Razvoj REST API servisa u Laravel okruženju

Macan, Doroteo

Sveučilište Jurja Dobrile u Puli
Fakultet informatike u Puli

DOROTEO MACAN

RAZVOJ REST API SERVISA U LARAVEL OKRUŽENJU

Diplomski rad

Pula, 2019.

Sveučilište Jurja Dobrile u Puli
Fakultet informatike u Puli

DOROTEO MACAN

RAZVOJ REST API SERVISA U LARAVEL OKRUŽENJU

Diplomski rad

JMBAG: 0303045994, redoviti student

Studijski smjer: Informatika

Predmet: Napredni algoritmi i strukture podataka

Znanstveno područje: Društvene znanosti

Znanstveno polje: Informacijske i komunikacijske znanosti

Znanstvena grana: Informacijski sustavi i informatologija

Mentor: doc. dr. sc. Tihomir Orehovački

Pula, rujan 2019.

IZJAVA O AKADEMSKOJ ČESTITOSTI

Ja, dolje potpisani Doroteo Macan, kandidat za magistra informatike ovime izjavljujem da je ovaj Diplomski rad rezultat isključivo mojega vlastitog rada, da se temelji na mojim istraživanjima te da se oslanja na objavljenu literaturu kao što to pokazuju korištene bilješke i bibliografija. Izjavljujem da niti jedan dio Diplomskog rada nije napisan na nedozvoljen način, odnosno da je prepisan iz kojega necitiranog rada, te da ikoji dio rada krši bilo čija autorska prava. Izjavljujem, također, da nijedan dio rada nije iskorišten za koji drugi rad pri bilo kojoj drugoj visokoškolskoj, znanstvenoj ili radnoj ustanovi.

Student

Doroteo Macan

U Puli, rujan, 2019 godine

IZJAVA

o korištenju autorskog djela

Ja, Doroteo Macan dajem odobrenje Sveučilištu Jurja Dobrile u Puli, kao nositelju prava iskorištavanja, da moj diplomski rad pod nazivom RAZVOJ REST API SERVISA U LARAVEL OKRUŽENJU koristi na način da gore navedeno autorsko djelo, kao cjeloviti tekst trajno objavi u javnoj internetskoj bazi Sveučilišne knjižnice Sveučilišta Jurja Dobrile u Puli te kopira u javnu internetsku bazu završnih radova Nacionalne i sveučilišne knjižnice (stavljanje na raspolaganje javnosti), sve u skladu s Zakonom o autorskom pravu i drugim srodnim pravima i dobrom akademskom praksom, a radi promicanja otvorenoga, slobodnoga pristupa znanstvenim informacijama.

Za korištenje autorskog djela na gore navedeni način ne potražujem naknadu.

U Puli, rujan, 2019 godine

Potpis

Doroteo Macan

SADRŽAJ

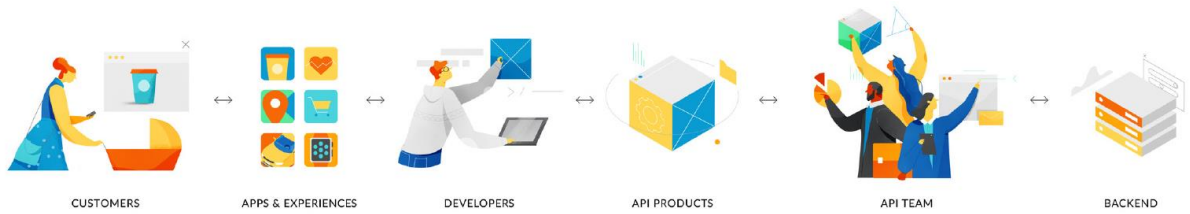
1. UVOD

U današnje doba se jedna aplikacija može pronaći na više različitih platformi. Najčešće su to Web, Windows, Mac, Android i iOS aplikacije. Programeri često započinju sa programiranjem za jednu željenu platformu, a kasnije kada se aplikacija popularizira i proširi dolazi do potražnje za istom i na drugim platformama. Programer tada istu aplikaciju mora programirati i za druge platforme, što sa sobom nosi mnoge probleme. Napraviti kopiju aplikacije za drugu platformu je često veći izazov nego napraviti originalnu aplikaciju. Time dolazi do velikog utroška vremena i moguće su pogreške koje se kasno detektiraju. Za prijenos na više drugačijih platformi često je potrebno uključiti i programere koji poznaju te tehnologije. Time se drastično povećavaju financijski troškovi. Programer, dok se bavi prijenosom na drugu platformu, ne može se baviti razvojem originalne aplikacije. Nakon što sve aplikacije dođu do iste točke razvoja potrebno ih je održavati. Svaka dodatna funkcionalnost zahtjeva novo programiranje na svim platformama. Sve ove probleme može djelomično riješit REST API servis.

Ovaj rad opisuje razvoj REST API servisa u kojem programer može svu logiku aplikacije napisati na jednom mjestu i istu koristiti sa bilo koje druge platforme. Time se u nekoliko aspekata olakšava razvoj aplikacija za druge platforeme. Aplikacije na drugim platformama ne moraju se brinuti o logici već samo o prikazu i prosljeđivanju podataka. Te aplikacije ne zanima što se dešava sa podacima. Kod promjena logike, aplikacije prikaza većinom nije potrebno niti prilagođavati jer se sve promjene događaju u pozadini na API-u. Puno je manja vjerojatnost da dođe do problema sa sinkronizacijom. Kod dodavanja novih funkcionalnosti nije potrebno iste imati na svim platformama. Sve ove pogodnosti smanjuju vremenske i financijske troškove, ako se aplikacija od samog početka stvara kao API aplikacija.

API nije namijenjen krajnjem korisniku, već je to aplikacija od programera za programera, kako bi se olakšao proces razvoja. Krajnji korisnik vjerojatno nikad neće izravno koristiti API kako bi pregledao ili spremao podatke. API servisi se najčešće razvijaju u tvrtkama koje znaju da će se aplikacija koristiti na nekoliko različitih platformi ili ako se osiguravaju za budućnost kako bi održavanje bilo lakše. Neke tvrtke žele pružiti pristup određenim podacima i funkcionalnostima sa slojem sigurnosti. API

omogućava da drugi programeri mogu koristiti podatke i funkcionalnosti koje mu oni dopuste bez da ima izravan pristup bazi podataka (Slika 1).



Slika 1. Uloga API-a unutar infrastrukture aplikacije (2018)

Ovaj rad se sastoji od uvoda, šest poglavalja o razvoju i primjeni servisa te zaključka. Drugo poglavlje opisuje problem i tehnologije koje su korištene za razvoj programa napisanih uz ovaj rad. Treće poglavlje opisuje što je Laravel okruženje (engl. framework), Laravelove alate i razvoj Web aplikacije u Laravel okruženju sa primjerima koda. Četvrto poglavlja opisuje REST API i razliku između Web i API aplikacije u Laravelu. Peto poglavlje prikazuje razvoj API servisa u Laravel okruženju sa isječcima koda iz programa koji je napisan u svrhu demonstracije razvoja za ovaj rad. Šesto poglavlje prikazuje dijelove programa Android aplikacije koji prikazuju primjere kako bi se API aplikacija, koja je opisna u petom poglavlju, mogla koristiti. Za razumijevanje ovog rada potrebno je znanje jezika PHP, SQL i Java.

2. OPIS PROBLEMA I KORIŠTENE TEHNOLOGIJE

Da se prikaže korisnost API-a uz ovaj rad je napravljena jednostavna API aplikacija kao primjer. Aplikacija za cilj ima praćenje rezultata odigranih društvenih igara. Mogućnosti koje aplikacija ima:

- Registracija korisnika
- Prijava korisnika
- Pregled i izmjena podataka o korisniku
- Pregled svih igara
- Dodavanje i brisanje igara iz vlastite knjižnice korisnika
- Dodavanje i brisanje drugih korisnika kao prijatelje
- Evidentiranje održanog igranja, označavanje prijatelja koji su sudjelovali u igri, unos i pregled rezultata

Za funkcionalnost ovo rada autor koristi i API koji je javno dostupan na internetu. Korišteni API je upotrjebljen za popunjavanje baze podataka sa popisom igara.

Razvoj REST API aplikacije podržava gotovo svako Web okruženje. Neka od popularnijih okruženja u kojima se može razvijati REST API aplikacija su: .NET, Spring, Laravel i sl.. .NET aplikacije se programiraju u Visual Basic ili C# programskim jezicima. Spring aplikacije se programiraju u Java programskom jeziku. Laravel aplikacije se programiraju u PHP programskom jeziku. Za razvoj REST API aplikacije ovog rada odabrano je Laravel okruženje koje je detaljno opisano u trećem poglavlju.

Razvijeni REST API se može upotrijebiti na bilo kojoj platformi. Za primjer korištenja razvijena je Android aplikacija. Od čitatelja ovog rada se očekuje poznavanje razvoja Android aplikacija kako bi imao potpuno razumijevanje šestog poglavlja.

3. LARAVEL

3.1 O LARAVELU

Laravel je okruženje otvorenog izvora (engl. open-source) koje služi za razvoj Web stranica pisanih u programskom jeziku PHP (Personal Home Page). PHP je skriptni programski jezik namijenjen za izradu dinamičkih Web stranica. PHP omogućava da se HTML stranica generira na strani servera te korisnik ne može vidjeti što se dogodilo u pozadini. Od korisnika Laravela se očekuje poznavanje osnova programiranja u PHP-u.

Laravel je kreirao Taylor Otwell u svrhu olakšavanja i ubrzavanja procesa razvoja Web aplikacija. Nastao je po uzoru na Ruby on Rails okruženju za PHP koji postoji od 2004. godine. Razvijen iz temelja, Laravel 1 je pušten u javnost 2011. godine. U samo nekoliko mjeseci razvijene su verzije 2 i 3. Taylor je tada odlučio ponovo napisati cijelo okruženje iz temelja, iz čega 2013. godine nastaje Laravel 4. Laravel 4 je postao standard u razvoju PHP Web stranica. Dvije godine kasnije očekivalo se unaprjeđenje na verziju 4.3, ali zbog velikih promjena i poboljšanja izašao je Laravel 5 koji je trenutno aktualan. Okruženje ima detaljno ispisanu dokumentaciju na internetu. Kreiran je tako da na pametan način pretpostavi što programer želi napraviti. Sa minimalnom količinom koda programer može dobiti željene podatke, dok i dalje postoji sloboda za potpunu modularnost koda ako programer pokušava napraviti nešto specifično. Programeru to znači da će novu Web stranicu moći napraviti u samo nekoliko jednostavnih koraka, ali to ga neće ograničiti da postigne željenu kompleksnost aplikacije. Popularnost je postigao i zbog mnogobrojnih alata, od onih za lokalno pokretanje aplikacije, vezu sa bazom, pretragu, autorizaciju i autentifikaciju, testiranje i slične radnje sa kojima se Web programer svakodnevno susreće. U nastavku su opisani neki od alata koje Laravel pruža. (Stauffer, 2017)

3.2 LARAVEL ALATI

3.2.1 COMPOSER

Composer je alat koji je potreban za instalaciju Laravela. Koristi se za upravljanje ovisnostima (engl. dependency) paketa i knjižnica (engl. library) kod razvoja PHP aplikacija. Nakon isnatalacije Composera instalacija Laravela je jednostavna. Naredba `composer global require "laravel/installer=~1.1"` u komandnoj liniji instalira Laravel. (Mundosaparte, 2019)

3.2.2 RAZVOJNE OKOLINE

Web aplikacije se lokalno pokreću pomoću alata kao što su MAMP, WAMP ili XAMPP. Laravel koristi interne alate Valet ili Homestead koji mogu zamijeniti alate koji se koriste globalno za PHP. Najjednostavniji primjer za pokretanje servera je naredba `php -S localhost:8000 -t public` u komandnoj liniji. Web stranici je tada moguće pristupiti preko URL-a *http://localhost:8000/* ili *http://127.0.0.1:8000/*.

3.2.3 ELOQUENT

Eloquent je alat koji ima intuitivnu interakciju sa bazama podataka poput MySQL, PostgreSQL, MSSQL i SQLite. Moguće je koristiti već postojeće baze i tablice podataka ili je moguće pomoću modela napisanih u Laravelu stvoriti nove tablice i/ili izmijeniti postojeće. Pisanje upita (engl. query) na bazu je pojednostavljeno. Eloquent je aktivni zapis koji može izvršavati CRUD (Create (hrv. kreiraj), Read (hrv. pročitaj), Update (hrv. ažuriraj), Delete (hrv. izbriši)) naredbe bez pisanja ijedne linije koda u SQL-u. Eloquent se bavi i vezom između tablica te ima mogućnost automatske raspodijele veće količine podataka na stranice.

3.2.4 ARTISAN

Artisan je Laravelovo sučelje za komandnu liniju. Artisan služi između ostalog za pokretanje migracija između modela i baze podataka, testova i planiranih zadataka. (Vo, 2014)

3.2.5 BLADE

Osim alata za rad sa podacima Laravel ima i Blade šablonu (engl. template) koja služi kao estetski privlačnija zamjena za PHP kod. (Vo, 2014)

3.3 STRUKTURA LARAVEL PROJEKTA

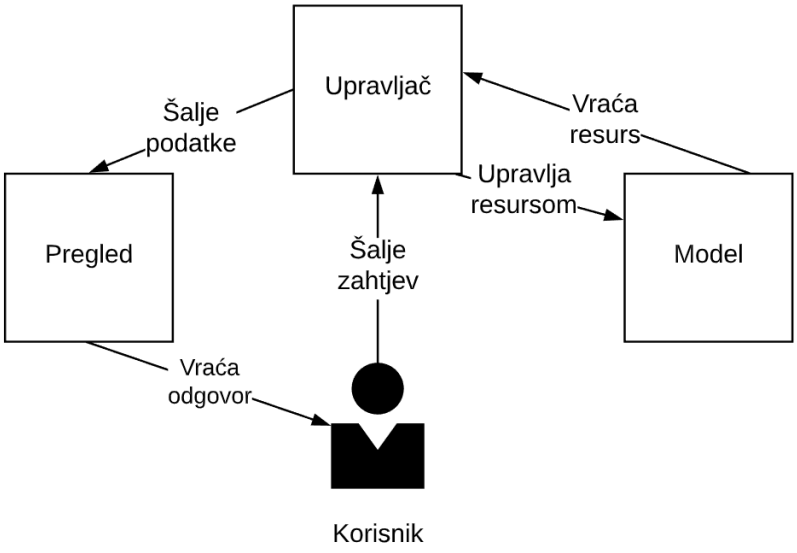
3.3.1 MVC

Laravel koristi Model-Pregled-Upravljač (engl. Model-View-Controller) arhitekturu izrade Web stranica. MVC označava da se aplikacija odvija na tri osnovna sloja. Slojevi su međusobno povezani i komuniciraju. (Stauffer, 2017)

Model je sloj koji je povezan sa bazom podataka, on dohvaća i sprema podatke. Model je jedini sloj koji nije povezan sa samim korisnikom. Korisnik sa modelom komunicira pomoću upravljača.

Korisnik šalje zahtjev upravljaču, upravljač komunicira sa modelom kako bi izvršio željeni zadatak. Model pruži podatke upravljaču, upravljač obradi podatke te obrađene podatke prenese pregledu.

Podaci se prezentiraju korisniku pomoću pregleda. Kod izrade normalne Web stranice prikaz je programiran u HTML-u, no u slučaju REST API-a prikaz nije HTML stranica već JSON format koji se kreira u upravljaču.



Slika 2. MVC arhitektura

Ovakva podjela na slojeve ima nekoliko prednosti. Kada jedna osoba radi front-end, a druga back-end, lako je odrediti tko će što raditi jer je razlika između slojeva jasna. Čak i osobe koje rade na istom sloju mogu raditi na različitim pregledima ili upravljačima. Metode se mogu grupirati u jedan upravljač ili rasporediti kroz više upravljača. Takav pristup olakšava snalaženje kod razvoja i ponovnog korištenja istog koda.

Mana ovakvog pristupa je ako jedna osoba mora raditi na sva tri sloja Web aplikacije. Tada je potrebno opširnije znanje programiranja nego kada se osoba može koncentrirati na samo jedan sloj. Navigacija kroz slojeve nije uvijek očita. Kod traženja pogrešaka greška može biti na bilo kojem od tri sloja, što je puno veća količina koda koju je potrebno provjeriti kako bi se pronašao i riješio problem. Promjene na jednom sloju mogu zahtijevati promjene i na druga dva sloja. Kada su metode raspoređene kroz previše upravljača teško je snalaženje i pronalaženje prave metode, dok u suprotnom, ako se metode rade u premalom broju upravljača biti će nagurane i čitanje koda će biti otežano.

Iako ne postoji jedinstveni način na koji bi programer trebao rasporediti kod, potrebno je iskustvo kako bi programer sam shvatio sistem koji njemu najviše odgovara. Nakon što osoba stekne iskustvo prednosti postaju vidljivije, a mene zanemarive.

3.3.2 STRUKTURA DIREKTORIJA

Nakon što je Laravel instaliran naredba `composer create-project laravel/laravel projectName --prefer-dist` kreira novi projekt sa datotekama i mapama za uobičajenu Laravel aplikaciju. Popis mapa i datoteka koje se nalaze u root (hrv. korijen) mapi projekta:

- *app* mapa → Osnovni dijelovi aplikacije: upravljači, modeli, naredbe i PHP kod.
- *boostrap* mapa → Datoteke koje Laravel koristi za pokretanje aplikacije.
- *config* mapa → Konfiguracija aplikacije.
- *databalse* mapa → Migracije i sjeme (engl. seeds) za bazu podataka.
- *public* mapa → Datoteke na koje poslužitelj usmjerava. Ova mapa sadržava *index.php* datoteku. Index datoteka služi kao upravljač koji preusmjerava na boostarap mapu i zadane usmjerivače. Mapa također sadržava slike, stilove i mapu sa skinutim sadržajem.

- *resources* mapa → Datoteke koje nisu PHP, poput pregleda, datoteka jezika, Sass/LESS datoteka i JavaScript datoteka.
- *routes* mapa → Datoteke sa definicijama usmjerivača i Artisan naredbi.
- *storage* mapa → Cash datoteke, log-ovi i kompilirane sistemske datoteke.
- *test* mapa → Unit i integracijski testovi.
- *vendor* mapa → Datoteke ovisnosti (engl. dependency). Git ignorira ovu mapu. Sve ovisnosti se na nekom drugom računalu ponovo skidaju kada se sa Composer-om pokrene aplikacija.
- *.env* datoteke → Varijable okoline (engl. environment), tj. varijable za koje se očekuje da budu drugačije ovisno o tome na kojem uređeju se pokreće aplikacija. Ove datoteke su ignorirane od srane Git-a.
- *artisan* datoteka → Omogućava korištenje Artisan naredbi u komadnoj liniji.
- *.gitignore* je datoteka → Popis datoteka i mapa koje se ne učitavaju na Git.
- composer.json i composer.lock datoteke → Konfiguracijske datoteke Composer-a, sadržavaju osnovne informacije o projektu i PHP ovisnosti.
- *gulpfile.js* datoteka → Opcionalna konfiguracija za Elixir i Gulp.
- *package.json* datoteka → Konfiguracijska datoteka za front-end.
- *phpunit.xml* datoteka → Konfiguracijska datoteka za PHPUnit, koji služi za testiranje.
- *readme.md* datoteka → Markdown datoteka koja daje uvod u Laravel.
- *sever.php* datoteka → Rezervni (engl. backup) server koji pokušava dopustiti manje sposobnim serverima da pregledaju Laravel aplikaciju.

3.3.3 USMJERIVAČI

Laravel definira svoje URL-ove sa usmjerivačima (engl. route). Ovaj *middleware* mehanizam u Laravelu filtrira HTTP zahtjeve koji mogu sadržavati varijabilne podatke. Zahtjeve preusmjerava na zadani upravljač koji obavi svoj dio zadatka u MVC arhitekturi. Usmjerivač se piše tako da se prvo pozove fasada „Route“, zatim HTTP glagol. Osnovni primjer je „get“ (engl. dohvati), u nastavku kao prvi argument se piše željeni izraz za usmjerivač, a kao drugi argument metoda koja vraća odgovor. Najjednostavnija Web stranica može biti napravljena u datoteci sa usmjerivačima (Primjer 1.).

```
//routes/web.php

<?php
Route::get('hello', function() {
      return 'Hello World';
});
```

Primjer 1. Jednostavna Web stranica napisana metodom u usmjerivaču

Metoda se pokreće kod slanja zahtjeva na Web stranicu sa URL-om Web adrese na koju je nadodan usmjerivač */hello*. Lokalna aplikacija koja se izvršava na portu 8000 prima zahtjev na *http:///localhost:8000/hello*. Korisnik u svom pregledniku vidi ispisanu rečenicu „Hello World“.

