id' => 10000,
                    'geo_x' => $faker->randomFloat($nbMaxDecimals =
6, $min = 45.7, $max = 45.9),
                    'geo_y' => $faker->randomFloat($nbMaxDecimals =
6, $min = 15.9, $max = 16.1),
                    'description' => $faker->paragraph($nbSentences
= 25),
                    'website' => 'http://' . $faker->domainName,
                ]);

                foreach(range(1, 2) as $images)
                {
                    $image = new Image();
                    $image->filename = $faker->imageUrl(700,
523, city);
                    $image->listing()->associate($listing);
                    $image->save();
                }
            }
        }
    }
}

```

```

    }
}
}
}
}

```

Iduća migracija je ona za gradove koje korisnik može odabrati iz padajućeg izbornika:

```
// app/database/seeds/create_places_table.php
```

```

public function up()
{
    Schema::create('places', function(Blueprint $table)
    {
        $table->increments('id');
        $table->string('slug', 50)->unique();
        $table->string('name', 40);
        $table->decimal('geo_x', 10, 6);
        $table->decimal('geo_y', 10, 6);
        $table->timestamps();
    });
}

```

Dio pripadajuće seed datoteke prikazan je u nastavku:

```
// app/database/seeds/PlacesTableSeeder.php
```

```

public function run()
{
    Place::create([
        'id'           => 10000,
        'slug'         => 'zagreb',
        'name'         => 'Zagreb',
        'geo_x'        => '45.797451',
        'geo_y'        => '15.979244',
    ]);
}

```

Posljednja migracija je ona za slike koje će korisnik pridijeliti oglasu kojeg predaje:

```
// app/database/seeds/create_images_table.php
```

```

public function up()
{
    Schema::create('images', function(Blueprint $table)
    {
        $table->increments('id');
        $table->integer('listing_id')->unsigned();
        $table->string('filename');
    });
}

```

```

        $table->timestamps();
    });
}

```

Potrebno je napomenuti da za slike nije potrebno raditi posebnu *seed* datoteku jer su slike već dodijeljene oglasu u *ListingsTableSeeder.php* datoteci koristeći metodu *associate*.

Da bi *seed* datoteke radile ispravno, potrebno ih je najprije registrirati u datoteci *app/database/seeds/DatabaseSeeder.php*:

```

// app/database/seeds/DatabaseSeeder.php

public function run()
{
    Eloquent::unguard();

    $this->command->info('Seeding Places table!');
    $this->call('PlacesTableSeeder');

    $this->command->info('Seeding Groups table!');
    $this->call('GroupsTableSeeder');

    $this->command->info('Seeding Users table!');
    $this->call('UserTableSeeder');

    $this->command->info('Seeding Listings table!');
    $this->call('ListingsTableSeeder');
}

```

Nakon kreiranja svih migracija i *seed* datoteka sve je spremno za pokretanje migracija i tzv. seedanje baze. Migracije se pokreću naredbom *php artisan migrate*, te je nakon toga potrebno seedati bazu naredbom *php artisan db:seed*.

8.3. Modeli i relacije

Nakon uspješnog kreiranja baze i njenog popunjavanja s testnim podacima, potrebno je kreirati odgovarajuće modele i njihove relacije sukladno tablicama u bazi.

Model *User* nije potrebno kreirati jer već dolazi u samoj instalaciji radnog okvira.

Međutim, potrebno je postaviti relacije na *Place* i *Listing* kao u nastavku:

```

// app/models/User.php

public function place()
{

```



```

        return $this->hasOne('Place', 'id', 'place_id');
    }

    public function listings()
    {
        return $this->hasMany('Listing');
    }

```

Model Listing ima sljedeće relacije:

```

// app/models/Listing.php

public function user()
{
    return $this->belongsTo('User');
}

public function place()
{
    return $this->hasOne('Place', 'id', 'place_id');
}

public function images()
{
    return $this->hasMany('Image');
}

```

Model Place ima sljedeće relacije:

```

// app/models/Place.php

public function users()
{
    return $this->hasMany('User');
}

public function listings()
{
    return $this->hasMany('Listing');
}

```

I na kraju model Image sa sljedećom relacijom:

```

// app/models/Image.php

public function listing()
{
    return $this->belongsTo('Listing');
}

```

8.4. Rute

Vrijeme je za kreiranje ruta koje će koristiti ova aplikacija. Pogledajmo datoteku:

