

# Razvoj web aplikacije u Laravelu

---

**Trbojević, Milan**

**Undergraduate thesis / Završni rad**

**2018**

*Degree Grantor / Ustanova koja je dodijelila akademski / stručni stupanj:* **University of Rijeka / Sveučilište u Rijeci**

*Permanent link / Trajna poveznica:* <https://um.nsk.hr/um:nbn:hr:195:006720>

*Rights / Prava:* [In copyright](#)/[Zaštićeno autorskim pravom.](#)

*Download date / Datum preuzimanja:* **2024-09-23**



*Repository / Repozitorij:*

[Repository of the University of Rijeka, Faculty of Informatics and Digital Technologies - INFORI Repository](#)



Sveučilište u Rijeci – Odjel za informatiku  
Preddiplomski jednopredmetni studij informatike

Milan Trbojević

# Razvoj *web* aplikacije u *Laravelu*

Završni rad

Mentorica: doc. dr. sc. Marina Ivašić-Kos

Rijeka, lipanj 2018

## Sadržaj

1. Uvod .....	3
2. Arhitektura web aplikacije i radno okruženje .....	5
2.1 MVC arhitektura .....	5
2.2 Radno okruženje (framework) .....	7
3. Uvod u Laravel .....	9
4. Laravelovo radno okruženje .....	10
4.1 Composer .....	10
4.2 PHP Artisan .....	11
4.3 Laravelove značajke i nužne postavke .....	12
5. Analiza strukture novog Laravelovog projekta .....	13
5.1 Analiza app direktorija .....	14
5.2 Eloquent, ORM, modeli i migracije .....	15
5.3 Http poddirektorij .....	19
5.4 Mehanizam za upravljanje tokovima (routing) .....	20
5.5 Resursni upravitelj i tokovi .....	21
5.6 Middleware .....	21
5.7 Analiza node_modules direktorija .....	22
5.8 Analiza public direktorija .....	22
5.9 Pogledi .....	23
5.10 Korisne kratice (shortcuts) .....	26
5.11 Ostali direktoriji i korisne datoteke .....	26
6. Alternative za korištenje sličnih radnih okruženja .....	28
7. Analiza komponenti aplikacije društvene mreže .....	29
7.1 Autentifikacija korisnika .....	29
7.2 Objave (statusi) .....	30
7.3 Like funkcionalnost .....	35
7.2 Messenger .....	37
7.3 Uređivanje korisničkog profila .....	40
7.4 Obavijesti (notifikacije) .....	43
7.5 Pretraga korisnika .....	46
8. Zaključak .....	49
9. Literatura .....	50
10. Prilozi .....	51
10.1 Laravelova instalacija .....	51

## 1. Uvod

*PHP* je kao programski jezik već dugo jedan od najpopularnijih skriptnih serverskih jezika. Tako da ni ne čudi činjenica da su i moja prva iskustva vezana upravo uz *PHP*. Kada krenete raditi na nešto većim i ozbiljnijim projektima bez obzira o kojem je programskom jeziku riječ, počnu se pojavljivati neki problemi koji s vremenom dosta usporavaju izradu vaše *web* aplikacije pa samim time smanjuju i vašu efikasnost kao programera što naposljetku negativno utječe na samog klijenta i korisnika aplikacije. Kao rješenje tih problema programeri najčešće koriste provjerena radna okruženja, gdje dolazimo do popularnog pojma *framework*. Tako je i "pala" odluka na *Laravel* [1], trenutno najpoznatiji *PHP*-ovo radno okruženje (*framework*) za izradu *web* aplikacija.

Uz samu analizu *Laravel* radnog okruženja (*frameworka*) i njegovih značajki izrađena je i mala društvena mreža (u usporedbi sa *Facebookom*) koja će poslužiti kao dobar praktičan primjer. *Web* aplikacija u obliku društvene mreže je odabrana na temelju učestalosti korištenja *web* aplikacija gdje su se društvene mreže pokazale jedne od najkorištenijih oblika *web* aplikacija (pri tome se najviše misli na populaciju studenata). Današnje društvene mreže sadrže mnogo korisnih značajki kako bi svojim korisnicima omogućile da što lakše ostanu u kontaktu sa svojim prijateljima, obitelji, obožavateljima itd. Upravo taj veliki broj korisnih značajki predstavlja i određeni izazov u izradi istih.

U ovom završnom radu najviše će biti naglaska na *backend* programiranje, odnosno funkcionalnost *web* aplikacije. Uz *Laravelov backend* za frontend je korišteno *Javascriptovo* radno okruženje *Vue.js*. *Vue.js* biti će spomenut u sklopu prikaza podataka.

Osim *Vue.js* u kombinaciji sa *Laravelom* se najčešće koriste i *Angular.js* te popularni *React.js*. *Web* aplikacija je izrađena u kombinaciji sa *Vue.js* zbog bolje podržanosti i sve veće popularnosti s obzirom na ostala *Javascript* radna okruženja.

Sve je usmjerneno ka tome da sve tvrdnje budu što je više moguće potkrepljene dijelom koda iz izrađene *web* aplikacije. Osim dijelova koda biti će zastupljene i neke vrste dijagrama kada bude više riječi o samoj arhitekturi. Završni rad nije zamišljen da se obrade baš sve značajke već da se one najbitnije i najkorištenije dobro analiziraju.

Kako bi se lakše mogli pratiti sadržaj završnog rada potrebno je predznanje iz kolegija kao što su: Baze podataka, Objektno orijentirano programiranje, Dinamičke *web* aplikacije, Modeliranje podataka, Računalne mreže i slični kolegiji. Dok će se ostali pojmovi kao što je pojam *framework*-a ili MVC arhitekture *web* aplikacije posebno pojasniti.

Ako izuzmemo uvod, neka od glavnih poglavlja koja će biti obrađena su:

- Pojam arhitekture *web* aplikacije i pojam radnog okruženja
- Povijesni razvoj *Laravela*
- *Laravelovo* radno okruženje
- Struktura novostvorene *web* aplikacije u *Laravelovom* okruženju
- Analiza najbitnijih stavki aplikacije društvene mreže izražene u *Laravelu*
- Prednosti i mane *Laravela*
- Alternative slične *Laravelovom* radnom okruženju



Slika 1 - *Laravelov* logo [[http://thingsaker.com/assets/images/blog/laravel\\_new.png](http://thingsaker.com/assets/images/blog/laravel_new.png)]

## 2. Arhitektura *web* aplikacije i radno okruženje

*Web* aplikacija je računalni program koji povezuje *web* preglednik i *web* tehnologije da bi izvršavao neki vrstu posla ili zadatka na internetu [2]. Funkcioniraju na način da:

1. Korisnik napravi zahtjev prema *web* serveru (npr. klikne na gumb) putem internetske mreže ili putem svog *web* preglednika.
2. Nakon toga *web* server prosljeđuje zahtjev određenom *web* aplikacijskom serveru.
3. *Web* aplikacijski server obradi zahtjev (npr. zahtjev za podacima u bazi podataka koja se nalazi na serveru), te izgenerira rezultat tog zahtjeva.
4. Rezultat se šalje prema *web* serveru koji je poslao zahtjev.
5. *Web* server nakon zaprimanja rezultata šalje povratnu informaciju o primitku te prikaže rezultat korisniku.

Najčešće se koriste u obliku *web* trgovina (*eBay*, *Amazon*, *AliExpress*), servisa za slanje *e-mail* pošte (*Gmail*, *Yahoo*), produkciju video zapisa (*Youtube*) itd. Neke od glavnih prednosti *web* aplikacije su da se one mogu koristiti neovisno o operacijskom sustavu ili uređaju, te ne zauzimaju prostor na tvrdom disku vašeg uređaja kako bi ih koristili.

### 2.1 MVC arhitektura

Što je *web* aplikacija veća ona naravno zahtjeva i više koda, više poddirektorija gdje će Vaše skripte bit razvrstane, više direktorija gdje su smješteni poddirektoriji razvrstani po funkciji koju obavljaju unutar *web* aplikacije itd. Dakle, najviše problema je prije svega vezano uz samu strukturu Vaše *web* aplikacije. Bolja struktura ili organizacija stavki Vaše *web* aplikacije omogućiti će nam bolji pregled obavljenog posla u svakome trenutku, lakšu izmjenu koda, nadopunjavanje novog koda itd. U kraćim crtama boljom organizacijom koda štedimo vrijeme koje bi inače potrošili na snalaženje u vlastitom kodu.

*MVC (model-view-controller)* je vrsta softverske arhitekture koja opisuje način na koji da strukturirate vašu aplikaciju [3]. Pokazuje koje su zadaće i interakcije svakog dijela vaše strukture. Trenutno je jedna od najpopularnijih arhitektura u današnjem razvoju *web* aplikacija. Osim *Laravela* koriste je i mnoga poznata radna okruženja: *Ruby on Rails (Ruby)*, *Codeigniter (PHP)*, *Zend (PHP)*, *Angular (Javascript)*, *Django (Python)*, *Flask (Python)* i mnogi drugi. [4]

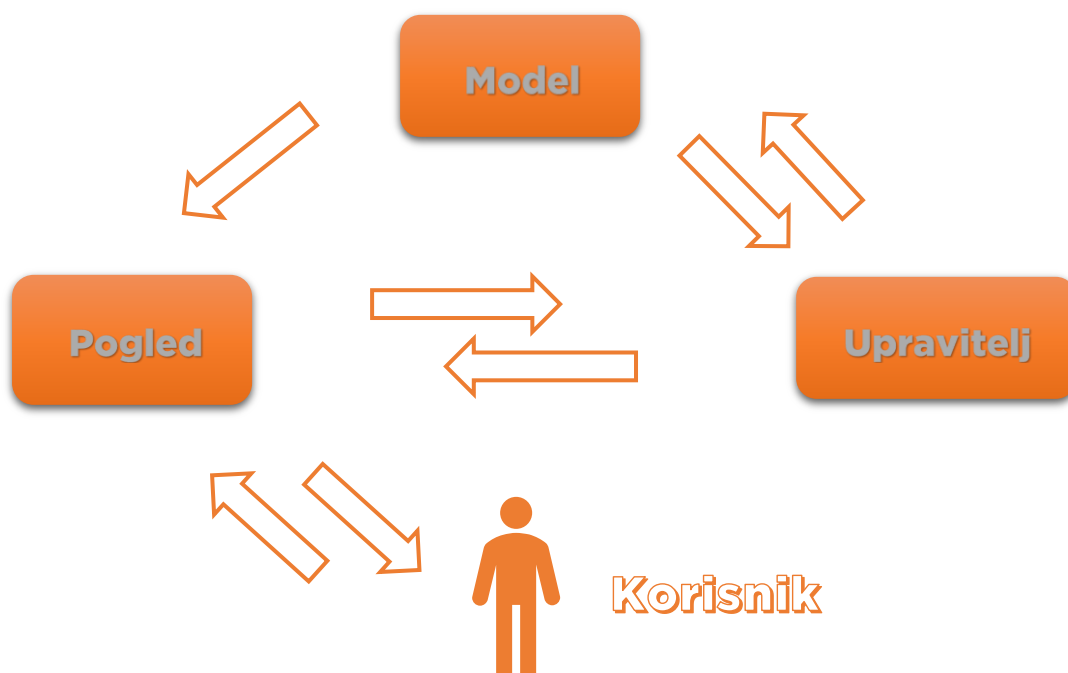
Kao što se može primjetiti *MVC* arhitektura se sastoji od 3 dijela:

**Model** – Dio arhitekture podređen podatkovnoj logici te odgovoran za interakcije sa bazom podataka. Komunicira sa upraviteljem te ovisno o radnom okruženju ponekad može ažurirati pogled. Obično predstavlja jednu ili više klasa, te ima ulogu poveznice između upravitelja i pogleda. Osim toga model je zadužen i za upravljanje i obradu podataka. U sebi najčešće sadrži metode koje mogu manipulirati podacima.

**Pogled (view)** – Predstavlja korisničko sučelje ili *UI (User Interface)* ili ono što korisnik zapravo vidi. Obično sastavljen od *HTML* i *CSS-a*. Komunicira sa upraviteljem, odnosno prikazuje dinamički dobivene vrijednosti iz upravitelja.

**Upravitelj (controller)** – Odgovoran da primi zahtjev korisnika neovisno radi li se o kliku na neki link, potvrdi ispunjene *web* forme, ili primjerice zahtjeva za ažuriranjem nekog podatka u bazi podataka. Njegova glavna uloga je zapravo uloga posrednika između modela i pogleda. On na zahtjev korisnika uzme podatke iz određenog modela i prosljedi ih prema pogledu, gdje je korisnik u mogućnosti vidjeti

tražene podatke. Naravno, ukoliko korisnik nema zahtjev za nekom vrstom podataka upravitelj je tada odgovoran samo za prikaz traženog pogleda.



Slika 2 – MVC dijagram

Kao što vidimo na dijagramu osim MVC dijela arhitekture prikazan je i korisnik kako bi bilo jasnije gdje "putuje" zahtjev korisnika u aplikaciji. Kada korisnik napravi zahtjev na pogledu (zahtjev može biti napravljen i samim otvaranjem pogleda), pogled komunicira sa upraviteljem koji dohvaća podatke od modela, upravitelj zatim izvršava potrebne radnje ( prikaz, ažuriranje, unos, ili brisanje podataka te povratnu informaciju prosljeđuje natrag u pogled. Model i upravitelj su u pozadinskom dijelu *web* aplikacije dok je pogled jedini vidljiv korisniku. Osim opisanog toka zahtjeva, moguća je i verzija gdje se iz modela podatci prosljeđuju pogledu bez naknadnog vraćanja upravitelju.

Kada je arhitektura *web* aplikacije strukturirana na MVC način uvelike je olakšano naknadno održavanje *web* aplikacije, njezino tesitranje te naravno izmjena napisanog koda. Možemo izmijeniti neki dio *web* aplikacije bez straha da će to imati loš utjecaj na druge dijelove aplikacije. U današnje vrijeme u razvoju *web* aplikacije najčešće sudjeluje više od jednog programera, što bi bez pravilne arhitekture napravilo probleme u praćenju razvitka *web* aplikacije. Korištenjem MVC arhitekture dok jedan programer primjerice mjenja bazu podataka drugi može nesmetano raditi na prikazu viewa i prakticki koja god promjena da se dogodi neće utjecati na rad *web* aplikacije.

Kako bi se povećala dinamičnost podataka najčešće se u pogledu arhitekture koriste *Template Engines* [5]. Odnosno sustavi grafičkih predložaka, koji osim dinamičkih podataka mogu prikazivati i varijable iz controllera koje ne bismo bili u mogućnosti prikazati korištenjem čistog *HTML-a*. Jedna od najbitnih značajki *Template Enginea* je ta da on uvelike smanjuje ponavljanje *HTML* koda tj. strukture stranice [5]. U mogućnosti smo primjerice napraviti jedinstveni predložak (*template*) i na svakoj slijedećoj *HTML* stranici koju stvorimo pisati samo sadržaj koji se razlikuje od glavnog predloška.

## 2.2 Radno okruženje (framework)

Možda ćemo najbolje shvatiti zašto se pojam radnog okruženja (*frameworka*) tako često koristi u današnjem svijetu web aplikacija tako da prije same definicije navedemo neke probleme koje on rješava [6]:

- Čini rad sa kompleksnim tehnologijama mnogo lakšim.
- Stvara "zdrave navike" u programiranju koje u budućnosti aplikacije stvaraju programsku okolinu fleksibilniju sa manje kvarova/pogrešaka (*bugova*).
- Svatko može lako testirati i debugirati kod aplikacije i ako nije sudjelovao u pisanju koda

Radno okruženje ili *framework* je struktura aplikacije koju kasnije nadograđujemo vlastitim funkcionalnostima [6]. Sama definicija radnog okruženja govori nam kako je to unaprijed pripremljen temelj web aplikacije sa određenom količinom složenosti (baziran na korisničkoj razini) koji programer može proširiti sa vlastitim kodom. Najčešće u sebi sadrži komplet softverskih biblioteka, kompajlere, interpretere ili aplikacijsko programsko sučelje (*application programming interface, API*). U globalu pruža okolinu koja olakšava određenom tipu programiranja razvitak softvera. U izgradnji softvera bez korištenja radnog okruženja uobičajeno je da sam programer uvijek definira tijekom programa te zahtjeve prema određenim bibliotekama. Dok je kod korištenja radnog okruženja situacija obrnuta (*inversion of control, IoC*). Osim već navedenih sastavnica radnog okruženja valja još spomenuti multimedijско stvaralaštvo (*multimedia authoring*), alate za lakše vođenje skripti (*scripting tools*) i srednji sloj web aplikacije (*middleware*).

Radno okruženje svojom instalacijom dolazi sa mnoštvom biblioteka koje su potrebne za izradu ipak nešto veće web aplikacije. No, želimo li napraviti neku vrstu

aplikacijskog programskog sučelja ili manju web aplikaciju, ipak nećemo koristiti većinu svojstava (*featurea*) koje nam klasično radno okruženje nudi.