Usmjerivač se nalazi u datoteci web.php koja se koristi kod razvoja Web stranice. Aplikacija koje je razvijena u svrhu ovog rada je API aplikacija. Za API aplikaciju usmjerivači su napisani u datoteci api.php. Iako funkcionalno iste, api.php koristi drugi *middleware* i u URL stranice nakon njene Web adrese potrebno je nadodati *'/api/'* te u nastavku usmjerivač koji se poziva. Bilo bi poželjno da metoda nije napisana u datoteci sa usmjerivačima, već u za to predviđenim upravljačima. Primjer 2. i Primjer 3. prikazuju kako kreirati istu Web stranicu kao i Primjer 1 ali sa metodom napisanom u datoteci upravljača:

```
//routes/web.php

<?php
Route::get('hello', HelloWorldController@getHelloWorld);
```

Primjer 2. Usmjerivač sa pozivom upravljača

```
//app/Http/Controllers/HelloWorldController.php

<?php
namespace app\Http\Controllers;
class HelloWorldController {
      public function getHelloWorld () {
            return 'Hello World';
      }
}
```

Primjer 3. Jednostavna Web stranica napisana metodom u upravljaču

Kod kreiranja metoda u upravljaču usmjerivač za drugi argument nema napisanu metodu već naziv upravljača prije znaka @ i naziv metode upravljača nakon znaka @. Okruženje na taj način zna u kojem upravljaču tražiti metodu i koju metodu izvršiti. U ovom primjeru okruženje kao odgovor korisniku vrati Web stranicu na kojoj je ispisana rečenica „Hello World“ isto kao i u prošlom primjeru. Već na ovako jednostavnom primjeru vidljivo je kako je Laravel jednostavan za stvaranje Web stranica. Po potrebi moguće je korak po korak ulaziti u dubinu okruženja i koristiti njegove mogućnosti kako bi se razvila funkcionalna Web stranica.

Parametri su dodatna mogućnost koju usmjerivači pružaju. Parametri se pišu kao riječi u prvom argumentu upravljača unutar vitičastih zagrada „{ }“ razdvojeni kosim crtama „/“. U argumentu metode je parametar PHP varijabla. Konvencionalno je da te riječi budu iste, ali nije obavezno. Primjer 4. prikazuje korištenje parametara u usmjerivaču.

```
//web.php

Route::get(hello/{name}, function ($name) {
      return 'Hello '.$name;
});
```

Primjer 4. Korištenje parametra u usmjerivaču

Pri pozivu adrese *http:///localhost:8000/hello/Ivan* korisniku se prikaže stranica na kojoj će biti ispisano „Hello Ivan“. U slučaju da usmjerivač ima više od jednog parametra bitno je paziti da argumenti budu ispisani istim redoslijedom sa lijeva na desno. Primjer 5. prikazuje korištenja više parametara.

```
//web.php

Route::get(hello/{first}/{second}, function ($surname, $name) {
      return 'Hello '.$name.' '.$surname;
});
```

Primjer 5. Korištenje više parametara u usmjerivaču

Pri pozivu na adresu *http:///localhost:8000/hello/Horvat/Ivan* korisniku se prikaže stranica na kojoj je ispisano „Hello Ivan Horvat“.

Moguće je napraviti i usmjerivače koji imaju neobavezne parametre. U tom slučaju korisnik ima slobodu odabrati želi li poslati podatak za taj parametar. Takvi parametri se pišu sa upitnikom „?“ prije zatvaranja vitičaste zagrade te se mora definirati zadana vrijednost ako korisnik ne unese vlastiti podatak. Primjer 6. prikazje korištanje neobaveznih parametara.

```
//web.php

Route::get(hello/{name?}, function ($name = 'World') {
      return 'Hello '.$name;
});
```

Primjer 6. Korištenje ne obaveznih parametara

Kada korisnik pošalje zahtjev na adresu *http://localhost:8000/hello/Ivan* prikaže mu se *„Hello Ivan“.* Ako korisnik odluči izostaviti parametar i pozove samo *http://localhost:8000/hello* prikaže mu se „Hello World“ jer je 'World' bio zadana vrijednost ako korisnik ne unese vlastitu. Moguće je da usmjerivač ima više parametara koji su neobavezni. Korisnik ne može koristiti parametre desno od parametra kojeg odluči izostaviti.

Programer može ograničiti korisnika na ono što smije unijeti pomoću regex-a tako da nakon usmjerivača napiše ulančanu metodu *where()* sa nazivom varijable i ograničenjima. Primjer 7. prikazuje korištenje regex-a:

```
//web.php

Route::get(game/{rating}', function ($rating) {
      return 'Game rating: '.$rating;
})->where('rating', '[0-9]+');
```

Primjer 7. Korištenje regex-a za unos broja putem parametra u usmjerivaču

U ovom primjeru korisnik kao argument mora upisati izraz koji se sastoji samo od brojeva između 0 i 9. Zahtjevi koji sadržavaju neki drugi znak izvan tog okvira ne bi vodili na ovaj usmjerivač. Da je na poveznicu umjesto niza brojeva od 0 do 9 napisana riječ sa bilo kojim malim ili velikim slovom vodila bi na usmjerivač iz Primjer 8.

```
//web.php

Route::get(game/{name}', function ($name) {
      return 'Game name: '.$name;
})->where('name', '[A-Za-z]+');
```

Primjer 8. Korištenje regex-a za unos riječi putem parametra u usmjerivaču

Usmjerivačima se mogu davati imena. Ta imena se mogu koristeiti kod referenciranja u kodu da se ne bi morala upotrebljavati puna poveznica. Nakon usmjerivača u ulančanu metodu *name()* kao argument se upiše željeno ime kao što je prikazano u Primjer 9.

```
//web.php

Route::get(game/{name}/{rating}', function ($rating, $ratin) {
      return $name.' has rating of '.$rating$;
})->name(games.rating);
```

Primjer 9. Usmjerivač sa imenom

Usmjerivač koji ima ime referencira se metodom *route()* u koju se za prvi argument upiše ime usmjerivača i za drugi argument lista podataka koji se upotrebljavaju za parametare usmjerivača. Primjer 10 prikazuje takav usmjerivač.

```
//resources/views/user.blade.php

<a href=
    "<?php echo route('games.rating', ['name' => 'Catan', 'rating' => '7');
?>"
```

Primjer 10. Poziv usmjerivača u pregledu

Ime usmjerivača može biti bilo koji izraz ali konvencionalno je da se napiše ime resursa u množini, zatim točka „.“ i na kraju naziv akcije. Primjeri imena usmjerivača za CRUD igre (engl. game):

- games.index → dohvaća popis svih igara
- games.create → kreira novu igru

- games.read → dohvaća informacije o igri
- games.update → izmjenjuje podatke o igri
- games.delete → briše igru

Laravel ima mogučnost grupiranja usmjerivača. Grupe se mogu koristiti kako bi se za određenu količinu usmjerivača koristio isti prefiks. Primjer 11. prikazuje grupu za usmjerivače igrara.

```
//web.php

Route::group(['prefix' => 'games'], function () {
      Route::get('all', function () {
            ...
      });
      Route::get('{id}', function ($id) {
          ...
      });
});
```

Primjer 11. Grupiranje usmjerivača po prefiksu

Grupe se mogu koristiti za pod domene. To se najčešće koristi kada se dio aplikacije želi prikazati kao zasebna aplikacija. Prednost ovakvog načina grupiranja usmjerivača je što se za pod domene mogu pisati parametri. Primjer 12. prikazuje grupiranje pod domene sa parametrom.

```
//web.php

Route::group(['domain' => '{company}.app.com], function () {
      Route::get('/', function ($company) {
            ...
      });
      Route::get('games/{id}', function ($company, $id) {
            ...
      });
});
```

Primjer 12. Grupiranje usmjerivača po pod domeni

Programer može kreirati grupu i nad njom primijeniti određeni *middleware*. Postoji više različitih *middleware*-a. Jedan od najčešćih je „auth“ *middleware* koji se koristi kod autorizacije prijavljenog korisnika i davanja dopuštenja pristupu usmjerivačima koji su

pod tim *middleware*-om. Kod korištenja REST API aplikacija koristi se „api“ *middleware*. API *middleware* se piše unutar *routes/api.php* datoteke. Primjer 13. prikazuje api middleware.

```
//routes/api.php

Route::group(['middleware' => ['api']], function () {
    Route::get('/', 'GameController@index);
    Route::post('games', 'GameController@store);
    Route::get('games/{id}', 'GameController@show);
    Route::put('games/{id}', 'GameController@update');
    Route::delete('games'/{id}', 'GameController@destroy);
});
```

Primjer 13. Grupiranje usmjerivača po middleware-u

3.3.4 PREGLEDI

Pregledi predstavljaju izgled Web stranice. Kako se ovaj rad bazira na REST API-u stranice neće biti prikazane sa HTML-om već sa JSON-om. Uz JSON i HTML stranica može biti prkazana i u XML-u. HTML pregled se poziva metodom *view()* unutar metode usmjerivača i nazivom pregleda kao argumentom te metode. Primjer 14. sadrži usmjerivač koji poziva pregled koji se nalazi u datoteci *resources/views/games/details.blade.php*.

```
//web.php

Route::get('users, function () {
      return view('games.details');
});
```

Primjer 14. Poziv pregleda u usmjerivaču

Kako su pregledi zasebna grana MVC arhitekture koja se ne koristi u razvoju API servisa ta tema neće biti dublje razrađena u ovom radu.

3.3.5 UPRAVLJAČI

Upravljači organiziraju i sadrže logiku aplikacije. Sve metode mogu biti napisane unutar jednog upravljača. Konvencija je da usmjerivači pozivaju metode iz različitih upravljača koji su podijeljeni po resursima. Upravljači najčešće sadržavaju CRUD metode ali i neke druge. Upravljač je moguće stvoriti pomoću Artisan naredbe. Naredba `php artisan make:controller GameController` stvara upravljač za resurs igre u za to predviđenoj mapi. Konvencija imenovanja upravljača je da su prve riječi naziv resursa, tj. imenice pisane u množini, a posljednja riječ je „Controller“. Naziv upravljača se piše PascalCase-om, tj. prvo slovo svake riječi treba biti napisano velikim slovom, ostatak slova u riječi malim slovom. Upravljači se nalaze na lokaciji *App/Http/Controllers/*. Laravel automatski ispuni datoteku sa klasom (Primjer 15.).

```
//http/controllers/GamesController.php

<?php
namespace App\Http\Controllers;
use Illuminate\Http\Request;
use App\Http\Requests;
class GamesController extends Controller{
      //
}
```

Primjer 15. Upravljač stvoren Artisan naredbom

Artisan naredba `php artisan make:controller GamesController --resource` stvara upravljač ispunjen sa CRUD metodama (Primjer 16.).

```
//GamesController.php

<?php

namespace App\Http\Controllers;

use Illuminate\Http\Request;

class GamesController extends Controller{
    public function index(){
        //
    }

    public function create(){
        //
    }
```

```
    public function store(Request $request){
        //
    }

    public function show($id){
        //
    }

    public function edit($id){
        //
    }

    public function update(Request $request, $id){
        //
    }

    public function destroy($id){
        //
    }
}
```

Primjer 16. Upravljač i CRUD metode stvorene Artisan naredbom

Metode koje se generiraju u upravljaču su:

- *index* → Prikaz svih elemenata resursa.
- *create* → Prikaz forme za stvaranje novog elementa resursa.
- *store* → Stvara novi element resursa. Kao argument prima objekt sa podacima.
- *show* → Prikaz elementa resursa. Kao argument prima id elementa.
- *edit* → Prikaz forme za izmjenu podataka elementa resursa. Kao argument prima id resursa kako bi se mogli prikazati stari podaci.
- *update* → Ažurira podatke elementa resursa. Kao argumente prima nove podatke i id elementa resursa.
- *destroy* → Briše element resursa. Kao argument prima id elementa.

Kod referenciranja upravljača u usmjerivaču nije potrebno napisati njegovu lokaciju jer Laravel pretpostavlja da smo upravljač stavili u za to predviđenu mapu. Ako nismo upravljač stavili u za to predviđenu mapu tada u usmjerivaču moramo napisati novu lokaciju. Primjer 17. prikazuje poziv upravljača koji se nalazi unutar druge mape.

```
//web.php

Route::get('/', 'Auth\AuthController@login);
```

Primjer 17. Poziv upravljača koji se nalazi na ugnježdenoj lokaciji

Kod izrade Web stranica upravljači najčešće služe za poziv pregleda i prosljeđivanje podataka koje su dobili od modela. Primjer 18. prikazuje takav poziv.

```
//GamesController.php

public function index(){
      return view(games.index')->with('games', Games::all());
}
```

Primjer 18. Poziv pregleda i prosljeđivanje podataka iz upravljača

Primjer 18. sadrži metodu koja poziva pregled pod nazivom index koji se nalazi u mapi „games“ na lokaciji pregleda. Ulančana metoda *with()* u prvom argumentu prima naziv varijable, u drugom argumentu vrijednost. U primjeru varijabla *games* sadržava podatke svih igara. Ti podaci su proslijeđeni na pregled i tamo se podešava način na koji će se oni prikazati. Podaci su u upravljaču dohvaćeni pomoću Eloquent fasade „Games“.

3.4 UNOS PODATAKA

Korisnik često želi unijeti podatke kako bi se izvršila željena akcija. Dio podataka moguće je unijeti u usmjerivaču pomoću parametara. Kod unosa veće količine podataka nepraktično je podatke upisivati kao parametre. Kako bi se podaci unijeli na drugi način potrebno je koristiti HTTP glagole (engl. verb), tj. akcije. Do sada su svi usmjerivači bili pozvani GET akcijom. Da bi korisnik upravljač mogao koristiti sa drugačijim načinom unosa podataka mora koristiti POST akciju. Primjer 19. prikazuje pozive različitih HTTP akcija.

```
//web.php

Route::get('games/create', 'GamesController@create');

Route::post('games/store, 'GamesController@store');
```

Primjer 19. Korištenje HTTP akcija

Usmjerivač *games/crate* sa HTTP akcijom GET služi za dohvaćanje forme za kreiranje nove igre. Kada korisnik ispuni formu i želi ju unijeti kao novi element šalje zahtjev na usmjerivač *games/store* sa HTTP POST akcijom. Aplikacija na taj način zna koje usmjerivače pozvati. Da oba usmjerivača imaju isti naziv Laravel bi pomoću HTTP akcije znao koji usmjerivač pozvati. Kada se tako pozove usmjerivač podaci se mogu unijeti pomoću Eloquent fasade *„Input“* (Primjer 20.) ili pomoću objekta „*Request“* (Primjer 21.).

```
//GamesController.php

use Illuminate\Support\facades\Input;
...
public function store(){
      $game = new Game;
      $game->name = Input::get('min_players');
      $game->surame = Input::get('max_players');
      $game->save();
      return redirect('games');
}
```

Primjer 20. Unos podataka Input fasadom

Varijabla *$game* se popunjava sa novim praznim modelom igre. Zatim se varijabla ispuni sa podacima unesenim pomoću fasade. Varijabla se spremi kao element tog resursa, a korisnik se preusmjeri na index stranicu svih igara.

Ako se ne želi koristiti fasada ili se jednostavno preferira raditi sa objektom *Request* moguće je unijeti podatke i na taj način. *Request* koristi podatke koji se nalaze u Laravelovom kontejneru (engl. container). Kontejner je mjesto gdje se izvršavaju svi Larevelovi usmjerivači. *Request* ima pristup svim podacima koji se pri izvršavanju nalaze u kontejneru pa tako ima pristup i podacima koje korisnik unio.

```
//GamesController.php

use Illuminate\Http\Request;
...
public function store(Request $request){
      $game = new Game;
      $game->name = $request->input('name');
      $game->min_players = $request->input('min_players');
      $game->max_players = $request->input('max_players');
      $game->save();
      return redirect('games');
}
```

Primjer 21. Unos podataka Request objektom

Kada se koristi *Request* za unos podataka potrebno je deklarirati varijablu *request* tipa *Request* kao prvi argument metode. Podaci se spremaju u varijablu koja služi za spremanje novih podataka u bazu. Kod izrada Web stranica ova dva načina nisu toliko različita. Korištenje jednog ili drugog često ovisi o preferenciji programera. Kako je ovaj rad usmjeren na izradu API servisa koristi se *Request* objekt jer nije moguće koristiti *Input* fasadu.

3.5 PREUSMJERAVANJA, ZAUSTAVLJANJA I PRILAGOĐENI ODGOVORI

3.5.1 PREUSMJERAVANJA

Metode mogu vraćati i neke druge strukture osim pregleda. Do sada je prikazano vraćanje pomoću *redirect()* (hrv. preusmjeri) metoda. Postoje dva osnovna načina na koje se može izvršiti preusmjeravanje. Jedan način je globalni pomoćnik za preusmjeravanje (engl. Redirect global helper), drugi je pomoću fasada. Pomoćnik za preusmjeravanje koristi dva načina. Primjer 22. prikazuje preusmeravanje na pregled zadan kao argument metode *redirect()*.

```
//web.php

Route::get('redirect', function () {
      return redirect('games');
});
```

Primjer 22. Preusmjeravanje sa argumentom metode

Primjer 23. prikazuje preusmjeravanje ulančanom metoda *to()* u koju se upiše naziv pregleda kao argument.

```
//web.php

Route::get('redirect', function () {
      return redirect()->to('games');
});
```

Primjer 23. Preusmjeravanje sa ulančanom metodom to()

Fasada radi se na sličan način samo što se ne korite ulančane metode *redirect()->to()* već fasada *Redirect::to()* i naziv u argumentu (Primjer 24.).

```
//web.php

Route::get('redirect', function () {
      return Redirect::to('games');
});
```

Primjer 24. Preusmjeravanje korištenjem fasade

Primjer 25. prikazuje metodu *to()* i sve njene argumente.

```
//web.php

function to($to = null, $status = 302, $headers = [], $secure = null)
```

Primjer 25. Metoda to() sa argumentima

Opisi argumenata *to()* metode: (Stauffer, 2017)

1. $to → Lokacija datoteke u projektu.
2. $status → HTTP status, ako nije ispunjen zadana vrijednost će biti 302 FOUND.
3. $header → Zaglavlje koje će biti poslano.
4. $secure → Može zamijeniti zadani status na *http* ili *https*, zadani status je jednak trenutnom, te je ovaj argument potreban samo ako se to želi promijeniti.

Preusmjeravanje je moguće na druge usmjerivače i izvršava se pomoću ulančane metode *route()* gdje se kao argument piše ime željenog usmjerivača. Primjer 26. prikazuje preusmjerenje na usmjerivač.

```
//web.php

Route::get('redirect', function () {
      return redirect()->route('games.index');
});
```

Primjer 26. Preusmjeravanje na usmjerivač

Primjer 27. prikazuje metodu *route()* i sve njene argumente:

```
//web.php

function route($to = null, $parameters = [], $status = 302, $headers = [])
```

Primjer 27. Metoda route() sa argumentima

Opisi argumenata *route()* metode:

1. $to → Lokacija datoteke u projektu.
2. $parameters → Lista parametara koje usmjerivač prima kao argumente.
3. $status → HTTP status, ako nije ispunjen zadana vrijednost će biti 302 FOUND.
4. $header → Zaglavlje koje će biti poslano.

Preusmjeravanje ima još nekoliko ulančanih metoda:

- *back()* → Način na koji Laravel koristi sesije omogućava da korisnik ide na prethodnu stranicu, može se koristiti skraćeno *back()*.
- *home()* → Povratak na početnu stranicu, tj. stranicu sa imenom „home“, ekvivalentno je *redirect-to('home')*.
- *refresh()* → Preusmjeri na stranicu na kojoj se korisnik trenutno nalazi.
- *away()* → Preusmjeri na URL izvan aplikacije, bez validacije.
- *secure()* → Preusmjeri kao i metoda *to()* sa zadanim parametrom za *$secure* postavljeno na „true“.

- *action()* → Omogući preusmjeravanje na drugu metodo na isti način kao što radi i kod usmjerivača gdje se za argument upisuje naziv upravljača i naziv metoda razdvojeni znakom @.
- *guest()* → Koristi se kada neautorizirani korisnik pokuša pristupiti određenim stranicama kojima ne bi trebao imati pristup te se preusmjeri na neku drugu stranicu, najčešće stranicu za prijavu.
- *intended()* → Izvrši se nakon što se neprijavljeni korisnik iz *guest()* metode prijavi u sustav te preusmjeri na stranicu kojoj je ranije pokušao pristupiti.