```
// app/routes.php

<?php

Route::get('/', array('as' => 'home', 'uses' =>
    'HomeController@showIndex'));

Route::get('/language/{locale}', ['as' => 'setLanguage',
    'uses' => 'LanguageController@setLocale']);

// Session Routes
Route::get('login', array('as' => 'login', 'uses' =>
    'SessionController@create'));
Route::post('login', array('as' => 'login', 'uses' =>
    'SessionController@store'));
Route::get('logout', array('as' => 'logout', 'uses' =>
    'SessionController@destroy'));
Route::resource('sessions', 'SessionController', array('only'
    => array('create', 'store', 'destroy')));

// User Routes
Route::get('register', array('as' => 'register', 'uses' =>
    'UserController@create'));
Route::get('users/{id}/activate/{code}',
    'UserController@activate')->where('id', '[0-9]+');
Route::get('resend', array('as' => 'resendActivationForm',
    function()
    {
        return View::make('users.resend');
    }
))) ;
Route::post('resend', 'UserController@resend');
Route::get('forgot', array('as' => 'forgotPasswordForm',
    function()
    {
        return View::make('users.forgot');
    }
))) ;
Route::post('forgot', 'UserController@forgot');
Route::post('users/{id}/change', 'UserController@change');
Route::get('users/{id}/reset/{code}', 'UserController@reset')->
    where('id', '[0-9]+');
Route::resource('users', 'UserController');

// Group Routes
Route::resource('groups', 'GroupController');
```

```
// Listing Routes
Route::resource('listings', 'ListingsController');
Route::get('users/{id}/listings',
'ListingsController@showByUserId');
```

Rute su detaljno obrađene u trećem poglavlju tako da sada neće biti previše riječi o njima. Treba samo primijetiti da se gotovo svaka ruta veže na akciju odgovarajućeg kontrolera, te korištenje RESTful rutiranja kod korisnika, oglasa i grupa.

8.5. Pregledi (views)

Na osnovu kreiranih ruta potrebno je kreirati odgovarajuće *view* datoteke. Prvi *view* koji je potrebno napraviti je tzv. *default* ili *master view* koji se najčešće sprema u */views/layouts* direktorij. U njemu je potrebno kreirati kostur stranice koji će se pojavljivati kroz cijelu aplikaciju. Tu spadaju otvaranje i zatvaranje *html*, *head* i *body* tagova, prikaz *meta* podataka, uključivanje *css* i *javascript* datoteka i sve ostalo što je potrebno. Sljedeća stvar je kreiranje *header* i *footer* datoteka. Njih je uobičajeno spremiti u */views/layouts/partials* direktorij. Nakon njihova kreiranja, potrebno ih je uključiti u *default view*. To se radi tako da u *default.blade.php* datoteci na odgovarajuće mjesto dodamo liniju `@include('layouts.partials.header')`, odnosno `@include('layouts.partials.footer')`.

Sve ostale *view* datoteke vezat će se na *default view* tako da ga proširuju (engl. *extend*). U *default.blade.php* na mjesto gdje treba doći sadržaj potrebno je dodati liniju `@yield('content')`. Pregled koji proširuje *default view* treba izgledati ovako:

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

@section('content')

{{-- Content --}}

@stop
```

8.6. Kontroleri

O kontrolerima je detaljno bilo riječi u četvrtom poglavlju. Oni su logika aplikacije i služe za predaju podataka iz baze do pregleda koji su kreirani u prethodnom koraku. Pogledom na kreirane rute, primjećuje se da su potrebni sljedeći kontroleri:

HomeController – služi za prikazivanje početne stranice,

LanguageController – služi za promjenu jezika aplikacije,

SessionController – služi za *login* i *logout* korisnika,

UserController – služi za kreiranje i aktivaciju korisnika, te uređivanje njihovih podataka,

GroupController – služi za kreiranje i uređivanje grupa,

ListingsController – služi za kreiranje novih oglasa i njihovo uređivanje.

