Razvoj aplikacije u Laravel okruženju

Dumbović, Valentina

SVEUČILIŠTE U ZAGREBU

PRIRODOSLOVNO–MATEMATIČKI FAKULTET

MATEMATIČKI ODSJEK

Valentina Dumbović

RAZVOJ APLIKACIJE U LARAVEL OKRUŽENJU

Diplomski rad

Voditelj rada:
Dr. sc. Goran Igaly

Zagreb, veljača 2017.

Ovaj diplomski rad obranjen je dana _____ pred ispitnim povjerenstvom u sastavu:

1. _____, predsjednik
2. _____, član
3. _____, član

Povjerenstvo je rad ocijenilo ocjenom _____.

Potpisi članova povjerenstva:

1. _____
2. _____
3. _____

Sadržaj

Uvod

U današnje vrijeme PHP dobiva titulu kao najpopularniji programski jezik diljem svijeta, a s nedavnom objavom njegove najnovije verzije PHP 7 postaje još bolji i stabilniji. Za velike projekte poput web stranica tvrtke ili web aplikacije koju koristi milijunski broj ljudi diljem svijeta, razvojni programeri (*developer*) rado biraju PHP. Razvoj računalnih znanosti i današnja potražnja za što brži i moderniji razvoj projekta je dovela do stvaranja razvojnog okruženja. Svaki zahtjevniji projekt ima mnogo redaka koda i ponavljajućih dijelova te tako projekt lako postaje nepregledan i neodrživ na dulje staze. PHP razvojno okruženje ubrzava razvoj projekta, pruža sigurno polazište, pomaže održati čitljivost i logiku strukture koda i time olakšava njegovo kasnije održavanje. Među mnoštvom razvojnih okruženjima danas ističe se Laravel čija popularnost eksponencijalno raste od svoje prve objave 2011. godine. Kao besplatno razvojno okruženje otvorenog koda Laravel je dostupan svima na Internetu i svoju popularnost zaslužuje svojom jednostavnošću, izražajnim i jasnim kodom i velikom zajednicom.

Struktura diplomskog rada

Poglavlje 1 govori o povijesti programskog jezika PHP, objašnjava gdje se koristi i ukratko opisuje osnove poput varijabli, funkcija.

Poglavlje 2 opisuje pojam razvojnog okruženja. Objašnjeno je zašto je nastalo, kada ga koristiti i koje su njegove prednosti odnosno mane naspram razvoja aplikacija iz temelja.

Poglavlje 3 objašnjava najčešće korišten oblikovni obrazac MVC kojeg koristi i Laravel. Predstavljeni su glavni pojmovi poput modela, pogleda.

Poglavlje 4 posvećeno je razvojnom okruženju Laravel. Započinje uvodom u povijest njegovog nastanka i razvoja te govori o njegovim posebnostima. Opisani su pomoćni alati poput Composer-a te novosti koje Laravel donosi poput Blade-a i Eloquent ORM.

Poglavlje 5 se detaljno bavi opisom najvažnijih dijelova Laravela. Opisan je proces instalacije, izgled novonastalog projekta u Laravelu, detaljno se objašnjavaju najvažniji pojmovi i uz pomoć primjera prikazuje njihovo izvršavanje.

Poglavlje 1

PHP

1.1 Uvod u PHP

"...PHP je lako dostupan i s njime se može raditi bilo što. U današnje vrijeme nije potrebno imati stručno znanje iz kompjuterskih znanosti kako bi stvorili nešto korisno i potrebno. Potrebna vam je samo knjiga poput ove, zajednica ljudi koji vam mogu pomoći, malo upornosti i zasukati rukave i već ste na putu stvaranja potpuno novog alata." Michael Bourque, VP, PTC, Organizator Boston PHP User Group ([6]).

Povijest PHP-a

PHP (*Hypertext Preprocessor*) je skriptni programski jezik interpreterskog tipa koji se koristi na sljedeća tri načina:

- skriptiranje sa strane poslužitelja
- skriptiranje naredbenog retka
- stvaranje GUI aplikacija sa klijentske strane

Prva verzija PHP-a je nastala 1994. kao skup Perl skripti koje je razvio Rasmus Lerdorf u svrhu brojanja posjeta na vlastitoj web stranici te ju nazvao PHP/FI (*Personal Home Page Tools/Forms Interpreter*). Kroz povijest je doživio četiri glavne velike promjene koje su dovele do složenog, kvalitetnog proizvoda kakav je danas.

Lerdorf je zbog povećane potrebe za funkcijama razvio novu verziju u programskom jeziku C koja je omogućavala rad s bazom podataka i stvaranje dinamičkih web stranica. 1995. godine je odlučio objaviti izvorni kod nove verzije PHP 1.0 i time omogućio razvojnim programerima slobodu korištenja njegovog proizvoda u vlastite svrhe i kako bi ga zajedno nastavili razvijati i poboljšavati. Neke od osnovnih funkcionalnosti modernog

PHP-a su korištenje varijabli po uzoru na Perl, varijable primljene iz HTTP formi se automatski obrađuju, omogućeno je uključivanje HTML sintakse.
U sljedećih godinu dana fokus razvoja se promijenio i 1996. godine objavljena je verzija PHP 2.0. Po prvi put se koristi izraz "skriptni jezik". U ovom izdanju, između ostalog je uvedena podrška za DBM (*Database Manager*), baze podataka Postgres95, cookies, mSQL i podrška za funkcije definirane od strane korisnika.