Često kada se korisnika preusmjeri na neku stranicu potrebno je proslijediti i neke podatke. Primjer 28. prikazuje preusmjeravanje koristeći ulančanu metodu *with()* i argument koji je napisan u obliku niza ključeva i podataka.

```
//web.php
Route::get('redirect', function () {
      return redirect('games')
      ->with([
            'success'=>true,
            'description'=>'Data seved successfully'
      ]);
});
```

Primjer 28. Preusmjeravanje sa podacima

Preusmjeravanje podataka u Laravelu omogućava i slanje pogrešaka. Primjer 29. prikazuje slanje pogrešaka nastalih kod validacije unesenih podataka.

```
//web.php

Route::post('redirect', function (Request $request) {
      $validator = Validator::make(
            $request->all()),
            $this->validationRules
      );
      if ($validator->fails()) {
          return redirect('redirect')
          ->withErrors($validator)
          ->withInput();
      }
});
```

Primjer 29. Preusmjeravanje pri pogrešci kod validacije

3.5.2 ZAUSTAVLJANJE ZAHTJEVA

Metodu je moguće zaustaviti metodama:

- *abort()* → Zaustavlja metodu. Neobavezni argumenti su HTTP status, poruka i zaglavlje.
- *abort_if()* → Zaustavlja metodu ako je ispunjen uvjet. Argument je uvjet. Neobavezni argumenti su HTTP status, poruka i zaglavlje.
- *abort_unless()* → Zaustavlja metodu ako nije ispunjen uvjet. Argument je uvjet. Neobavezni argumenti su HTTP status, poruka i zaglavlje.

3.5.3 PRILAGOĐENI ODGOVORI

Odgovore je moguće prilagoditi po potrebi. Postoji opcija korištenja „response()“ pomoćnika ili „Response“ fasade. U svrhu svih primjera u ovom radu koristi se pomoćnik. Popis ulančanih metoda za pomoćnik:

- *make()* → Kreiranje vlastitih odgovora. Prvi argument je poruka koja se želi poslati, neobavezni drugi argument je HTTP status kod i treći je zaglavlje.
- *download()* → Omogućava korisniku da preuzme željenu datoteku (pdf, slika ili slično). Prvi argument je naziv datoteke koja se nalazi u aplikaciji, drugi argument je naziv pod kojim će se ta datoteka preuzeti kod korisnika.
- *file()* → Prikaz datoteke (pdf, slika ili slično) koja se nalazi u aplikaciji.
- *json()* → Odgovor u JSON formatu. Metoda kao argument prima listu, kolekciju, objekt ili bilo što drugo. Laravel neovisno o tipu argumenta koji je dat metodi to pretvori u JSON format. Ovo je najbitnija metoda za kreiranje odgovora u REST API aplikaciji.

3.6 RAD S PODACIMA

Laravelu korisnik može proslijediti podatke pomoću GET akcije sa parametrima i URL segmentima ili POST akcije slanjem podataka u JSON formatu, request pomoćnika i Request fasade. (Stauffer, 2017)

Popis ulančanih metoda koje se mogu koristiti nad varijablom *$request*, tipa Request, koja sadržava podatke:

- *all()* → Dohvaća niz svih ulaznih podataka neovisno o tome na koji način su proslijeđeni. Primjer 30. prikazuje dohvaćanje i odgovor. Jedan podatak je poslan kao segment URL-a, drugi je poslan u tijelu (engl. body) POST metode.

```
URL: localhost:8000/request?first=one
Body: second:two
```

```
public function request(Request $request){
      return response()->json($request->all());
}
```

```
Odgovor:
{
 "first": "one",
 "second": "two"
}
```

Primjer 30. Ulančana metoda all()

- *except()* → Dohvaća niz svih ulaznih podataka osim onih polja iznimka (engl. exception) koja su zadana kao argumenti metode. Primjer 31. prikazuje dohvaćanje i vraćanje istih podataka kao i prošli primjer sa iznimkom zadanog polja „first“.

```
URL: localhost:8000/request?first=one
Body: second:two
```

```
public function request(Request $request){
      return response()->json($request->except('first'));
}
```

```
Odgovor:
{
 "second": "two"
}
```

Primjer 31. Ulančana metoda except()

- *only()* → Dohvaća samo (engl. only) niz ulaznih polja koja su zadana kao argumenti metode. Primjer 32. prikazuje dohvaćanje i vraćanje istih podataka kao i prošli primjer, no ovaj put odabire samo zadano polje „first“:

```
URL: localhost:8000/request?first=one
Body: second:two
```

```
public function request(Request $request){
      return response()->json($request->only('first'));
}
```

```
Odgovor:
{
 "first": "one"
}
```

Primjer 32. Ulančana metoda only()

- *exists()* ili *has()* → Dohvaća niz svih ulaznih podataka i provjerava postoje (engl. exists) li polja koja su zadana kao argument metode. Primjer 33. prikazuje dohvaćanje istih podataka kao i prošli primjeri i provjerava postoje li među njima polja „first“, „second“ i „third“. Ukoliko postoje vraća „true“, inače vraća „false“:

```
URL: localhost:8000/request?first=one
Body: second:two
```

```
public function request(Request $request)
    {
        return response()->json([
            'first-exists' => $request->exists('first'),
            'second-exists' => $request->exists('second'),
            'third-exists' => $request->exists('third')
        ]);
    }
```

```
Odgovor:
{
 "exists-first": true,
 "exists-second": true,
 "exists-third": false
}
```

Primjer 33. Ulančana metoda exists()

- *input()* → Slično kao metoda *only()* čita samo polje koje je u argumentu, razlika je što *only()* može čitati niz polja dok *input()* čita samo jedno polje. Kao drugi argument ima zadanu vrijednost ako to polje ne postoji. Primjer 34. prikazuje dohvaćanje istih podataka kao prošli primjeri i ispisuje vrijednosti polja „first“, „second“ i „third“, te ako koje polje ne postoji ispisuje zadanu vrijednost „default value“:

```
URL: localhost:8000/request?first=one
Body: second:two
```

```
public function request(Request $request)
    {
        return response()->json([
            'first' => $request->input('first', "default value"),
            'second' => $request->input('second', "default value"),
            'third' => $request->input('third', "default value")
        ]);
    }
```

```
Odgovor:
{
 "first": "one",
 "second": "two",
 "third": "default value"
}
```

Primjer 34. Ulančana metoda input()

Podaci se mogu slati i kao dio višedimenzionalnog niza podataka. Podatke se tada čita pomoću naziva polja i točkom odvojenih naziva pod polja. Primjer 35. prikazuje čitanje višedimenzionalnog niza podataka.

```
URL: localhost:8000/request?first=one
Body:
second[0][word]:two [0]
second[0][number]:2 [0]
second[1][word]:two [1]
second[1][number]:2 [1]
```

```
public function request(Request $request)
    {
        return response()->json([
            'second' => $request->only('second'),
            'second.0.number' => $request->only('second.0.number'),
            'second.*.word' => $request->only('second.*.word'),
        ]);
    }
```

```
Odgovor:
{
 "second": [
     {
         "word": "two [0],
         "number": "2 [0]"
     },
     {
         "word": "two [1],
         "number": "2 [1]"
     }
 ],
 "second.0.number": "2 [0]",
 "second.*.word": [
     "two [0]",
     "two [1]"
 ],
}
```

Primjer 35. Čitanje višedimenzionalnog niza podataka

Podaci se mogu slati i primati u JSON formatu. Slijedeći primjer prikazuje rad sa istim setom podataka kao iz prošlog primjera, no ovaj put su proslijeđeni u JSON formatu. Iako je unos podataka drugačiji rad s podacima se nije promijenio jer Laravel pretpostavlja što programer pokušava napraviti. Primjer 36. prikazuje zahtjev napisan u JSON formatu.

```
URL: localhost:8000/request?first=one
Body:
{
"second": [
      {
            "word": "two [0]",
            "number": "2 [0]"
        },
        {
            "word": "two [1]",
            "number": "2 [1]"
        }
    ]
}
```

```
public function request(Request $request)
    {
        return response()->json([
            'second' => $request->only('second'),
            'second.0.number' => $request->only('second.0.number'),
            'second.*.word' => $request->only('second.*.word'),
        ]);
    }
```

```
Odgovor:
{
 "second": [
     {
         "word": "two [0],
         "number": "2 [0]"
     },
     {
         "word": "two [1],
         "number": "2 [1]"
     }
 ],
 "second.0.number": "2 [0]",
 "second.*.word": [
     "two [0]",
     "two [1]"
 ],
}
```

Primjer 36. Zahtjev napisan u JSON formatu

3.7 VALIDACIJA

3.7.1 ZAHTJEVI VALIDACIJE

Validacija podataka se može napisati ručno i metodom *validate().* Neki od mogućih zahtjeva validacije su: (Rees, 2016)

- *boolean, date, integer, string…* → Je li podatak zadanog tipa.
- *accepted* → Ima li polje boolean vrijednost „true“. Najčešće se koristi kod prihvaćanja uvjeta pružanja usluge (engl. Terms of service).
- *after:{datum}* ili *before:{datum}* → Ima li polje datumsku vrijednost prije, tj. nakon, zadane.
- *date_equals:{date}* → Odgovara li datum zadanom.
- *alphabetic* → Sastoji li se vrijednost polja samo od znakova abecede.
- *alpha_num* → Sastoji li se vrijednost polja samo od znakova abecede i brojeva.

- *alpha_dash* → Sastoji li se vrijednost polja samo od znakova abecede, brojeva, crtica i povlaka.
- *array* → Je li vrijednost niz.
- *between:{min},{max}* → Je li vrijednost između min i max vrijednosti.
- *confirmed* → Postoji li polje sa istim nazivom kao i zadano samo sa nadodanim „__confirmation“, ako postoji, uspoređuje vrijednosti. Najčešće se koristi kod potvrde lozinke.
- *digits:{broj}* → Ima li vrijednost zadani broj znamenki.
- *digits_between:{min},{max}* → Ima li vrijednost više znamenki od min i manje od max.
- *distinct* → Nema li duplih vrijednosti u nizu vrijednosti.
- *email* → Je li vrijednost napisana u formatu email adrese.
- *exists:{tablica},{stupac}* → Postoji li vrijednost u zadanoj tablici i stupcu.
- *file* → Je li vrijednost datoteka.
- *filled* → Je li polje ispunjeno.
- *image* → Je li slika.
- *ip, ipv4* ili *ipv6* → Je li vrijednost u zadanom formatu IP adrese.
- *json* → Je li u JSON formatu.
- *min:{min}* ili *max:{max}* → Vrijednost mora biti veća ili jednaka od min, tj. manja ili jednaka od max.
- *nullable* → Dopušta polju da bude null.
- *numeric* → Je li vrijednost broj.
- *present* → Postoji li polje, vrijednost ne mora biti ispunjena.
- *regex:{patern}* → Je li vrijednost u formatu zadanom regex-om.
- *not_regex:{patern}* → Nije li vrijednost u formatu zadanom regex-om.
- *required* → Vrijednost je obavezna.
- *same:{polje}* → Odgovara li vrijednost polja vrijednosti zadanog polja.
- *unique:{tablica},{stupac}* → Je li vrijednost jedinstvena u zadanom stupcu tablice.

3.7.2 METODA VALIDATE

Metoda *validate()* radi na način da joj se kao prvi argument proslijedi objekt i kao drugi argument zahtjevi koje objekt mora zadovoljiti. Zahtjevi se pišu unutar string-a odvojeni ravnom crtom „|“. Ako objekt zadovolji sve zahtjeve metoda normalno nastavlja sa radom. Ako objekt ne zadovolji sve zahtjeve baca se *ValidationException*. Primjer 37. prikazuje validaciju metodom *validate()*.

```
//UserController.php

public function request(Request $request){
      $this->validate($request, [
            'required' => 'required',
            'small' => 'max:5',
            'number' => 'integer',
            'email' => 'email|required',
            'password' => 'confirmed|required'
      ]);
}
```

Primjer 37. Metoda validate()

3.7.3 RUČNA VALIDACIJA

Moguće je kreirati zahtjeve za polja pomoću fasade „Validator“ i metode *make()*. Prvi argument su podaci. Drugi argument je niz zahtjeva unutar string-a odvojeni ravnom crtom „|“. Ukoliko je validacija neuspjela programer sam odlučuje kako će rukovati sa greškama koje postoje. Primjer 38. prikazuje bezuspješnu validaciju sa ispisom grešaka.

```
URL: localhost:8000/request
Body:
 name:
 age:1994
 height:1.80
 email:doroteo.gmail.com
 password:myPassword
```

```
public function request(Request $request){
      $errors = Validator::make($request->all(), [
            'name' => 'required',
            'surname' => 'present',
            'age' => 'integer|max:99',
            'height' => 'integer',
            'email' => 'email|required',
            'password' => 'confirmed|required'
      ])->errors();

      if (count($errors)){
            return response()->json($errors);
      }
      else{
            return response()->json($request);
      }
}
```

```
Odgovor:
{
    "name": [
        "The name field is required."
    ],
    "surname": [
        "The surname field must be present."
    ],
    "age": [
        "The age may not be greater than 99."
    ],
    "height": [
        "The height must be an integer."
    ],
    "email": [
        "The email must be a valid email address."
    ],
    "password": [
        "The password confirmation does not match."
    ],
}
```

Primjer 38. Validacija

3.8 BAZA PODATAKA

3.8.1 „MODEL FIRST“ PRISTUP

Laravel ima nekoliko načina za interakciju sa bazom podataka. Najpoznatije rješenje i ono koje je korišteno za razvoj ovog projekta je Eloquent. Eloquent je ActiveRecord ORM (object-relational mapper). ActiveRecord je knjižnica Ruby okruženja za rad sa SQL bazama podataka. ActiveRecord omogućava da se objekt u aplikaciji može

povezati sa tablicom baze, da se stupcima može pristupiti iz metoda i utjecati na njih, da se može raditi sve CRUD operacije nad bazom podataka i pojednostavljeno je kreiranje SQL upita. Jedan model pripada jednoj tablici baze i zaslužan je za dohvaćanje, prikazivanje i uređivanje podataka u toj tablici. Laravel također ima i mogućnost migracija tablica, tj. moguće je stvoriti ili izmijeniti tablice pomoću modela u koje se upiše struktura tablice. Takav pristup se zove „Model First“ (hrv. prvo model). Korišten za stvaranje baze podataka u projektu koji je napravljen uz ovaj rad. (Rees, 2016)

3.8.2 KREIRANJE I UREĐIVANJE TABLICA

Migracija se kreira tako što se pokrene Artisan naredba u komandnoj liniji. Naziv migracije se sastoji od datuma kreiranja iste i naziva koji je korisnik zadao. Migracije se nalaze na lokaciji *database/mingrations*. Naredbom `php artisan make:migration create_games_table` se kreira datoteka sa nazivom sličnim `2018_08_02_231653_create_games_table.php`. Ovakva migracija sama ne prepoznaje je li za stvaranje ili za izmjenjivanje tablica te je potrebno ručno definirati čemu služi. Druga mogućnost je pri pokretanju naredbe za stvaranje migracije nadodati `--create=naziv_tablice` za stvaranje nove tablice ili `--table=naziv_tablice` za izmjenjivanje postojeće tablice.

Artisan naredbe za migracije:

- `php artisan migrate:install` → Stvaranje tablice migracija u bazi podataka. Naredbu nije potrebno ručno pokretati već se pokreće sa bilo kojom drugom naredbom za migracije ako tablica već ne postoji.
- `php artisan migrate` → Pokretanje svih migracija kronološki po datumu stvaranja. Baza podataka u sebi ima tablicu migracija koja ima zabilježene sve migracije koje su izvršene. Artisan nikad neće pokrenuti već izvršene migracije.
- `php artisan migrate:reset` → Biti će poništene sve migracije koje su izvršene.
- `php artisan migrate:refresh` → Biti će poništene sve migracije koje su izvršene, te će se ponovo pokrenuti sve postojeće migracije.

- `php artisan migrate:rollback` → Poništi samo migracije izvršene sa posljednjim pokretanjem migriranja. Moguće je nadodati `--step=x` na naredbu, gdje je x količina migriranja koja se želi poništiti. Kreće od posljednje izvršene migracije po datumu.
- `php artisan migrate:status` → Ispisuje popis svih migracija i uz svaku oznaku Y (Yes, hrv. da) ako je migracija izvršena ili N (No, hrv. ne) ako nije.

Migracija u sebi sadržava dvije metode, jedna je *up()* metoda koja se izvršava kod pokretanja migracije, druga je *down()* metoda koja se izvršava kod poništavanja migracije. Metode su djelomično ispunjene ako je pri kreaciji nadodano `--create` ili `--table` na naredbu.

Kod kreiranja tablice *down()* metoda u sebi ima samo fasadu koja izbriše tablicu (Primjer 39.). Metodu nije potrebno izmjenjivati.

```
//2018_08_02_231653_create_games_table.php

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

Primjer 39. Metoda za brisanje tablice

Metoda *up()* je kreirana sa Artisan kreiranjem migracije, sa fasadom koja je gotovo prazna i potrebno ju je ispuniti *Blueprint* metodama za kreiranje stupaca (Primjer 40.). Predefinirani stupci su za id i spremanje vremena. U metodu *timestamps()* za spremanje vremena spadaju „created_at“ stupac za spremanje vremena kreiranja podatka i „updated_at“ stupac za spremanje vremena posljednje izmjene podatka.

```
//2018_08_02_231653_create_games_table.php

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

Primjer 40. Metoda za stvaranje tablice

Osnovne Blueprint metode za strukturu stupaca:

- *integer(naziv_stupca)* → Stupac tipa integer.
- *string(naziv_stupca, duljina)* → Stupac tipa *string*, drugi argument nije obavezan.
- *binary(naziv_stupca)* → Stupac tipa *blob*, podaci u binarnom formatu.
- *boolean(naziv_stupca)* → Stupac tipa *boolean*, tj. u MySQL-u tipa *tiniyint(1).*
- *char(naziv_stupca, duljina)* → Stupac tipa *char.*
- *datetime(naziv_stupca)* → Stupac tipa *datetime.*
- *decimal(naziv_stupca, preciznost, skala)* → Stupac tipa *decimal*.
- *double(naziv_stupca, ukupno_znamenki, znamenke_nakon_decimale)* → Stupac tipa *double.*
- *enum(naziv_stupca, [opcije, ..])* → Stupac tipa enum sa mogućim opcijama proslijeđenim u nizu u drugom argumentu.
- *float(naziv_stupca)* → Stupac tipa *float*, tj. u MySQL-u tipa *double.*
- *json(naziv_stupca)* → Stupac tipa *json.*
- *text(naziv_stupca)* → Stupac tipa *text.*
- *time(naziv_stupca)* → Stupac tipa *time.*
- *timestamp(naziv_stupca)* → Stupac tipa *timestamp.*
- *uuid(naziv_stupca)* → Stupac tipa *uuid*, tj. u MySQL-u char(36).

Specijalne Blueprint metode za strukturu stupaca:

- *increments(naziv_stupca)* → Stupac tipa *integer* koji stvara inkrement sa svakim novim podatkom, najčešće se koristi kod stupca ID-a tablice.
- *timestamps()* → Stupci „created_at“ i „updated_at“ tipa *timestamp* koje Laravel automatski popunjava kod određenih CRUD operacija.
- *rememberToken()* → Stupac „remember_token“ tipa varchar(100) koji Laravel koristi kod prijave korisnika u sustav.
- *softDeletes()* → Stupac „deleted_at“ tipa *timestamp* u koje se upisuje vrijeme kada je podatak meko izbrisan.

- *morphs(naziv_stupca)* → Stupci „naziv_stupca_id“ tipa *integer* i „naziv_stupca_type“ tipa *string*, koristi se kod polimorfnih veza.

Ulančane Blueprint metode za strukturu stupaca:

- *nullable()* → Dopušta vrijednost *null.*
- *default('zadana_vrijednost')* → Ako korisnik nije specificirao vrijednost postavlja na zadanu.
- *unsigned()* → Dopušta samo pozitivne brojeve i nulu.
- *first()* → Postavlja stupac na početak tablice.
- *after(naziv_stupca)* → Postavlja stupac nakon zadanog stupca.
- *unique()* → Vrijednosti moraju biti jedinstvene.
- *primiry()* → Stupac predstavlja primarni ključ.
- *index()* → Stupac predstavlja index.

Blueprint metode izmjene strukture stupaca:

- *change()* → Nadodavanje ove ulančane metode na prilagođene metode koje su primijenjene nad nekim stupcem.
- *renameColumn(stari_nazv, novi_naziv)* → Izmjena naziva stupca.
- *dropColumn(naziv_stupca)* →Brisanje stupca.
- *dropUnique()* → Stupac više ne mora biti jedinstven.
- *dropPrimary()* → Stupac više ne predstavlja primarni ključ.
- *dropIndex()* → Stupac više ne predstavlja index.