Na primjeru *UserControllera* pogledajmo kako ispravno koristiti kontrolere. Uzmimo kao primjer metodu *store()* koja je zadužena za obradu registracije korisnika. Kada se korisnik pokuša registrirati, potrebno je napraviti sljedeće: dohvatiti podatke iz forme koje je korisnik unio, validirati te podatke, prikazati korisniku validacijske greške ako postoje ili kreirati korisnika u bazi podataka ako je validacija uspjela, poslati korisniku email dobrodošlice, itd. Ako bi za svaku metodu u kontroleru proceduralno pisali logiku aplikacije, oni bi vrlo brzo postali nepregledni. Iz tog razloga, dijelovi logike u kontroleru razdvajaju se u zasebne klase koje onda u kontroleru dohvaćamo koristeći *dependency injection* kroz konstruktor. Npr. validacija je izdvojena u zasebnu klasu, obrada podataka iz forme također je izdvojena u zasebnu klasu i obrađuje se koristeći *repository pattern*. U konstruktoru također definiramo filtere koje pojedine metode moraju zadovoljiti. Zasebne klase stavljaju se u novi direktorij kojeg je potrebno kreirati. Praksa je da se taj direktorij nazove imenom aplikacije koja se razvija. U ovom slučaju, direktorij s izdvojenim klasama zove se *Završni*. Metoda *store* prikazana je u nastavku:

```
public function store()
{
    // Form Processing
    $result = $this->registerForm->save( Input::all() );

    if( $result['success'] )
    {
        Event::fire('user.signup', array(
            'email' => $result['mailData']['email'],
            'userId' => $result['mailData']['userId'],
            'activationCode' =>
                $result['mailData']['activationCode']
        ));

        // Success!
        Session::flash('success', $result['message']);
    }
}
```

```

        return Redirect::route('home');

    } else {
        Session::flash('error', $result['message']);
        return Redirect::action('UserController@create')
            ->withInput()
            ->withErrors( $this->registerForm->errors() );
    }
}

```

8.7. Ostalo

Kada je neki podatak potrebno prikazati u svim pregledima u aplikaciji, koriste se **view composeri**. Npr. za prikaz imena i prezimena ulogiranog korisnika u headeru aplikacije ili za prikaz dostupnih jezika u footeru aplikacije.

Višejezičnost se vrlo jednostavno implementira u Laravel radnom okviru. U *app/lang* direktoriju potrebno je kreirati 2 nova direktorija (*/en* i */hr*) i organizirati php datoteke sukladno potrebama. U php datotekama napravi se polje po principu ključ vrijednost. Zatim se u *view* datotekama fraze pripremaju za prijevod koristeći Laravel *helper* funkciju *trans*, npr. *trans('users.created')*. U konfiguracijsku datoteku *app.php* potrebno je dodati sljedeću liniju: *'languages' = array('en', 'hr')*. Ostatak konfiguracije može se pronaći u zasebnom direktoriju *Zavrsni/Languages* kao i *LanguageControlleru*. Također je potrebno kreirati i rute odgovorne za promjenu jezika.

Većina korištenih **filtera** su standardni Laravel filteri uključeni s radnim okvirom koji provjeravaju da li je posjetitelj gost ili ulogirani korisnik. Vlastito kreirani filteri provjeravaju da li je korisnik u određenoj grupi (*inGroup*), te da li ima pravo uređivati oglas (*canEditListing*). Tu je također *before* filter koji određuje jezik koji aplikacija treba prikazati.

Email notifikacije rade pomoću Laravel događaja (engl. *events*). Npr. kada se korisnik registrira, u kontroleru kreiramo događaj *'user.signup'* koristeći metodu *Event::fire*. U datoteci *app/observables.php*, koristeći metodu *Event::listen*, slušamo kada će se očekivani događaj desiti i zatim ga šaljemo na klasu *UserMailer* u direktoriju *Zavrsni/Mailers*.

8.8. Postavljanje na server

Laravel aplikacije vrlo se jednostavno postavljaju na server. Koristeći *git*, sa servera je potrebno povući aplikacijski kod. Nakon toga je potrebno pokrenuti naredbu *composer install*, te zatim pokrenuti migracije i seed datoteke naredbama *php artisan migrate* i *php artisan db:seed*. Jedina razlika između lokalne konfiguracije i konfiguracije na serveru je u bazi podataka. Iz tog razloga potrebno je u *root* direktoriju kreirati datoteke *.env.local.php* i *.env.php*. Sadržaj *.env.local.php* datoteke je sljedeći:

```
<?php

return [
    'DB_HOST'      => 'localhost',
    'DB_USERNAME'   => 'homestead',
    'DB_PASSWORD'   => 'secret',
    'DB_NAME'       => 'new'
];
```

Ekvivalentnu *.env.php* datoteku potrebno je postaviti na server s odgovarajućim vrijednostima. Ove datoteke potrebno je dodati u *.gitignore* kako se ne bi verzionirale i otkrile osjetljive podatke za spajanje na bazu podataka.