Pa tako osim samog radnog okruženja valja spomenuti i pojam umanjenog ili mikro radnog okruženja (*microframework*). Najveći naglasak mikro radnog okruženja je naravno na njegovoj brzini.



Slika 3 - Web radna okruženja [<http://www.systemseeders.com/wp-content/uploads/2018/04/blog-banner.jpg>]



Mikro radno okruženje dolazi bez većine klasičnih svojstava kao što su:

- Autentifikacija korisnika, autorizacija korisnika, uloge (*roles*)
- Povezanost sa bazom podataka preko objektno orijentiranog *mappinga*
- Validacija formi
- *Template Engine* itd.



Slika 4 - Web mikro radna okruženja

### 3.Uvod u *Laravel*

*Laravel* je *web* radno okruženje bazirano na *PHP-u* odnosno *Symfony* radnom okruženju [8]. Njegov osnivač je Taylor Otwell. Prije svega glavni cilj je bio popularizacija razvoja *web* aplikacija koje se zasnivaju na *MVC* arhitekturi. Njegova prva verzija nastala je kao Otwellov pokušaj da osigura napredniju alternativu tada popularnom *CodeIgniter* radnom okruženju [9].

Prva beta inačica *Laravela* pojavila se 9. lipnja 2011. godine, dok je prva službena verzija pod nazivom *Laravel 1* izašla krajem istog mjeseca. Nakon prva tri *Laravela* odlučeno je da se za *Laravel 4* napravi potpuna "prerada" koda. Izlaskom *Laravel 5.1* rečeno je kako će ta verzija imati nešto dugoročniju podršku (*long-term support, LTS*) od najmanje 3 godine. Te je planiran izlazak inačica svake dvije godine.

*Laravel 1* – Objavljen krajem lipnja 2011. godine. Uključena podrška za autentifikaciju korisnika, lokalizacija, modeli, pogledi (*views*), sesije (*sessions*), mehanizam za kontrolu toka zahtjeva u aplikaciji (*routing*) i ostali mehanizmi. Gdje je naravno najviše nedostajala podrška za upravitelje (*controllers*) što je bila prepreka da *Laravel* već tada bude radno okruženje sa *MVC* arhitekturom.

*Laravel 2* – Već nakon nepuna tri mjeseca, objavljena je nova verzija gdje je došlo do značajnog napretka od strane autora i tada već značajno velike zajednice *Laravel* korisnika (*community*). Glavni novitet nove *Laravelove* inačice bili su upravitelji (*controllers*). Osim upravitelja dodan je princip inverzije kontrole (*IoC*), i sustav predložaka (*templating system*) naziva *Blade*.

*Laravel 3* – U korištenje pušten u veljači 2012. godine. Noviteti su bili: sučelje naredbenog retka (*command-line interface, CLI*) naziva *Artisan*, podrška za nešto bolje upravljanje bazama podataka, migracije, podrška za rukovanje *eventima* te sustav za upravljanje paketima naziva *Bundles*. Jedan od najznačajnijih skokova *Laravelove* popularnosti vezan je upravo uz izlazak ove inačice.

*Laravel 4* – Kodnog imena *Illuminate* objavljen u svibnju 2013. godine, obilježila je potpuna "prerada" *Laravel* radnog okruženja. Od četvrte *Laravelove* verzije za raspored njegovih datoteka odgovoran je *Composer* koji ima ulogu menadžera za pakete. *Laravel* je od tada postao skup paketa koji u organizirano radno okruženje pretvara menadžer za pakete, u ovom slučaju *Composer*. Od ostalih noviteta važno je istaknuti *seeding* baze podataka te podršku za sustav slanja poruka i e-mailova.

*Laravel 5* – Objavljen je u veljači 2015.godine uzrokovan skupom sitnih promjena na *Laravel 4.3* inačici. Nova svojstva osim nove strukture direktorija, uključuju: *Scheduler* (periodično izvršavanje naloga), *Flysystem* (namjenjen za udaljenu pohranu podataka) te *Elixir* (služi za upravljanje novim paketima). Kako je za *Laravel 5.1* inačicu zamišljen već spomenuti *LTS*. Što se tiče noviteta značajniju promjenu *Laravel* je doživio *Laravel 5.4* inačicom gdje valja spomenuti: *Laravel Dusk*, *Laravel Mix*, nove *Blade* komponente, te napretke na razini *routinga*. Zadnja inačica *Laravel 5.6* objavljena je sedmog veljače 2018. godine.

Svake godine se u SAD-u (prva polovina godine) i Europi (druga polovina godine), odnosno Amsterdamu održavaju konferencije pod imenom *Laracon*, na kojima se prati razvoj i uporaba *Laravel web* radnog okruženja.

## 4. *Laravelovo* radno okruženje

U nastavku će biti navedena dva najvažnija alata koja se koriste pri izradi *web* aplikacije u *Laravelu*. Oba alata koriste se kako bi ubrzali postupke poput stvaranja novog modela, upravitelja, pogleda kao i instaliranje novih paketa u našem projektu.

### 4.1 *Composer*

*Composer* je alat za upravljanje paketima u aplikacijama napisanim u *PHP-u*. Koristeći *Composer* možemo "instalirati" i ažurirati sve pakete koji su na potrebni u aplikaciji. "Instalirati" paket u ovom slučaju znači preuzeti (*download*) sve datoteke od željenog paketa i spremiti tih na određenom mjesto (direktorij) u našoj aplikaciji.

Da bi smo bili u mogućnosti koristiti *Composer* moramo kao preduvjet imati instaliran *PHP* interpreter. Ovisno o svom operacijskom sustavu preuzmemo verziju *Composera* sa njegove službene *web* stranice ([getcomposer.org](http://getcomposer.org)) gdje se nalaze i upute za instalaciju. Kada smo preuzeli *Composer* i instalirali ga na svom računalu spremni smo za njegovo korištenje. U komandnoj liniji ili terminalu naredbom ***composer*** možemo još jednom provjeriti jesmo li pravilno napravili instalaciju.

Struktura *Composer* naredbe: ***composer require naziv-razvijatelja (developera)/naziv paketa {verzija-paketa}***. Definiramo koji novi paket želimo preuzeti i spremiti u našu aplikaciju. Po defaultu stvaranjem nove *Laravel web* aplikacije stvori se *composer.json* datoteka u kojoj su u obliku JSON (Javascript Object Notation) teksta zapisani svi instalirani paketi aplikacije potrebni za njezin rad gdje se naravno instalacijom novih paketa *composer.json* datoteka ažurira. Naredbom ***composer install*** *Composer* nam osigurava da će sa našim instaliranim paketom doći i svi potrebni paketi koji se nalaze u njegovoj konfiguraciji i pomažu pri njegovom radu.

```
(c) 2018 Microsoft Corporation. All rights reserved.
C:\Users\riverDeer>composer

Composer version 1.6.3 2018-01-31 16:28:17

Usage:
  command [options] [arguments]

Options:
  -h, --help                Display this help message
  -q, --quiet               Do not output any message
  -V, --version             Display this application version
  --ansi                    Force ANSI output
  --no-ansi                 Disable ANSI output
  -n, --no-interaction     Do not ask any interactive question
  --profile                 Display timing and memory usage information
  --no-plugins              Whether to disable plugins.
  -d, --working-dir=WORKING-DIR
                           If specified, use the given directory as working directory.
  -v|vv|vvv, --verbose     Increase the verbosity of messages: 1 for normal output, 2 for more verbose output and 3 for debug
```

Slika 5 – *Composerov* Help screen

Nakon instalacije paketa u ovom slučaju novog *Laravel* projekta *Composer* će ažurirati i direktorij *vendor* (svi nužni paketi koji su zapisani u *composer.json* datoteci) te datoteku *composer.lock* (sadrži informacije o verziji *web* aplikacije).

## 4.2 PHP Artisan

*Artisan* je sučelje naredbenog retka koje koristi *Laravel*. Zaslužno je za mnogo korisnih naredbi koje će ubrzati rad na aplikaciji. Nakon što se pozicioniramo unutar direktorija našeg *Laravel* projekta naredbom ***php artisan list*** izlistati će nam se sve naredbe koje možemo koristiti sa *Artisanom*.

Svaka naredba u sebi sadrži pomoćni ekran/prozor (*help screen*) koji prikazuje i opisuje „prirodu“ određene naredbe, ako nam zatreba pomoć u vezi određene naredbe ključnu riječ ***help*** napisati ćemo prije naredbe o kojoj želimo dobiti dodatne informacije (npr. ***php artisan help migrate*** ).

```
D:\xampp\htdocs\socialnetwork>php artisan help migrate
Usage:
  migrate [options]

Options:
  --database[=DATABASE] The database connection to use.
  --force                Force the operation to run when in production.
  --path[=PATH]          The path to the migrations files to be executed.
  --realpath             Indicate any provided migration file paths are pre-resolved absolute paths.
  --pretend              Dump the SQL queries that would be run.
  --seed                Indicates if the seed task should be re-run.
  --step                Force the migrations to be run so they can be rolled back individually.
  -h, --help            Display this help message
  -q, --quiet           Do not output any message
  -V, --version         Display this application version
  --ansi                Force ANSI output
  --no-ansi             Disable ANSI output
  -n, --no-interaction Do not ask any interactive question
  --env[=ENV]           The environment the command should run under
  -v|vv|vvv, --verbose Increase the verbosity of messages: 1 for normal output, 2 for more verbose output and 3 for debug

Help:
  Run the database migrations
```

Slika 6 - *Artisanov help screen*

Uz *Artisana* instalacijom *Laravela* dobivamo i *Tinker* koji nam omogućuje da u bilo kojem trenutku možemo napraviti testnu interakciju sa modelima, *eventima* itd. Kako bi pokrenuli *Tinker* okruženje potrebno je samo pokrenuti naredbu *Artisana* ***php artisan tinker***.

Kako analiziranje svih naredbi *Artisana* i ne bi imalo prevelikog smisla, napomenuti valja da je sa *Artisanom* moguće napraviti i vlastite naredbe pomoću: ***php artisan make:command <ime nove naredbe>***.

### 4.3 *Laravelove* značajke i nužne postavke

*Laravel* se prije svega odlikuje svojstvima koja omogućuju nešto brži razvoj *web* aplikacije [1]. U njemu su olakšani neki dijelovi koda koji se često ponavljaju u aplikacijama. Kao primjerice registracija i prijava korisnika u aplikaciju (autentifikacija), sesije, keširanje, mehanizam za kontrolu toka zahtjeva itd.

Analiza *Laravelovih* značajki u ovom završnom radu zamišljena je na način da se nakon instaliranog novog *Laravel* projekta redom analiziraju novo stvoreni direktoriji i novo stvorene značajke pomoću *Composer*a ili *Artisana*.

Na samom početku valja naglasiti da je za instalaciju novog *Laravel* projekta bitno da već unaprijed pripremimo svoje računalo sa potrebnim alatima:

- Lokalni server – bilo bi najbolje koristiti neki od provjerenih primjerice: *XAMPP* ili *WAMP*, za ovu *web* aplikaciju korišten je *XAMPP* sa čijom instalacijom dobivate *MySQL* server za baze podataka, *PHP* te *phpMyAdmin*.
- *Composer* – u uvodu već spomenuto komandno sučelje

Kako se u izradi ove *web* aplikacije koristi popularni *MySQL* server, opcionalno možete koristiti i neki *SQL* klijent, kako bi vam bilo nešto lakše manipulirati podacima u bazi podataka. Za izrađenu *web* aplikaciju korišten je *HeidiSQL* [10].

## 5. Analiza strukture novog *Laravelovog* projekta

Osnovna struktura novog *Laravelovog* projekta sadrži direktorije:

- *app* – u njemu se nalazi "jezgra" aplikacije. Dakle, klase (modeli), upravitelji, te *middleware* (mehanizam koji služi za kontrolu zahtjeva klijenta prema serveru).
- *bootstrap* – direktorij koji služi kao "lijepilo" u aplikaciji koje povezuje (*Bootstrapping*) ključne komponente *Laravela* te konfigurira automatsko učitavanje (*autoloading*). Osim toga ovaj direktorij sadrži *cache* direktorij koji u svojim datotekama ima spremljene podatke predmemorije.
- *config* – direktorij u kojem se nalaze konfiguracijske datoteke. Najznačajnije su *app.php* – sadrži osnovne postavke aplikacije, te *database.php* – koja sadrži postavke za bazu podataka.
- *database* – sadrži u sebi migracije, *seed-ove* (služe za popunjavanje tablica baze podataka sa testnim podacima, i tvornice (*factory*) – gdje definiramo oblik naših testnih podataka.
- *node\_modules* – ovaj direktorij nije u osnovnoj strukturi *Laravel* aplikacije. Nastaje pokretanjem naredbe *npm install* unutar komandne linije (preduvjet za pokretanje naredbe je instalirani Node.js koji nam omogućuje npm globalnu naredbu). Direktorij u sebi sadrži poddirektorije u obliku instaliranih modula tj. biblioteka, za *frontend* dio aplikacije.
- *public* – u sebi sadrži *index.php* datoteku koja ima ulogu početne točke aplikacije. Osim "početne datoteke" u ovom direktoriju se nalaze i "javne" datoteke (najčešće Javascript i CSS datoteke koje po *defaultu* pripadaju *Bootstrap* alatu za *frontend* naše aplikacije).
- *resources* – direktorij koji sadrži sve poglede naše *web* aplikacije.
- *routes* – ovaj direktorij sadrži sve potrebne route ili tokove potrebne za rad *Laravel* aplikacije. Dvije glavne datoteke unutar ovog direktorija su *web.php* i *api.php*.
- *storage* – služi kao lokalno spremište podataka, sadrži podatke o *Blade* predlošcima, predmemoriji aplikacije, sesijama i drugo.
- *tests* – sadrži testne skripte aplikacije.
- *vendor* – sadrži sve datoteke potrebne za rad *Composer*a.

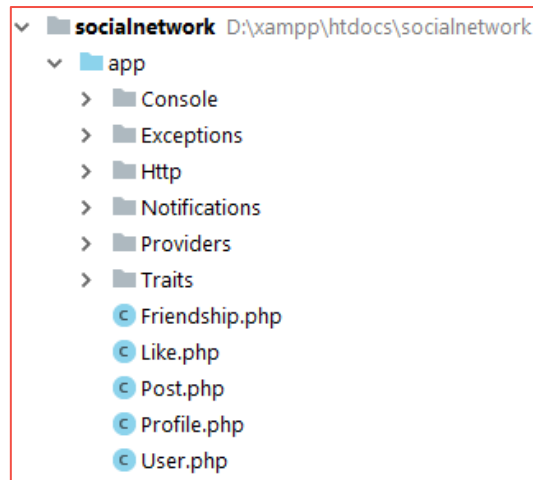
Prije nego krenemo redom na analizu svakog pojedinog direktorija, unutar *Laravela* primjetiti ćemo i *.env* datoteku koja nam služi kao konfiguracijska datoteka za našu *web* aplikaciju.

Neke od postavki *.env* koje možemo postaviti su:

- "APP\_KEY" – ključ za kriptiranje aplikacije. Ukoliko *Laravel web* aplikaciju nismo stvorili putem *Composer*a ključ za kriptiranje aplikacije neće biti stvoren. Njega možemo stvoriti pomoću *Artisana* naredbom *php artisan key:generate*.
- "DB\_USERNAME", "DB\_PASSWORD", "DB\_DATABASE" – korisničko ime, lozinka i naziv baze podataka sa kojom je povezana naša aplikacija.
- "MAIL\_DRIVER", "MAIL\_HOST" – *driver* za upravljanje *e-mailovima*, domaćin koji nam omogućava korištenje uslugu odabrane e-pošte. Osim navedenih postavki za slanje *e-maila* možemo konfigurirati i korisničko ime te lozinku *web* servisa kojeg koristimo.
- "PUSHER\_APP\_ID", "PUSHER\_APP\_KEY", "PUSHER\_APP\_SECRET" – postavke *Pusher web* servisa koji služi za rad sa notifikacijama u aplikaciji. Važno je uz *Pusher*ove postavke postaviti i "BROADCAST\_DRIVER" tj. *Pusher*ov službeni driver kao *pusher*.
- "ALGOLIA\_APP\_ID", "ALGOLIA\_SECRET" – postavke za *Algolijin* API koji služi za pretraživanje *search* unutar naše aplikacije.

## 5.1 Analiza *app* direktorija

Klikom na *app* direktorij u *Laravel* projektu. Primjetiti ćemo 5 klasa: *User.php*, *Profile.php*, *Post.php*, *Like.php* i *Friendship.php*. Klase koje vidimo predstavljaju modele odnosno *Model* dio *MVC* arhitekture *Laravel web* aplikacije.