Nakon objave verzije PHP 3.0 1998. godine koju su stvorili Andi Gutmans i Zeev Suraski, zajednica korisnika se naglo povećala sa nekoliko tisuća na desetke tisuća korisnika i stotine tisuća web poslužitelja diljem svijeta. Bila je to prva verzija koja podsjeća na današnji PHP. Dvojac je u suradnji s Lerdorfom nastavio na razvoju PHP-a kako bi dodali nove funkcionalnosti koje su im tada nedostajale i stvoriti posve novi, samostalan programski jezik. Nova svojstva poput mogućnosti korištenja različitih baza podataka, protokola, API-ja, dodavanja novih funkcionalnosti, objektno orijentiranog programiranja i konzistentna sintaksa zaslužne su za veliki rast popularnosti treće verzije.

Konačno, četvrta verzija je izdana 2000. godine pod utjecajem više razvojnih programera, među kojima su glavni bili Suraski i Gutmans. Sloj između samog jezika i web poslužitelja se apstrahirao, uvedena je podrška za HTTP sesije, dana je podrška za mnoštvo web poslužitelja i dodan je napredniji sustav analiziranja oznaka (*tag*) koji se sastoji od faze sintaksne analize i faze izvršavanja pod novim imenom Zend engine.

Danas se koristi verzija PHP 5.6, s početnom verzijom 5.0 iz 2004. godine, s time da je 1. prosinca 2016. objavljena verzija 7.1. PHP 7.0. Najnovija verzija je najveće ažuriranje objavljeno u povijesti PHP-a i donosi velika poboljšanja poput dvostruke brzine naspram PHP 5.0, smanjeno korištenje memorije, dosljedne 64-bitna podrške, anonimne klase, poboljšane hijerarhije iznimaka i još puno više. Može se sa sigurnošću reći da se danas PHP koristi na nekoliko desetaka milijuna domena diljem svijeta i taj broj neprekidno raste.

Primjena PHP-a

PHP ima tri primjene od kojih se najčešće koristi ona za skriptiranje sa strane poslužitelja. Prva namjena PHP-a je upravo bilo kreiranje dinamičnih web stranica. Poseban PHP interpreter prolazi kroz cijelu web stranicu, pronalazi mjesta na kojima su označene PHP naredbe i izvršava ih, nakon čega se ta prerađena stranica šalje klijentu odnosno web pregledniku. Uglavnom se kombinira sa web poslužiteljima (server) kao što su Apache i nginx. Danas je PHP popularan alat i za stvaranje XML dokumenata, grafike, PDF dokumenata i više.

Rjeđe se koristi za skriptiranje naredbenog retka i stvaranje GUI aplikacija sa klijentske strane. Moguće je izvoditi skripte iz naredbenog retka za zadatke koji se tiču sistemske administracije poput stvaranja sigurnosne kopije podataka. Također, koristeći ekstenziju PHP-GTK mogu se stvarati GUI aplikacije za različite platforme.

1.2 Osnove PHP-a

PHP stranica je poput obične HTML stranice uz dodatne PHP naredbe koje se miješaju sa HTML naredbama. PHP skripte imaju poseban nastavak po kojemu se prepoznaju: ".php" i naredbe pisane u PHP-u su izdvojene u posebnim oznakama (*tag*) koje označavaju početak i kraj naredbe, s time da se pojedina naredba može isprepletati sa HTML naredbama:

```
<html>
    <head>
        <title>PHP skripta</title>
    </head>
    <body>
        <?php echo "Tekst koji ispisuje PHP naredba!"; ?>
    </body>
</html>
```

Web poslužitelj će prepoznati nastavak ".php" i započeti dodatnu obradu skripte od strane PHP interpretera i slati odgovor web pregledniku u obliku HTML skripte. PHP naredbe se smještaju unutar oznaka "<?*php*" i "? >" i međusobno odvajaju oznakom ";", a njihova interpretacija se unosi u HTML skriptu na predviđeno mjesto.

Nazivi klasa i funkcija koje definira korisnik i ugrađenih ključnih riječi poput echo, class, while i slično, nisu osjetljivi na velika i mala slova. Nazivi varijabli započinju sa znakom "$" i bilo kojim slovom ili znakom "_" (ne smiju započinjati brojem no mogu sadržavati broj) i osjetljivi su na velika i mala slova.

```
//ekvivalentni pozivi
echo("ispis teksta");
ECHO("ispis teksta");
EcHo("ispis teksta");
//razlicite varijable
$varijabla = 1;
$Varijabla = 2;
$VARIJABLA = "neki string";
```

Postoje sljedeće vrste podataka u PHP-u:

• sklarani: integer (cijeli broj), float (realni broj), string (niz slova), Boolean (istinito, lažno)

• kolekcije: array (niz), object (razred)

• posebni tipovi: izvori, NULL (ne postojeće)

```
$cijeliBroj = 10;
$realanBroj = -2.5;
$oktalni = 0123;
$heksadecimalni = 1x2B;
$nekiString = "Ovo je niz slova sa oznakom za novi red\n";
$novi_string .= "Nadodan tekst na nekiString";
$boolean = false;
$nizImena = array("Ivan", "Marko", "Luka");
$nizSaKljucevima = array('Prvi' => "Ivan", 'Drugi' => "Marko", 'Treci' =>
    "Luka");

$nizImena[3] = "CetvrtoIme";

class Osoba{
    public $ime = '';

    function ime ($novoIme = NULL){
        if (!is_null($novoIme)) {
        $this->ime = $novoIme;
        }
        return $this->ime;
    }
}
$objektMarko = new Osoba;
$objektMarko->ime('Marko');
echo "Dobavljanje imena iz objekta: {$objektMarko->ime}";
```