Primjer 41. prikazuje stvaranje tablice.

```
//2018_08_02_231653_create_games_table.php

Schema::create('games', function (Blueprint $table) {
    $table->increments('id');
    $table->string('name');
    $table->integer('min_players')->nullable();
    $table->integer('max_players')->nullable();
    $table->decimal('rating')->nullable();
    $table->boolean('is_expansion')->default(false);
    $table->integer('bgg_game_id')->nullable()->unique();
    $table->timestamps();
});
```

Primjer 41. Stvaranje tablice

Primjer 42. prikazuje izmjenu strukture tablice.

```
//2018_08_03_130650_update_games_table.php

Schema::table('games', function ($table) {
    $table->string('name',100)->change();
    $table->renameColumn('rating','average_rating');
    $table->dropColumn('bgg_game_id');
});
```

Primjer 42. Izmjena strukture tablice

3.8.3 VEZE MEĐU TABLICAMA

Veze među tablicama je moguće kreirati u migraciji. Potrebno je specificirati koji stupac je vanjski ključ za primarni ključ druge tablice. Moguće je odrediti što se dogodi sa podacima iz tablice djeteta pri brisanju podatka iz tablice roditelja.

Metode za kreiranje veza:

- *foreign*(naziv_stupca) → Stupac koji predstavlja vanjski ključ.
- *references*(naziv_stupca) → Stupac na koji je vanjski ključ povezan.
- *on*(naziv_tablice) → Tablica na koju se vanjski ključ odnosi.
- *onDelete*(cascade / set null / restrict) → Što se dogodi pri brisanju podataka iz tablice roditelja.
- *dropForeign*(naziv_stupca) → Stupac više ne predstavlja vanjski ključ.

Primjer 43. prikazuje stvaranja veza.

```
//2018_08_03_130650_update_games_table.php

Schema::table('games', function ($table) {
     $table->foreign('developer_id')->references('id')->on('developers');
});
```

Primjer 43. Stvaranje veza među tablicama u migraciji

Iako je stvaranje veza praktičan način za upravljanjem bazom podataka u ovom projektu su veze među tablicama napravljene na razini aplikacije a ne na razini baze podataka. Odabrano je da se logika nalazi na razini aplikacije kako bi se lakše sa njom upravljalo.

3.8.4 KREIRANJE UPITA

Postoje dva načina na koja se mogu kreirati upiti. Jedan je uobičajeni, drugi je preko Eloquent alata. Prvo je opisano dohvaćanje i manipulacija podacima na uobičajeni način. Laravel ima *Fluent Interface* gdje se upit može razdvojiti na više manjih dijelova što olakšava čitanje koda. Primjeri 44., 45., 46. i 47. prikazuju ne *Fluent* upite.

```
//GameController.php

$games = DB::select(
      'select * from games where min_players = ?,
      [$minPlayers]
);
```

Primjer 44. Ne Fluent dohvaćanje podataka

```
//GameController.php

DB::insert(
      'insert into games (name, min_players, max_players) values (?,?,?)',
      [$name, $minPlayers, $maxPlayers]
);
```

Primjer 45. Ne Fluent unos podataka

```
//GameController.php

DB::update(
      'update games set min_players = ? where id = ?',
      [$minPlayers, $id]
);
```

Primjer 46. Ne Fluent ažuriranje podataka

```
//GameController.php

DB::delete(
      'delete from games where id = ?',
      [$id]
);
```

Primjer 47. Ne Fluent brisanje podataka

3.8.5 DOHVAĆANJE PODATAKA

Kod Fluent pisanja upita umjesto jedne metode koja sadržava cijeli upit, piše se više ulančanih metoda koje imaju svoje parametre. Svaka metoda se razlikuje i funkcionira na drugačiji način.

Popis metoda koje se može koristiti za kreiranje upita:

- *select()* → Kao argumente prima niz naziva stupaca koje korisnik želi pročitati. Moguće je naredbom „as“ stvoriti pseudonim.
- *addSelect()* → Drugačiji način odabira stupaca ako korisnik ne želi odabrati sve stupce u jednom nisu već kroz nekoliko metoda.
- *where()* → Metoda služi za postavljanje uvjeta koji podaci će biti odabrani. Moguće je više uvjeta napisati u nizu ili lančano nadodati novi *where()*. Argumenti su naziv stupca, operator i vrijednost sa kojom se uspoređuje. Ako je operator '=' nije ga potrebno pisati, dovoljno je samo naziv stupca i vrijednost sa kojom se uspoređuje.
- *orWhere()* → Nadovezuje se na *where()* metodu. Mora biti zadovoljen jedan od uvjeta. Potrebno je biti oprezan kod pisanja kompleksnijih *orWhere()* metoda.
- *whereBeetween()* → Odabire sve podatke čije se vrijednosti nalaze između dvije zadane vrijednosti, uključujući zadane. Prvi argument je naziv stupca, drugi argument je niz vrijednosti gdje je prva niža vrijednost, a drugi viša.

- *whereIn()* → Odabire sve podatke čije se vrijednosti nalaze među zadanim vrijednostima. Prvi argument je naziv stupca, drugi argument je niz vrijednosti.
- *whereNull()* → Odabire sve podatke gdje je vrijednost null. Argument je naziv stupca.
- *whereNotNull()* → Odabire sve podatke gdje nije vrijednost null. Argument je naziv stupca.
- *whereRaw()* → Dopušta korištenje jednog string-a za kreiranje upita. Nije preporučeno korištenje jer je slaba točka kod napada ubrizgavanjem (engl. injection attack).
- *distinct()* → Odabire samo podatke sa različitim vrijednostima, ne ponavlja duplikate.

Popis metoda za modificiranje podataka:

- *orderBy()* → Sortira podatke. Prva argument je naziv stupca, drugi argument je način sortiranja, uzlazno ('asc') ili silazno ('desc').
- *groupBy()* → Grupira podatke. Argument je naziv stupca po kojem se grupira.
- *having()* → Grupira podatke uz mogućnost filtriranja.
- *havingRaw()* → Grupira podatke uz mogućnost filtriranja unutar jednog string-a. Nije preporučeno koristiti jer je slaba točka kod napada ubrizgavanjem.
- *skip()* → Koliko podataka s početka niza se preskače. Najčešće se koristi kod kreiranja stranica. Argument je broj podataka.
- *take()* → Koliko podataka se odabire. Najčešće se koristi kod kreiranja stranica. Argument je broj podataka.
- *latest()* → Sortira podatke od posljednjeg prema najstarijem po datumu iz stupca 'created_at'
- *oldest()* → Sortira podatke od najstarijeg prema posljednjem po datumu iz stupca 'created_at'
- *inRandomOrder()* → Sortira podatke nasumično.

Na kraju ulančane metode potrebno je nadodati jednu od dodatnih metoda kako bi se pokrenuo upit:

- *get()* → Dohvaća sve rezultate upita.

- *first()* → Dohvaća prvi rezultat upita.
- *firstOrFail()* → Dohvaća prvi rezultat upita. Ako ne postoji niti jedan rezultat baca iznimku.
- *find()* → Dohvaća jedini rezultat upita. Argument je id podatka.
- *findOrFail()* → Dohvaća jedini rezultat upita. Argument je id podatka. Ako ne postoji rezultat baca iznimku.
- *value()* → Dohvaća vrijednost zadanog stupca prvog rezultata. Argument je naziv stupca.
- *count()* → Dohvaća broj koliko upit ima rezultata.
- *min()* → Dohvaća najmanju vrijednost zadanog stupca. Argument je naziv stupca
- *max()* → Dohvaća najveću vrijednost zadanog stupca. Argument je naziv stupca
- *sum()* → Dohvaća vrijednost koja je zbroj svih rezultata zadanog stupca. Argument je naziv stupca.
- *avg()* → Dohvaća prosječnu vrijednost svih rezultata jednog stupca. Argument je naziv stupca.

3.8.6 PRIDRUŽIVANJE TABLICA

Kako bi se tablice pridružile (engl. join) kod kreiranja upita potrebno je iskoristiti *join()* metodu. Metoda ima četiri argumenta, prvi je naziv tablice sa kojom se radi pridruživanje, drugi argument je naziv tablice popraćen nazivom stupca, treći je operator i četvrti je naziv druge tablice popraćen nazivom stupca (Primjer 48.).

```
//GameController.php

$games = DB::table('games')
      ->join('publishers', 'publisher.id', '=', 'games.publisher_id')
      ->select('publisher.name', 'game.name')
      ->get();
```

Primjer 48. Pridruživanje tablica

3.8.7 UNIJA UPITA

Moguće je stvoriti uniju dva upita pomoću metode *union()* ili *unionAll()* (Primjer 49.).

```
//GamesController.php

$minPlayersGames = DB::table('games')
      ->where('min_players', '<=', $players)
      ->get();

$games = DB::table('games')
      ->where('max_players', '>=', $players)
      ->union($minPlayersGames)
      ->get();
```

Primjer 49. Unija tablica

3.8.8 UNOS PODATAKA

Podatke je moguće unijeti pomoću metode *insert()* ili *insertGetId()* koji vraća id novog podatka. Metoda kao argumente prima niz u kojem je specificiran naziv stupca i vrijednost koja se u njega sprema (Primjer 50.).

```
//GamesController.php

DB::table('games')->insert([
      'name' => $name,
      'min_players' => $minPlayers,
      'max_players' => $maxPlayers,
]);
```

Primjer 50. Unos podataka u tablicu

3.8.9 AŽURIRANJE PODATAKA

Podatke je moguće ažurirati pomoću metode *update()*. Metoda kao argumente prima niz u kojem je specificiran naziv stupca i vrijednost koja se u njega sprema. Metodom *where()* specificiraju se redovi koje se želi ažurirati (Primjer 51.).

```
//GamesController.php

DB::table('games')
      ->where('id',$id)
      ->update([
            'name' => $name,
            'min_players' => $minPlayers,
            'max_players' => $maxPlayers,
      ]
);
```

Primjer 51. Ažuriranje podataka tablice

3.8.10 BRISANJE PODATAKA

Podatke je moguće izbrisati pomoću metode *delete()*. Metodom where() specificiraju se redovi koje se želi izbrisati (Primjer 52.).

```
//GamesController.php

DB::table('games')
      ->where('id',$id)
      ->delete()
);
```

Primjer 52. Brisanje podataka iz tablice

3.9 ELOQUENT

3.9.1 KREIRANJE MODELA

Eloquent je najpraktičniji alat za manipulaciju podataka u Laravelu. Slijedeća potpoglavlja će objasniti korištenje Eloquenta u kombinaciji sa već opisanim metodama kreiranja upita. (Stauffer, 2017)

Kako bi se mogla koristiti Eloquent fasada potrebno je kreirati modele za sve tablice baze podataka. Modele je moguće kreirati Artisan naredbom. U naredbi se naziv modela po konvenciji piše imenicom u jednini PacalCase-om. Naredba za kreiranje modela, npr. za igre je `php artisan make:model Game`. Kako su model i migracija povezani moguće ih je kreirati jednom Artisan naredbom. Nakon Artisan naredbe za kreiranje modela potrebno je dopisati `--migration` ili `--m`. Kada se na ovakav način stvori model nije potrebno izvršiti naredbu za stvaranje migracija. Laravel zna naziv modela iz jednine pretvoriti u konvencionalni naziv migracije gdje je naziv u množini. Naziv napisan PascalCase-om Laravel prilagodi na snake_case za naziv tablice.

Primjer 53. prikazuje postavljanje drugačijeg naziva tablice.

```
//app/models/Games.php

protected $table = 'board_games';
```

Primjer 53. Ručno postavljanje naziva tablice

Laravel pretpostavlja da je id primarni ključ. Primjer 54. prikazuje postavljanje drugog stupca za primarni ključ.

```
//Games.php

protected $primaryKey = 'bgg_game_id';
```

Primjer 54. Ručno postavljanje primarnog ključa

Ako tablica nema *timestamps* stupce za bilježenje vremena stvaranja i izmjene, potrebno je u modelu izmjeniti vrijednost kako Eloquent ne bi manipulirao tim stupcima (Primjer 55.).

```
//Games.php

public $timestamps = false;
```

Primjer 55. Postavljanje stupaca za vrijeme kreiranja i ažuriranja

3.9.2 DOHVAĆANJE REZULTATA U ELOQUENTU

Dohvaćanje podataka Eloquent fasadom slično je uobičajenom dohvaćanju, no umjesto fasade „DB“ (Primjer 56.) ulančane metode se izvršavaju nad modelom (Primjer 57.). Nije potrebno specificirati tablicu nad kojom se radi jer je model povezan sa istom.

```
//GamesController.php

$allGames = DB::table('games')->get();
```

Primjer 56. Dohvaćanje podataka DB fasadom

```
//GamesController.php

$allGames = Game::all();
```

Primjer 57. dohvaćanje podataka Eloquent fasadom

3.9.3 DOHVAĆANJE JEDNOG REZULTATA

Korištenjem Eloquent-a za dohvaćanje jednog rezultata mogu se koristiti bilo koje od metoda *find(), findOrFail(), first(), firstOrFail()* (Primjer 58.).

```
//GamesController.php

$game = Game::findOrFail($id);
```

Primjer 58. Dohvaćanje jednog podatka

3.9.4 DOHVAĆANJE VIŠE REZULTATA

Za dohvaćanje više rezultata može se koristiti metoda *get()* kada se zadaje neki uvjet ili *all()* ako se žele dohvatiti svi rezultati (Primjer 59.). Metoda *get()* se također može koristiti za dohvaćanje svih rezultata. Oboje metode kao rezultat vraćaju kolekcije podataka.

```
//GamesController.php

$games = Game::all();
```

Primjer 59. Dohvaćanje svih podatka

3.9.5 UNOS PODATAKA

Postoje dva načina na koja je moguće unijeti nove podatke pomoću Eloquent fasade. Prvi način je kreirati novu instancu klase i ručno definirati polja, te spremiti instancu metodom *save()* (Primjer 60.).

```
//GamesController.php

$game = new Game([
      'name' => $name,
      'min_players' => $minPlayers,
      'max_players' => $maxPlayers
]);
$game->rating = $rating;
$game->save();
```

Primjer 60. Unos podataka metodom save()

Drugi način za unos podataka je metodom *create()* gdje se niz podataka prosljeđuje kao argument (Primjer 61.). U ovom slučaju nije potrebno pozvati metodu *save().*

```
//GamesController.php

$game = Game::create([
      'name' => $name,
      'min_players' => $minPlayers,
      'max_players' => $maxPlayers
);
```

Primjer 61. Unos podataka metodom save()

Ovakvim unosima podataka Eloquent će ispuniti polja *created_at* i *updated_at* sa trenutnim vremenom.

3.9.6 AŽURIRANJE PODATAKA

Postoje dva načina za ažuriranje podataka. Prvi način je da se u varijablu dohvati podatak iz baze, izmijeni vrijednosti i pozove metoda *save()* (Primjer 62.).

```
//GamesController.php

$game = Game::find($id);
$game->min_players = $minPlayers;
$game->save();
```

Primjer 62. Ažuriranje podataka metodom save()

Drugi način je da se pronađe rezultat koji se ažurira metodom *update()* koja kao argument sadrži niz izmjena (Primjer 63.).

```
//GamesController.php

$game = Game::find($id)->update(['min_players' => $minPlayers]);
```

Primjer 63. Ažuriranje podataka metodom update()

Ovakvim unosima podataka Eloquent će ažurirati polje updated_at sa trenutnim vremenom.

3.9.7 BRISANJE PODATAKA

Postoje dva osnovna načina za brisanje podataka. Prvi način je dohvatiti jedan ili više podataka u varijablu nad kojom se izvrši metoda *delete()* (Primjer 64.).

```
//GamesController.php

$game = Game::find($id);
$game->delete();
```

Primjer 64. Brisanje podataka metodom delete()

Drugi način je metodom *destroy()* koja kao argument prima niz ID-eva koje se želi izbrisati (Primjer 65.).

```
//GamesController.php

Game::destroy([1, 2, 3]);
```

Primjer 65. Brisanje podataka metodom destroy()

3.9.8 MEKO BRISANJE

Postoji i „mekši“ način brisanja podataka (engl. soft delete) gdje se podaci ne izbrišu već se označe kao da su obrisani, ali i dalje postoje u bazi i moguće ih je vratiti u prijašnje stanje. Meko izbrisani podaci imaju ispunjen stupac *deleted_at*. Programer bi svaki upit morao pisati *whereNull()* metodu kako bi dohvatio ne izbrisane podatke, što dovodi do mogućnosti ljudske pogreške i nepotrebnog filtriranja kod svakog upita. Zato Eloquent automatski obavlja filtriranje te se programer ne mora obazirati na to u kojem je stanju stupac *deleted_at*. Prednost je što je podatke moguće vratiti na prijašnje stanje. Nedostaci su, ako ijedan drugi alat koristi bazu osim Eloquenta obavezno je paziti na *deleted_at* stupac. Baza postaje sve veća ako se nepotrebni podaci ne čiste. Preporučeno je koristiti meko brisanje samo kod tablica za koje je bitno da se podaci mogu ponovo dohvatiti nakon brisanja. Meko brisanje se omogući tako da se doda

stupac *deleted_at* u migraciji (Primjer 66.), te *SoftDelete* svojstvo i *deleted_at* u varijablu *$dates* u modelu (Primjer 67.). Nakon što je omogućeno *delete()* i *destroy()* izvršavaju meko brisanje.

```
//2018_08_04_030603_create_games_table.php

Schema::table('games', function (Blueprint $table) {
      $table->softDeletes();
});
```

Primjer 66. Migracija koja omogućava meko brisanje podataka

```
//Game.php

class Game extends Model
{
      use SoftDeletes;
      protected $dates = ['deleted_at'];
}
```

Primjer 67. Model koji omogućava meko brisanje podataka

3.9.9 DOHVAĆANJE MEKO IZBRISANIH PODATAKA

Za dohvaćanje meko izbrisanih podataka uz ne izbrisane podatke koristi se ulančana metoda *withTrashed()*, a za dohvaćanje samo meko izbrisanih podataka metoda *onlyTrashed()* (Primjer 68.).

```
//GamesController.php

$all = Game::withTrashed()->get();
$deleted = Game::onlyTrashed()->get();
```

Primjer 68. Dohvaćanje meko izbrisanih podataka

3.9.10 VRAĆANJE MEKO IZBRISANIH PODATAKA

Za vratiti meko izbrisane podatke koristi se metoda *restore()* nad instancom ili upitom. Primjer 69. prikazuje vraćanje podataka.

```
//GamesController.php

$game = Game::onlyTrashed()->first();
$game->restore();
```

Primjer 69. Vraćanje meko izbrisanih podataka

3.9.11 BRISANJE MEKO IZBRISANIH PODATAKA

Ako podatak želimo izbrisati iz baze podataka, a ne samo meko izbrisati, potrebno je pokrenuti metodu *forceDelete()* nad instancom ili upitom (Primjer 70.).

```
//GamesController.php

Game::onlyTrashed()->forceDelete();
```

Primjer 70. Brisanje meko obrisanih podataka

3.10 ZAŠTITA STUPACA

Ponekad je bolje korisnicima zabraniti mogućnost pristupa određenim stupcima kako ne bi koristili zlonamjerne podatke. To je moguće izvesti tako da se dopusti ili zabrani pristup određenim stupcima preko modela. Ispuni se *$fillable* varijabla sa nizom stupaca koje korisnik smije izmjenjivati ili *$guarded* varijabla sa nizom stupaca koje korisnik ne smije izmjenjivati. Ako se korisniku želi dati pristup svim stupcima potrebno je *$guarded* ispuniti praznim nizom ili ako se korisniku želi zabraniti pristup svakom stupcu potrebno je *$fillable* ispuniti praznim nizom.