Slika 7 - *app* direktorij

Ostatak *app* direktorija sastoji se od poddirektorija:

- *Console* – unutar njega nalazi se datoteka *Kernel.php* u kojoj se registriraju *Artisanove* naredbe
- *Exceptions* – sadrži skup iznimaka u aplikaciji. Glavna datoteka u ovom poddirektoriju je *Handler.php* koja ima ulogu upravljača iznimkama.
- *Http* – poddirektorij koji sadrži najveći logički dio aplikacije. Najvažniji dio ovog poddirektorija čine upravitelji i *middleware*.
- *Notifications* – po *defaultu* nije napravljen pri *Laravelovoj* početnoj instalaciji. Stvori se kada se u komandnoj liniji pokrene naredba: ***php artisan make:notification <ime obavijesti>***. Unutar direktorija sadrži datoteke kao vrste obavijesti koje šalje aplikacija. Prijmjerice, unutar aplikacije društvene mreže to je *NewFriendRequest.php* datoteka koja se pokreće i prikazuje kada jedan korisnik drugom pošalje zahtjev za prijateljstvo.
- *Providers* – sadrži tkz. sve davatelje usluga *Providere* aplikacije. Davatelji usluga omogućuju aplikaciji pripremu za nadolazeće zahtjeve.
- *Traits* – poddirektorij koji sadrži datoteke koje su sačinjene od skupa metoda. Gdje se te metode koriste za istu funkcionalnost. Datoteke unutar ovog direktorija namjenjene su da još više raspodjele "posao" unutar aplikacije i spriječe gomilanje koda u bilo kojoj datoteci direktorija.

Ostali poddirektoriji se naravno kao što je slučaj sa obavijestima *Notifications* stvaraju pokretanjem *Artisan* naredbi. Pa tako osim navedenih imamo: *Events*, *Listeners*, *Jobs* i druge.

## 5.2 Eloquent, ORM, modeli i migracije

Kako bi bila jasnija uloga *Modela* tj. ovih klasa, potrebno je pojasniti pojam objektno-relacijskog mapiranja. Objektno-relacijsko mapiranje [11], omogućuje nam da stvorimo "virtualnu objektnu bazu podataka" koju je lakše koristiti sa objektnim programskim jezikom. Kako bi korištenje objektno-relacijskog mapiranja bilo jednostavnije često se koriste razne biblioteke.

*Laravel* također posjeduje svoju ORM biblioteku naziva *Eloquent*. Glavna svrha korištenja *Eloquent* je izbjegavanje pisanja dugih i često nepreglednih *sql* upita. *Eloquent* također omogućava da se jedan model (klasa) poveže sa jednom tablicom u bazi podataka. Kako ni o tome ne bi morali brinuti *Laravel* slijedi neku vrstu konvencija o pisanju. Nije nužno da ih slijedimo, ali je vrlo korisno i ubrzava izradu web aplikacije. Model bi uvijek trebao biti u jednini dok bi naziv tablice trebao biti u množini. Ako za primjer uzmemo *User.php* model, tablica u bazi podataka koja mu odgovara je *create\_users\_table.php*. Naziv tablice može kao prefiks sadržavati i vremensku oznaku kada je napravljena no, o migracijama više ćemo govoriti kada budemo analizirali *database* direktorij projekta.

Kako je već rečeno da nam *Eloquent* pomaže u izbjegavanju pisanja kompliciranih *sql* upita, zato se pobrinuo da umjesto njih koristimo jednostavnije i preglednije *Eloquent* metode.

Neke od nešto korištenijih metoda su:

- "all" – u slučaju da želimo primjerice dohvatiti sve korisnike naše društvene mreže napisati ćemo "`$users = User::all();`" kao rezultat dobiti ćemo kolekciju *Eloquent* modela svih korisnika.
- "get" – ako koristimo neki uvjet pri dohvaćanju naših korisnika onda koristimo ovu metodu nadovezanu na uvjet. Pri registraciji korisnika sa 0 su u bazu podataka spremljeni korisnici ženskog spola, a sa 1 muškog spola. Sa "`$users = User::where("gender", 0)->get();`" će se u *users* varijablu spremiti objekti svih korisnika ženskog spola. "get" nam je potreban kako bismo dohvatili sve te objekte nakon izvršenog uvjeta kojeg smo postavili.
- "first" – pomoću nje dohvaćamo samo prvi zapis kao rezultat, primjerice "`$user = User::first();`"
- "save", "update", "delete" – ove metode zamjenjuju uobičajene *CRUD* funkcionalnosti: spremanje, ažuriranje i brisanje zapisa u bazi podataka. Primjerice ako želimo kod 56. korisnika izmijeniti spol napisati ćemo "`$user = User::find(redni broj korisnika);`", gdje smo pronašli korisnika i spremili ga u varijablu *user*. Kako bi ažurirali korisnika napisati ćemo još "`$user->update();`".

Postoji još nekoliko korisnih metoda od kojih je za aplikaciju društvene mreže nešto korištenija "orderBy" metoda koja nam omogućava organizaciju dobivenih rezultata. *Eloquent* metode primjetiti ćete kod analize koda upravitelja unutar čijih se metoda one najviše koriste.

Ne bismo mogli objasniti pojam modela a da ne pojasnimo pojam migracije, koja je usko povezana sa samim modelom. Migracije ukratko predstavljaju naše sheme tablica. Mogu biti stvorene kada stvaramo i sami model ili zasebno naredbom ***php artisan make:migration <ime migracije>***. Poštujući *Laravelovu* konvenciju nazivanja migracija nazvati ćemo "create\_<naziv tablice u bazi podataka u množini>\_table". Svaka migracija sastoji se od dvije metode: "up" i "down". Unutar "up" metode definiramo izgled naše tablice u bazi podataka nakon što migracija bude migrirana. "down" metoda briše staru tablicu sa istim imenom ako ona postoji. Na primjeru migracije korisnika društvene mreže možemo vidjeti kako izgleda migracija (Slika 8.)



```

<?php

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

class CreateUsersTable extends Migration
{
    public function up()
    {
        Schema::create('users', function (Blueprint $table) {
            $table->increments('id');
            $table->boolean('gender');
            $table->string('name');
            $table->string('username')->unique();
            $table->date('dob');
            $table->string('avatar');
            $table->string('cover');
            $table->string('email')->unique();
            $table->string('password');
            $table->rememberToken();
            $table->timestamps();
        });
    }

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

```

Slika 8 – create\_users\_table.php

Kao što se vidi iz Slike 8, ključnom riječi "\$table" definiramo tip podataka te naziv za svaki stupac tablice. Po *defaultu* kada stvorimo novu migraciju automatski nam se stvori i prvi stupac tablice koji ima ulogu primarnog ključa u tablici. U primjeru migracije na Slici 8 to je redak "\$table->increments('id');" što po konvenciji označava primarni ključ *user\_id* tablice *users* koji ima uključen *autoincrement* ili automatsko povećavanje vrijednosti za 1 ukoliko mu se ne dodijeli vrijednost.

id	gender	name	username	dob	avatar	cover	email
1	0	Mrs. Ana Streich	mrs-ana-streich	1978-08-01	public/defaults/avatars/female.png	public/defaults/covers/cover.jpg	eddie70@example.net
2	1	Mr. Charles Dach Sr.	mr-charles-dach-sr	1921-04-12	public/defaults/avatars/female.png	public/defaults/covers/cover.jpg	nkoss@example.net
3	0	Miss Serenity Kuvallis	miss-serenity-kuvallis	1982-03-11	public/defaults/avatars/female.png	public/defaults/covers/cover.jpg	ovandervort@example.com
4	1	Dimitri Blanda	dimitri-blanda	1985-04-26	public/defaults/avatars/female.png	public/defaults/covers/cover.jpg	sromaguera@example.org
5	0	Kyra Cruickshank Sr.	kyra-cruickshank-sr	1972-02-16	public/defaults/avatars/female.png	public/defaults/covers/cover.jpg	blake.king@example.net
6	0	Prof. Donny Dooley	prof-donny-dooley	1976-09-12	public/defaults/avatars/female.png	public/defaults/covers/cover.jpg	pfeffer.mae@example.org
7	1	Tia Schimmel	tia-schimmel	1953-01-18	public/defaults/avatars/female.png	public/defaults/covers/cover.jpg	ladarius.kihn@example.net
8	1	Edmund Jones Sr.	edmund-jones-sr	1938-11-25	public/defaults/avatars/female.png	public/defaults/covers/cover.jpg	aracely50@example.org
9	1	Ms. Bria Rowe I	ms-bria-rowe-i	1959-04-02	public/defaults/avatars/female.png	public/defaults/covers/cover.jpg	elody.walsh@example.net
10	1	Dr. Blake Stokes	dr-blake-stokes	1947-04-20	public/defaults/avatars/female.png	public/defaults/covers/cover.jpg	upton.lola@example.org
11	1	Dr. Tad Feeney MD	dr-tad-feeney-md	1923-08-03	public/defaults/avatars/female.png	public/defaults/covers/cover.jpg	gerlach.hollie@example.org
12	0	Katelin Deckow	katelin-deckow	2006-05-03	public/defaults/avatars/female.png	public/defaults/covers/cover.jpg	hoppe.megane@example.com
13	0	Ofelia Shanahan	ofelia-shanahan	1949-07-08	public/defaults/avatars/female.png	public/defaults/covers/cover.jpg	uriah.damore@example.com
14	1	Ursula Bartell	ursula-bartell	1980-12-07	public/defaults/avatars/female.png	public/defaults/covers/cover.jpg	luis.mueller@example.com

Slika 9 - Izgled tablice users u bazi podataka

```

<?php

namespace App;

use App\Traits\Friendable;
use Illuminate\Notifications\Notifiable;
use Laravel\Scout\Searchable;
use Illuminate\Foundation\Auth\User as Authenticatable;
use Illuminate\Support\Facades\Storage;

class User extends Authenticatable
{
    use Notifiable, Friendable, Searchable;

    protected $fillable = ['gender', 'name', 'username', 'dob',
        'cover', 'avatar', 'email', 'password'];

    protected $dates = ['dob'];

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

    public function getAvatarAttribute($avatar)
    {
        return asset(Storage::url($avatar));
    }

    public function profile()
    {
        return $this->hasOne(Profile::class);
    }

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

```

Slika 10 - User.php

Osim primarnog ključa stvori nam se i redak "\$table->timestamps();" koji označava dva stupca tablice sa imenima *created\_at* i *updated\_at*. Kako im sama imena govore *created\_at* stupac posjeduje vremensku oznaku kada je stvoren zapis u tablici, dok *updated\_at* sadržava zapis kada je napravljena posljednja promjena na određenom retku tablice.

Migracije migriraju pomoću naredbe **php artisan migrate**. Nakon pokrenute naredbe u komandnoj liniji, u našoj bazi podataka na lokalnom serveru stvoriti će se tablice gdje svaka tablica odgovara točno jednoj migriranoj migraciji.

Popis svih migracija aplikacije nalazi se u *database* direktoriju unutar poddirektorija *migrations*.

Novi model stvoriti ćemo naredbom ***php artisan make:model -m <ime modela>***. Novi model nam se po *defaultu* stvara unutar *app* direktorija. Dodavanjem "-m" u naredbu kreira se i njegova vlastita migracija.

Za primjer analize modela uzeti ćemo kod unutar *User.php* klase. Inače, to je klasa koja u sebi sadrži model podataka za svakog pojedinog korisnika društvene mreže.

Za *User.php* model klasu definirali smo "šfillable" svojstvo. To svojstvo označava niz stupaca u tablici baze podataka za koje je dozvoljeno dodijeljivanje vrijednosti od strane korisnika/klijenta koji koristi aplikaciju. "šfillable" svojstvo koristimo i kako bi se zaštitili od nepoželjnih unosa u bazu podataka. Kada smo u niz svojstava "šdates" dodali ime stupca u tablici baze podataka, tada smo si omogućili korištenje *Laravelove* biblioteke *Carbon* koja je zadužena za rad sa datumima. *Carbonove* metode možemo koristiti i unutar pogleda. Osim ova dva svojstva u izradi ove aplikacije korišteno je i svojstvo "šhidden" koje označava stupce u tablici baze podataka koji su skrivenog (hidden) tipa.

Model je predodređen da se u njemu pišu i *get* i *set* metode. Koje se u *Laravelovom* žargonu nazivaju *Accessors* i *Mutators*. *Laravel* posjeduje konvenciju pisanja i za *Eloquent* attribute. Prema konvenciji ako koristimo *Accessor* pisati ćemo ga u obliku: ***get<ime atributa u CamelCase obliku>Attribute***, ako koristimo *Mutator* sintaksa je ista osim što ključnu riječ *get* zamjenimo sa *set*.

Veze između modela također se definiraju unutar samog koda svakog od modela. Tako u *User.php* modelu imamo vezu *User.php* modela prema modelima: *Profile.php*, *Post.php* i *Like.php*. Gdje možemo isčitati da: svakom korisniku društvene mreže (*User.php*) pripada točno jedan korisnički profil (*Profile.php*), svakom korisniku društvene mreže pripada mnogo napisanih objava (*Post.php*) te da svakom korisniku društvene mreže pripada mnogo *likeova*, odnosno postupaka gdje je on izjavio da mu se sviđa određena objava koja je objavljena od strane drugog korisnika društvene mreže.

Prije definiranih svojstava ključnom riječi *use* unutar klase navedeno je kako će se koristiti datoteke koje sadrže skup metoda. Datoteke koje sadrže skup metoda nalaze se u poddirektoriju *Traits* unutar *app* direktorija. Na primjeru *User.php* modela to je *Friendable.php* datoteka koja sadrži sve metode potrebne za određivanje "prijateljstva" na aplikaciji društvene mreže.

### 5.3 Http poddirektorij

Predstavlja jedan od glavnih "stupova" aplikacije. Njegov glavni dio čine upravitelji *Controllers* i *Middleware*. Upravitelji su zaslužni za logički dio aplikacije. Novi upravitelj stvaramo naredbom *php artisan make:controller <ime upravitelja>*. Novi upravitelj se nakon naredbe stvara u poddirektoriju *Controllers*.

```
public function wall($user_id)
{
    $user = User::where('id', $user_id)->first();

    $wallPosts = array();

    foreach ($user->posts->sortByDesc('created_at') as $wallPost):
        array_push($wallPosts, $wallPost);
    endforeach;

    return $wallPosts;
}
```

Slika 11 - Metoda iz *ProfileController.php* upravitelja

Njihova je glavna zadaća kako im samo ime kaže da upravljaju sa korisničkim zahtjevima prema *web* aplikaciji. Za svaki zahtjev korisnika pobrine se određena metoda upravitelja. Konvencija nazivanja upravitelja je: **<naziv modela kojim upravlja>Controller**. Umjesto modela upravitelj može upravljati i nekim drugim dijelom arhitekture aplikacije, ali u praksi najčešće upravlja modelom.

Na primjeru slike (Slika 11), možemo vidjeti jednu od metoda *ProfileController.php* upravitelja. Prikazana *wall* metoda služi za prikaz tkz. "zida" korisnika društvene mreže, tj. prikuplja objave na njegovom profilu koje je taj korisnik objavio. Kao argument ovoj metodi šalje se redni broj korisnika u bazi podataka što nam omogućuje da lakše pronađemo tog korisnika u bazi podataka i spremimo ga u *\$user* varijablu. Nakon toga definiramo polje *\$wallPosts* u koje ćemo spremiti korisnikove objave.

S obzirom na to da nam je poznato kako je *User.php* model povezan sa modelom objave na društvenoj mreži *Post.php* modelom, pomoću *\$user* varijable možemo pristupiti točno onim objavama koje je objavio naš pronađeni korisnik kada se nadovežemo na *posts* metodu u *User.php* modelu. Na pronađene objave je dodana *sortByDesc()* pomoćna *Laravel funkcija* koja nam omogućuje sortiranje tih objava prema *created\_at* atributu odnosno atributu koji označava vrijeme kada je objavljena objava. Nakon toga potrebno je samo u *foreach* petlji svaku objavu proslijediti u *\$wallPosts* polje.

Kao vrijednost koju metoda upravitelja vraća može se koristiti i opcija gdje metoda vraća pogled uz prosljeđenu varijablu. Te se primjerice ta varijabla koja često sadrži skupinu od nekoliko rezultata (polje) može iterirati unutar samog pogleda. Način za takvu vrstu iteracije će biti prikazan u dijelu kada se bude govorilo detaljnije o pogledima i *Blade* vrsti predložka.

Kod stvaranja novog upravitelja postoji opcija stvaranja resursnog upravitelja. Resursni upravitelj stvaramo tako da naredbi stvaranja upravitelja dodamo "—resource". Pri čemu su nam osim klasičnog predloška upravitelja dodane i resursne metode:

- *index* metoda – u njoj se najčešće dohvaćaju sve jedinice nekog modela/zapisa.
- *create* metoda – najčešće vraća pogled u obliku forme za stvaranje novog zapisa.
- *store* metoda – omogućuje spremanje zapisa u bazu podataka. Kao parametar prima sve zapise unutar potvrđene forme. Poslanim zapisima putem forme pristupamo instancom *Request* paketa u *Laravelu*, koji je po *defaultu* uključen u svakom resursnom upravitelju.
- *show* metoda – služi za prikaz određenog zapisa. Kao parametar prima redni broj ili *id* zapisa u bazi podataka.
- *edit* metoda – najčešće prikazuje formu za izmjenu već postojećeg zapisa. . Kao parametar prima redni broj ili *id* zapisa u bazi podataka.
- *update* metoda – služi za spremanje promjena o zapisu u bazu podataka. . Kao parametar prima redni broj ili *id* zapisa u bazi podataka i *Request* instancu.
- *destroy* metoda – briše zapis iz baze podataka. Kao parametar prima redni broj ili *id* zapisa u bazi podataka.