PHP sadržava ugrađene funkcije poput *if, while, echo, strlen* i slično, a mogu se i definirati funkcije od strane korisnika. Funkcije se definiraju na sljedeći način:

```
function nazivFunkcije ($prviArgument, $drugiArgument, ...) {
    //tijelo funkcije primjerice:
    if($prviArgument != "0"){
        //...
    }
    return Nesto;
}

//pozivanje funkcije
$odgovor = nazivFunkcije($prvi, $drugi, ...)
```

Programski jezik PHP podržava sve često korištene baze podataka poput MySQL, PostgreSQL, Oracle, Sybase, SQLite i ODBC.

Poglavlje 2

Razvojno okruženje

Radi dinamičke prirode programiranja sa PHP-om, vrlo je lako pogriješiti i izgubiti se u kodu jer za razliku od ostalih programskih jezika, u PHP-u ne postoji alat koji bi ukazao na pogreške prije pokretanja programa. Pogreške će programer primijetiti tek nakon pokretanja programa, a moguće je čak da mnoge prođu nezapaženo jer naizgled ne ometaju rad ili logiku programa. Taj problem danas uvelike olakšava upotreba razvojnog okruženja.

2.1 Definicija i svojstva

Općenito govoreći, pojam razvojno okruženje (*framework*) predstavlja univerzalni, konceptualni ili stvarni alat osmišljen u svrhu olakšanog razvoja nečega korisnog. U području računalnog programiranja, razvojno okruženje je slojevita struktura za višekratnu upotrebu koja se može sastojati od već gotovih kodova ili programa te omogućava selektivne izmjene istih od strane korisnika, čineći ga glavnim polazištem za razvoj softverskih produkata, aplikacija, dijelova operacijskog sustava... Pruža parcijalne funkcionalnosti i navodi korisnika u radu primjerice, što je moguće programirati, kako će ti programi korespondirati, definira sučelje za programiranje, standardizaciju komunikacije; i slično.

Za usporedbu, razvojno okruženje je opsežnije od protokola i propisuje više stvari od obične strukture te ga odlikuju značajna svojstva koja ga razlikuju od programskih biblioteka kao što su:

Inverzija kontrole: tok kontrole programa kod razvojnog okruženja nije određen od strane korisnika

Proširivost: korisnik može proširiti razvojno okruženje dodavajući (ili rjeđe izmjenjujući) selektivne funkcionalnosti

Nepromjenjivost koda: kod razvojnog okruženja se ne bi trebao mijenjati iako dopušta njegovo proširivanje od strane korisnika

Pozicija razvojnog okruženja u programiranju

Što čini razvojno okruženje zanimljivim i koja je njegova pozicija danas u programiranju? Do nedavno je razvoj softvera zahtijevao velik trud i znanje, bilo je od izuzetne važnosti savladati u potpunosti programski jezik i pridavati puno pažnje suptilnim pogreškama u kodu, što u konačnici dovodi do kvalitetnijeg produkta. Danas se pitanja o sintaksi, ispravljanje pogrešaka, prebacivanje u druge programske jezike, dodavanje novih funkcionalnosti i ostalo, svodi na automatsko ispravljanje prevoditeljom (*compiler*), pozivanje već postojećih API-ja (*Application Programing Interfaces*) i korištenje razvojnog okruženja. Stoga se pozornost prebacuje sa savladavanja tih vještina na razumijevanje API-a i što se s njima može postići. Razvojno okruženje tu također izlazi ususret početnicima osiguravajući ispravnu interakciju sa bazom podataka i slojevima koda. Praksa programiranja se promijenila u toliko što se danas možemo osloniti na tuđi programski kod pa i gotove poluprodukte u stvaranju vlastitih. Stoga je danas isplativije za programera naučiti sve o zadanom razvojnom okruženju nego uložiti puno vremena u svladavanje sintakse nekog jezika na kojem se ono temelji.

IDE (*Integrated Development Enviroment*) je softverska aplikacija koja pruža sveobuhvatnu podršku programerima pri razvoju softvera. Sastoji se uglavnom od editora izvornog koda, ugrađenih automatizacijskih alata i programa za pronalaženje programskih pogrešaka (*debugger*). Od velike je pomoći pri radu u razvojnom okruženju jer navodi programera na jednostavnije greške kao što su postavljanje točka-zareza, ispravno korištenje tipa podatka ili parametara u funkciji, pružajući time programeru slobodu razmišljanja na višim razinama programiranja i ostvarivanje ideja jer pritom ne mora brinuti o pravilima sintakse ili ponavljajućim dijelovima koda.

Prednosti i mane

Zašto koristiti razvojno okruženje danas, odnosno koje su njegove prednosti? Prije svega, važna je učinkovitost nekog djelovanja. Poslovi koji bi inače zahtijevali mnogo utrošenog vremena i redaka programskog koda se svode na jednostavno pozivanje i povezivanje s već ugrađenim funkcijama. Razvoj softvera se time ubrzava, pojednostavljuje i zahvaljujući već provjerenim gotovim kodovima, odvija učinkovitije. Svoju popularnost današnja razvojna okruženja temelje na velikim sigurnosnim implementacijama što korisniku pruža sigurnost razvoja, te cijeni i podrški. Većina razvojnih okruženja je dostupna besplatno na Internetu što za posljedicu ima veliku zajednicu korisnika koji međusobno komuniciraju i lako mogu dojaviti proizvođačima otkrivene propuste ili pogreške. Pozadina sigurnosnih implementacija je rezultat dugogodišnjih testiranja što od strane proizvođača što od korisnika. Korisnici zapravo preuzimaju ulogu testera dok god koriste taj produkt i svojim povratnim informacijama pridonose sigurnosti razvojnog okruženja kroz dulje razdoblje.