Sada je potrebno u konfiguracijskoj datoteci *database.php* prilagoditi *mysql* polje na sljedeći način:

```
'mysql' => array(
    'driver'      => 'mysql',
    'host'        => getenv('DB_HOST'),
    'database'    => getenv('DB_NAME'),
    'username'    => getenv('DB_USERNAME'),
    'password'    => getenv('DB_PASSWORD'),
    'charset'     => 'utf8',
    'collation'   => 'utf8_unicode_ci',
    'prefix'      => '',
)
```

Funkcija koja dohvaća odgovarajuće vrijednosti je *getenv()*.

Zadnja stvar koju je potrebno napraviti je automatski odrediti u kojem okruženju se nalazimo, lokalnom ili produkcijskom. To se može učiniti u datoteci *bootstrap/start.php* na sljedeći način:

```
$env = $app->detectEnvironment(array(
    'local' => array('homestead'),
    'production' => array('exquisite-pebble'),
));
```

Homestead predstavlja naziv lokalnog servera, a *exquisite-pebble* je u ovom slučaju ime produkcijskog servera.

Treba spomenuti i servis Laravel *Forge*³, kreiran od strane samog autora Laravela Taylora Otwell, koji cijeli proces postavljanja na server uvelike olakšava.

³ <https://forge.laravel.com>

9. Zaključak

Kroz ovaj rad obrađene su osnove radnog okvira Laravel i njegove najbitnije funkcionalnosti. Također je prikazana struktura direktorija i način na koji Laravel implementira MVC (*model-view-controller*) arhitekturu.

Iako vrlo mlad radni okvir, Laravel je izuzetno cijenjen i vrlo popularan među *developerima*. Prema anketi iz prosinca 2013. u kojoj su *developeri* glasali o trenutno najpopularnijim PHP radnim okvirima, Laravel je zauzeo prvo mjesto ispred Phalcona, Symfony2, CodeIgnitera i ostalih.⁴ U kolovozu 2014. Laravel je postao najpopularniji i najpraćeniji PHP projekt na stranici GitHub.⁵

Sa svojom jednostavnom sintaksom, odličnom dokumentacijom i izvrsnim materijalima za učenje, vrlo je lako savladati osnove i izraditi nešto korisno u jako kratkom vremenu.

Jedina zamjerka koja se spominje o radnom okviru Laravel je njegov prebrzi razvoj. Zaista, prva verzija Laravela predstavljena je u lipnju 2011. Već nakon nepunih pola godine (studenj 2011.) izlazi nova verzija Laravel 2. Četiri mjeseca nakon toga (veljača 2012.) izlazi stabilna verzija Laravel 3 uz novu web stranicu. Trenutna verzija radnog okvira je Laravel 4, koji je predstavljen u svibnju 2013. Verzija 4 bitno se razlikuje od prijašnjih verzija i napisana je praktički iz početka. Sastoji se od kolekcija *composer* paketa koji međusobno povezani čine radni okvir. Najavljena verzija Laravel 5 treba izaći u veljači 2015. Uz brojna poboljšanja i dodatke, najveća razlika očekuje se u strukturi direktorija. Trenutni razvoj moguće je pratiti na stranici GitHub.⁶

U svakom slučaju, Laravel je radni okvir na kojeg treba obratiti pozornost. Svakako bih ga preporučio početnicima za razvijanje web aplikacija, ali i iskusnim korisnicima. Osnovni koncepti lako se uče, aplikacije se razvijaju brzo i jednostavno te je zaista užitak raditi s ovim sjajnim radnim okvirom.

⁴ Bruno Skvorc, "Best PHP Frameworks for 2014" – sitepoint.com (prosinac 2013.)

⁵ Most popular and watched PHP projects – github.com (kolovoz 2014.)

⁶ Laravel develop branch – github.com (studenj 2014.)

10. Literatura

1. Christopher Pitt and Taylor Otwell: Laravel 4 Cookbook, Projects you can build to learn Laravel 4; Leanpub 2013.
2. Dayle Rees: Code Bright, Web application development for the Laravel framework version 4 for beginners; Leanpub, 2014.
3. Taylor Otwell: From Apprentice To Artisan, Advanced Architecture With Laravel 4; Leanpub, 2013.
4. Taylor Otwell: Official Laravel documentation, <http://laravel.com/docs/4.2>
5. Jeffrey Way: The best Laravel and PHP screencasts, <https://laracasts.com/>