#### 5.4 Mehanizam za upravljanje tokovima (routing)

Kao što su modeli i ORM usko povezani sa migracijama, tako su upravitelji usko povezani sa *routingom* ili mehanizmom koji određuje tokove zahtjeva u aplikaciji. Sve tokove naše aplikacije možemo pronaći u datoteci *web.php* u direktoriju *routes* naše aplikacije.

Na donjem primjeru koda iz aplikacije (Slika 12), možete vidjeti kako se definira tok u aplikaciji. Na početku koristimo ključnu riječ *Route* nakon toga se piše vrsta zahtjeva koji koristimo. U specifikaciji svakog toka (u zagradi), pišemo naziv toka (npr. *profile/{username}*). Vitičaste zagrade koristimo kada želimo naglasiti da se u tom dijelu definiranog toka nalazi parametar (u ovom slučaju *username*). Nakon definiranog imena moramo odabrati koji upravitelj će koristiti novi tok kao i metodu odvojenu za znakom "@" (npr. *ProfileController@update*). Te na kraju opcionalno odabiremo *alias* (npr. *profile.edit*) ili neko kraće ime za tok koje će nam biti jednostavnije koristiti nego da pišemo cijeli naziv toka svaki puta kada na njega šaljemo zahtjev.

```
Route::get('/profile/{username}', ['uses'=>'ProfileController@index',  
'as'=>'profile']);  
  
Route::get('/profile/{username}/edit', ['uses'=>'ProfileController@edit',  
'as'=>'profile.edit']);  
  
Route::post('/profile/{username}', ['uses'=>'ProfileController@update',  
'as'=>'profile.update']);
```

Slika 12 - Dio koda web.php datoteke

## 5.5 Resursni upravitelj i tokovi

Kada koristimo resursni upravitelj u mogućnosti smo sve tokove vezane uz resurnog upravitelja dodati sa ključnom riječi *resource*. U aplikaciji društvene mreže kao resursni upravitelj korišten je *PostController.php*. Koji upravlja objavama korisnika na društvenoj mreži. Prikazuje ih na naslovnoj stranici društvene mreže, omogućuje stvaranje novih, uređivanje brisanje i pregled. Svi tokovi ovo upravitelja dodani su isključivo jednom linijom koda ("Route::resource('posts', PostController);") i nije potrebno definirati svaki tok zasebno.

Tokovi koji su dodani su:

- `http://<ime domene>/posts` – za pristup se koristi GET te odgovara *index* metodi upravitelja.
- `http://<ime domene>/posts/create` – koristi GET, njegova funkcionalnost je unutar *create* metode upravitelja.
- `http://<ime domene>/posts/store` – koristi POST, povezan sa *store* metodom upravitelja.
- `http://<ime domene>/posts/{id}` – koristi GET, povezan je sa *show* metodom upravitelja. Kako bi mogao prikazati sve podatke o nekom zapisu u njegovom zahtjevu najčešće se šalje redni broj ili *id* parametar željenog zapisa.
- `http://<ime domene>/posts/{id}/edit` – koristi GET, povezan sa *edit* metodom upravitelja.
- `http://<ime domene>/posts/update/{id}` – koristi PUT, povezan je sa *update* metodom upravitelja. Kod zahtjeva se šalje parametar rednog broja ili *id* parametar zapisa koji se ažurira.
- `http://<ime domene>/posts/delete/{id}` – koristi DELETE, povezan je sa *delete* metodom upravitelja. Kod zahtjeva se šalje parametar rednog broja ili *id* parametar zapisa koji se želi izbrisati.

## 5.6 Middleware

Kako bi naša *web* aplikacija u *Laravelu* djelovala uz određeni stupanj sigurnosti, za to se pobrinuo *middleware* ili srednji sloj *web* aplikacije zadužen za kontrolu korisničkih zahtjeva. *Laravel* aplikacija po *defaultu* ima već napravljen *middleware* naziva "auth", koji omogućuje pristup određenim tokovima samo ako je korisnik već prijavljen sa korisničkim imenom i lozinkom za korištenje aplikacije. Novi *middleware* može napraviti pomoću naredbe: **`php artisan make:middleware <ime middlewarea>`**. Nakon što smo pokrenuli naredbu, novi *middleware* stvoren je unutar *Middleware* poddirektorija.

```
public function __construct()
{
    $this->middleware('auth');
}
```

Slika 13 - Korištenje *middlewarea* - metoda upravitelja

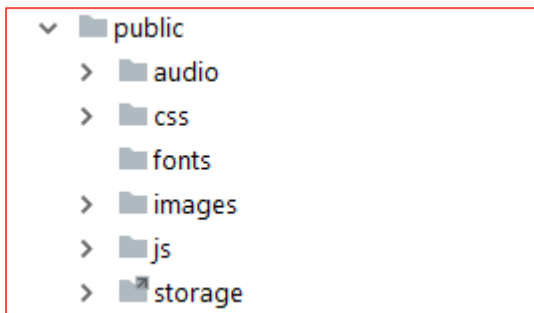
```
Route::group(['middleware' => 'auth'], function () {  
  
    Route::get('/profile/{username}', ['uses' =>  
        |'ProfileController@index', 'as' => 'profile']);  
});
```

Slika 14 - Korištenje *middlewarea* - metoda *web.php* datoteke

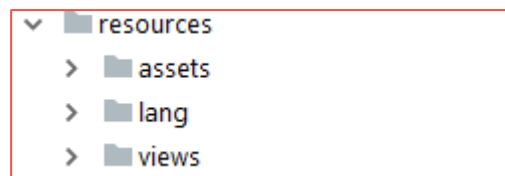
*Middleware* možemo koristiti u metodi upravitelja ili unutar metode *web.php* datoteke. Globalna metoda "*middleware()*;" kao parametar prima klasu ili ključnu riječ *middlewarea* koji želimo koristiti. Na donjim primjerima možete vidjeti načine korištenja *middlewarea* unutar metode *web.php* datoteke te unutar metode upravitelja.

## 5.7 Analiza *node\_modules* direktorija

Kako je u samom uvodu rečeno naglasak će najviše biti na *backend* ili logičkom dijelu *web* aplikacije. *Node\_modules* direktorij stvara se pokretanjem naredbe *npm install*. Koja stvara skupinu modula za korištenje. Tada ukoliko imamo instalirani *Node.js* na našem operacijskom sustavu možemo pristupiti tim modulima / paketima i instalirati ih unutar *Laravel* projekta. Paketi se instaliraju također pomoću naredbe *npm install* sa dodatkom imena paketa. "NPM" naredba u ovoj aplikaciji najviše je korištena za *frontend* funkcionalnosti.



Slika 15 - Struktura *public* direktorija

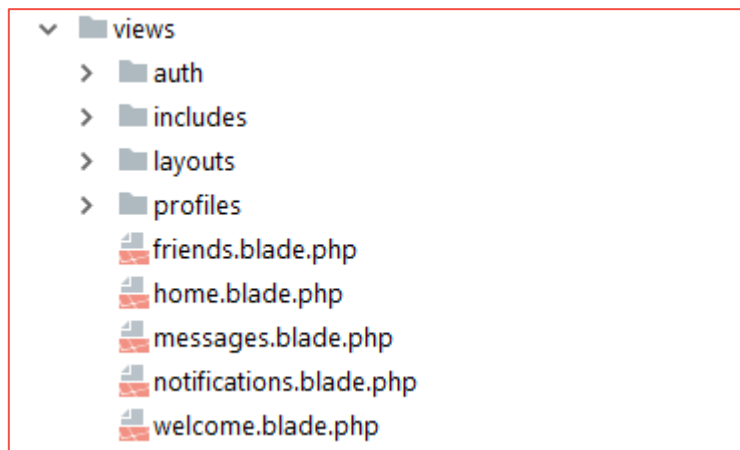


Slika 16 - Struktura *resources* direktorija

## 5.8 Analiza *public* direktorija

Unutar *Public* direktorija dodane su tkz. javne datoteke. Najčešće su raspoređene u nekoliko direktorija kao što je prikazanom na slici (Slika 15).

Poddirektorij naziva *css* sadrži datoteke sa *.css* ekstenzijom. *CSS* datoteke unutar poddirektorija služe za uređivanje dizajna *web* aplikacije. Po *defaultu* se u *css* poddirektoriju nalazi *app.css* datoteka u kojoj je najnovnija verzija *Boostrapa* (jedna od najpoznatijih *css* i *javascript* biblioteka za lakši raspored elemenata na stranici i njihov dizajn). Poželjno je prema *Laravelovoj* konvenciji da unutar *public* direktorija stvaramo i vlastite nove *javascript* i *css* datoteke. *Audio* direktorij sadrži audio zapise, *images* direktorij slike, a *fonts* specijalizirane verzije fontova. *Storage* se razlikuje od ostalih direktorija jer prije svega njegova ikona ima oblik *shortcuta* ili poveznice. *Storage* poddirektorij samo je poveznica na lokalno spremište također naziva *Storage* u korijenskom direktoriju aplikacije.



Slika 17 - Struktura *views* direktorija

Direktorij *resources* također ima ulogu jednog od korištenijih direktorija u izradi aplikacije. U njemu se nalaze *views*(pogledi) koji predstavljaju jedan od stupova *MVC* arhitekture te *assets* unutar kojeg su spremljene *frontend* komponente aplikacije u *Vue.js* radnom okruženju. *Lang* poddirektorij po *defaultu* ima podršku za engleski jezik za našu aplikaciju gdje kasnije možemo s obzirom na postavke lokacije naše aplikacije postaviti alternativne poruke na drugim jezicima.

## 5.9 Pogledi

*Views* poddirektorij sadrži sve poglede naše aplikacije. Razvrstani su u nekoliko poddirektorija ovisno o dijelu aplikacije kojem pripadaju. Nerazvrstani pogledi pripadaju zasebnim funkcionalnostima zato se i ne nalaze niti u jednom poddirektoriju. Ono što možemo primijetiti je dodatak *blade* na uobičajenu *PHP-ovu* ekstenziju. Upravo taj dodatak predstavlja još jednu veliku prednost *Laravel* radnog okruženja. Riječ je naravno o *Blade Template Engine-u* [5] ili *Blade* mehanizmu za predloške.

*Blade* predlošci nam omogućuju: [5]

- Izradu glavnog predloška ( u slučaju aplikacije društvene mreže to je *master.blade.php* datoteka)
- Proširivanje glavnog predloška prilikom stvaranja novih pogleda
- Prikaz podataka generiranih od strane upravitelja
- Logičke petlje sa nešto urednijom i čitljivijom sintaksom od klasične *PHP* sintakse

Prije nego krenemo s analizom koda jedne od *blade* datoteka potrebno je napomenuti kako se unutar *layouts* direktorija nalaze glavni predlošci, a unutar *includes* direktorija pomoćni dijelovi stranice koji se uključuju po potrebi u glavne predloške ( *navigacija*, *header*, *footer* itd.).

Kako je već rečeno glavni predlošci služe kako ne bismo ponavljali određene dijelove strukture stranice prilikom pravljenja svakog novog pogleda. Glavne predloške koristi gotovo svaki pogled koji ga proširuje ovisno o tome što želi prikazati. *@yield* oznake služe kako bismo naznačili linije koda koje će se u novom pogledu proširivati. Osim oznake *@yield* potrebno je i naznačiti naziv tog odjeljka (na Slika



18 vidimo `@yield('title')` liniju koja označava dio pogleda gdje je "rezervirano" mjesto za proširenje naslova pogleda(stranice). `@yield('content')` biti će dio pogleda gdje će se nalaziti glavni sadržaj stranice. `@include` oznaka služi kako bi se dodao kod iz nekog drugog manjeg predloška. U slučaju glavnog predloška ove aplikacije to je navigacija. U glavni predložak uključene vrlo često i `css` te `javascript` skripte čije klase i funkcije želimo da dijele svi novostvoreni pogledi.

```
<!DOCTYPE html>
<html lang="{{ app()->getLocale() }}">
<head>
  <meta charset="utf-8">
  <meta http-equiv="X-UA-Compatible" content="IE=edge">
  <meta name="viewport" content="width=device-width, initial-scale=1">

  <!-- CSRF Token -->
  <meta name="csrf-token" content="{{ csrf_token() }}">

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

  <!-- Styles -->
  <link href="{{ asset('css/app.css') }}" rel="stylesheet">
  <link href="{{ asset('css/main.css') }}" rel="stylesheet">
</head>

<body>
<div id="app">
  <init></init>|
  @include('includes.navigation')
  @yield('content')
  @if(Auth::check())
    <notification :id="{{ Auth::user()->id }}"></notification>
  @endif
  <audio id="notification_sound">
    <source src="{{ asset('audio/notification.mp3') }}">
  </audio>
</div>

<!-- Scripts -->
<script src="{{ asset('js/app.js') }}"></script>
<script src="{{ asset('js/main.js') }}"></script>
</body>
</html>
```

Slika 18 - Kod `master.blade.php` datoteke

Naravno, nakon što smo napravili glavni predložak potrebno je napraviti poglede koji će proširivati sadržaj glavnog predloška. Novi pogled stvaramo kao i svaku novu PHP datoteku, osim što je potrebno dodati dodatak *blade* (<ime datoteke>.blade.php).

Na Sliku 19 možemo vidjeti primjer koda jednog od pogleda. Ovaj pogled predstavlja naslovnu stranicu društvene mreže gdje su prikazane objave korisnika koji su prihvatili vaš zahtjev za prijateljstvo (*friend request*). Kako pogled ima otprilike stotinjak linija koda nije prikazan u cijelosti već samo jedan njegov dio koji će biti dovoljan kako bi razumijeli proširivanje pogleda i stvaranje novih. Sa oznakom *@extends* definira se poveznica sa glavnim predloškom koji će naš pogled koristiti. Linije koda koje su kod glavnog predloška bile označene sa *@yield* oznakom, kod proširivanja zamjenjene su oznakama *@section('ime odjeljka')* i *@endsection* koja označava kraj proširenog sadržaja.

```

@extends('layouts.master')

@section('title')NewsFeed @endsection

@section('content')
    <div class="container-fluid">
        <div class="row">

            <!-- Modal for new post-->
            <div class="modal fade" id="newPostModal" role="dialog">
                <div class="modal-dialog">
                    <!-- Modal content-->
                    <div class="modal-content">
                        <div class="modal-header">
                            <h4 class="modal-title">Post something
                            great !</h4>
                            <button type="button" class="close" data-
                            dismiss="modal">&times;</button>
                        </div>
                        <div class="modal-body">
                            <create-post></create-post>
                        </div>
                    </div>
                </div>
            </div>
        </div>
    </div>

```

Slika 19 - Kod home.blade.php datoteke

Osim oznaka koje nam uvelike olakšavaju i ubrzavaju stvaranje novih stranica naše aplikacije, u pogledu možemo direktno prikazivati podatke generirane od strane upravitelja. Upravitelj koji vraća pogled sa varijablom ( npr. *return->view('friends')->compact('friends');*). Omogućio je da se u pogledu

```

@extends('layouts.master')

@section('title')Friends @endsection

@section('content')
    <div class="container-fluid spacing">
        <div class="row">
            <div class="card col-md-5 offset-3" style="padding: 20px;">
                <div class="card-header text-center">List of all friends</div>
                @foreach($friends as $friend)
                    <a class="img-thumbnail spacing friend-row"
                        href="{{ route('profile', ['username' => $friend->username ]) }}">
                         {{ $friend->name }}
                        @foreach($friendships as $friendship)
                            <span class="float-right">
                                @if($friendship->requested_user == $friend->id || $friendship->requester == $friend->id )
                                    Friends since: {{ $friendship->created_at->diffForHumans() }}
                                @endif
                            </span>
                        @endforeach
                    </a>
                @endforeach
            </div>
        </div>
    </div>
@endsection

```

Slika 20 - Kod *friends.blade.php* datoteke

pod nazivom *friends.blade.php* koristi *\$friends* varijabla. Moguće je da i unutar upravitelja ispišemo sve dobivene rezultate odnosno *\$friends*, ali to možemo napraviti i u pogledu nešto jednostavnijom sintaksom. Svaka logička petlja kao i sekcija (*@section*) ima svoj početak, primjerice *@foreach(\$friends as \$friend)* i kraj *@endforeach*. Što se tiče upraviteljskog dijela "posla" varijablu osim sa funkcijom *compact* možemo proslijediti i sa *with* funkcijom koja kao prvi parametar uzima željeni alias varijable koju prosljeđujemo, dok je drugi parametar sama varijabla.

## 5.10 Korisne kratice (shortcuts)

*Laravel* u svom radnom okruženju omogućuje "kratice" kako ne bismo morali pisati cijelu putanju do željenih datoteka. Tako je u ovoj aplikaciji najčešće korištena *asset* kratica koja označava putanju do *public* direktorija naše aplikacije (npr. *asset('css/app.css')*), te *Auth* globalna klasa koja označava trenutno prijavljenog korisnika ( npr. *Auth::user()*). Pomoću *Auth* klase možemo pristupiti željenim atributima prijavljenog korisnika, čiji model podataka odgovara *User.php* modelu.