Kvaliteta podrške se ostvaruje kroz svu potrebnu dokumentaciju uz produkt, kao i dostupnosti tima za pružanje podrške te foruma na kojima si korisnici mogu međusobno pomagati u kratkom roku. Moguće je izraditi i vlastito razvojno okruženje iako većina razvojnih programera ipak bira neko od već postojećih popularnih okruženja radi već navedenih prednosti te je preporučljivo da se na taj pothvat odluče samo iskusni PHP programeri.

Korištenje razvojnog okruženja u izradi aplikacija ima i svoje mane. Koristeći ga bez temeljnog znanja o programskom jeziku koji ga podržava ne može se steći kvalitetno znanje o jeziku jer se u konačnici programer uči koristiti razvojno okruženje, a ne i sam jezik. Činjenica da se koriste tuđi, već gotovi kodovi donosi sa sobom sigurnosni rizik i oduzimanje kontrole nad razvojem. Ukoliko postoji sigurnosni propust u razvojnom okruženju, sam korisnik nije u mogućnosti to popraviti već može jedino obavijestiti proizvođače i pričekati ispravak ili nadogradnju. Također je već gotove kodove teško mijenjati, razvojno okruženje je u tom smislu ograničeno jer ne možemo ići van njegovih okvira i mogućnosti koje nudi. Dostupnost na Internetu dovodi do opasnosti zloupotrebe. Svatko je u mogućnosti pribaviti kodove razvojnog okruženja, proučiti njegov rad i iskoristiti sigurnosne propuste ili druge mane u napadu na neku web stranicu/aplikaciju izrađenu u tom okruženju.

2.2 Razvojno okruženje u praksi

Važno je prije pokretanja projetka odrediti koliko bi korištenje razvojnog okruženja bilo od koristi razvojnom programeru i/ili korisniku. Ukoliko će ubrzati vrijeme razvoja, olakšati korištenje aplikacije korisniku, poboljšati performanse i stabilnost aplikacije, tada je korištenje PHP razvojnog okruženja preporučljivo. Kod razvoja manjih aplikacija ili aplikacija kod kojih je sigurnost najveći prioritet (primjerice online bankarstvo) bolje je nadgledati cijeli razvoj i pisati vlastiti programski kod.

Pri odabiru razvojnog okruženja gleda se nekoliko faktora: iskustvo programera i jednostavnost u korištenju, u kojoj mjeri pojednostavljuje i ubrzava razvoj, kako poboljšava performanse aplikacije, koje funkcionalnosti i gotove dijelova koda pruža, popularnost među razvojnim programerima te dostupnu podršku. Ukoliko programer nije stručnjak za PHP, preporučljivo je izbjegavati razvojna okruženja koja nemaju dovoljnu podršku ili ona uopće ne postoji, koja imaju premalu zajednicu korisnika te valja paziti na one proizvedene od strane pojedinaca. Njihovo korištenje može dovesti do nepredvidivih pogrešaka jer se ne možemo u potpunosti pouzdati u ispravnost i djelotvornost razvojnog okruženja.

Danas postoji veliki izbor PHP razvojnih okruženja, a među popularnijima su: Codelgniter, Symfony, Laravel, CakePHP, Zend Framework, Yii 2, Phalcon, Slim, FuelPHP, Kohana (vidi [5]).

Pogreške pri korištenju

Jedna od prednosti korištenja razvojnog okruženja je upravo smanjenje pogrešaka pri pisanju koda, no razvojno okruženje ne otklanja ih u potpunosti. Moguće ga je neispravno koristiti zbog neznanja o programskom jeziku ili neshvaćanja kako dano razvojno okruženje zapravo funkcionira. Važno je provjeriti kompatibilnost baze podataka i web poslužitelja s korištenim razvojnim okruženjem. Ukoliko se to zanemari može doći do slabije performanse aplikacije ili njenog nepokretanja. Symfony primjerice zahtijeva sljedeće: verzija PHP-a mora biti najmanje 5.5.9, JSON mora biti omogućen, ctype mora biti omogućen, php.ini mora imati postavke date.timezone. Ako se određeni zahtjevi razvojnog okruženja zanemare, velika je vjerojatnost da se neće moći koristiti ili će neispravno raditi. Proces instaliranja može koji put određivati kasnije funkcioniranje razvojnog okruženja i utjecati na probleme koji mogu doći, stoga je važno slijediti instrukcije i pravilno podešavati željene komponente kako se kasnije ne bi trošilo vrijeme na ispravljanje pogrešaka koje su se lagano mogle izbjeći.