3.11 VEZE

3.11.1 VEZA JEDAN NA JEDAN

Veza jedan na jedan se stvara tako da se unutar modela kreira metoda koja vraća rezultat metode *hasOne()* gdje se kao argument naznači sa kojim modelom je model povezan, tj. kojem modelu on pripada kao dijete (Primjer 71.). (McCool, 2012)

```
//Game.php

class Game extends Model
{
	public function developer()
	{
		return $this->belongsTo(Developer::class);
	}
}
```

Primjer 71. Veza jedan na jedan

3.11.2 VEZA JEDAN NA MNOGO

Veza jedan na mnogo se stvara tako da se unutar modela kreira metoda koja vraća rezultat metode *hasMany()* gdje se kao argument naznači sa kojim modelom je model povezan, tj. kojem modelu je on roditelj (Primjer 72.). (McCool, 2012)

```
//Developer.php

class Developer extends Model
{
	public function games()
	{
		return $this->hasMany(Game::class);
	}
}
```

Primjer 72. Veza jedan na mnogo

3.11.3 KORIŠTENJE VEZA

Kako bi se pristupilo povezanim tablicama dovoljno je pozvati kreiranu metodu i kolekcija će za svaki rezultat imati vrijednosti iz povezane tablice (Primjer 73.).

```
//GamesController.php

$games = Game::all();
foreach ($games as $game) $game->developer;

$developers = Developer::all();
foreach ($developers as $developer) $developer->games;
```

Primjer 73 Korištenje veza mnogo na jedan i jedan na mnogo

4. REST API U LARAVEL OKRUŽENJU

4.1 O REST API-U

Reprezentativni prijenos stanja (engl. Representational State Transfer , REST) je tip arhitekture za kreiranje aplikacijskog programskog sučelja (engl. Application Programming Interface, API). API je zapravo skup pravila i specifikacija koje programer slijedi kako bi stvorio servis. Drugi programer koji zna pravila, ne mora sam pisati cijele programe već koristi postojeći servis kako bi uštedio vrijeme i trud. Za API je specifično da koristi HTTP protokol i njegove akcije, za odgovor vraća JSON i nema stanje. (Richardson & Amundsen, 2013)

Resursi se mogu jedinstveno referencirati pomoću URI-a, tako da se napiše naziv resursa, te id nakon kose crte „/“ (npr. *games/42*). Više različitih metoda može se pokrenuti sa istim URI-em, tako da se koriste HTTP glagoli. URI *games/42* bi imao posve drugačiju svrhu ako se pozove uz GET, PUT ili DELETE glagol. GET bi vjerojatno vratio vrijednosti za resurs koji ima pripadajući ID, PUT bi ažurirao resurs sa pripadajućim ID-om i DELETE bi obrisao resurs sa pripadajućim ID-om. API nema stanje što znači da se korisnik ne može prijaviti u servis na uobičajeni način. Kako nema stanje, zahtjevi bi gotovo uvijek trebali imati isti rezultat neovisno o tome tko ih izvrši. Ovakva pravila mijenjaju način korištenja Laravela. U suštini za logiku aplikacije nije bitno je li ona napravljena za Web stranicu ili za API. Razlika je da kao odgovor metode aplikacija ne vraća preglede već JSON. (2012)

Front-end aplikacije nije uobičajeni HTML kojiu je namjenjen korisniku već JSON format kojeg drugi programer koristiti u bilo kojem programskom jeziku. Eloquent ima dodatne mogućnosti kod raspodjele rezultata na stranice što je opisano u slijedećem potpoglavlju. Kako API servis nema stanje opisan je i način na koji se autorizacija i autentifikacija izvršavaju.

4.2 API UPRAVLJAČ

4.2.1 STRUKTURA REST API UPRAVLJAČA

Struktura REST API upravljače je vrlo slična uobičajenoj strukturi upravljača. Stvaranjem upravljača sa Artisan naredbom stvore se CRUD metode.

Korištenje metoda u API upravljaču (Richardson & Ruby, 2007):

- index() → Vraća sve rezultate. Primjer zahtjeva: GET → *games*
- store(Request *$request*) → Stvara novi resurs. Primjer: POST → games
- show($id) → Vraća zadani resurs. Primjer zahtjeva: GET → games/42
- update(Request *$request*, *$id*) → Ažurira se postojeći resurs. Primjer zahtjeva: PUT → games/42
- destroy($id) → Briše zadani resurs. Primjer zahtjeva: DELETE → games/42
- create() → Ne koristi se jer forma nije potrebna.
- edit() → Ne koristi se jer forma nije potrebna.

4.2.2 OBILJEŽAVANJE STRANICA

Eloquent ima mogućnost za rastavljanje podataka na stranice. Kako se ne bi moralo dohvatiti veću količinu podataka moguće ih je rastaviti na dijelove korištenjem metode *paginate()* sa argumentom koliko podataka se želi dohvatiti u jednom zahtjevu. Metoda *paginate()* se koristi umjesto metode *get()*. Eloquent zna koju stranicu treba dohvatiti pomoću parametra „page“. Ako parametar nije ispunjen podrazumijeva se da korisnik želi prvu stranicu. Unutar metode upravljača nije potrebno koristiti parametar jer Eloquent zna na što se parametar odnosi i sam ga upotrijebi. Osim što vraćeni podaci budu rastavljeni na stranice, Eloquent vrati i druga polja koje daju dodatne informacije o stranici.

Opis polja koja se stvore kod kreiranja stranica:

- *path* → URL zahtjeva bez parametara.
- *first_page_url* → URL prve stranice.
- *last_page_url* → URL posljednje stranice.
- *prev_page_url* → URL prethodne stranice.

- *next_page_url* → URL sljedeće stranice.
- *per_page* → Koliko rezultata se prikazuje po stranici.
- *last_page* → Koja stranica je posljednja.
- *total* → Koliko sveukupno ima rezultata.
- *from* → Od kojeg rezultata se prikazuje u ovom zahtjevu.
- *to* → Do kojeg rezultata se prikazuje u ovom zahtjevu

Primjer 74. prikazuje polja koja opisuju stranicu.

```
GET -> /games?page=2

"path": "http://localhost:8000/api/games",
"first_page_url": "http://localhost:8000/api/games?page=1",
"last_page_url": "http://localhost:8000/api/games?page=4",
"prev_page_url": "http://localhost:8000/api/games?page=1",
"next_page_url": "http://localhost:8000/api/games?page=3",
"per_page": 100,
"last_page": 4,
"total": 370,
"from": 101,
"to": 200,
```

Primjer 74. Opis trenutne stranice

4.3 AUTORIZACIJA I AUTENTIFIKACIJA

4.3.1 KORIŠTENJE TOKENA PRI AUTORIZACIJI I AUTENTIFIKACIJI

Za autorizaciju i autentifikaciju je vrlo bitno to da REST API ne sprema stanje. To što aplikacija nema stanje označava da se korisnik ne može prijaviti u aplikaciju te ju onda koristiti jer se ta prijava ne sprema. REST se zasniva na tome da se stanje šalje između sustava i korisnika. Kod autorizacije to označava da korisnik stvori zahtjev za prijavu i pošalje ga API-u, API obradi zahtjev te korisniku pošalje odgovor u kojem se nalazi token. Token je potvrda da se korisnik uspješno prijavio u sustav. Korisnik do određenog trenutka može koristiti taj token. Token služi pri slanju drugih zahtjeva kako bi dokazao da je prijavljen u sustav i dokazao svoj identitet. Na taj način dolazi do slanja stanja između API-a i korisnika. (Cooksey, 2014)

4.3.2 JSON WEB TOKEN

Neki od poznatih tokena su JSON Web Token (JWT), Simple Web Token (SWT) i Security Assertion Markup Language Token (SAML). U svrhu ovog rada koristi se preporučeni JSON Web Token. JWT je preporučen nad SWT-om zbog povećane sigurnosti pri šifriranju podataka. Što se tiče sigurnosti JWT i SAML su na istoj razini. Razlika između JWT-a i SAML-a dolazi kod tipa podataka kojeg šifriraju. SAML je šifrirani XML, JWT je šifrirani JSON. JSON i XML su čitljivi ljudima, imaju hijerarhiju podataka i često su korišteni u različitim programskim jezicima. Razlika je što je JSON orijentiran na podatke dok je XML orijentiran prema prikazu datoteke. Kod prijenosa podataka XML sadržava suvišne dijelove i riječi koje zauzimaju prostor dok je JSON kratak i lakši za raščlanjivanje. Upravo to pogoduje kod šifriranja tokena jer je lakše šifrirati i raščlaniti takve podatke.

JWT se sastoji od tri dijela:

1. zaglavlje (engl. header)
2. sadržaj (engl. payload)
3. potpis (engl. signature).

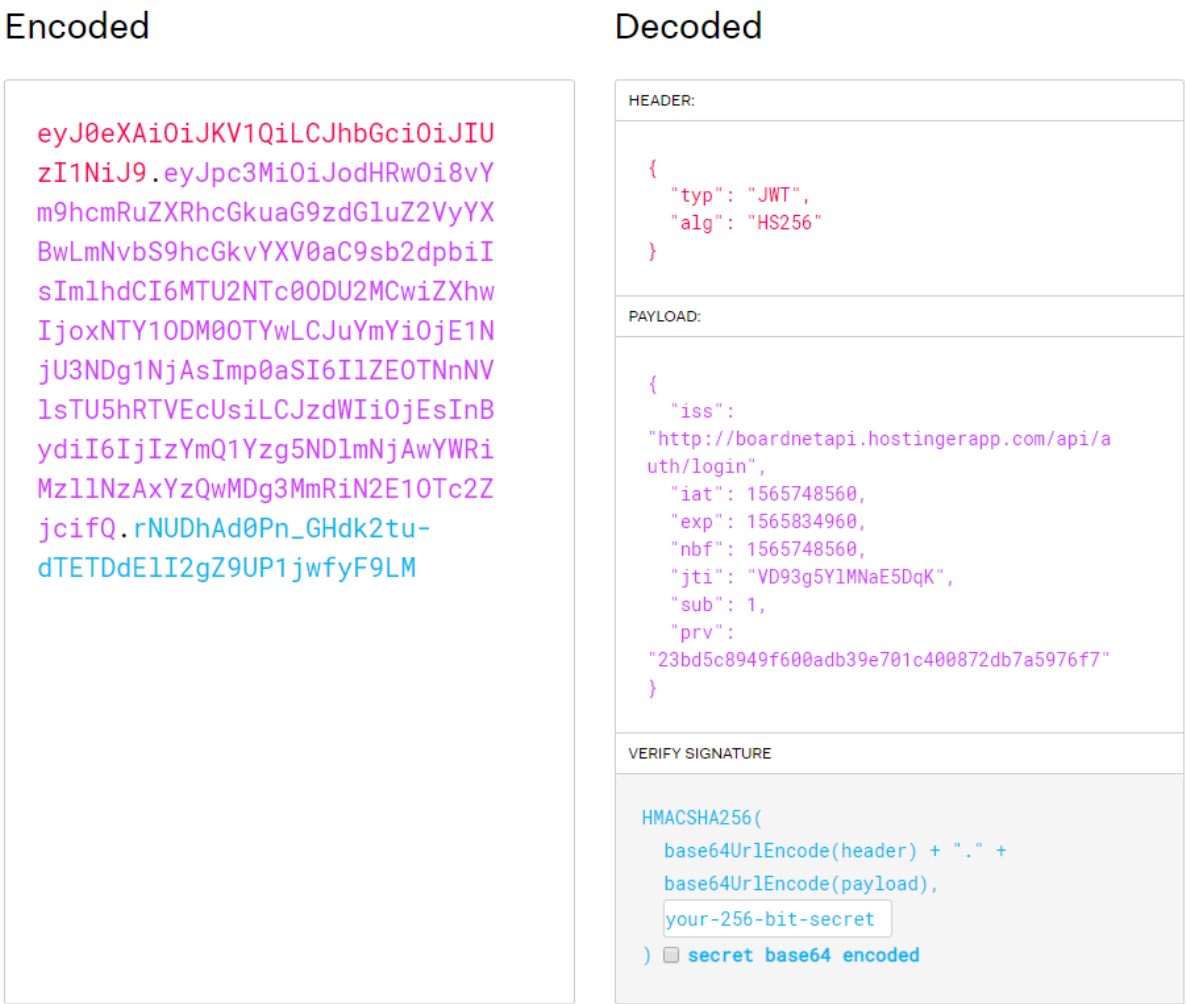
Svaki dio je niz alfanumeričkih znakova međusobno spojenih točkom „.“. Zaglavlje sadžava informaciju o tipu tokena i algoritmu šifriranja. Token je JWT, a algoritam HS256. Sadržaj ima podatke, tj. stanje koje se šalje.

Kod registriranih korisnika u sadržaj spadaju sljedeća polja:

- *iss* (Issuer) → Tko je kreirao token. U Laravel aplikaciji to je usmjerivač za prijavu korisnika.
- *iat* (Issued at) → Vrijeme kada je token kreiran, izraženo u sekundama.
- *exp* (Expires at) → Vrijeme do kada token vrijedi, izraženo u sekundama.
- *nbf* (Not walid before) → Vrijeme od kada token vrijedi, izraženo u sekundama.
- *jti* (JWT ID) → Jedinstveni id JWT-a.
- *sub* (Subject) → Korisnik kojem token pripada.
- *prv* (User Provider Class) → Koristi se kada ima nekoliko razina sigurnosti pri autorizaciji.

Za dobiti potpis potrebno je uzeti šifrirano zaglavlje, šifrirani sadržaj, tajnu (engl. secret), algoritam šifriranja te to potpisati. Potpis služi za provjeru da zaglavlje i sadržaj nisu zlonamjerno modificirani pri prijenosu tokena. Tajna je izraz koji je programer zadao da se koristi pri šifrirajnu podataka. Poželjno je imati što složeniji izraz za tajnu kako bi se umanjila mogučnost zlonamjernog izmjenjivanja tokena. (Peyrott, 2018)

Slika 3. prikazuje primjer JWT-a koji je dobiven kao rezultat prijave na API koji je stvoren u svrhu ovog rada. Na slici se mogu vidjeti sva tri dijela tokena, šifrirana sa lijeve strane i dešifrirana sa desne strane.



Slika 3. JWT token i pripadajući JSON (Auth0, 2019)

5. PROGRAMSKO RJEŠENJE

5.1 OPIS RJEŠENJA

Uz pomoć prethodnih cjelina stvoren je projekt koji služi kao API aplikacija za bilježenje rezultata odigranih društvenih igara. Kroz sljedeća poglavlja opisani su resrusi korisnika, igrara, knjžnica igara, prijatelja, igranja, igrača i timova sa pripadajućim usmjerivačima, modelima, migracijama, upravljačima i metodama. U dodatku se nalazi popis svih usmjerivača sa metodama koje pozivaju (Tablica 1.)

5.2 KORISNICI

5.2.1 REGISTRACIJA I PRIJAVA

Za registraciju i prijavu se koriste predefinirane metode koje automatski pružaju token za autentifikaciju i autorizaciju. Koristi se model User (hrv. korisnik). Model i migracija su automatski kreirani sa kreiranjem projekta. Migracija je ispunjena sa željenim poljima (Primjer 75.).

```
//2014_10_12_000000_create_users_table.php

class CreateUsersTable extends Migration{
    public function up(){
        Schema::create('users', function (Blueprint $table) {
            $table->increments('id');
            $table->string('username')->unique();
            $table->string('email')->unique();
            $table->string('password');
            $table->string('bgg_username')->nullable()->unique();
            $table->string('name')->nullable();
            $table->string('surname')->nullable();
            $table->timestamp('date_of_birth')->nullable();
            $table->rememberToken();
            $table->timestamps();
        });
    }

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

Primjer 75. Migracija za tablicu korisnika

Tablica korisnika u bazi sadrži i sljedeća polja:

- *id* → Identifikacijsko polje, jedinstveno za svakog korisnika. Automatski generiran broj pri kreiranju novog elementa.
- *username* → Korisničko ime, korisnik ga koristi kod prijave, mora biti jedinstveno.
- *email* → Da bi se korisnik registrirao mora unijeti vlastiti email. Mora biti jedinstveni, svaki korisnik bi trebao imati samo jedan račun.
- *password* → Lozinka koju si korisnik odabere pri registraciji. Koristi se kod prijave. Lozinka je kriptirana.
- *bgg_username* → Ako korisnik ima račun na Web stranici Bord Game Geek može ga ovdje spremiti da ga drugi korisnici mogu vidjeti. Nije obavezno.
- *name* → Korisnikovo ime. Nije obavezno.
- *surname* → Korisnikovo prezime. Nije obavezno.
- *date_of_birth* → Korisnikov datum rođenja. Nije obavezan.
- *rememberToken* → Token koji se koristi kod Web stranica napravljenih u Laravelu. Nije potreban za API.
- *timestamps* → Datumi kada je korisnik kreiran i posljednji put izmjenjen.

Primjer 76. prikazuje isječak koda modela korisnika:

```
//User.php

class User extends Authenticatable implements JWTSubject{
    protected $guarded = [];

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

    public function getJWTIdentifier(){
        return $this->getKey();
    }

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

```
    public function libraries(){
        return $this->hasMany('App\Models\Library');
    }

    public function friends_user(){
        return $this->hasMany('App\Models\Friend');
    }

    public function friends_friend(){
        return $this->hasMany('App\Models\Friend');
    }

    public function plays()
    {
        return $this->hasMany('App\Models\Play');
    }

    public function players()
    {
        return $this->hasMany('App\Models\Player');
    }
}
```

Primjer 76. Model korisnika

Niz *$guarded* je prazan što označava da se svako od polja može izmjenjivati. Niz *$hidden* (hrv. sakriveno) je ispunjen poljima *password*, *remember_token, created_at i updated_at* zato što ne postoji potreba da korisnik ima pristup tim podacima kako ne bi ugrozio sigurnost. Metode *getJWTIdentifier()* i *getJWTCustomClaims()* su automatski stvorene metode koje se koriste kod autentifikacije i autorizacije. Sve ostale metode su veze prema drugim tablicama. Metode *hasMany()* označavaju da jedan element u tablici korisnika može imati više elemenata drugih tablica.

Primjer 77. prikazuje isječak koda za usmjerivače registracije i prijave. Upravljači i metode na koje usmjerava su automatski stvorene i pripadaju *ApiRegisterController*-u i *ApiAuthController*-u koji se nalaze unutar mape *Auth*.

```
//api.php

Route::group(['middleware' => ['api']], function () {
      Route::prefix('auth')->group(function () {
          Route::post('register', 'Auth\ApiRegisterController@register');
          Route::post('login', 'Auth\ApiAuthController@login');
      });
      ...
};
```

Primjer 77 .Usmjerivači za registraciju i prijavu

Usmjerivači se nalaze okruženi *middleware*-om *'api'*. Svi usmjerivači koji će biti u slijedećim primjerima će se nalaziti unutar ovog *middleware*-a iako to neće biti ispisano u isječcima. Unutar *middleware*-a se nalazi grupa koja dijeli prefiks, u ovom slučaju je to prefiks *'auth'*. Model korisnika je podijeljen na dva prefiksa, *'auth'* i *'user'*. Metode *'auth'* predefinirane su za registraciju i prijavu korisnika dok su *'user'* metode ručno programirane. Metode *register* i *login* se pozivaju HTTP akcijom POST jer je potrebno zahtjev ispuniti nekim podacima. Usmjerivači za 'user' metode sadrže CRUD i ostale operacije koje se vrše nad modelom korisnika. Osim korisnika svi ostali modeli imaju zasebne grupe u kojima su usmjerivači koji dijele prefiks, te je taj prefiks jednak nazivu modela. Za registraciju je potrebno poslati zahtjev na Web stranicu *URL/api/auth/login* što poziva metodu *register* (Primjer 78.).

```
//ApiRegisterController.php

public function register(Request $request){
      $errors = $this->validator($request->all())->errors();
      if(count($errors)){
            return response(['success' => false,
            'errors' => $errors], 400);
      }

      event(new Registered($user = $this->create($request->all())));

      return response(['success' => true, 'user' => $user]);
}
```

Primjer 78. Metoda registracije

Metoda registracije može biti podijeljena na tri dijela. Prvi dio vrši validaciju podataka te ako podaci nisu u željenom formatu API vraća odgovor sa listom pogrešaka i HTTP statusom 400, što označava da je zahtjev pogrešno poslan i da korisnik mora nešto izmijeniti da bi zahtjev bio odobren. Primjer 79. prikazuje isječak metode validacije. Drugi dio metode registracije stvara novog korisnika. Ispunjava polje username i email sa unesenim podacima, a polja password ispuni sa kriptiranom verzijom lozinke koju je korisnik unio. Lozinka se kriptira kako bi se osiguralo da nitko ne može pročitati korisnikovu lozinku iz baze podataka. Primjer 80. prikazuje isječak metode stvaranja korisnika. Metoda u posljednjem dijelu registracije vraća kao odgovor podatke id, username i email korisnika.

```
//RegisterController.php

protected function validator(array $data) {
      return Validator::make($data, [
            'username' => 'required|string|max:255|unique:users,username',
            'email' => 'required|string|email|max:255|unique:users,email',
            'password' => 'required|string|min:4|confirmed'
        ]);
}
```

Primjer 79. Validacija registracije

```
//RegisterController.php

protected function create(array $data){
      return User::create([
            'username' => $data['username'],
            'email' => $data['email'],
            'password' => Hash::make($data['password']),
      ]);
}
```