## 5.11 Ostali direktoriji i korisne datoteke

Do sada smo analizirali sve najbitnije direktorije *Laravelovog* radnog okruženja. Jedino što nam preostaje je napomenuti još neke datoteke koje svojim djelovanjem mogu upotpuniti našu aplikaciju te je samim time učiniti profesionalnijom.

Što se tiče direktorija *routes* najkorišteniju *web.php* datoteku smo uspostavili kao mjesto koje nam služi za registriranje tokova u našoj aplikaciji. Slična njoj datoteka *api.php*, koja se nalazi u istom direktoriju, služi za registriranje tokova prilikom izrade aplikativnog programskog sučelja ili *API-ja*. Gdje će naravno, prilikom izrade aplikativnog programskog sučelja upravo u toj datoteci biti registrirana većina tokova.

U direktoriju *database* osim popisa migracija naše aplikacije, možemo primjetiti još dva poddirektorija. Točnije, *factories* poddirektorij i *seeds* poddirektorij. Njih koristimo kada želimo vidjeti/testirati kako naša aplikacija radi sa više zapisa u bazi podataka. U slučaju aplikacije društvene mreže korištene su u

smislu generiranja testnih korisnika i objava. *Factory* datoteke služe za generiranje testnih objekata u našoj aplikaciji, dok *seed* datoteke imaju ulogu mehanizama za ispunjavanje tablice u bazi podataka.

Na gornjem primjeru možemo vidjeti način definiranja jedne od *factory* datoteka. Za popunjavanje testnih atributa najčešće se koriste *PHP-ove* funkcije. Osim *PHP-ovih* funkcija možemo koristiti već u *Laravel* ugrađenu biblioteku *Faker* pa pomoću varijable *\$faker* generirati testne podatke (*fake data*).

U *seed* datoteka definiramo primjerice sa koliko testnih zapisa želimo popuniti našu bazu podataka, ili ako generiramo više testnih objekata, odnos između njih. Kako bi pokrenuli *seeder* i popunili bazu podataka testnim podacima trebamo pokrenuti naredbu ***php artisan db:seed***. Njenim pokretanjem djelujemo na *run* funkciju glavne *seed* datoteke (*DatabaseSeeder.php*).

```
$factory->define(App\User::class, function (Faker $faker) {
    static $password;
    $name = $faker->name;
    $gender = [0, 1];
    return [
        'name' => $name,
        'email' => $faker->unique()->safeEmail,
        'username' => str_slug($name),
        'gender' => array_rand($gender),
        'dob' => $faker->dateTimeThisCentury,
        'avatar' => 'public/defaults/avatars/female.png',
        'cover' => 'public/defaults/covers/cover.jpg',
        'password' => $password ?: $password = bcrypt('secret'),|
        'remember_token' => str_random(10),
    ];
});
```

Slika 21 - Kod *UserFactory.php* datoteke

```
public function run()
{
    $this->call(UsersTableSeeder::class);
    $this->call(PostsTableSeeder::class);
}
```

Slika 22 - *Run* funkcija *DatabaseSeeder.php* datoteke

## 6. Alternative za korištenje sličnih radnih okruženja

Ukoliko programer već ima iskustva u izradi *web* aplikacija u nekom drugom programskom jeziku, postoje alternative koje će pružiti vrlo sličnu podršku unutar svojih radnih okruženja. U nastavku će biti nabrojane neke od najkorištenijih i ukratko njihove prednosti i mane:

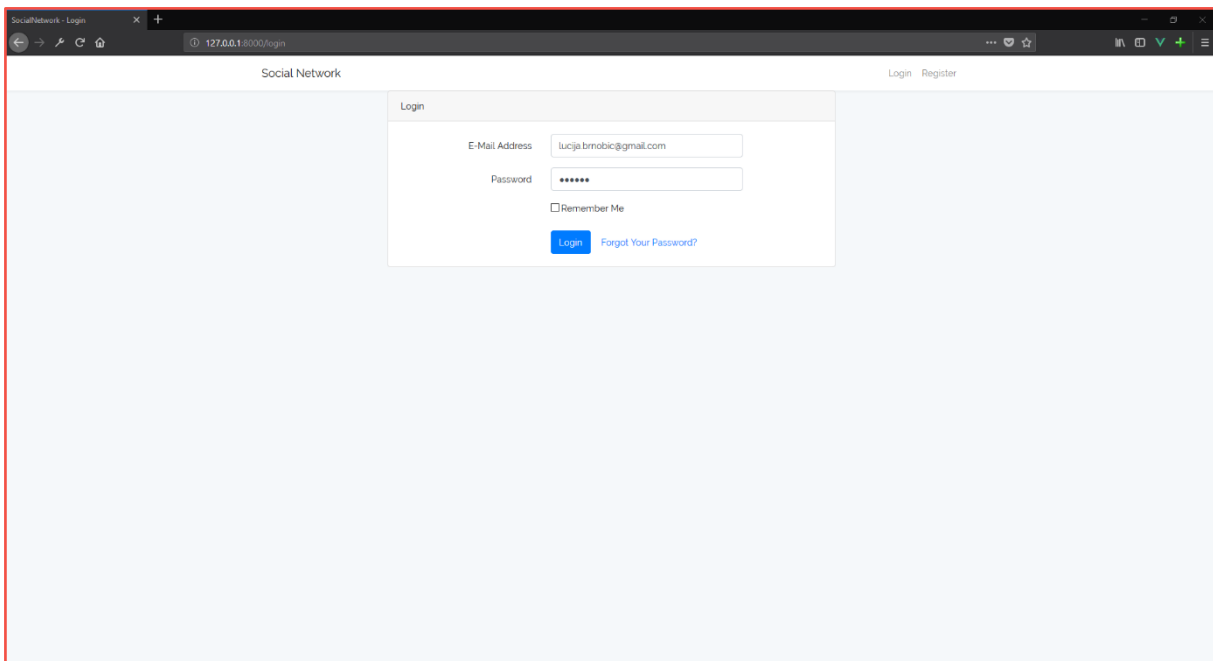
- *Ruby on Rails* – radno okruženje koje koristi *Ruby* programski jezik. Idealan za korištenje na manjim projektima ali i nešto većim. Mnogo dostupnih dodataka (*pluginova*) i vrlo dobra podrška od strane *Ruby* zajednice programera. Neke od mana su te što *Ruby* programski jezik i nije toliko popularan za neku drugu vrstu aplikacija osim *web* aplikacija.
- *Flask* ili *Django* – radna okruženja koja koriste *Python* programski jezik. Vrlo jednostavni za korištenje. Minimalistični i vrlo fleksibilni. Ukoliko radite na manjim projektima ili razvijate *api* onda je najbolje koristiti *Flask* (mikro radno okruženje). Dok je *Django* u pravom smislu radno okruženje za nešto veće i zahtjevnije *web* aplikacije. Pri izradi zahtjevnijih *web* aplikacija stranice se nešto sporije otvaraju pa je definitivno potrebno uzeti u obzir i optimizaciju aplikacije.
- *Express.js* – radno okruženje koje koristi *Javascript* programski jezik. Odlikuje se velikom podrškom za dodatne pakete i dodatke. Vrlo lako i jednostavno postavljanje okruženja. Daleko najpopularnije radno okruženje za *Node.js*.

## 7. Analiza komponenti aplikacije društvene mreže

U ovom poglavlju biti će objašnjen postupak izrade najbitnijih komponenti aplikacije društvene mreže. Osim *backend* dijela programiranja biti će spomenut i *fronted* dio *web* aplikacije (*Vue.js* [13]), koji je unutar ove aplikacije imao ulogu prikaza podataka generiranih od strane *Laravela*.

### 7.1 Autentifikacija korisnika

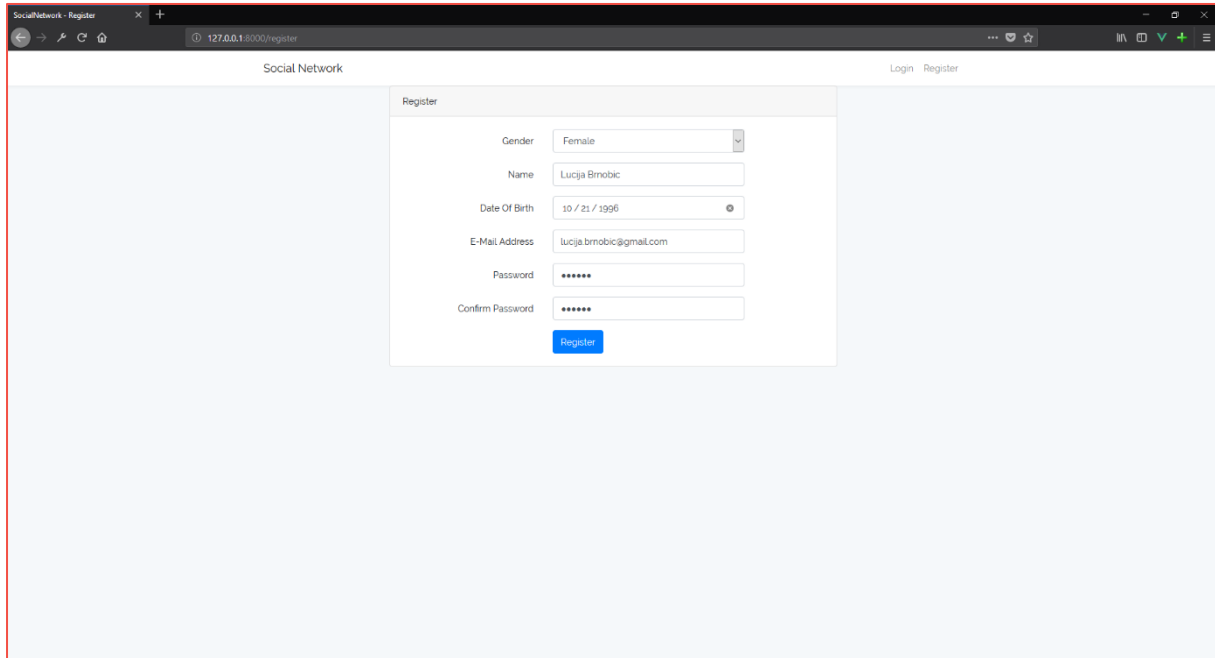
Implementacija autentifikacije izvršava se pokretanjem naredbe ***php artisan make:auth***. Nakon pokretanja naredbe generiran je "HomeController" upravitelj koji se koristi za upravljanje zahtjevima korisnika nakon njegove prijave u aplikaciju (*Login*) i pogledi za prijavu korisnika i njegovu registraciju. Osim *HomeControllera* unutar projekta primjetiti ćemo i *LoginController* te *RegisterController* upravitelje unutar kojih se može definirati gdje će korisnik biti preusmjeren nakon prijave ili registracije, a to radimo izmjenom varijable "\$redirectTo". *ForgotPasswordController* upravitelj kako mu samo ime kaže zadužen je za po *defaultu* slanje *emaila* linkom za resetiranje lozinke, dok *ResetPasswordController* upravitelj sadrži logički dio zadužen za resetiranje lozinke.



Slika 23 - Prozor za prijavu korisnika

Nakon implementacije autentifikacije moguće je koristiti i nove autentifikacijske metode. Primjerice, ukoliko želimo dohvatiti trenutno prijavljenog korisnika to možemo napraviti pomoću metode "*Auth::User()*" gdje nam metoda vraća trenutno prijavljenog korisnika. Metoda "*Auth::id()*" vraća samo primarni ključ prijavljenog korisnika, dok metodom "*Auth::check()*" možemo provjeriti je li korisnik prijavljen u aplikaciji.

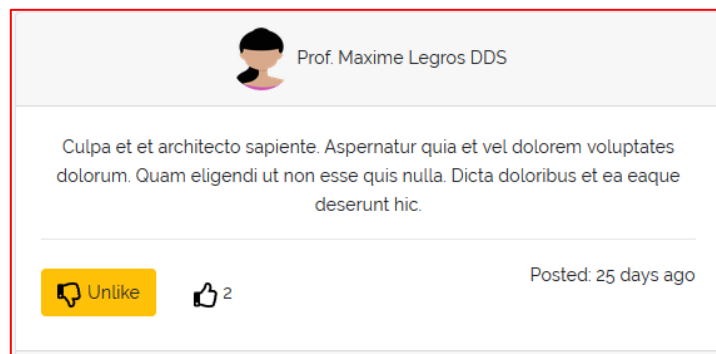
Također, novostvoreni *middleware* "*auth*" omogućuje nam da npr. zabranimo pristup određenim korisnicima ukoliko nisu autentificirani. Nakon migracije tablica naredbom ***php artisan migrate*** generira nam se i tablica "*password\_resets*" unutar naše baze podataka zajedno sa tablicom "*users*". Kada smo uspješno izvršili migraciju imamo u potpunosti implementiranu autentifikaciju.



Slika 24 - Prozor za registraciju korisnika

## 7.2 Objave (statusi)

Ovaj odlomak se odnosi na objave na društvenoj mreži koje kreiraju korisnici. Te su objave vidljive na korisničkom profilu određenog korisnika ili na naslovnoj stranici. Svaka objava korisnika sadrži korisnikov *id* odnosno *user\_id* te sadržaj napisane objave. Tako da je prvo napravljen model pomoću naredbe ***php artisan make:model Post -m***. Nakon izvršene naredbe izgeneriran nam je novi model i njegova migracija *create\_posts\_table.php*.



Slika 25 - Izgled objave

Osim modela i migracije potreban nam je i upravitelj kojeg smo stvorili pomoću ***php artisan make:controller PostController --resource*** naredbe. Kako smo specificirali da želimo stvoriti resursni upravitelj, unutar upravitelja imamo sve potrebne metode za CRUD (*Create-Read-Update-Delete*) funkcionalnosti. Na slici (Slika 26) možemo vidjeti kod *PostController.php* datoteke tj. upravitelja za objave. Uobičajena je praksa da *index* metoda dohvaća sve zapise iz baze podataka, no u ovom slučaju dohvaćanje objava je odvojeno u posebne upravitelje. Ulogu *index* metode u *FeedController.php* upravitelju (prikaz koda na slici Slika 26) ima metoda *feed*. Zadužena je za prikaz objava na naslovnoj

stranici društvene mreže. Objave su filtrirane na način da se prikazuju objave samo onih korisnika sa kojima je prijavljen korisnik "prijatelj" na društvenoj mreži. U "filtriranju" objava pomogao nam je skup funkcija u *Friendable.php* datoteci. U kodu je sa ključnom riječi *use* naglašeno da će se koristiti ta skupina funkcija.

```
<?php
namespace App\Http\Controllers;

use App\Post;
use App\Traits\Friendable;
use Illuminate\Http\Request;
use Illuminate\Support\Facades\Auth;

class FeedController extends Controller
{
    use Friendable;

    public function feed()
    {
        $friends = Auth::user()->friends();

        $feed = array();

        foreach ($friends as $friend):
            foreach ($friend->posts->sortByDesc('created_at') as $post):
                array_push($feed, $post);
            endforeach;
        endforeach;

        return $feed;
    }
}
```

Slika 26 - Kod *FeedController.php* upravitelja

U varijablu *\$friends* spremili smo sve "prijatelje" od trenutno prijavljenog korisnika pomoću funkcije *friends* (Slika 26). Zatim je pod varijablom *\$feed* deklarirano novo polje u koje ćemo spremiti sve objave. Kako postoji veza između modela korisnika društvene mreže (*User.php*) i modela objave (*Post.php*), u petlji za svakog "prijatelja" prijavljenog korisnika dohvaćamo sve objave i sortiramo ih u silaznom poretku. Na kraju vraćamo skupinu svih objava koje će biti na naslovnoj stranici u obliku varijable *\$feed*.



Metode koje koristimo u *PostController.php* upravitelju su *store*, *edit*, *update*, i *destroy* metoda.

```

public function store(Request $request)
{
    return Post::create([
        'content' => $request->get('content'),
        'user_id' => Auth::id()
    ]);
}

public function edit($id)
{
    $post = Post::findOrFail($id);
    return response()->json($post);
}

public function update(Request $request, $id)
{
    $post = Post::findOrFail($id);
    $post->content = $request->get('content');
    $post->save();
    return response()->json($post);
}

public function destroy($id)
{
    $post = Post::findOrFail($id);
    $post->delete();
    return $id;
}
}

```

Slika 27 - Kod *PostController.php* upravitelja

*Store* metoda zadužena je za spremanje novog modela objave u bazu podataka. *Post* model pomoću metode *create* stvara novi objekt objave gdje se njezin sadržaj (*content*) dohvaća pomoću *\$request* varijable (instancija koja nam omogućuje pristup podacima iz nadolazećeg zahtjeva [14]), a redni broj kreatora objave *user\_id* pomoću *Auth::id()* klase koja dohvaća *id* (redni broj) trenutno autentificiranog korisnika.