Poglavlje 3

MVC

3.1 Uvod u MVC

Definicija

PHP razvojno okruženje pomaže u bržem razvoju web aplikacije, daje kvalitetniji odnosno stabilniji kostur aplikacije, smanjuje količinu ponavljajućih dijelova koda i uvelike olakšava kasnije mijenjanje i održavanje aplikacije. Glavna ideja na kojoj se temelji i koja to omogućava je *Model-View-Controller* (MVC) koji je u zadnjih nekoliko godina najčešće korišten obrazac u području programiranja. Prikupljanje podataka od korisnika, njihova obrada i skladištenje u određenom formatu čini veliki dio rada web aplikacije, a uz današnju struju čestih promjena, važno je osigurati jednostavno održavanje i prilagodbu različitim uređajima. Ideja je odvojiti sve dijelove i odgovornosti aplikacije za koje se to može učiniti, u zasebne strukture/klase, odnosno kreiranje slojeva aplikacije. MVC je obrazac arhitekture aplikacije, propisuje način strukturiranja aplikacije u zasebne komponente i njihove odgovornosti te njihovu interakciju. Standard je podijeliti aplikaciju u prezentacijski sloj, sloj poslovne logike te podaktovni sloj.

MVC se sastoji od tri komponente odnosno klase po kojima je i dobio naziv:
- model
- *view* (pogled)
- *controller* (upravljački dio)

Model

Model se sastoji od glavnih programskih podataka i pripadnih funkcija. Ovaj sloj aplikacije se bavi upravljanjem i obradom podataka kao što su informacije o korisniku, informacije o objektima iz baze, SQL upiti. Predstavlja jednu ili više klasa sa stanjima te ima ulogu poveznice između upravljačkog dijela i pogleda i ne može se direktno pozvati. Korisnik prvo poziva upravljački dio primjerice izmjenom nekog dokumenta. Zatim se poziva metoda iz modela koja upravljaču vraća tražene podatke te metoda mijenja svoje stanje. Metoda dalje obavještava pogled o promjenama i pogled ponovno poziva metode iz modela kako bi dobio potrebne informacije o podacima koje prikazuje. Sam model nikad ne pokreće zahtjeve već obrađuje zahtjeve koje mu šalju upravljački dio i pogled. Može imati pasivnu i aktivnu ulogu:

• kod jednostavnijih primjera gdje nije nužna interakcija modela sa pogledom i upravljačkim dijelom kažemo da je model pasivan

• ukoliko se mijenja stanje modela i postoje interakcije sa pogledom i upravljačkim dijelom kažemo da je model aktivan

U praksi se češće koristi model s aktivnom ulogom jer on predstavlja poslovnu logiku. Moguće je koristiti isti model za različite klase pogleda i upravljačkog dijela.

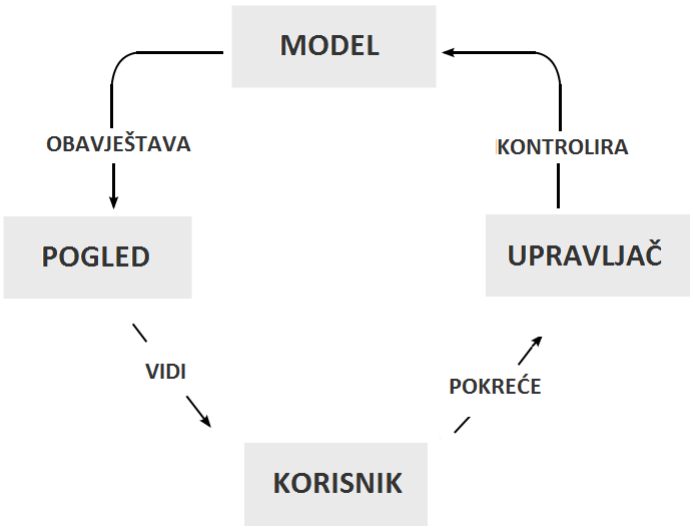
Upravljački dio (*controller*)

Upravljački dio odražava poslovnu logiku aplikacije odnosno pravila i funkcija koje ostvaruju programsku logiku. Odgovoran je za prihvaćanje i upravljanje unosom podataka odnosno zahtjeva korisnika (GET, POST, klik na element) i pretvara ih u odgovarajuće zahtjeve koje šalje modelu i pogledu. Sadrži sve kontrole aplikacija i određuje koje se promjene trebaju dogoditi u modelu i pogledu pa je na taj način direktno odgovoran za tok kontrole programa. Kada korisnik aktivira neku kontrolu aplikacije, upravljački dio poziva metodu modela da na odgovarajući način promijeni stanje. Za otkrivanje neke akcije korisnika zaduženi su osluškivači (*listeners*). U slučaju web aplikacije upravljački sloj čini prvi sloj koji se pokreće kada se u pretraživaču poziva adresa aplikacije.

Pogled (*view*)

Pogled je odgovoran za prikaz podataka, koordinira izgled grafičkog korisničkog sučelja odnosno ono što korisnik vidi kao primjerice; obrazac za unos podataka, tablica sa podacima, gumbi, slike... Pogled prikazuje objekte iz modela i omogućuje korisniku mijenjanje podataka, no nije odgovoran za njihovo pohranjivanje već tu odgovornost prosljeđuje modelu. Jedino je pogled vidljiv korisniku, dok su model i upravljački dio u pozadinskom dijelu aplikacije. Stoga se pogled definira pomoću HTML-a, JavaScript-a, CSS-a, JSON-a

i sličnih programskih jezika. Model obavještava pogled o promjenama stanja na što pogled poziva metode iz modela kako bi dobio informacije o trenutnom, novom prikazu objekata.