Primjer 80. Kreiranje korisnika

Metoda prijave (Primjer 81.) sprema podatke koje je korisnik poslao u zahtjevu u varijablu *$credentials*. Pomoću te varijable, ako su podaci točni, pokušava stvoriti JWT token koji vraća korisniku kod prijave. Korisnik taj token može zadano vrijeme koristiti kod poziva ostalih metoda ako je za to autoriziran. Ako je autorizacija neuspjela metoda obavijesti korisnika da se uneseno korisničko ime ne slaže sa lozinkom.

```
//Auth/ApiAuthController.php

public function login(Request $request){
      $credentials = $request->only('username', 'password');

      if (!$jwt = JWTAuth::attempt($credentials)) {
            return response()->json([
                  'response' => 'error',
                  'message' => 'invalid_credentials',
            ], 401);
      }
      return response()->json([
            'response' => 'success',
            'result' => ['token' => $jwt]
      ]);
}
```

Primjer 81. Metoda prijave

5.2.2 CRUD I OSTALE METODE

CRUD operacije nad bazom izvršavaju metode *index*, *store*, *show*, *update* i *destroy* (Primjer 82.). Za ovaj model se preskače metoda *store*, tj. stvaranje korisnika jer tu radnju obavlja metoda registracije.

```
//api.php

Route::prefix('users')->group(function () {
      Route::get('', 'UserController@index');
      Route::get('{user_id}', 'UserController@show')
                  ->where('user_id', '[0-9]+');
      Route::put('{user_id}', 'UserController@update')
                  ->where('user_id', '[0-9]+');
      Route::delete('{user_id}', 'UserController@destroy')
                  ->where('user_id', '[0-9]+');
      Route::get('{username}', 'UserController@showByUsername');
      Route::get('search/name/{name}', 'UserController@searchByName');
      Route::get('search/username/{username}',
                  'UserController@searchByUsername');
});
```

Primjer 82. Usmjerivači korisnika

Metoda *index* (Primjer 83.) vraća sve elemente tablice korisnika. Podaci su raskomadani po 50 na stranici da se ne bi slala prevelika količina podataka. U metodi se nigdje ne vidi da se za prikaz određene stranice zahtjeva parametar *page* iz zahtjeva, to izvršava metoda *paginate().* Podaci se vraćaju tako da se stvore polje *'success'* (hrv. uspijeh) koje u sebi sadrži boolean vrijednost i polje *'result'* (hrv. rezultat) koje u sebi sadrži kolekciju koju želimo da bude rezultat metode. Metoda *json()* pretvara bilo koji tip podatka u JSON format te ga kao takvog šalje korisniku. Metoda kao prvi argument prima niz polja, a kao drugi neobavezni prima HTTP status kod. Ako HTTP status kod nije zadan metoda šalje status 200, što znači da je rezultat zahtjeva OK. Uz HTTP status kod nije potrebno da postoji polje *'success'*, no ono se svejedno koristi radi lakšeg raščlanjivanja podataka. Ako je došlo do greške na serveru šalje se HTTP status kod 500.

```
//UserController.php

public function index () {
      $users = User::paginate(50);
      return response()->json(['success' => true, 'result' => $users]);
}
```

Primjer 83. Metoda index korisnika

Metoda *show* (Primjer 84.) za cilj ima vratiti podatke o jednom korisniku. Zahtjev u sebi mora sadržavati id korisnika čije podatke želi dohvatiti. Prvi dio metode provjerava token dobiven u zahtjevu. Ako je korisnik autentificiran metoda nastavlja izvršavanje, inače vraća grešku. Primjer 85. prikazuje isječak metode autentifikacije. Metoda *show* provjerava postoji li željeni korisnik. Ako željeni korisnik ne postoji vraća HTTP status 404, što znači da traženi resurs ne postoji. Ako korisnik postoji šalje taj resurs kao odgovor.

```
//UserContorller.php

public function show ($user_id) {
   $this->authenticateRequest();
   if ($this->authMessage != null)
       return response()->json(['success' => false,
            'result' => $this->authMessage]);

   $user = User::find($user_id);
   if ($user == null)
       return response()->json(['success' => false,
            'result' => "There is no user with this id"], 404);

   return response()->json(['success' => true, 'result' => $user]);
}
```

Primjer 84. Metoda show korisnika

```
//Controller.php

public function authenticateRequest(){
      try{
            $this->loggedUser = JWTAuth::parseToken()->authenticate();
      } catch (\Exception $exception) {
            $this->authMessage = $exception->getMessage();
            $this->authStatusCode = 401;
      }
}
```

Primjer 85. Metoda autentifikacije

```
//Controller.php

public function authorizeRequest($user_id){
      if ($this->loggedUser != null && $this->loggedUser->id != $user_id){
            $this->authMessage='You can not execute action with this user';
            $this->authStatusCode = 403;
      }
}
```

Primjer 86. Metoda autorizacije

Metoda *update* (Primjer 87.) za svrhu ima ažurirati podatke korisnika. Metoda ima dva argumenta, prvi je *$request* tipa Request, što korisnik šalje u tijelu zahtjeva, a drugi *$user_id*, što korisnik šalje kao parametar zahtjeva. Metoda provjerava postoji li korisnik u bazi pod zadanim id-om, ako ne postoji metoda vraća grešku. Korisnik koji je poslao zahtjev mora priložiti svoj token radi autentifikacije, ali u ovom slučaju i radi autorizacije. To se radi zato što ne može svaki korisnik imati pravo izmjene podataka drugih korisnika. Autorizacija provjerava pripada li token korisniku nad kojim se pokušava napraviti izmjena podataka. Ako ne pripada šalje se HTTP status 403, što označava da zahtjev nije autoriziran za izvršiti ovu metodu. Primjer 86. prikazuje isječak metode autorizacije. Ako je zahtjev autoriziran metoda provjerava jesu li svi podaci ispravno uneseni metodom validacije. To je u ovom slučaju bitno provjeriti jer iako korisnik može mijenjati username i email i dalje je potrebno osigurati da su oni jedinstveni u bazi podataka. Bitno je i da druga polja ispunjavaju uvijete. Primjer 88. prikazuje isječak metode validacije. Ako je validacija uspješno prošla kod dohvaćenog korisnika podaci se ažuriraju i spremaju u bazu, te se vraćaju kao odgovor na zahtijev.

```
//UserController.php

public function update (Request $request, $user_id) {
      $user = User::find($user_id);
      if ($user == null)
            return response([
                  'success' => false,
                  'result' => "There is no user with this id."
            ], 404);

      $this->authenticateRequest();
      $this->authorizeRequest($user_id);
      if ($this->authStatusCode != null)
            return response()->json([
                  'success' => false,
                  'result' => $this->authMessage
            ],$this->authStatusCode);

      $errors = $this->userUpdateDataValidator(
            $request->all(), $user_id)->errors();
      if(count($errors))
            return response([
                  'success' => false,
                  'result' => $errors
            ], 400);

      $user->username = $request->get("username");
      $user->email = $request->get("email");
      $user->name = $request->get("name", null);
      $user->surname = $request->get("surname", null);
      $user->date_of_birth = $request->get("date_of_birth", null);
      $user->bgg_username = $request->get("bgg_username", null);
      $user->save();
      return response(['success' => true, 'result' => $user], 200);
}
```

Primjer 87. Metoda update korisnika

```
//UserController.php

protected function userUpdateDataValidator(array $data, $user_id){
    return Validator::make($data, [
       'username'
          =>'required|string|max:255|unique:users,username,'.$user_id,
       'email'
          =>'required|string|email|max:255|unique:users,email,'.$user_id,
       'date_of_birth'
          =>'date|date_format:Y-m-d|after:1900-01-01|before:today'
    ]);
}
```

Primjer 88. Metoda validacije ažuriranja korisnika

Metoda destroy (Primjer 89.) ima za cilj izbrisati zadanog korisnika iz baze podataka. Metoda provjerava postoji li korisnik koji se želi izbrisati, je li zahtjev autentificiran i autoriziran za brisanje tog podatka. Kako je odlučeno da se sva logika odvija na razini aplikacije, a ne na razini baze podataka potrebno je prije brisanja resursa korisnika obraditi sve resurse djece tog korisnika. Neki resursi se u potpunosti brišu, a neki samo postavljaju vanjske ključeve na vrijednost *null*. Kada su svi resursi djece obrađeni briše se resurs korisnika, te se vraća njegov *username* kao odgovor na zahtjev.

```
//UserController.php

public function destroy ($user_id){
        $user = User::find($user_id);
        if ($user == null)
            return response([
                  'success' => false,
                  'result' => "User does not exist"
                  ], 200);

        $this->authenticateRequest();
        $this->authorizeRequest($user_id);
        if ($this->authMessage != null)
            return response()->json([
                  'success' => false,
                  'result' => $this->authMessage
                  ]);

        $libraries = $user->libraries;
        foreach ($libraries as $library) $library->delete();

        $friends_user = $user->friends_user;
        foreach ($friends_user as $friend) $friend->delete();

        $friends_friend = $user->friends_friend;
        foreach ($friends_friend as $friend) $friend->delete();

        $plays = $user->plays;
        foreach ($plays as $play) $play->delete();

        $players = $user->players;
        foreach ($players as $player)
        {
            $player->user_id = null;
            $player->save();
        }

        $user->delete();

        return response([
            'success' => true,
            'result' => ['deleted' => $user->username]
        ], 200);
}
```

Primjer 89. Metoda destroy korisnika

Metoda *showByUsername* (Primjer 90.) radi na istom principu kao i metoda *show*, razlika je da se u metodi *showByUsername* korisnik može dohvatiti preko username-a umjesto id-a. To je napravljeno kako bi korisnicima bilo lakše i jasnije poslati zahtjev. Kako se metode *show* i *showByUsername* pozivaju HTTP akcijom GET, imaju isti URL za poziv i primaju parametar na istom mjestu, bilo je potrebno osigurati da se prava metoda pozove u pravom trenutku. To je napravljeno tako da se kod usmjerivača provjerava je li parametar broj ili ne (Primjer 82.). Ako je parametar broj poziva se metoda *show*, inače se poziva metoda *showByUsername*.

```
//UserController.php

public function showByUsername ($username)
{
    $this->authenticateRequest();
    if ($this->authMessage != null)
        return response()->json([
            'success' => false,
            'result' => $this->authMessage
        ]);
    $user = User::where('username', $username)->first();
    if ($user == null)
        return response()->json([
            'success' => false,
            'result' => "User does not exist."
        ]);
    return response()->json(['success' => true, 'result' => $user]);
}
```

Primjer 90. Metoda showByUsername korisnika

Metode *searchByName* i *searchByUsername* (Primjer 91.) za cilj imaju dohvatiti sve korisnike koji se slažu sa zadanim parametrom. Ovisno o pozvanoj metodi korisnike se pretražuje po korisničkom imenu ili imenu i prezimenu.

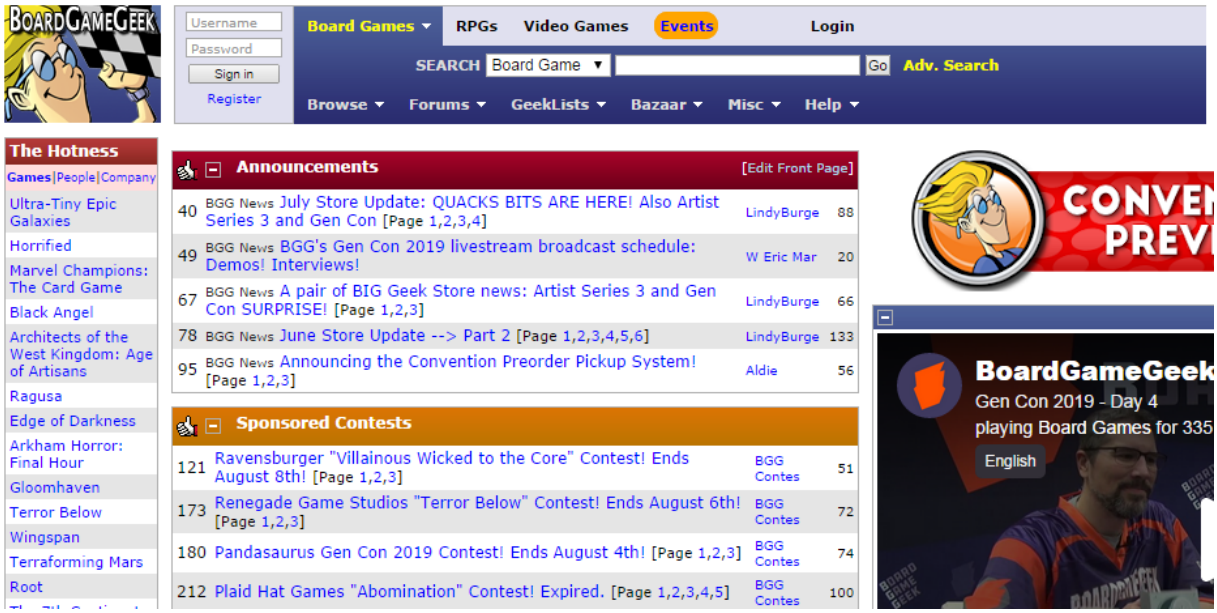
```
//UserController.php

public function searchByUsername($username){
    $users = User::select(['id', 'username', 'name', 'surname'])
            ->where('username', 'LIKE', '%' . $username . '%')->get();
    if ($users->isEmpty())
        return response(['success' => true, 'result' => null], 200);
    return response(['success' => true, 'result' => $users], 200);
}
```

Primjer 91. Metoda searchByUsername korisnika

5.3 IGRE

Model igre je osmišljen na drugačiji način nego ostali modeli ove aplikacije. Kako na svijetu postoji prevelika količina igara da bi se sve ručno dodale u bazu odlučeno je koristiti podatke sa Web stranice Board Game Geek (BGG). Slika 4. prikazuje stranicu.



Slika 4. Web stranica BoardGameGeek (https://boardgamegeek.com/, 16. srpanj 2019.)

BGG stranica ima sličnu svrhu kao i aplikacija ovog rada, no postoji samo u obliku Web stranice. BGG nema niti jednu drugu službenu aplikaciju za mobilne ili stolne uređaje. BGG pruža API sa podacima igara koje imaju u svojoj bazi (https://bgg-json.azurewebsites.net/). Ovime se prikazuje korisnost API-a koje jedan programer može pružiti drugome. Aplikacija pomoću BGG API-a dohvatiti informacije o igrama te ih spremiti u vlastitu bazu. Kako je baza podataka koju posjeduju jako velika nije kreiran API zahtjev koji pruža popis svih igara. Postoji nekoliko drugih načina na koji se može dohvatiti te podatke. Jedan način je da se napravi zahtjev sa id-om igre koji ona ima na BGG-u, drugi način je da se dohvate sve igre koje BGG korisnik ima u svojoj knjižnici, te treći način da se dohvati 50 igara koje se trenutno najviše pretražuju. Treći način dohvaća samo naziv i id igre te je tada potrebno pozvati igru pomoću id-a. Aplikacija može koristiti svaki od ta tri načina. Jednom kada je igra dodana u bazu podataka nije ju više potrebno pozivati sa BGG-a. Primjer 92. prikazuje isječak migracije igara koja ima sva polja kao i igra koja se dobije kao odgovor sa BGG

API-a. Primjer 93. prikazuje model igre koji je povezan na druge modele. Primjer 94. prikazuje usmjerivače za igre.

```
//2018_08_02_231653_create_games_table.php

public function up(){
        Schema::create('games', function (Blueprint $table) {
            $table->increments('id');
            $table->string('link')->nullable();
            $table->integer('bgg_game_id')->nullable()->unique();
            $table->string('name');
            $table->string('image')->nullable();
            $table->string('thumbnail')->nullable();
            $table->integer('min_players')->nullable();
            $table->integer('max_players')->nullable();
            $table->integer('playing_time')->nullable();
            $table->boolean('is_expansion')->nullable()->default(false);
            $table->integer('year_published')->nullable();
            $table->decimal('bgg_rating')->nullable();
            $table->decimal('average_rating')->nullable();
            $table->integer('rank')->nullable();
            $table->timestamps();
        });
    }
```

Primjer 92. Migracija za tablicu igara

```
//Game.php

class Game extends Model
{
    protected $guarded = [];

    public function libraries(){
        return $this->hasMany('App\Models\Library');
    }

    public function plays(){
        return $this->hasMany('App\Models\Play');
    }
}
```

Primjer 93. Model igre

```
//api.php

Route::prefix('games')->group(function () {
    Route::post('bgg/id',
      'GameController@storeFromBggGame');
    Route::post('bgg',
      'GameController@storeFromBggLibrary');
    Route::post('bgg/hotness', 'GameController@storeFromBggHotness');
    Route::get('', 'GameController@index');
    Route::get('{game_id}/{username?}',
      'GameController@show')->where('game_id', '[0-9]+');
    Route::put('{game_id}',
      'GameController@update')->where('game_id', '[0-9]+');
    Route::delete('{game_id}',
      'GameController@destroy')->where('game_id', '[0-9]+');
    Route::get('{bgg_game_id}/{username?}',
      'GameController@showByBggId')->where('bgg_game_id', '[0-9]+');
    Route::put('{bgg_game_id}',
      'GameController@updateByBggId')->where('bgg_game_id', '[0-9]+');
    Route::delete('{bgg_game_id}',
      'GameController@destroyByBggId')->where('bgg_game_id', '[0-9]+');
    Route::get('search/name/{name}', 'GameController@searchByName');
    Route::get('search/letter/{letter}','GameController@searchByLetter');
});
```

Primjer 94. Usmjerivači igre

Prve tri metode koje se nalaze u usmjerivačima su metode koje sadržavaju pozive na BGG API pomoću kojeg se ispunjava baza podataka. Sve tri metode se pozivaju HTTP akcijom POST da se obilježi unos podataka u bazu iako korisnik to sam ne radi.

Metoda *storeFromBggLibrary* (Primjer 95.) u varijablu *$json* sprema sadržaj koji je dobiven kao odgovor na zahtjev koji je poslan BGG API-u. U zahtjevu je kao parametar poslan username BGG korisnika. Sadržaj dobiven metodom *json_decode()* sprema se u obliku liste u varijablu *$list*. Ako je lista jednaka null to označava da taj BBG korisnik ne postoji. Ako je lista ispunjena igrama, petlja za svaku igru provjerava je li već u bazi podataka, ako nije dodaje ju. Na kraju kao rezultat zahtjeva vraća brojač koliko je igara dodano u bazu podataka sa liste.

```
//GameController.php

public function storeFromBggLibrary(Request $request){
    try {
        $bgg_username = $request->get("bgg_username");
        $url = "https://bgg-json.azurewebsites.net
                  /collection/".$bgg_username;
        $json = file_get_contents($url);
        $list = json_decode($json, true);
        if ($list == null)
            return response([
                  'success' => false,
                  'result' => 'This user does not exist on Board Game Geek'
            ]);
        $counter = 0;
        foreach ($list as $bgg_game) {
            $exist = Game::where('bgg_game_id', '=', $bgg_game['gameId'])
                        ->first();
            if ($exist != null) continue;
            ++$counter;
            Game::create([
                'bgg_game_id' => $bgg_game['gameId'],
                'name' => $bgg_game['name'],
                'image' => $bgg_game['image'],
                'thumbnail' => $bgg_game['thumbnail'],
                'average_rating' => $bgg_game['averageRating'],
                'rank' => $bgg_game['rank'],
                'year_published' => $bgg_game['yearPublished'],
                'min_players' => $bgg_game['minPlayers'],
                'max_players' => $bgg_game['maxPlayers'],
                'playing_time' => $bgg_game['playingTime']
            ]);
        }
        return response()->json([
            'success' => true,
            'result' => ['added' => $counter]
        ]);
    }
    catch (Exception $e){
        return response()->json([
            'success' => false,
            'result' => $e->getMessage()
        ]);
    }
}
```

Primjer 95. Metoda storeFromBggLibrary igre

Sljedećih nekoliko usmjerivača su na metode *index*, *show*, *update* i *destroy*. Ove metode imaju istu svrhu i rade na isti način kao i metode za korisnika, ovaj put nad bazom igara. Razlika je samo u metodi *show* koja ima neobavezni parametar *username*. Ako je parametar *username* ispunjen metoda također vrati informaciju ima li zadani korisnik ovu igru u svojoj knjižnici. Metode *showByBggId*, *updateByBggId*,

destroyByBggId rade isto kao i metode *show*, *update* i *destroy* samo koristi *id* igre sa BGG-a. Posljednje dvije metode služe za pretraživanje igara. Metoda *searchByName* vraća popis svih igara koje u nazivu sadrže traženi pojam. Metoda *searchByLetter* vraća popis svih igara koje započinju sa željenim slovom ili brojem.