*Edit* metoda preko parametra *\$id* traži objavu koja odgovara istom *\$id* te vraća u obliku *json* objekta podatke o traženoj objavi. U varijablu *\$post* sprema se model objave pomoću metode *findOrFail(\$id)*. Ukoliko objava nije pronađena *Laravel* će na izbaciti poruku izuzetka/odstupanja (*Exception Message*).

*Update* metodom pomoću također parametra *\$id* i metode *findOrFail(\$id)* pronalazimo objavu čije atribute želimo izmijeniti, te nakon toga atribute pronađene objave izjednačimo sa novim postavljenim vrijednostima iz poslane forme pomoću *\$request* varijable. Na kraju je još potrebno napraviti *save*

naše objave sa novim atributima. Na kraju metoda kao u slučaju sa *edit* metodom vraća *json* objekt izmjenjene objave.

*Destroy* metoda služi za brisanje objava. Pomoću parametra *\$id* i metode *findOrFail(\$id)* pronalazimo objavu za brisanje u bazi podataka. Pronađena objava spremjena je u varijablu *\$post* na koju nadovezujemo funkciju *delete()* koja briše objavu iz baze podataka. U ovoj *destroy* metodi metoda vraća *\$id* izbrisane objave.

*home.blade.php* pogled je zadužen za prikaz naslovne stranice društvene mreže. U njegovom je kodu pod *html* oznakom `<feed></feed>` označena komponenta *frontend* dijela aplikacije (*Vue.js*). *Feed.vue* komponenta svojom metodom *get\_feed()* vrši *get* zahtjev na */feed* putanju u aplikaciji, a ta putanja označava *feed* metodu *FeedController.php* upravitelja.

```

<template>
  <div class="container-fluid">
    <div class="text-center" v-if="loading">
      Loading...
    </div>
    <div class="text-center" v-if="postsNumber === 0">
      No posts.
    </div>
    <div class="row spacing" v-for="(post,index) in posts" :key="index">
      <div class="post-container">
        <div class="card">
          <div class="card-header">
            
            <span class="text-center">{{ post.user.name }}</span>
          </div>
          <div class="card-body text-center">
            {{ post.content }}
            <hr/>
            <div class="float-left">
              <like :id="post.id"></like>
            </div>
            <div class="float-right">
              <span>Posted: {{ moment(post.created_at).fromNow() }}</span>
            </div>
          </div>
          <div class="card-footer">
            <post-comments :post_id="post.id"></post-comments>
          </div>
        </div>
      </div>
    </div>
  </div>
</template>

```

Slika 28 - Kod *Feed.vue* datoteke (1)

Odgovorom na zahtjev (*response*) dobivamo popis svih objava u obliku *json* objekta. Sve dobivene objave u odgovoru spremamo u polje *posts* u *store.js* datoteci koja nam služi kao spremnik za lakši prikaz podataka. *Store.js* datoteka nam je korisna jer iz nje možemo globalno pristupati bilo kojim spremljenim podacima [15]. Pomoću *computed* svojstva i njegove *posts* funkcije dohvaćamo iz *store.js* spremnika sve naše objave. Unutar `<template></template>` oznaka napisan je izgled naše naslovne stranice. Objave su prikazane u obliku "kartica", izlistane su pomoću *for* petlje koja prolazi kroz *posts* polje (`v-for="(posts,index) in posts"`), gdje su ključevi za svaku izlistanu objavu prikazani klasičnom *javascript* notacijom (npr. `{{ post.content }}` će prikazati vrijednost sadržaja izlistane objave). Unutar *Feed.vue* koda mogu se primjetiti podređene komponente *Like.vue* (oznaka `<like></like>`) te *PostComments.vue* (oznaka `<post-comments></post-comments>`). Objema komponentama se kao parametar proslijeđuje *id* objave kako bi se lakše dohvatio broj *likeova* odnosno broj komentara za prikazanu objavu.

```
import Like from "../Like.vue";
import PostComments from "../PostComments.vue";

let moment = require("moment");
export default {
  created() {
    this.get_feed();
  },

  data() {
    return {
      moment: moment,
      loading: true,
    };
  },

  computed: {
    posts() {
      return this.$store.getters.all_posts;
    },

    postsNumber() {
      return this.$store.getters.all_posts_count;
    }
  },

  components: {
    Like,
    PostComments
  },

  methods: {
    get_feed() {
      this.loading = true;
      this.$http.get("/feed").then(response => {
        if (response.bodyText === "[]") {
          this.loading = false;
        }
        response.body.forEach(post => {
          this.$store.commit("add_post", post);
          this.loading = false;
        });
      });
    }
  }
};
```

Slika 29 - Kod *Feed.vue* datoteke (2)

### 7.3 Like funkcionalnost

*Like* funkcionalnost jedna je od stavki bez koje je gotovo nezamisliva bilo koja vrsta društvene mreže. To je jedan od najlakših načina da se "rangiraju" objave korisnika. *Like* komponentu već ste mogli primjetiti u kodu *Feed.vue* komponente. Smještena je u podnožju (*footeru*) svake objave.

```

<template>
  <div>
    <button class="btn btn-success" v-if="!auth_user_liked_post" @click="like_post()">
       Like
    </button>
    <button class="btn btn-warning" v-else @click="unlike_post()">
       Unlike
    </button>
    <a class="btn btn-link" data-toggle="modal" :data-target="'#likersModal'+this.id">
      <span class="btn btn-default" v-if="likers.length != 0">
        {{ likers.length }}
      </span>
    </a>

    <div class="modal fade" :id="'likersModal'+this.id" role="dialog">
      <div class="modal-dialog">
        <!-- Modal content -->
        <div class="modal-content">
          <div class="modal-header">
            <h4 class="modal-title">List of likers</h4>
            <button type="button" class="close" data-dismiss="modal">&times;</button>
          </div>
          <div class="modal-body">
            <ul class="list-group">
              <li class="list-group-item" v-for="(like,index) in post.likes" :key="index">
                <a :title="like.user.name" :href="'/profile/' + like.user.username">
                  
                  <span>{{ like.user.name }}</span>
                </a>
              </li>
            </ul>
          </div>
        </div>
      </div>
    </div>
  </div>
</template>

```

Slika 30 - Kod Like.vue datoteke (1)

Za *like* funkcionalnost potreban nam je *LikeController.php* upravitelj, *Like.php* model i *create\_likes\_table.php* datoteka koju migracijom pretvaramo u *likes* tablicu u bazi podataka. U *LikeController.php* upravitelju imamo dvije metode. Prva metoda *like* kao parametar prima *post\_id* (redni broj objave u bazi podataka) gdje pomoću nje *findOrFail()* funkcijom spremamo željenu objavu u *\$post* varijablu. Zatim sa funkcijom *create* stvaramo novi *like* gdje je *post\_id* jednak parametru kojeg smo dobili u metodi a *user\_id* trenutno prijavljenom korisniku koji je kliknuo *like*. Metoda vraća novostvoreni *like* objekt. Druga metoda *unlike* također prima parametar *\$post\_id*, pronalazi objavu pomoću *findOrFail()*. Zatim koristeći *where* funkciju filtrira one *like* zapise (redove u bazi podataka) koji pod atributom *user\_id* imaju *id* jednak *id-u* trenutno prijavljenog korisnika. Na filtrirane objave dodajemo i uvjet (ponovno *where* funkcijom) gdje uzimamo objave kojima je *post\_id* atribut jednak *id-u* naše pronađene objave spremljene u *\$post* varijabli. Na kraju specificiramo funkcijom *first()* da želimo uzeti prvu objavu sa karakteristikama "filtera" kojeg smo napravili. Kada smo pronašli *like* funkcijom *delete()* ga brišemo iz baze podataka. Metoda vraća *\$id* izbrisanog *like-a*.

U kodu *Like.vue* komponente možemo primjetiti dva gumba i *modal* prozor. Prvi gumb na svoj klik poziva funkciju za stvaranje/dodavanje novog *likea* (*like\_post()* funkcija) za željenu objavu a drugi gumb miče taj *like* ukoliko ga želimo ukloniti (*unlike\_post()* funkcija). Pored gumbova nalazi se brojač sa *like* ikonom koji nam govori koliko *likeova* ima objava. Klikom na "brojač" *likeova* otvara nam se *modal* sa popisom korisnika koji su pritisnuli *like* gumb za objavu.



Slika 31 - Izgled like dijela objave

*Computed* svojstvo u ovoj komponenti zaduženo je za dohvaćanje svih korisnika koji su kliknuli *like* gumb na objavu (*likers* polje), provjeru nalazi li se trenutno prijavljeni korisnik u polju *likers* te naravno pronalazak objave (*post* funkcija) iz polja *posts* (*store.js* datoteka) po proslijeđenom parametru *id* iz nadređene *Feed.vue* komponente.

*like\_post* metoda radi *get* zahtjev na */like\_post* putanju uz parametar *id* koji označava redni broj objave. Putanja odgovara *like* metodi *LikeController.php* upravitelja. Odgovor na zahtjev poslužiti će da ažuriramo polje *likes* u *store.js* datoteci prosljeđujući objekt odgovora u *update\_post\_likes* funkciju .

```
export default {
  props: ["id"],
  computed: {
    likers() {
      let likers = [];
      this.post.likes.forEach(like => {
        likers.push(like.user.id);
      });

      return likers;
    },
    auth_user_liked_post() {
      let check_index = this.likers.indexOf(this.$store.state.auth_user.id);

      if (check_index === -1) {
        return false;
      } else {
        return true;
      }
    },
    post() {
      return this.$store.state.posts.find(post => {
        return post.id === this.id;
      });
    }
  },
}
```

Slika 32 - Kod *Like.vue* datoteke (2)

*unlike\_post* metoda također radi *get* zahtjev, ali na putanju */unlike\_post* koja odgovara *unlike* metodi *LikeController.php* upravitelja. Odgovor na zahtjev poslužiti će da ažuriramo polje *likes* u *store.js* datoteci prosljeđujući objekt odgovora u *unlike\_post* funkciju .

```

methods: {
  like_post() {
    this.$http.get("/like_post/" + this.id).then(response => {
      this.$store.commit("update_post_likes", {
        id: this.id,
        like: response.body
      });
    });

    this.$swal({
      type: "success",
      position: "bottom-left",
      text: "Post liked",
      showConfirmButton: false,
      timer: 3000
    });
  },

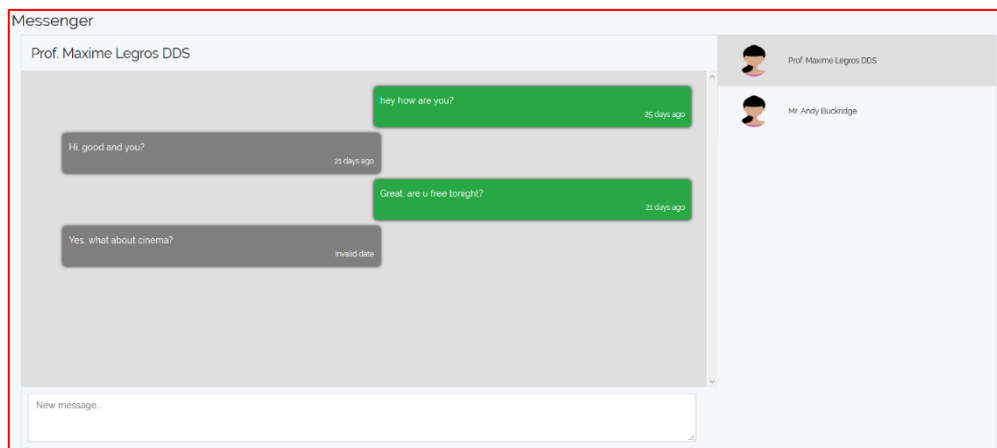
  unlike_post() {
    this.$http.get("/unlike_post/" + this.id).then(response => {
      this.$store.commit("unlike_post", {
        id: this.id,
        like_id: response.body
      });
    });
  }
}
}

```

Slika 33 - Kod Like.vue datoteke (3)

## 7.2 Messenger

Messenger je u aplikaciji smješten u poseban pogled *messages.blade.php* koji u sebi sadrži *html* oznaku komponente `<messenger></messenger>` (*Messenger.vue*). Njoj se kao svojstvo proslijeđuje *id* trenutno prijavljenog korisnika.



Slika 34 - Izgled Messenger prozora

Najprije je potrebno napraviti model poruke odnosno *Message.php*, on u sebi sadrži: tekst poslana poruke (*message*), *id* korisnika koji je poslao poruku (*from*) i *id* korisnika koji prima poruku (*to*). *MessageController.php* je upravitelj koji dohvaća poruke za određeni razgovor i stvara novu poruku nakon zahtjeva za slanje. *conversation* metoda zadužena je za dohvaćanje poruka između prijavljenog korisnika i korisnika odabranog u listi razgovora (kontakt korisnik). Kao parametar prima *\$contact\_id*

odnosno *id* kontakta sa kojim želi započeti razgovor. U *\$messages* varijablu spremaju se sve poruke između prijavljenog korisnika i kontakta. Prvo se dohvate sve poruke koje smo poslali odabranom korisniku, zatim se na njih nadovežu poruke koje je prijavljenom korisniku poslao kontakt i u kojima je primatelj poruke (*to* atribut) prijavljeni korisnik.

```
public function messages()
{
    return view('messages');
}

public function conversation($contact_id)
{
    $messages = Message::where(function($q) use ($contact_id) {
        $q->where('from', Auth::user()->id);
        $q->where('to', $contact_id);
    }->orWhere(function($q) use ($contact_id) {
        $q->where('from', $contact_id);
        $q->where('to', Auth::user()->id);
    }->get());

    return response()->json($messages);
}

public function send_message(Request $request)
{
    $message = Message::create([
        'from' => Auth::user()->id,
        'to' => $request->contact_id,
        'message' => $request->message,
    ]);

    broadcast(new NewMessage($message));

    return response()->json($message);
}
```

Slika 35 - Kod *MessageController.php* upravitelja

```
<?php

namespace App\Events;

use App\Message;
use Illuminate\Broadcasting\Channel;
use Illuminate\Queue\SerializesModels;
use Illuminate\Broadcasting\PrivateChannel;
use Illuminate\Broadcasting\PresenceChannel;
use Illuminate\Foundation\Events\Dispatchable;
use Illuminate\Broadcasting\InteractsWithSockets;
use Illuminate\Contracts\Broadcasting\ShouldBroadcast;

class NewMessage implements ShouldBroadcast
{
    use Dispatchable, InteractsWithSockets, SerializesModels;

    public $message;

    public function __construct(Message $message)
    {
        $this->message = $message;
    }

    public function broadcastOn()
    {
        return new PrivateChannel('messages.' . $this->message->to);
    }

    public function broadcastWith()
    {
        return ["message" => $this->message];
    }
}
```

Slika 36 - Kod *NewMessage.php* eventa

*send\_message* metoda zadužena je za slanje poruke odabranom kontaktu. U varijablu *\$message* spremamo novostvorenu poruku. Atribut *from* jednak je *id-u* prijavljenog korisnika. Atribut *to* i *message* dohvaćamo pomoću *\$request* varijable po imenima (*name*) iz poslane forme. Metoda vraća u obliku *json* objekta novu poruku koja je stvorena.

Za bolju funkcionalnost *messengera* koristiti ćemo pomoćni web servis *Pusher* [17]. On će nam omogućiti komunikaciju između dva korisnika u realnom vremenu. Unutar *Laravel* projekta društvene mreže već postoje postavke za *Pusher web* servis, ali su zakomentirane u kodu *bootstrap.js* datoteke koja se nalazi uz *Vue.js* komponente u projektu. Potrebno ih je maknuti van komentara te u *.env*

datoteci ispuniti pristupne podatke koje dobijemo besplatnom registracijom na službenoj stranici *Pushera*. Prva stvar koju moramo napraviti je novi *event* [18] prema kojem će web servis (*Pusher*) prepoznati što se dogodilo u aplikaciji i prema tome postupati. Novi *event* stvaramo pomoću **php artisan make:event NewMessage** naredbe. Kada *NewMessage event* bude pokrenut on će biti znak web servisu da je poruka poslana i da treba biti prosljeđena korisniku.

Na slici Slika 35 možemo vidjeti kod *NewMessage eventa*. Datoteka ima 3 metode i javni atribut *\$message* koji označava sadržaj poruke. U konstruktoru smo javnom atributu dodijelili vrijednost instance poslane poruke. Metoda *broadcastOn()* označava ime kanala (*channel*) koji ima "prijem" za *NewMessage event*, u ovom slučaju to je privatni kanal kako bi pristup konverzaciji bio omogućen samo za osobu koja šalje i za osobu koja prima poruku, u nastavku je još naznačeno kako će ime kanala biti *messages.{id korisnika kojem se šalje poruka}*. Metoda *broadcastWith()* vraća sadržaj poruke kako bi ga mogli prikazati u *frontend* dijelu aplikacije. Privatnost kanala provjerava se u *channels.php* datoteci koja se nalazi u *routes* folderu. Na donjoj slici nalazi se kod *channels.php* datoteke. Funkcije unutar ove datoteke nam služe kako bi provjerili ima li autentificirani korisnik ovlasti za slušanje nekog

```
<?php

Broadcast::channel('messages.{id}', function ($user, $id) {
    return $user->id == (int) $id;
});

Broadcast::channel('App.User.{id}', function ($user, $id) {
    return (int) $user->id === (int) $id;
});
```

Slika 37 - *Channels.php*

kanala. Npr. provjerava se parametar *id* u nazivu kanala '*messages.{id}*' (isti naziv kanala korišten u *broadcastOn()* metodi u *NewMessage.php* datoteci) sa *id-em* prijavljenog korisnika.