Slika 3.1: MVC podjela aplikacije

3.2 Prednosti i mane

Ovakvo strukturiranje pomaže nezavisnom razvoju aplikacije, olakšava testiranje i naknadno održavanje ili izmjenu. Kroz razvoj ona može doživjeti puno promjena što od tehničke dokumentacije, pronalaženja i ispravljanja programskih pogrešaka (*bug*) do izmjena samog koda. Ovakva arhitektura to olakšava i upravo zato je MVC danas među najpopularnijim obrascima za izradu aplikacije. S druge strane, nije preporučljivo koristiti MVC kod manjih odnosno jednostavnih aplikacija. Takve vrlo jednostavne kodove nije potrebno dodatno strukturirati na manje dijelove jer bi vrlo vjerojatno bilo teže razumjeti takav kod nego primjerice jednu veliku klasu koja ima već sve potrebne funkcije. Kako se aplikacija razvija i postaje kompliciranija, podjela koda na tri dijela na ovaj način postaje nužna. Izmjene, dodavanje i testiranje se tada svodi na izmjene dijelova koda unutar klase bez utjecaja na druge dijelove aplikacije.

Izdvajanje koda zaslužnog za izgled aplikacije omogućuje da se ona prikaže na različite načine bez ponovnog prepisavanja logike i podataka. Nezavisan razvoj aplikacije podrazumijeva da je radi ovakve kontrole tipa ”podijeli pa vladaj” moguće da različiti razvojni

programeri rade paralelno na različitim dijelova koda. Danas postoje ljudi specijalizirani za posebne dijelove aplikacije kao primjerice izgled korisničkog sučelja, rad s bazom podataka, implementaciju sigurnosnih postavki i kod takvih timova važnost ovakve arhitekture je očita.

Poglavlje 4

Laravel

Laravel je besplatno PHP razvojno okruženje otvorenog koda namijenjeno razvoju web aplikacija korištenjem MVC obrasca. U kratkom vremenu je zadobio pažnju razvojnih programera i od 2015. godine je proglašen jednim od najpopularnijih PHP razvojnih okruženja diljem svijeta (Sitepoint online anketa[1]) ponajprije radi svoje jednostavnosti, izražajne sintakse, jasne strukture i vrlo detaljne dokumentacije.

4.1 Povijest Laravela

Uvidjevši nedostatke dotadašnjih razvojnih okruženja kao što su nepostojanje ugrađene podrške za autentikaciju korisnika i autorizaciju, Taylor Otwell, .net razvojni programer, je započeo vlastiti projekt koristeći ideje iz .net infrastrukture u lipnju 2011. godine i istog mjeseca objavio razvojno okruženje pod nazivom Laravel. Prva verzija je pružala ugrađenu podršku za autentikaciju, lokalizaciju, sesije, jednostavni mehanizam za kontrolu toka (*routing*), keširanje (*caching*), forme te modele i poglede. Druga verzija je objavljena u rujnu iste godine dodavši podršku za upravljače i tako ga upotpunivši do pravog MVC razvojnog okruženja. Također je proširen metodama validacije, obilježavanjem stranica, proširenom Eloquent ORM, omogućeno je instaliranje paketa putem naredbenog retka, dodana podrška za inverziju kontrole (*IoC)* i stotinjak testova za pojedine komponente razvojnog okruženja. Zajednica razvojnih programera je dobro prihvatila novo razvojno okruženje i njegov dosadašnji razvoj no ne i uklanjanje podrške za module ostalih proizvođača (*third party modules*).

U veljači 2012. je objavljena verzija Laravel 3 s brojnim novim svojstvima poput sučelja naredbenog retka (*Command-line interface*, CLI) nazvano Artisan, podrške za sus-

[1] Sitepoint online anketa:
`https://www.sitepoint.com/best-php-framework-2015-sitepoint-survey-results/`

tave upravljanja bazama podataka, uveden je sustav pakiranja naziva Bundles i vraćena je podrška za module ostalih proizvođača. S ovom verzijom je počeo nagli rast popularnosti Laravela. Već pet mjeseci nakon objave, unatoč njegovoj tadašnjoj popularnosti i stabilnosti kao razvojnog okruženja, kreatori su odlučili iznova napisati cijelo razvojno okruženje u obliku skupa paketa distribuiranih putem Composer-a. Time je započet rad na sljedećoj verziji pod imenom Illuminate.

Laravel 4 je pod imenom Illuminate ubrzo objavljen u svibnju 2013. godine. Predstavljao je značajan napredak sa svojom novom arhitekturom i mogućnostima nadogradnje. Ponovno napisana od temelja, ova verzija Laravela je donijela neke od svojstava koje dosad nije nudilo niti jedno drugo razvojno okruženje. Uveden je modularni sustav pakiranja, različiti pristupi relacijskim bazama podataka, inicijalno kreiranje baze podataka (*database seeding*), podrške za redove poruka (*message queue*), rad sa e-poštom i za odgođeno brisanje baza podataka pod nazivom *soft deletion* (mekano brisanje). Dio zajednice korisnika je smatrao kako česte izmjene razvojnog okruženja smanjuju njegovu vjerodostojnost. Naime, iako učestalo objavljivanje novih verzija ukazuje na njegovo razvijanje, s druge strane prisiljava korisnike na konstantnu prilagodbu projekata novijim verzijama te određene izmjene (poput cijele arhitekture) na zahtjevnijim aplikacijama jednostavno nisu bile održive. Uz neka dodatna svojstva i poboljšanje testiranja, korisnici su tražili i veću stabilnost. Za razliku od prijašnjih verzija, uveden je raspored objavljivanja prerađenih izdanja sa manjim izmjenama (popravci pogrešaka, dodavanje/brisanje svojstva) svakih šest mjeseci. Uz dodatne jedinice testiranja s kojima je omogućeno testiranje čak 100% komponenti razvojnog okruženja, Laravel 4 konačno postaje stabilno i pouzdano razvojno okruženje te ubrzo i jedno od najpopularnijih diljem svijeta.