5.4 KNJIŽNICA IGARA

Knjižnica (engl. library) igra predstavlja popis svih igara koje korisnik posjeduje. Primjer 96. prikazuje migraciju knjižnice:

```
//2019_05_25_161332_create_libraries_table.php

public function up(){
    Schema::create('libraries', function (Blueprint $table) {
        $table->increments('id');
        $table->integer('user_id');
        $table->integer('game_id');
        $table->timestamps();
    });
}
```

Primjer 96. Migracija za tablicu knjižnica

U primjeru možemo vidjeti polja za vanjske ključeve prema tablicama korisnika i igre. Veza jedan na više prema modelu knjižnice se pojavljuje u modelima korisnika i igre. Jedan korisnika može imati više igara i jedna igra može pripadati većem broju korisnika. Knjižnica će u ovom slučaju imati vezu prema tim modelima sa metodom *belongsTo()*. Metoda označava da jedan resurs knjižnice pripada jednom korisniku, tj. jednoj igri. Primjer 97. je isječak modela knjižnice.

```
//Library.php

class Library extends Model{
    protected $guarded = [];

    public function game(){
        return $this->belongsTo('App\Models\Game');
    }

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

Primjer 97. Model knjižnice

Primjer 98.Primjer 98. Usmjerivači knjižnice prikazuje usmjerivače za metode knjižnice. Metode *index*, *store*, *update* i *destroy* izvršavaju CRUD operacije i rade na istom principu kao i za model korisnika. Metoda *showByUser* vraća popis igara koje zadani korisnik ima u knjižnici. Metoda *showByBggId* vraća popis korisnika koji zadanu igru imaju u knjižnici. Metoda *destroyByUserAndGame* ima istu svrhu kao i metoda *destroy* no kao parametar ne prima id resursa već id-eve vanjskih ključeva i pomoću toga pronalazi pravi resurs koji treba obrisati. Primjer 99. prikazuje isječak metode *showByUser.*

```
//api.php

Route::prefix('libraries')->group(function () {
    Route::get('', 'LibraryController@index');
    Route::post('', 'LibraryController@store');
    Route::get('{library_id}',
      'LibraryController@show')->where('library_id', '[0-9]+');
    Route::put('{library_id}',
      'LibraryController@update')->where('library_id', '[0-9]+');
    Route::delete('{library_id}',
      'LibraryController@destroy')->where('library_id', '[0-9]+');
    Route::get('user/{username}', 'LibraryController@showByUser');
    Route::get('game/{bgg_game_id}',
      'LibraryController@showByBggId')->where('bgg_game_id', '[0-9]+');
    Route::delete('user/{username}/game/{bgg_game_id}',
      'LibraryController@destroyByUserAndGame')
      ->where('bgg_game_id', '[0-9]+');
});
```

Primjer 98. Usmjerivači knjižnice

```
//Library.php

public function showByUser($username){
    $user = User::where('username','=',$username)->first();
    if ($user == null)
        return response()->json([
            'success' => false,
            'result' => "User does not exist."
         ], 404);

    $games = Game::whereHas('libraries', function ($query) use ($user) {
        $query->where('user_id', '=', $user->id);
    })->orderBy('name','asc')->get();

    return response(['success' => true, 'result' => $games], 200);
}
```

Primjer 99. Metoda showByUser knjižnice

Metoda showByUser prvo provjerava postoji li korisnik, ako ne vraća grešku. Zatim stvara popis svih igara koje filtrira metodom *whereHas()*. Metoda *whereHas()* koristi metodu *libraries* iz modela kako bi pristupila knjžnici, te preko modela knjižnice sa metodom *where()* pronazi željenog korisnika. Varijabla *$games* se ispuni listom igara koje korisnik ima u knjižnici i to se vrati kao odgovor na zahtjev.

5.5 PRIJATELJI

Prijatelji (engl. friends) predstavlja vezu između dva korisnika. Korisnik može na svoju listu prijatelja dodati bilo kojeg drugog korisnika. Drugi korisnik ne mora prihvatiti zahtjev i ne smatra se da je sa time prvi korisnik njegov prijatelj. Primjer 100. prikazuje migraciju za tablicu prijatelja.

```
//2019_05_27_180308_create_friends_table.php

public function up(){
    Schema::create('friends', function (Blueprint $table) {
        $table->increments('id');
        $table->integer('user_id')->nullable();
        $table->integer('friend_id')->nullable();
        $table->timestamps();
    });
}
```

Primjer 100. Migracija za tablicu prijatelja

Migracija sadržava dva vanjska ključa koja pokazuju na tablicu korisnika. Prvi vanjski ključ predstavlja prvog korisnika koji je dodao nekoga na listu prijatelja, dok drugi vanjski ključ predstavlja drugog korisnika koji je dodan na popis od prvog korisnika. Primjer 101. prikazuje model prijatelja.

```
//Friend.php

class Friend extends Model {
    protected $guarded = [];

    public function user(){
        return $this->belongsTo('App\Models\User', 'user_id', 'id');
    }

    public function friend(){
        return $this->belongsTo('App\Models\User', 'friend_id', 'id');
    }
}
```

Primjer 101. Model prijatelja

Model prijatelja ima dvije veze na tablice korisnika. Zbog toga je potrebno specificirati naziv vanjskih ključeva iz tablice prijatelja i naziv primarnog ključa iz tablice korisnika. Primjer 102. prikazuje isječak iz koda usmjerivača.

```
//api.php

Route::prefix('friends')->group(function () {
    Route::get('', 'FriendController@index');
    Route::post('', 'FriendController@store');
    Route::get('{friends_id}',
      'FriendController@show')->where('friends_id', '[0-9]+');
    Route::put('{friends_id}',
      'FriendController@update')->where('friends_id', '[0-9]+');
    Route::delete('{friends_id}',
      'FriendController@destroy')->where('friends_id', '[0-9]+');
    Route::get('user/{username}', 'FriendController@showByUser');
    Route::get('friend/{friend_username}',
      'FriendController@showByFriend');
    Route::get('are-friends/user/{username}/friend/{friend_username}',
      'FriendController@areFriends');
    Route::delete('user/{username}/friend/{friend_username}',
       'FriendController@destroyByUserAndFriend');
});
```

Primjer 102. Usmjerivači prijatelja

Metode *index, store, show, update i destroy* rade na isti način kao i metode za model korisnika. Metoda *showByUser* vraća popis svih korisnika koje je zadani korisnik dodao na popis prijatelja. Metoda *showByFriend* vraća popis svih korisnika koji su zadanog korisnika dodali na popis prijatelja. Metoda *areFriends* vraća boolean vrijednost je li prvi zadani korisnik dodao drugog zadanog korisnika na popis prijatelja. Metoda *destroyByUserAndFriend* radi isto što i metoda *destroy* samo što za parametar umjesto *id*-a prijateljstva prima *id*-eve korisnika i njegovog prijatelja te pomoću vanjskih ključeva pronalazi i briše željeni resurs.

5.6 IGRANJE

Igranje (engl. play) prati rezultate igre. Smisao aplikacije je korištenje ove funkcionalnosti kako bi si spremalo i pregledavalo rezultate. Korisnik može evidentirati da je igrao sa drugim korisnicima ili upisati ime neke osobe koja nije registrirana u sustavu. Primjer 103. prikazuje isječak koda iz migracije igranja:

```
//2019_05_27_214938_create_plays_table.php

public function up() {
    Schema::create('plays', function (Blueprint $table) {
        $table->increments('id');
        $table->integer('user_id');
        $table->integer('game_id');
        $table->enum('mode',
            ['SOLO', 'FFA', 'TEAM', 'COOP', 'MASTER'])->nullable();
        $table->integer('duration')->nullable();
        $table->timestamps();
    });
}
```

Primjer 103. Migracija za tablicu igranja

Migracija ima vanjske ključeve na tablice korisnika i igre. Migracija stvara polje enumerator za različite načine (engl. mode) igranja. Moguće je unijeti samo zadane izraze u bazu te se kod validacije podataka u metodi provjerava je li unesen dopušteni izraz.

Igranja mogu biti podijeljena na sljedeće načine:

- SOLO → Igrač igra sam protiv igre.
- FFA → Svaki igrač sam za sebe (engl. Free For All).
- TEAM → Igrači su podijeljeni u timove.
- COOP → Kooperativan način igre.
- MASTER → Igra ima glavnu osobu koja vodi igru, dok svi drugi igraju protiv te osobe.

Model igranja (Primjer 104.) pripada kombinaciji korisnika i igre, tj. resurs sadržava vezu na korisnika koji je stvorio igranje te na igru koja se igrala. Model stvara vezu na igrače i/ili timove koji su sudjelovali u igranju. Primjer 105. prikazuje isječak koda usmjerivača za igranja.

```
//Play.php

class Play extends Model{
    protected $guarded = [];

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

    public function game(){
        return $this->belongsTo('App\Models\Game');
    }

    public function teams(){
        return $this->hasMany('App\Models\Team');
    }

    public function players(){
        return $this->hasMany('App\Models\Player');
    }
}
```

Primjer 104. Model igranja

```
//api.php

Route::prefix('play')->group(function () {
    Route::get('', 'PlayController@index');
    Route::post('', 'PlayController@store');
    Route::get('{play_id}',
      'PlayController@show')->where('play_id', '[0-9]+');
    Route::put('{play_id}',
      'PlayController@update')->where('play_id', '[0-9]+');
    Route::delete('{play_id}',
      'PlayController@destroy')->where('play_id', '[0-9]+');
    Route::get('friends-not-in-play/{play_id}',
      'PlayController@showFriendsNotInPlay')->where('play_id', '[0-9]+');
    Route::get('user/{username}', 'PlayController@showByUser');
});
```

Primjer 105. Usmjerivači igranja

Metode *index, store, show, update i destroy* rade na gotovo isti način kao i metode za korisnika. Metoda *show* dodatno vraća i detaljne informacije o korisniku koji je stvorio igranje, informacije o igri koja se igrala, popis svih igrača i/ili timova koji su sudjelovali u igranju. Metoda *showFriendsNotInPlay* vraća popis svih prijatelja od korisnika koji nisu sudjelovali u ovom igranju, ova metoda može služiti kako bi korisnik lakše pronašao prijatelje koje još nije dodao u igranje, a želi. Metoda *showByUser* vraća popis svih igranja koje je korisnik odigrao.

5.7 IGRAČ

Igrač (engl. player) predstavlja korisnika koji je sudjelovao u igranju. Jedan korisnik može sudjelovati u više igranja i jedno igranje može imati više igrača. Korisnik može sudjelovati u igri i kao dio tima. Primjer 106. prikazuje migraciju igrača.

```
//2019_06_06_000435_create_players_table.php

public function up()
{
    Schema::create('players', function (Blueprint $table) {
        $table->increments('id');
        $table->integer('play_id');
        $table->integer('user_id')->nullable();
        $table->integer('team_id')->nullable();
        $table->string('name');
        $table->boolean('won')->nullable();
        $table->integer('points')->nullable();
        $table->timestamps();
    });
}
```

Primjer 106. Migracija za tablicu igrača

Model ima vanjske ključeve na tablice korisnika, igranja i timova, te polja za ime igrača, je li igrač pobijedio (engl. won) i koliko je bodova (engl. points) sakupio. Obavezna polja su vanjski ključ igranja i ime igrača. Igrač mora sudjelovati u nekoj igri i mora imati ime da se razlikuje od drugih igrača. To može biti postavljeno ime ili ime korisnika. Vanjski ključ korisnika nije obavezno polje. Ne mora svaki igrač biti registriran u sustav ali mora biti moguće dodati takvog korisnika. Vanjski ključ tima nije obavezan jer nije svako igranje timsko. Polje pobjede nije obavezno jer se svaka igra ne igra na pobjedu. Aplikacija može služiti samo za trenutno spremanje bodova nekog igranja. Polje bodova također nije obavezno jer nije svaka igra na bodove. Primjer 107. prikazuje model igrača.

```
//Player.php


class Player extends Model{
    protected $guarded = [];

    public function play(){
        return $this->belongsTo('App\Models\Play');
    }

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

    public function team(){
        return $this->belongsTo('App\Models\Team');
    }
}
```

Primjer 107. Model igrača

Model ima vezu sa tablicama korisnika, igranja i timova. Model igrača može pripadati svakoj od tih tablica. Primjer 108. prikazuje isječak koda usmjerivača za igrače.

```
//api.php

Route::prefix('player')->group(function () {
    Route::get('', 'PlayerController@index');
    Route::post('', 'PlayerController@store');
    Route::get('{player_id}',
      'PlayerController@show')->where('player_id', '[0-9]+');
    Route::put('{player_id}',
      'PlayerController@update')->where('player_id', '[0-9]+');
    Route::delete('{player_id}',
      'PlayerController@destroy')->where('player_id', '[0-9]+');
    Route::get('play/{play_id}',
      'PlayerController@showByPlay')->where('play_id', '[0-9]+');
    Route::get('team/{team_id}',
      'PlayerController@showByTeam')->where('team_id', '[0-9]+');
    Route::get('user/{user_id}',
      'PlayerController@showByUser')->where('user_id', '[0-9]+');
});
```

Primjer 108. Usmjerivači igrača

Metode *index, store, show, update i destroy* funkcioniraju gotovo isto kao i kod metoda korisnika. Metoda *show* vraća dodatne informacije o korisniku, igranju, igri i timu u kojem je igrač sudjelovao. Metoda *showByPlay* vraća popis svih igrača koji su

sudjelovali u igranju. Metoda *showByTeam* vraća popis svih igrača nekog tima. Metoda *showByUser* vraća popis svih igrača gdje je označen zadani korisnik.

5.8 TIM

Tim (engl. team) predstavlja skupinu igrača koji su sudjelovali u nekom igranju. Tim ne mora postojati za svako igranje već samo za timske igre. Primjer 109. prikazuje migraciju za timove.

```
//2019_06_06_001019_create_teams_table.php

public function up() {
    Schema::create('teams', function (Blueprint $table) {
        $table->increments('id');
        $table->integer('play_id');
        $table->string('name');
        $table->boolean('won')->nullable();
        $table->integer('points')->nullable();
        $table->timestamps();
    });
}
```

Primjer 109. Migracija za tablicu timova

Migracija sadržava vanjski ključ na tablicu igranja, polja imena tima, pobjednika i broj bodova tima. Polja funkcioniraju isto kao i ta polja kod igrača. Primjer 110 prikazuje isječak koda modela tima.

```
//Team.php

class Team extends Model{
    protected $guarded = [];

    public function play(){
        return $this->belongsTo('App\Models\Play');
    }

    public function players(){
        return $this->hasMany('App\Models\Player');
    }
}
```

Primjer 110. Model tima

Model ima vezu na modele igranja i igrača. Tim pripada jednom igranju. Tim može imati više igrača u sebi. Primjer 111. prikazuje usmjerivače za timove.

```
//api.php

Route::prefix('team')->group(function () {
    Route::get('', 'TeamController@index');
    Route::post('', 'TeamController@store');
    Route::get('{team_id}',
      'TeamController@show')->where('team_id', '[0-9]+');
    Route::put('{team_id}',
      'TeamController@update')->where('team_id', '[0-9]+');
    Route::delete('{team_id}',
      'TeamController@destroy')->where('team_id', '[0-9]+');
    Route::get('play/{play_id}',
      'TeamController@showByPlay')->where('play_id', '[0-9]+');
});
```

Primjer 111. Usmjerivači tima

Metode *index, store, show, update i destroy* funkcioniraju gotovo isto kao i kod metoda korisnika. Metoda *showByPlay* vraća popis svih timova koji su sudjelovali u nekom igranju.

Ovime su opisane sve tablice, modeli, usmjerivači i neki bitniji upravljači ovog API servisa. Kao što je ranije opisano osoba može koristiti ovaj servis po potrebi. Krajnji korisnik vjerojatno nikad neće koristiti sam API kako bi unio ili pročitao podatke iz baze. Sljedeća cjelina ukratko prikazuje kako se ovakav API može primijeniti da bude dostupan krajnjem korisniku.

6. PRIMJER KORIŠTENJA RAZVIJENOG API SERVISA

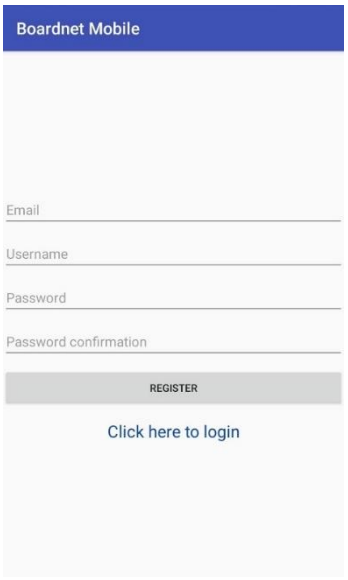
6.1 OPIS KORIŠTENJA

Razvijeni API servis sam po sebi nema veliku primjenu za krajnjeg korisnika, ali pomoću njega može se stvoriti mnogo različitih stolnih, mobilnih ili Web aplikacija bez da se mora brinuti o programiranju logike, spajanju na bazu podataka ili sličnih koraka koji se moraju napraviti kod razvoja uobičajenih aplikacija. Dovoljno je stvoriti prave forme za unos podataka i prikazati podatke koje API pruža. Da bi se prikazao razvoj jedne takve aplikacije stvorena je Android aplikacija koja koristi razvijeni API servis. (DiMarzio, 2017)

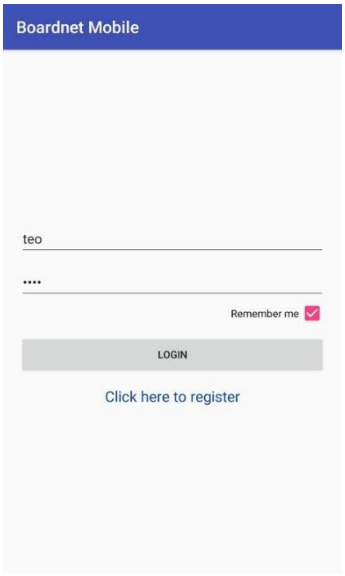
Sljedeća poglavlja će prikazati dohvaćanje podataka sa API-a i slanje podataka u zahtjevu na API. Cjelina je napisana na način da se podrazumijeva da čitatelj rada zna razvoj Android mobilne aplikacije. Primjeri koda i opisi se bave logikom korištenja API-a, a ne razvojem stvaranja forme i prikazivanja podataka u mobilnoj aplikaciji.

6.2 REGISTRACIJA I PRIJAVA

Za korištenja većine metoda API-a potreban je token za autentifikaciju i autorizaciju. Mobilna aplikacija ima forme za registraciju i prijavu. Slika 5. prikazuje formu za registraciju. Slika 6. prikazuje formu za prijavu.



Slika 5. Zaslon registracije



Slika 6. Zaslon prijave

Primjer 112. prikazuje isječak koda za stvaranje zahtjeva registracije korisnika.

```
//Register.java

RequestQueue requestQueue = Volley.newRequestQueue(Register.this);
Map<String, String> params = new HashMap<String, String>();
params.put("email", emailBox.getText().toString());
params.put("username", usernameBox.getText().toString());
params.put("password", passwordBox.getText().toString());
params.put("password_confirmation",
passwordConfirmationBox.getText().toString());
JsonObjectRequest jsonObjectRequest = new JsonObjectRequest(
        Request.Method.POST,
        URL + "/auth/register",
        new JSONObject(params),
        new Response.Listener<JSONObject>() {
            @Override
            public void onResponse(JSONObject response) {
                try {
                    onSuccessDo(response);
                } catch (JSONException e) {
                    Toast.makeText(Register.this,
                        e.toString(), Toast.LENGTH_LONG).show();
                    progress.dismiss();
                }
            }
        },
        new Response.ErrorListener() {
            @Override
            public void onErrorResponse(VolleyError e) {
                onErrorDo(e);
            }
        });
requestQueue.add(jsonObjectRequest);
```

Primjer 112 Stvaranje zahtjeva za registraciju

Stvaranje i slanje zahtjeva se sastoji od nekoliko koraka. Nakon što korisnik upiše podatke u formu sa Slika 5. i klikne na tipku „REGISTER“ (hrv. registriraj) pokreće se metoda registracije. Za slanje zahtjeva potrebno je ispuniti *RequestQueue*. *RequestQueue* je klasa knjižnice *Volley*. Radi asinkrono i stvara dretvu (engl. thread) u kojoj će se poslati zahtjev i primiti odgovor. Zahtjev je potrebno ispuniti u obliku *JsonObjectRequest*-a. *JsonObjectRequest* prima nekoliko argumenata. (Segato, 2015)

Prvi argument je HTTP akcija koja poziva željeni zahtjev. Kako je ranije napomenuto za registraciju je potrebno koristiti POST akciju. Drugi argument je URL na koji se zahtjev šalje. U primjeru je varijabla ispunjena URL-om API servisa u što spada i

prefiks */api*. Za registraciju se zadani URL popuni sa usmjerivačem za registraciju, tj. */auth/register*. Treći argument je *JSONObject* koji sadržava parametre koji se šalju u tijelu zahtjeva. Parametri se u *JSONObject* unose u obliku mape koja za ključ i za vrijednost ima tip String, tj. niz znakova. Kako JSON prihvaća bilo koji tip podatka nije potrebno vrijednosti u mapi označavati sa njihovim pravim tipom, kao što su Integer za broj, DateTime za datum i slično. Takav tip pristupa uvelike olakšava stvaranje, slanje i čitanje podataka. Ako postoji potreba da programer iskoristi točno određeni tip podataka ima mogućnost to napraviti. Četvrti argument obrađuje odgovor ako je primljen HTTP status kod 200, tj. status kod koji označava da je API uspješno obradio zahtjev i vratio odgovor. Peti argument obrađuje odgovor ako je primljen bilo koji drugi HTTP status kod osim 200, tj. status kod koji označava da API nije uspješno obradio zahtjev i vratio odgovor. U primjeru vidimo da se u četvrtom i petom argumentu pozivaju dodatne metode koje obrađuju odgovor. Primjer 113 prikazuje isječak koda koji se izvršava ako je zahtjev uspješno izvršen.