U *frontend* dijelu *messengera* imamo glavnu komponentu *Messenger.vue* unutar koje imamo podređene komponente: *ContactList.vue* i *Conversation.vue*. *ContactList.vue* komponenta nalazi se na desnoj strani prozora unutar koje su svi korisnici kojima možemo slati poruke (korisnici sa kojima smo "prijatelji" na društvenoj mreži). *Conversation.vue* komponenta sastoji se od *NewMessage.vue* i *MessagesFeed.vue* komponente. *NewMessage.vue* komponenta sastoji se od bloka za pisanje poruke, dok se unutar *MessagesFeed.vue* komponente nalaze sve poruke između nas i odabranog korisnika.

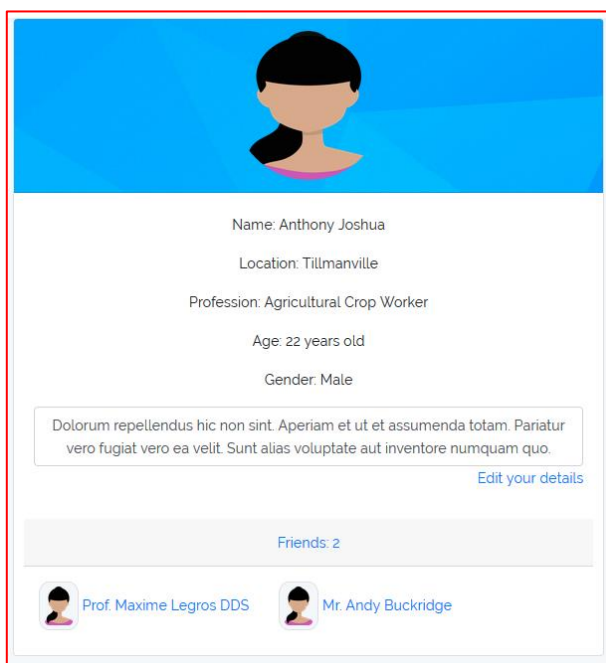
*Messenger.vue* komponenti se iz *messages.blade.php* pogleda kao što smo mogli vidjeti na slici Slika (messages.blade.php) šalje *id* trenutno prijavljenog korisnika. U *data()* svojstvu definirana su 3 najbitnija podatka, a to su: *selectedContact* – označava trenutno odabranog korisnika za razgovor, *messages* – polje sa svim porukama između korisnika i *contacts* – polje sa svim kontaktima. U *mounted* svojstvu pomoću *Laravel Echo* [19] definiramo naziv kanala i "slušamo" *NewMessage event* koji nam signalizira novu poruku. Kada je *Echo* primjetio *NewMessage event* pomoću metode *handle\_incoming\_message()* prosljeđujemo novu poruku u polje sa ostalim porukama. U *mounted* svojstvu se radi i *get* zahtjev na */my\_friends* putanju sa parametrom *id-ja* od prijavljenog korisnika. U odgovoru dobivamo popis svih kontakata koje prosljeđujemo u *contacts* polje. U metodama se nalazi i *start\_conversation()* metoda kojoj se kao parametar prosljeđuje objekt kontakta sa kojim želimo započeti razgovor. Vršiti se *get* zahtjev na */conversation* putanju sa parametrom *id-ja* od objekta kontakta. Vrijednost podatka *selectedContact* postaje dobiveni kontakt, a kao odgovor na zahtjev dobivamo poruke koje prosljeđujemo u *messages* polje.

*Conversation.vue* komponenti se prosljeđuje kontakt za koji je potrebno dohvatiti konverzaciju i skupina poruka za tu konverzaciju. Ova komponenta koristi metodu *send\_message()* kojoj se prosljeđuje sadržaj poruke napisan u bloku od strane korisnika. U metodi se vrši *post* zahtjev na */conversation/send\_message* putanju sa objektom u kojem se nalazi *id* kontakta kojem se poruka šalje i sadržaj poruke. U odgovoru na zahtjev dobivamo cijeli objekt novostvorene poruke koji možemo



iskoristiti da ga odmah dodamo u *messages* polje sa ostalim porukama bez ponovnog slanja zahtjeva. *NewMessage.vue* komponenta sastoji se od bloka za pisanje poruke. Na pritisak *ENTER* tipke pokrećemo *send()* metodu. *Send* metoda uzima vrijednost napisanu u tekstualnom bloku i pokreće *event send* koji nam je potreban za pokretanje *send\_message()* metode u nadređenoj *Messenger.vue* komponenti. *ContactsList.vue* komponenta služi za prikazivanje svih kontakata. Njoj se iz *Messenger.vue* komponente prosljeđuju kontakti. Nakon klika na jedan od kontakata pokreće se metoda *selectContact* koja služi kao okidač za *selected event*. *select event* nam je potreban kako bismo u nadređenoj komponenti (*Messenger.vue*) pokrenuli *start\_conversation* metodu.

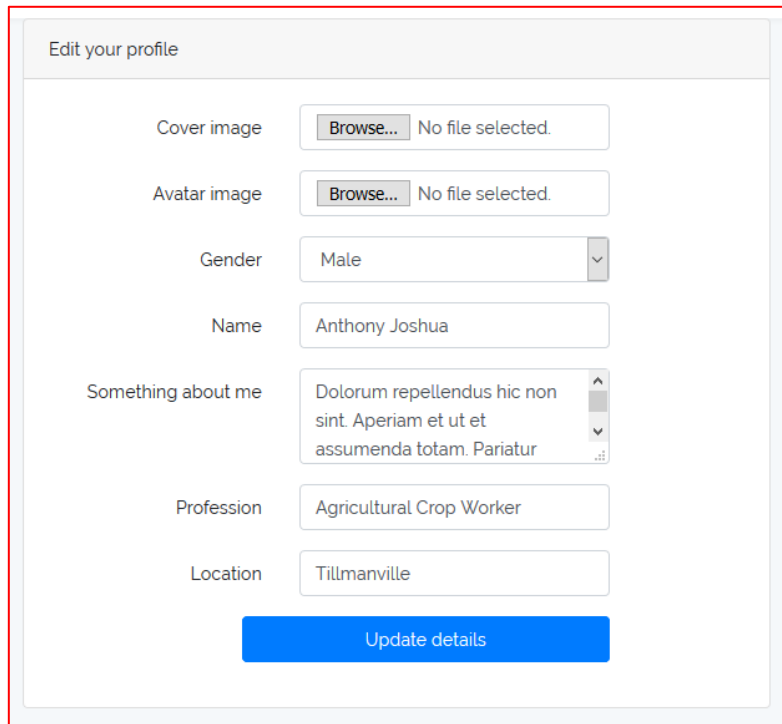
### 7.3 Uređivanje korisničkog profila



Slika 38 - Izgled korisničkog profila

Za uređivanje vlastitog profila u aplikaciji napravljen je novi model *Profile.php* zajedno sa migracijom *create\_profiles\_table.php* kako bi se "proširio obujam" korisničkih podataka. U *Profile.php* modelu dodani su atributi *profession* (zanimanje korisnika), *about* (kratak opis korisnika sa kojim se želi predstaviti ostalim korisnicima kada posjete njegov profil), *location* (mjesto na kojem korisnik trenutno boravi). Svaki profil povezan je sa *id-jem* korisnika. Tako se svaki profil sastoji od maloprije navedenih atributa, *id-ja* korisnika kojem profil pripada i *id-ja* profila. Kada korisnik klikne u padajućem izborniku na *My Profile* stavku, šalje se *get* zahtjev na */profile* putanju sa parametrom korisničkog imena korisnika. Npr. za korisnika imena Ivo Ivić korisničko ime bi bilo *ivo-ivic*, a putanja do njegovog profila bi bila */profile/ivo-ivic*. Klikom na link *Edit your details* šalje se *get* zahtjev na */profile/{korisnicko ime}/edit* putanju koja odgovara *edit* metodi *ProfileController.php* upravitelja. Pomoću *edit* metode i prosljeđenog korisničkog imena možemo u *edit* metodi dohvatiti dosadašnje podatke za korisnika. Metoda vraća pogled zajedno sa pronađenim korisnikom. Za ažuriranje profila zadužena je *update* metoda upravitelja. Nakon ispunjenih novih podataka, i klikom na gumb *Update details* šalje se *post* zahtjev na */profile/{korisnicko ime}* putanju. U *update* metodi provjerava se sa *Validatorom* [19] ukoliko je promjenjeno ime korisnika zadovoljava li potrebne uvjete (npr. minimalna ili maksimalna dužina imena). Nakon toga pomoću korisničkog imena pronalazi se korisnik, a nakon toga i njegov

profil. Zatim se vrši provjera je li prilikom izmjene podataka bilo *uploada* nove profilne ili naslovne slike. Ukoliko se to dogodilo slike se spremaju u lokalni spremnik, a ukoliko nije preskače se taj dio ažuriranja. Na kraju se funkcijom *save()* spremaju izmjenjeni podatci za profil i osnovne korisničke podatke korisnika. Metoda vraća početni prikaz profila sa izmjenjenim podacima i *flash* porukom [20] ukoliko je uspješno obavljeno ažuriranje.



The screenshot shows a web form titled "Edit your profile". It contains several input fields: "Cover image" and "Avatar image" both have "Browse..." buttons and "No file selected." text; "Gender" is a dropdown menu set to "Male"; "Name" is a text input with "Anthony Joshua"; "Something about me" is a text area with placeholder text; "Profession" is a text input with "Agricultural Crop Worker"; and "Location" is a text input with "Tillmanville". A blue "Update details" button is at the bottom.

Slika 39 - Izgled prozora za uređivanje profila

```
public function __construct()
{
    $this->middleware('auth');
}

public function index($username)
{
    $user = User::where('username', $username)->first();
    return view('profiles.profile')->with('user', $user);
}

public function edit($username)
{
    $user = User::where('username', $username)->first();

    if (Auth::user()->id != $user->id) {
        return redirect()->back();
    }
    return view('profiles.edit')->with('user', $user);
}
```

Slika 40 - Kod ProfileController.php datoteke

```
public function update($username, Request $request)
{
    $validator = Validator::make($request->all(), [
        'name' => 'required|string|max:255',
    ]);

    if ($validator->fails()) {
        return redirect()->back()
            ->withErrors($validator)
            ->withInput();
    }

    $user = User::where('username', $username)->first();
    $profile = Profile::findOrFail($user->id);

    $user->gender = $request->get('gender');
    $user->name = $request->get('name');

    // updating cover image //
    if ($request->hasFile('cover')) {
        Auth::user()->update([
            'cover' => $request->cover->store('public/covers')
        ]);
    }

    // updating avatar image //
    if ($request->hasFile('avatar')) {
        Auth::user()->update([
            'avatar' => $request->avatar->store('public/avatars')
        ]);
    }

    $profile->location = $request->get('location');
    $profile->about = $request->get('about');
    $profile->profession = $request->get('profession');

    $user->save();
    $profile->save();

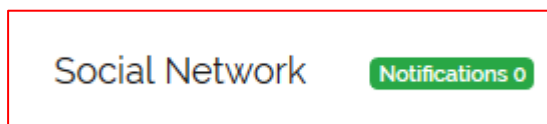
    $request->session()->flash('alert-success', 'Profile was successfully updated!');

    return redirect('/profile/' . $username);
}
```

Slika 31 - Update metoda ProfileController.php upravitelja

## 7.4 Obavijesti(notifikacije)

Jedna od notifikacija koja se koristi u aplikaciji stvorena je pomoću naredbe *php artisan make:notification NewFriendRequest.php*. Notifikacije se nalaze unutar *app* direktorija u poddirektoriju *Notifications*. Kao što se može iz imena notifikacije zaključiti radi se o notifikaciji koju korisnik dobije prilikom novog zahtjeva za prijateljstvo. Notifikacije se prosljeđuju korisnicima u realnom vremenu tako da će se i kod notifikacija koristiti *Pusher web* servis.



Slika 42 - Statusna traka za notifikacije

U slučaju ove notifikacije ona se poziva u *add\_friend()* metodi *FriendshipController.php* upravitelja. Metodi se prosljeđuje *id* korisnika kojem se šalje zahtjev za prijateljstvo i pomoću *add\_friend()* funkcije iz *Friendable.php* skupine funkcija (*Traits*) dodajemo novog prijatelja. No, kada želimo pri tome u istom trenutku i obavijesiti korisnika da smo mu poslali zahtjev za prijateljstvo, koristimo funkciju *notify()* u kojoj pozivamo novu instancu klase *NewFriendRequest* sa prosljeđenim parametrom korisnika koji šalje zahtjev a to je zapravo trenutno prijavljeni korisnik.

```
public function add_friend($id)
{
    $resp = Auth::user()->add_friend($id);

    User::find($id)->notify(new NewFriendRequest(Auth::user()));

    return $resp;
}
```

Slika 43 - *add\_friend* metoda *FriendshipController.php* upravitelja

U kodu *NewFriendRequest.php* notifikacije možemo primjetiti 3 metode. U konstruktoru se kao što je bio slučaj i kod *NewMessage.php* eventa javni atribut *\$user* dodijeli dobivenom parametru koji je poslan *notify()* funkcijom. U *via* metodi specificiramo koje sve oblike obavijesti želimo napraviti. U ovom slučaju to je *mail*, *broadcast* i *database* opcija. Za *mail* opciju koristimo *Mailtrap web* servis [21]. Koji je ukratko testni *smtp* server i odlično će poslužiti za testiranje naše aplikacije. *Mailtrap* postavke već su pripremljene unutar *.env* datoteke i bitno je upisati vlastite šifre/ključeve koje se dobiju prilikom registracije na službenoj stranici *Mailtrapa*. Naše notifikacije imaju i svoju *create\_notifications\_table.php* migraciju u kojoj su u bazi podataka zapisane sve notifikacije. U metodi *toMail()* specificiramo kako će izgledati *email* koji korisnik dobije, a *toArray()* metoda uzima podatke koji su nam korisni u *frontend* dijelu aplikacije.

```

<?php
namespace App\Notifications;

use Illuminate\Bus\Queueable;
use Illuminate\Notifications\Notification;
use Illuminate\Contracts\Queue\ShouldQueue;
use Illuminate\Notifications\Messages\MailMessage;

class NewFriendRequest extends Notification implements ShouldQueue
{
    use Queueable;

    public $user;

    public function __construct($user)
    {
        $this->user = $user;
    }

    public function via($notifiable)
    {
        return ['mail', 'broadcast', 'database'];
    }

    public function toMail($notifiable)
    {
        return (new MailMessage)
            ->line('You received a new friend request from' . $this->user->name)
            ->action('View users profile', route('profile', ['username' => $this->user->username]))
            ->line('Thank you for using our social network!');
    }

    public function toArray($notifiable)
    {
        return [
            'name' => $this->user->name,
            'username' => $this->user->username,
            'message' => ' sent you a friend request.'
        ];
    }
}

```

Slika 44 - Kod *NewFriendRequest.php* datoteke

*Notification.vue* komponenta uključena je u *master.blade.php* pogledu (koji čini okvir svake stranice u aplikaciji) *html* oznakom `<notification></notification>`. Komponenti se proslijeđuje kao svojstvo *id* trenutno prijavljenog korisnika.

```

@if(Auth::check())
    <notification :id="{{ Auth::user()->id }}"></notification>
@endif
<audio id="notification_sound">
    <source src="{{ asset('audio/notification.mp3') }}">
</audio>

```

Slika 45 - Dio koda *master.blade.php* datoteke

Prije učitavanja komponente provjerava se je li korisnik prijavljen u aplikaciju (autentificiran) jer je jedino tada u mogućnosti primiti notifikacije. U *Notification.vue* komponenti se koristi metoda *listen()*. Pomoću te metode i *Laravel Echo* radi zahtjev prema privatnom kanalu sa *id-jem* prijavljenog korisnika. Odgovor na naš zahtjev omogućuje nam pristup poruci notifikacije koju ispujemo u *alert* prozoru. Za ovu aplikaciju korišten je *SweetAlert2* [22] paket koji nam omogućuje korištenje stiliziranih *alert* prozora. Nakon što je prikazana notifikacija ona se sprema u *notifications* polje u *store.js* datoteci (spremniku), te se nakon toga pokreće zvuk notifikacije u `<audio></audio>` elementu sa *id-jem* *notification\_sound*.

```

export default {
  mounted(){
    this.listen()
  },
  props: ['id'],
  methods: {
    listen(){
      Echo.private('App.User.' + this.id)
        .notification(notification) =>{
          this.$swal({
            position: 'bottom-left',
            text: notification.name + notification.message,
            showConfirmButton: false,
            timer: 3000
          });

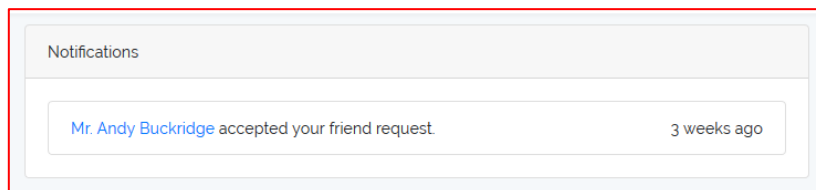
          this.$store.commit('add_notification', notification);

          document.getElementById('notification_sound').play();
        })
    }
  }
}

```

Slika 46 - Kod *Notification.vue* komponente

Za izgled statusne trake za notifikacije zadužena je *UnreadNotifications.vue* komponenta. Ona se nalazi u *navigation.blade.php* pogledu. U njezinom *computed()* svojstvu pomoću funkcije *all\_notifications\_count()* poziva se dužina polja *notifications* koja označava i broj nepročitanih notifikacija iz *store.js* datoteke. Metoda *get\_notifications()* kako joj samo ime kaže dohvaća sve notifikacije i sprema ih u *notifications* polje *store.js* datoteke.