Posljednja verzija Laravel 5 ([4]) je objavljena u veljači 2015. godine. Uveden je paket Scheduler koji omogućava organiziranje periodičkih zadataka, apstraktni sloj Flysystem koji omogućuje korištenje udaljenog spremišta na način poput lokalnog datotečnog sustava, unaprijeđeno rukovanje paketima uz pomoć Elixir-a, pojednostavljeno je vanjsko rukovanje autentikacijom uz opcionalni paket Socialite te je uvedena nova interna stablasta struktura za razvijenu aplikaciju. Izvorni kod Laravela se nalazi na GitHub web stranici i licenciran je pod uvjetima MIT License.

4.2 O značajnostima

Laravel se odlikuje svojstvima koja omogućuju brzi razvoj aplikacije. Implementira prečace i olakšice za česte i ponavljajuće dijelova koda poput autentikacije korisnika, sesije, keširanje, upravljanje tokom kontrole. Prepoznatljiv je po visokoj razini apstrakcije uz pomoću koje

olakšava izradu aplikacija.

Composer je alat za upravljanje zavisnostima u PHP-u. Omogućava deklariranje biblioteka o kojima projekt ovisi i vodi brigu o njihovoj instalaciji i ažuriranju. Ne upravlja paketima kao standardni upravitelj paketima, već ih instalira unutar direktorija projekta pri svakom stvaranju novog projekta.

Middleware odnosno međusoftver, je softver koji djeluje između aplikacije i mreže. Pruža mehanizam za filtriranje HTTP zahtjeva koji dolaze aplikaciji. Laravel donosi nekoliko međusoftvera koji su smješteni unutar direktorija app/Http/Middleware:

- EncryptCookies
- RedirectIfAuthenticated
- TrimStrings
- VerifyCsrfToken

Drugi po redu naveden je zadužen za provjeru je li korisnik aplikacije prošao autentikaciju. Ukoliko utvrdi suprotno, taj međusoftver će preusmjeriti korisnika na pogled za prijavu, inače dozvoljava da se zahtjev provede. Laravel omogućuje definiranje dodatnih međusoftvera za izvođenje raznih zadataka poput dodavanja zaglavlja odgovorima koje šalje aplikacija ili međusoftvera koji bi prijavljivao sve nadolazeće zahtjeve.

Laravel donosi vlastiti alat za predloške Blade za poglede (*templating engine*) koji za razliku od drugih ne zabranjuje korištenje običnog PHP koda u pogledima. Alat za predloške uvelike umanjuje posao izrade *front-end* koda i pruža bolju funkcionalnost. Blade se očituje u nastavku ".blade.php". Svi pogledi koje stvaramo se spremaju u mapu app/resources/views.

Primjer 4.2.1. *Dio koda u PHP-u se zapisuje na način propisan Blade-om:*

view.php

```
<?php foreach($subjekti as $subjekt) ?> ...
    echo <?= $subjekt->svojstvo ?>; ...
  <?php endforeach ?>
```

view.blade.php

```
@foreach($subjekti as $subjekt)
    echo {{ $subjekt->svojstvo }};
@endforeach
```

Glavna prednost korištenja Blade-a je nasljeđivanje predložaka i stvaranje sekcija. Za definiranje izgleda pogleda definiramo skriptu "layout.blade.php" u kojoj deklariramo mjesto sekcije "@yield('nazivSekcije')" koje će svaki pogled ispuniti na svoj način. Tada unutar pogleda pišemo sljedeće:

```
@extends('layout')

@section('nazivSekcije')
    //HTML i PHP kod primjerice
    <div>
        <p> Dobrodosao {{ $user->name }} ! </p>
    </div>
@endsection
```

Podaci koji se prosljeđuju pogledu se dohvaćaju unutar zagrada "{{ }}" sa nazivom varijable. Funkcije poput *foreach* ili *if/else* izjava se ugrađuju u kod jednostavno dodajući oznaku "@" bez posebnog pisanja oznaka za PHP kod i HTML kod. Svi pogledi koriste BootStrapp CSS razvojno okruženje, no nije ga nužno koristiti.

PHP Artisan je sučelje naredbenog retka koje dolazi u paketu sa Laravelom. Posebna okolina Tinker je jednostavna i interaktivna okolina za korisnike koja pruža brojne korisne naredbe koje mogu poslužiti tijekom razvoja aplikacije poput naredbi za rad sa bazom podataka, praćenja rada na aplikaciji ili upravljanja sesijama i događajima (*event*). Jako je zgodna naredba inspire koja ispisuje inspirirajući citat tijekom rada.

Primjer 4.2.2. *PHP Artisan Tinker*

```
//pokretanje
php artisan Tinker
Psy Shell v0.8.1 (PHP 7.0.13 IcO cli) by Justin Hileman
>>>
//primjer naredbe
>>>DB::table('users')->first();
=> {#680
    +"id": "1",
    +"name": "Ivan",
    +"email": "ivan@mail.com"
    +"created_at": "2017.01.01. 18:43:18",
    +"updated_at": 2017.01.01. 18:43:18",
    }
>>>inspire
```

```
Well begun is half done. - Aristotle
>>>
```

Eloquent ORM

Eloquent je moćan i izražajan alat, odnosno *Object Relational Mapper* (ORM[2]), za upravljanje bazom podataka. Svaki Eloquent model predstavlja jednu tablicu u bazi podataka i proširuje klasu Illuminate\Database\Eloquent\Model, odnosno svakoj tablici pripada jedan Model uz pomoć kojega se komunicira s istom. To podrazumijeva postavljanje upita za određene podatke iz tablice kao i umetanje novih podataka.