```
//Register.java

private void onSuccessDo(JSONObject response) throws JSONException {
    if (response.getString("success").equals("true")) {
        Toast.makeText(Register.this,
            "Registration successful",
            Toast.LENGTH_LONG).show();
        Intent myIntent = new Intent(getBaseContext(), Login.class);
        myIntent.putExtra("username",
            response.getJSONObject("user").getString("username"));
        startActivity(myIntent);
    } else {
        Toast.makeText(Register.this,
            response.getString("errors"),
            Toast.LENGTH_LONG).show();
    }
}
```

Primjer 113. Uspješno izvršavanje zahtjeva

Odgovor na zahtjev je spremljen u varijablu *response*. Podacima u toj varijabli možemo pristupiti metodama *get(), getString(), getInteger(), getBoolean(), getJSONObject(), getJSONArray()* i sličnim. Kao argument metode se unosi ključ polja koje pokušavamo dohvatiti. Kod uspješno izvršenog zahtjeva izvršava se metoda koja dohvaća JSON objekt *user* te iz njega korisničko ime 'username'. To korisničko ime se prosljeđuje dalje

na novu aktivnost prijave gdje se automatski upotrebljava za ispunjavanje polja korisničkog imena.

Ako zahtjev nije uspješno izvršen dobiva se neki HTTP status kod koji ne spada u grupu 200. Te statuse je potrebno zasebno obraditi. Za najčešće statuse napisane su poruke. Primjer 114. prikazuje metodu koja se izvršava u petom argumentu, tj. ako zahtjev nije uspješno obrađen.

```
//Register.java

private void onErrorDo(VolleyError e) {
    if (e.networkResponse.statusCode == 400) {
        Toast.makeText(Register.this,
            "Error 400: Bad request.",
             Toast.LENGTH_LONG).show();
    } else if (e.networkResponse.statusCode == 401) {
        Toast.makeText(Register.this,
            "Error 401: The request has not been applied because it lacks
            valid authentication credentials for the target resource.",
            Toast.LENGTH_LONG).show();
    } else if (e.networkResponse.statusCode == 403) {
        Toast.makeText(Register.this,
            "Error 403: The server understood the request but refuses to
            authorize it.",
            Toast.LENGTH_LONG).show();
    } else if (e.networkResponse.statusCode == 404) {
        Toast.makeText(Register.this,
            "Error 404: Requested resource not found",
            Toast.LENGTH_LONG).show();
    } else if (e.networkResponse.statusCode == 500) {
        Toast.makeText(Register.this,
            "Error 500: Something went wrong at server end",
             Toast.LENGTH_LONG).show();
    } else {
        Toast.makeText(Register.this,
            "Error: " + e.toString(),
            Toast.LENGTH_LONG).show();
    }
    progress.dismiss();
}
```

Primjer 114. Ne uspješno izvršavanje zahtjeva

Sljedeći primjer prikazuje isječak koda koji se dešava u metodi u četvrtom argumentu zahtjeva za prijavu, tj. u metodi uspješno izvršenog zahtjeva za prijavu. Prvi argument je HTTP akcija POST. Drugi je poziv usmjerivač za prijavu. Treći su parametri username i password za prijavu korisnika. Peti je isti kod svih zahtjeva. Primjer 115. prikazuje metodu koja se pokreće kod uspješno izvršenog zahtjeva za prijavu.

```
//Login.java

private void onSuccessDo(JSONObject response) {
    try {
        if (response.getString("response").equals("success")) {
            preferences.edit().putString(
                  "token",
                  response.getJSONObject("result").getString("token")
            ).apply();
            preferences.edit().putString(
                  "username",
                  usernameBox.getText().toString()
            ).apply();
            finish();
            startActivity(new Intent(Login.this, MainActivity.class));
        } else {
            try {
                if (response.getString("message")
                        .equals("invalid_credentials")) {
                    Toast.makeText(Login.this,
                        "Wrong username or password.",
                        Toast.LENGTH_LONG).show();
                } else if (response.getString("message")
                        .equals("failed_to_create_token")) {
                    Toast.makeText(Login.this,
                        "Failed to create token.",
                        Toast.LENGTH_LONG).show();
                }
            } catch (JSONException e) {
                Toast.makeText(Login.this,
                  e.toString(), Toast.LENGTH_LONG).show();
            }
        }
    } catch (JSONException e) {
        Toast.makeText(Login.this,
            e.toString(), Toast.LENGTH_LONG).show();
    }
}
```

Primjer 115. Uspješno izvršen zahtjev prijave na API

Varijabla preferences označava unutarnju memoriju aplikacije i vrijednosti unutar te varijable će se prenositi kroz sve aktivnosti. Ako je zahtjev bio uspješan pod ključ *token* unutar prefrences se sprema token koji je korisnik dobio kao rezultat prijave. Token se koristiti u aplikaciji za autentifikaciju i autorizaciju ostalih zahtjeva. Također sprema se i username koji se koristi u kreiranju zahtjeva za prijavljenog korisnika.

6.3 PRIKAZ I SLANJE PODATAKA

Većinu stranica mobilne aplikacije prikazivati će neke podatke koje je aplikacija primila od API-a. Podatke mobilna aplikacija prima kao odgovor na zahtjev u JSON formatu. Podatke je potrebno raščlaniti. Za takav postupak potrebno je znati točno kakve se podatke prima od API-a i pod kojim su ključem. Uspješno raščlanjene podatke programer može upotrijebiti na bilo koji način. API često pruža više podataka nego što je programeru potrebno u tom trenutku. Detaljni podaci se šalju kako se svrha nekog zahtjeva ne bi ograničila na samo taj jedan zadatak. Gotovo svi podaci za mobilnu aplikaciju dohvaćeni su HTTP GET akcijom. Primjer 116. prikazuje dohvaćanje i prikaz podataka za profil korisnika. (Kapil, 2016)

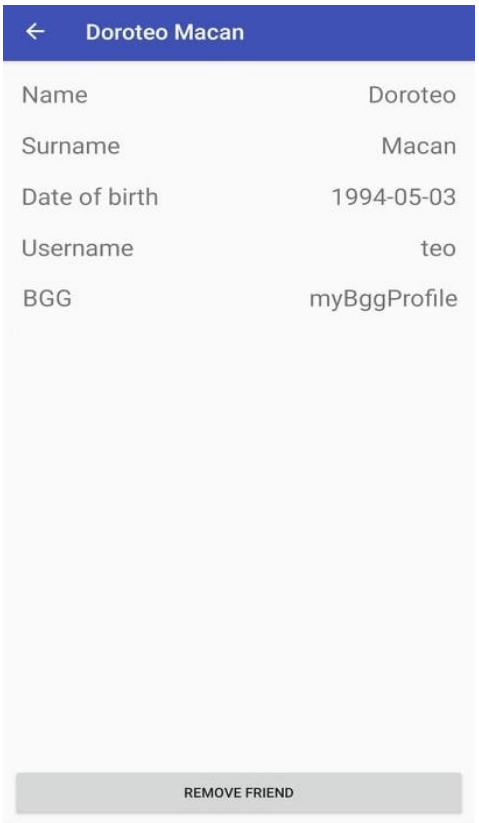
```
JsonObjectRequest jsonObjectRequest = new JsonObjectRequest(
      Request.Method.GET,
      URL + "/users/" + preferences.getString("username", ""),
      null,
      new Response.Listener<JSONObject>() {
            @Override
            public void onResponse(JSONObject response) {
                  try {
                        if (response.getBoolean("success")) {
nameEditText.setText(response.getJSONObject("result").getString("name"));
surnameEditText.setText(response.getJSONObject("result").getString("surna
me"));
dateOfBirthEditText.setText(response.getJSONObject("result").getString("d
ate_of_birth").substring(0, 10));
usernameEditText.setText(response.getJSONObject("result").getString("user
name"));
emailEditText.setText(response.getJSONObject("result").getString("email")
);
bggUsernameEditText.setText(response.getJSONObject("result").getString("b
gg_username"));
user_id= response.getJSONObject("result").getInt("id");
                        } else {
Toast.makeText(Profile.this, response.getString("result"),
Toast.LENGTH_LONG).show();
                        }
                  } catch (JSONException e) {
Toast.makeText(Profile.this, "Error: " + e.toString(),
Toast.LENGTH_LONG).show();
                  }
            }
      },
      new Response.ErrorListener() {
            @Override
            public void onErrorResponse(VolleyError e) {
```

```
                    errorResponse(e, Profile.this);
                    if (e.networkResponse.statusCode == 401) {
                          finish();
                          Intent myIntent = new Intent(
                                getBaseContext(),
                                Login.class);
                          startActivity(myIntent);
                    }
              }
        }) {
        @Override
        public Map<String, String> getHeaders() {
                    HashMap<String, String> header =
                          new HashMap<String, String>();
                    header.put("Authorization", "Bearer " +
                          preferences.getString("token", ""));
              return header;
        }
};
```

Primjer 116. Dohvaćanje i prikaz podataka u mobilnoj aplikaciji

Primljeni odgovor na zahtjev je raščlanjen i pomoću ključa su dohvaćeni podaci koji su iskorišteni za prikaz profila korisnika. Slika 7. i Slika 8. prikazuju različite prikaze podataka u mobilnoj aplikaciji koji su dobiveni kao odgovor istog API zahtjeva. Kako bi

Slika 7. Zaslon profila korisnika



Slika 8. Zaslon profila prijatelja

korisnik dobio ove podatke od API-a potrebno je da se zahtjev autentificira. Zaglavlje zahtjeva se ispuni sa autorizacijskim tokenom koji je korisnik dobio pri prijavi u sustav.

Podaci se najčešće stvaraju HTTP akcijom POST i ažuriraju HTTP akcijom PUT. POST i PUT akcija rade na sličan način. Za obije akcije se očekuje da su podaci proslijeđeni u tijelu zahtjeva u obliku parametara u JSON formatu. Primjer 117. prikazuje kako kreirati podatke sa željenim ključem te kako te podatke prosljediti u zahtjevu. Mobilna aplikacija prima odgovor od API-a isto kao da je poslala HTTP GET zahtjev. Dobiveni odgovor može upotrijebiti po potrebi.

```
Map<String, String> params = new HashMap<String, String>();
params.put("username", usernameEditText.getText().toString());
params.put("email", emailEditText.getText().toString());
params.put("name", nameEditText.getText().toString());
params.put("surname", surnameEditText.getText().toString());
params.put("bgg_username", bggUsernameEditText.getText().toString());
params.put("date_of_birth", dateOfBirthEditText.getText().toString());

JsonObjectRequest jsonObjectRequest = new JsonObjectRequest(
      Request.Method.PUT,
      URL + "/users/" + user_id,
      new JSONObject(params),
      new Response.Listener<JSONObject>() {
      @Override
      public void onResponse(JSONObject response) {
      try {
            if (response.getBoolean("success")) {
                  preferences.edit().putString(
                        "username",
                        response.getJSONObject("result")
                              .getString("username")).apply();
                  Toast.makeText(Profile.this,
                        "Save successful", Toast.LENGTH_LONG).show();
            }
            ...
);
```

Primjer 117. Ažuriranje podataka korisnika

7. ZAKLJUČAK

REST API aplikacija daje programerima slobodu kod razvoja i korištenja API. Da je aplikacija koja je razvijena za ovaj rad napisana u nekom drugom programskom jeziku i u nekom drugom okruženju rezultat bi i dalje mogao biti potpuno isti. Korisnik API-a ne može primjetiti razliku između API-a koji je napisan u Laravel okruženju od API-a koji su napisani npr. u .NET ili Spring okruženju. Također programer koji koristi API kao back-end neke aplikacije nije ograničen gdje taj API može upotrijebiti. Kao primjer korištenja razvijenog API-a uz ovaj rad je razvijena Android aplikacija, no isto tako su mogle biti razvijene Web, stolne i mobilne aplikacije koje bi izgledale i bile korištene na drugačiji način. Sloboda koja se dobiva kod razvoja REST API aplikacija omogućava programeru da radi u željenom programskom jeziku. Tvrtkama takav način razvoja može uštedjeti novac i olakšati održavanje aplikacija. Ono što dopušta slobodu kod takvog načina rada je JSON format za prijenos podataka. Sva okruženja znaju raditi sa ovim formatom podataka jer je jednostavan za raščlanjivanje. Gotovo svaki objekt je moguće pretvoriti u JSON format. Prijenos tih podataka internet mrežom je podatkovno jeftiniji nego neki drugi formati prijenosa. Sve te pogodnosti, od slobode korištenja bilo koje tehnologije pri razvoju ili korištenju API-a, jednostavnosti i niskoj zahtjevnosti razvoja, do jeftinog prijenosa podataka pogodovale su popularizaciji razvoju aplikacija kao REST API servisa.

DODATAK

Tablica 1. Popis svih metoda API aplikacije

HTTP Akcija	URL	Metoda
Registracija i prijava		
POST	api/auth/login	...\Auth\ApiAuthController@login
POST	api/auth/register	...\Auth\ApiRegisterController@register
Korisnik		
GET	api/users	...\UserController@index
GET	api/users/search/name/{name}	...\UserController@searchByName
GET	api/users/search/username/{username}	...\UserController@searchByUsername
GET	api/users/{user_id}	...\UserController@show
DELETE	api/users/{user_id}	...\UserController@destroy
PUT	api/users/{user_id}	...\UserController@update
GET	api/users/{username}	...\UserController@showByUsername
Igra		
GET	api/games	...\GameController@index
POST	api/games/bgg/hotness	...\GameController@storeFromBggHotness
POST	api/games/bgg/id/{bgg_game_id}	...\GameController@storeFromBggGame
POST	api/games/bgg/library/{bgg_username}	...\GameController@storeFromBggLibrary
GET	api/games/search/letter/{letter}	...\GameController@searchByLetter
GET	api/games/search/name/{name}	...\GameController@searchByName
PUT	api/games/{bgg_game_id}	...\GameController@updateByBggId
DELETE	api/games/{bgg_game_id}	...\GameController@destroyByBggId
GET	api/games/{bgg_game_id}/{username?}	...\GameController@showByBggId
PUT	api/games/{game_id}	...\GameController@update
DELETE	api/games/{game_id}	...\GameController@destroy
GET	api/games/{game_id}/{username?}	...\GameController@showByBggId
Knjižnica		
GET	api/libraries	...\LibraryController@index
POST	api/libraries	...\LibraryController@store
GET	api/libraries/game/{bgg_game_id}	...\LibraryController@showByBggId

GET	api/libraries/user/{username}	...\LibraryController@showByUser
DELETE	api/libraries/user/{username}/game/{bgg_game_id}	...\LibraryController@destroyByUserAndGame
DELETE	api/libraries/{library_id}	...\LibraryController@destroy
PUT	api/libraries/{library_id}	...\LibraryController@update
GET	api/libraries/{library_id}	...\LibraryController@show
Prijatelj		
POST	api/friends	...\FriendController@store
GET	api/friends	...\FriendController@index
GET	api/friends/arefriends/user/{username}/friend/{friend_username}	...\FriendController@areFriends
GET	api/friends/friend/{friend_username}	...\FriendController@showByFriend
GET	api/friends/user/{username}	...\FriendController@showByUser
DELETE	api/friends/user/{username}/friend/{friend_username}	...\FriendController@destroyByUserAndFriend
DELETE	api/friends/{friends_id}	...\FriendController@destroy
PUT	api/friends/{friends_id}	...\FriendController@update
GET	api/friends/{friends_id}	...\FriendController@show
Igranje		
POST	api/play	...\PlayController@store
GET	api/play	...\PlayController@index
GET	api/play/friends-not-in-play/{play_id}	...\PlayController@showFriendsNotInPlay
GET	api/play/user/{username}	...\PlayController@showByUser
DELETE	api/play/{play_id}	...\PlayController@destroy
PUT	api/play/{play_id}	...\PlayController@update
GET	api/play/{play_id}	...\PlayController@show
Igrač		
POST	api/player	...\PlayerController@store
GET	api/player	...\PlayerController@index
GET	api/player/play/{play_id}	...\PlayerController@showByPlay
GET	api/player/team/{team_id}	...\PlayerController@showByTeam
GET	api/player/user/{user_id}	...\PlayerController@showByUser
GET	api/player/{player_id}	...\PlayerController@show
PUT	api/player/{player_id}	...\PlayerController@update
DELETE	api/player/{player_id}	...\PlayerController@destroy

Tim		
POST	api/team	...\TeamController@store
GET	api/team	...\TeamController@index
GET	api/team/play/{play_id}	...\TeamController@ showByPlay
GET	api/team/{team_id}	...\TeamController@show
PUT	api/team/{team_id}	...\TeamController@update
DELETE	api/team/{team_id}	...\TeamController@destroy

LITERATURA

Apigee, 2018. *Web API Design: The Missing Link.* [Mrežno]
Dostupno na: https://cloud.google.com/files/apigee/apigee-web-api-design-the-missing-link-ebook.pdf
[Pokušaj pristupa 26 srpanj 2019].

Auth0, 2019. *JWT.* [Mrežno]
Dostupno na: https://jwt.io/
[Pokušaj pristupa 28 srpanj 2019].

Cooksey, B., 2014. *An Introduction to APIs.* s.l.:Zapier.

DiMarzio, J. F., 2017. *Android Programing with Andrid Studio.* Indianapolis: John Wiley & Sons, Inc..

Kapil, 2016. *Android Volley Tutorial.* [Mrežno]
Dostupno na: http://www.androidtutorialpoint.com/networking/android-volley-tutorial/
[Pokušaj pristupa 20 srpanj 2019].

Massé, M., 2012. *REST API Design Rulebook.* Sebastopol: O'Reilly Media, Inc..

McCool, S., 2012. *Laravel Starter.* Birmingham: Packt Publishing Ltd..

Mundosaparte, D., 2019. *Laravel Documentation - 5.8.* [Mrežno]
Dostupno na: https://laravel.com/docs/
[Pokušaj pristupa 15 srpanj 2019].

Peyrott, S. E., 2018. *The JWT Handbook.* s.l.:Auth0 Inc..

Rees, D., 2016. *Laravel: Code Bright.* s.l.:Leanpub.

Richardson, L. & Amundsen, M., 2013. *RESTful Web APIs.* Sebastopol: O'Reilly Media, Inc..

Richardson, L. & Ruby, S., 2007. *RESTful Web Services.* Sebastopol: O'Reilly Media, Inc..

Segato, G., 2015. *An Introduction to Volley.* [Mrežno]
Dostupno na: https://code.tutsplus.com/tutorials/an-introduction-to-volley--cms-23800
[Pokušaj pristupa 22 srpanj 2019].

Stauffer, M., 2017. *Laravel: Up and Running.* Sebastopol: O'Reilly Media, Inc..

Vo, J., 2014. *Learning Laravel: The Easiest Way.* s.l.:Jack Vo.

POPIS SLIKA

POPIS PRIMJERA

SAŽETAK

Ovaj rad se bavi razvojem REST API servisa u PHP okruženju Laravel. Laravel koristi MVC arhitekturu razvoja aplikacija. Servis je upotrijebljen u Android mobilnoj aplikaciji. Razvijeni API se može koristiti za bilježenje rezultata odigranih društvenih igrara. Podaci su spremljeni na MySQL bazu podataka koja je stvorena „model-first“ pristupom. Razvoj API servisa i mobilne aplikacije popračen je isječcima koda koji su detaljno opisani.

Ključne riječi: REST API, Laravel, okruženje, MVC, web aplikacija, PHP, MySQL, Android aplikacija

ABSTRACT

This thesis deals with the development of REST API service in the PHP framework Laravel. Laravel uses the MVC architecture for aplication development. System is used in the Android mobile application. The developed API can be used to record the results of borard games played. The data is stored on MySQL databases which is created by a "model-first" approach. The API service and mobile application development is accompanied with code snippets that are described in detail.

Keywords: REST API, Laravel, framework, MVC, web aplication, PHP, MySQL, Android aplication