Slika 47 - Popis notifikacija

Klikom na statusnu traku otvara se *notifications.blade.php* pogled gdje se mogu vidjeti sve notifikacije. Koje su u pogled proslijeđene putem *notifications()* metode u *HomeController.php* upravitelju. U metodi se sve notifikacije nakon njenog pokretanja pretvaraju u pročitanu (*read notifications*).

```

Route::get('/get_notifications', function () {
  return Auth::user()->unreadNotifications;
});

```

Slika 48 - Putanja za nepročitanu notifikaciju

```
public function notifications()
{
    Auth::user()->unreadNotifications->markAsRead(); // by default sorted created_at
    return view('notifications')->with('notifications', Auth::user()->notifications);
}
```

Slika 49 - Notifications metoda HomeController.php upravitelja

```
<template>
  <div>
    <a class="nav-link" href="/notifications">
      <span class="badge badge-success"> Notifications {{ all_notifications_count }} </span>
    </a>
  </div>
</template>

<script>
  export default {
    mounted() {
      this.get_notifications()
    },

    computed: {
      all_notifications_count() {
        return this.$store.getters.all_notifications_count
      }
    },

    methods: {
      get_notifications() {
        this.$http.get('/get_notifications')
          .then((notifications) => {
            notifications.body.forEach((notification) => {
              this.$store.commit('add_notification', notification)
            })
          })
      }
    }
  }
</script>
```

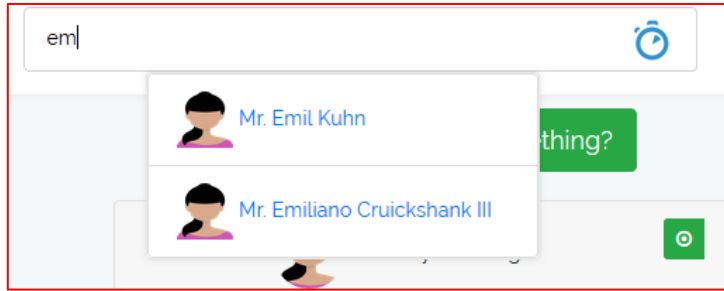
Slika 50 - Kod UnreadNotifications.vue datoteke

## 7.5 Pretraga korisnika

Za pretragu korisnika korišten je *Laravel Scout* [23] i *Algolia* [24] web servis. *Laravel Scout* nam omogućava da pretražujemo unutar svoje aplikacije sa fokusom na modele koje smo napravili. Samim time svakom promjenom zapisa u bazi podataka, pretraga je sinkronizirana, jer se bazira na modelu (u ovom slučaju to je *User.php* model). Nakon registracije na službenoj stranici *Algolia web* servisa dobili smo pristupne podatke za naš korisnički profil koje zapisujemo u *.env* datoteci. Nakon toga potrebno je u modelu kojeg pretražujemo, a to je *User.php* model naznačiti sa ključnom riječi *use* da ćemo koristiti *Searchable* skup funkcija.

```
ALGOLIA_APP_ID=XMOONZHM6E
ALGOLIA_SECRET=58bc6ddeb90c6984bac559a455def7a9
```

Slika 51 - Algolia pristupni podatci u .env datoteci



Slika 52 - Izgled polja za pretraživanje

Kako bi *Laravel Scout* znao koji ćemo model koristiti u pretrazi potrebno je još pokrenuti ***php artisan scout:import "App\User"*** naredbu. Kada je naredba izvršena sigurni smo da je naš model u potpunosti sinkroniziran sa *Alodia web* servisom.

Za prikaz polja za pretraživanje koristiti ćemo *Search.vue* komponentu. Ona se sastoji od polja u koje se upisuje ime korisnika. Kako se upisuje ime korisnika istovremeno se generira padajuća lista sa rezultatima najbližim napisanom tekstu u polju.

```
let algoliasearch = require('algoliasearch');
let client = algoliasearch('XMOONZHM6E', '58bc6ddeb90c6984bac559a455def7a9');
let index = client.initIndex('users');

export default {
  data() {
    return {
      query: '',
      hiddenResults: false,
      results: []
    }
  },
  watch: {
    query() {
      if (this.query.length > 0) {
        this.hiddenResults = true;
      } else {
        this.hiddenResults = false;
      }
    }
  },
  methods: {
    search() {
      index.search(this.query, (err, response) => {
        this.results = response.hits
      });
    }
  }
}
```

Slika 53 - Kod *Search.vue* (1) komponente



Na početku skripte uključuje se *algoliasearch* paket/modul. Za njega se uspostavljaju potrebni ključevi i definira se ime tablice za model koji je sinkroniziran na *web* servisu. U *data()* svojstvu imamo *query* podataka koji odgovara upisanom tekstu u polju za pretraživanje, *hiddenResults* odnosno rezultate pretraživanja koji su po *defaultu* skriveni dok ništa nije upisano u polje za pretraživanje. I imamo *results* polje u koje spremamo sve dobivene rezultate pretraživanja. *watch* svojstvo nadzire dužinu napisane riječi za pretraživanje te ukoliko nije ništa napisano (duljina *query* stringa je jednaka nuli) rezultati su i dalje skriveni. U protivnom je prikazana padajuća lista sa rezultatima. U *search()* metodi radimo zahtjev na *Algolia web* servis sa upisanom riječi. Odgovor spremamo u *results* polje.

```
<template>
  <div class="container-fluid">
    <div class="row">
      <div class="col-md-8 form-control">
        <input class="col-md-11 search-input" type="text" @keyup="search" v-model="query"
          placeholder="Search for other users...">
        <a href="https://www.algolia.com/"><span></span></a>
      </div>
    </div>
    <div class="search-results-div" v-show="hiddenResults">
      <div class="list-group" v-if="results.length">
        <span class="list-group-item" v-for="(user, index) in results" :key="index">
          <a :href="/profile/' + user.username'"> {{ user.name}} </a>
        </span>
      </div>
    </div>
  </div>
</template>
```

Slika 54 - Kod *Search.vue* komponente (2)

## 8. Zaključak

Danas se sve više susrećemo sa učestalom potrebom izrade i korištenja *web* aplikacija. Motiviran vlastitim afinitetom upravo prema toj vrsti aplikacijskih ili programabilnih rješenja, odlučio sam se na ipak nešto veće i detaljnije istraživanje, koje sam i odlučio "pretočiti" u ovaj završni rad. Smatram da mi je istraživanje puno pomoglo da mi se jednostavno poslože neke do sada za mene apstraktne stvari, točnije da shvatim kako bi trebala izgledati moderna, stabilna i najvažnije lako održiva *web* aplikacija.

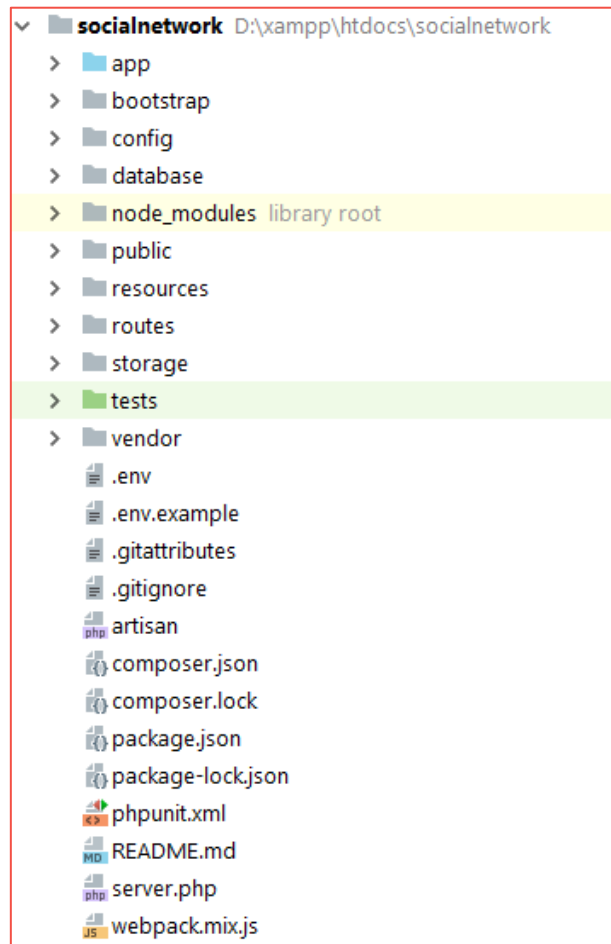
## 9.Literatura

- [1] *Laravel*, <https://laravel.com>, pristupljeno 15.5.2018
- [2] *What is Web application*, <https://www.maxcdn.com/one/visual-glossary/web-application/>, pristupljeno 15.5. 2018.
- [3] *MVC Arhitecture*, <https://www.tutorialspoint.com/mvc-framework/mvc-framework-introduction.htm>, pristupljeno 15.5.2018
- [4] *Most popular frameworks*, <https://hotframeworks.com/>, prisutpljeno 15.5.2018.
- [5] *Blade Template Engine*, <https://laravel.com/docs/5.6/blade>, pristupljeno 15.5.2018.
- [6] *What Is A Framework*, <https://www.codeproject.com/Articles/5381/What-Is-A-Framework>, pristupljeno 28.5.2018.
- [7] *The Rise of Micro-Frameworks*, <https://blog.appdynamics.com/engineering/php-microframework-vs-full-stack-framework/>, pristupljeno 28.5.2018.
- [8] *Laravel History*, <https://en.wikipedia.org/wiki/Laravel>, pristupljeno 29.5.2018.
- [9] *Symfony*, <https://symfony.com/>, pristupljeno 29.5.2018.
- [10] *HeidiSQL*, <https://www.heidisql.com/>, pristupljeno 29.5.2018.
- [11] *Object-relational mapping*, [https://en.wikipedia.org/wiki/Object-relational\\_mapping](https://en.wikipedia.org/wiki/Object-relational_mapping), pristupljeno 30.5.2018.
- [12] *Composer*, <https://getcomposer.org/>, pristupljeno 30.5.2018.
- [13] *Vue JS*, <https://vuejs.org/v2/guide/>, pristupljeno 30.5.2018.
- [14] *Laravel Requests*, <https://laravel.com/docs/5.6/requests>, pristupljeno 30.5.2018.
- [15] *What is Vuex?*, <https://vuex.vuejs.org/>, pristupljeno 6.6.2018.
- [16] *Pusher*, <https://pusher.com/>, pristupljeno 6.6.2018.
- [17] *Events*, <https://laravel.com/docs/5.6/events>, pristupljeno 6.6.2018.
- [18] *Laravel Echo*, <https://laravel.com/docs/5.6/broadcasting>, pristupljeno 13.6.2018.
- [19] *Laravel Validator*, <https://laravel.com/docs/5.6/validation>, pristupljeno 13.6.2018.
- [20] *Flash Data*, <https://laravel.com/docs/5.6/session>, pristupljeno 13.6.2018.
- [21] *Mailtrap*, <https://mailtrap.io/>, pristupljeno 15.6.2018.
- [22] *SweetAlert2*, <https://sweetalert2.github.io/>, pristupljeno 15.6.2018.
- [23] *Laravel Scout*, <https://laravel.com/docs/5.6/scout>, pristupljeno 15.6.2018.
- [24] *Algolia*, <https://www.algolia.com/>, pristupljeno 15.6.2018.
- [25] *Laravel Alternatives*, <https://www.slant.co/options/9622/alternatives/~laravel-5-alternatives>, pristupljeno 20.6.2018.

## 10. Prilozi

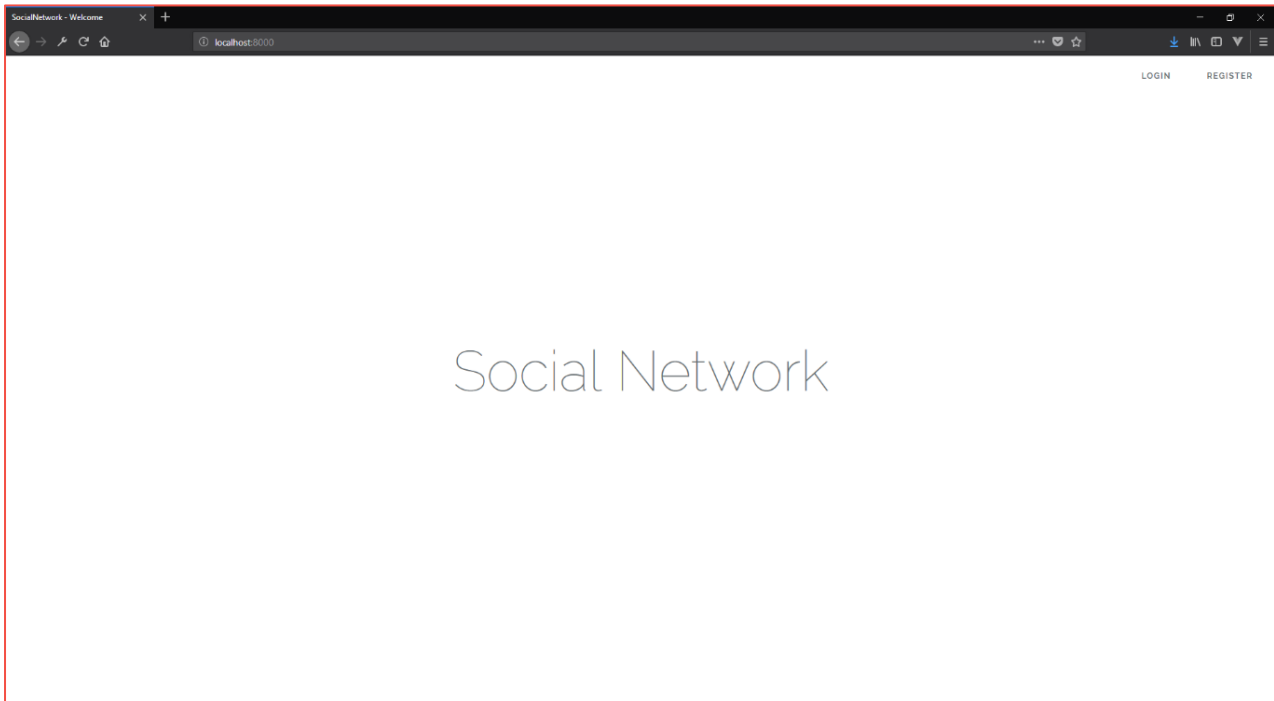
### 10.1 Laravelova instalacija

Nakon što smo se pozicionirali u direktorij na našem računalu gdje želimo da nam se stvori novi *Laravel* projekt, u slučaju ove *web* aplikacije to je unutar *htdocs* direktorija u *xampp* direktoriju na lokalnom D disku računala. U slučaju da koristite *WAMP* to bi bio direktorij pod nazivom *www*. *Composer* naredbom ***composer create-project laravel/laravel socialnetwork*** pokrećemo "instalaciju" novog *Laravel* projekta čija struktura izgleda kao na slici (Slika 7).



Slika 55 - Struktura novog *Laravel* projekta

Možda ste mogli već sada primjetiti s obzirom na raspored direktorija u novom projektu i *MVC* arhitekturu gdje će se nalaziti određene datoteke. Kako bi smo provjerili ispravnost novonastale *web* aplikacije pokrenuti ćemo lokalni server sa naredbom *php artisan serve*. Te bi smo na adresi "http://localhost:8000" pretraživača trebali dobiti prozor nalik prozoru na slici (Slika 8). Pokrenuti lokalni server mora raditi cijelo vrijeme u pozadini dok mi radimo na našoj *web* aplikaciji te ga se ne smije gasiti.



Slika 56 - Prikaz *web* aplikacije u *web* pregledniku

Naravno, po *defaultu* nakon instalacije novog *Laravelovog* projekta umjesto *Social Network* pisati će *Laravel* kao naslov te neće biti poveznica na prijavu i registraciju kao što se vidi u desnom gornjem uglu.