Novi model se stvara u naredbenom retku koristeći Artisan naredbu ”php artisan make:model ModelZaTablicu”.

Modeli se zadano spremaju unutar app direktorija i zadan izgled novostvorenog modela je sljedeći:

```
<?php

namespace App;
use Illuminate\Database\Eloquent\Model;

class ModelZaTablicu extends Model
{
    //
}
```

Jako je važno kako imenujemo tablice u bazi podataka i njihove modele. Eloquent sam zaključuje da je tablica imenovana nazivom u množini neke riječi (engleskog jezika) povezana sa modelom naziva iste riječi u jednini (engleskog jezika). Tako na primjer, ako imamo tablicu naziva ”photos”, tada pripadajući model trebamo nazvati ”Photo”. To se naravno može zaobići ako definiramo svojstvo ”$table” unutar modela deklarirajući kojoj tablici model pripada:

```
class Photo extends Model
{
    protected $table = 'moje_fotografije';
}
```

[2]ORM čine sastavni dijelovi koji pomažu u radu s bazom podataka tako da pretvaraju pristup bazi u više prijateljski, objektno orijentirani način rada.

Eloquent također prepostavlja postojanje primarnog ključa svake tablice uzlazne vrijednosti tipa cijeli broj i po zadanome je to stupac ”id”. No i to se može zaobići definirajući svojstvo ”$primaryKey” i deklarirajući željeni primarni ključ. Ako taj ključ nije ulaznog ”integer” tipa i to definiramo postavljajući svojstvo ”$incrementing” na false:

```
class Photo extends Model
{
    protected $table = 'moje_fotografije';

    protected $primaryKey = 'naziv';

    public $incrementing = false;
}
```

Eloquent modele koristimo za definiranje relacija među tablicama i možemo ih koristiti kao graditelje upita na bazu podataka za tablicu kojoj pripadaju.

Primjer 4.2.3. *Neke metode za upravljanje podacima iz baze podataka koje koriste modele kao graditelje upita su all, get/first, find:*

```
<?php

use App\Photo;  //omogucava koristenje modela u toj skripti

$pictures= Photo::all(); //dohvacanje svih redova tablice kojoj pripada
    model Photo

foreach ($pictures as $picture) {
    echo $picture->title;
}

//bez 'use \App\Photo'
$pictures = App\Photo::where('year', 2017)->get();
$picture = App\Photo::first();

$picture = App\Photo::find(1); //pronalazak po svojstvu id
$picturesByIds = App\Photo::find([1, 5, 10]); //pronalazak tri elementa po
    svojstvu id
```

Neke stupce tablice podataka želimo zaštiti od izmjene od strane korisnika kao primjerice atribut ”id”. Na početku klase modela koristeći svojstvo ”protected $fillable” deklariramo koji atributi su promjenjivi od strane korisnika:

```
class Photo extends Model
{
    protected $fillable = ['title'];
}
```

Svi Eloquent modeli su zaštićeni od takozvane ”mass-asignment” pojave. To se događa kada korisnik pokuša unijeti slučajno ili zlonamjerno neočekivani parametar putem nekog zahtjeva. Takav parametar može narušiti sigurnost baze podataka (primjerice korisnik se može postaviti kao administrator aplikacije). Sa svojstvom ”$fillable” deklariramo koje atribute korisnik može mijenjati po volji (unutar granica ispravnog parametara). Na taj način možemo sigurno koristiti metodu za stvaranje novog elementa tablice *create*.

```
$picture = App\Photo::create(['title' => 'Neka slika']);
```

S druge strane, one atribute koje želimo zaštititi od izmjene od strane korisnika deklariramo koristeći svojstvo ”$guarded” i deklariramo također na početku klase danog modela. Nije potrebno koristiti oba svojstva istovremeno unutar istog modela jer oba govore koji atributi se mogu, a koji ne mogu koristiti, odnosno ukoliko smo deklarirali atribute koji su izmjenjivi sa ”$fillable” tada smo automatski deklarirali atribute koji nisu izmjenjivi i obratno. Ako želimo da su svi atributi izmjenjivi od strane korisnika, tada jednostavno deklariramo ”protected $guarded =[];” to jest da ne postoji atribut kojeg se ne može mijenjati.

Elementi tablice se uz pomoć modela brišu koristeći metode *delete* i *destroy*. Element koji želimo izbrisati prvo treba pronaći odnosno definiramo što točno brišemo iz baze podataka. Tada koristimo metodu *delete*, a metodu *destroy* koristimo kada znamo ”id” traženog elementa pa nije potrebno dohvaćati element:

```
$picture = App\Photo::find(1); //dohvacanje

$picture->delete(); //brisanje

App\Photo::destroy(1);

App\Photo::destroy([1, 2, 3]); //brisanje vise elemenata
```

4.3 Usporedba razvojnih okruženja

Svako razvojno okruženje je prikladno za izradu web apllikacije, no ipak svako služi drugačijoj svrsi.

Primjerice, Symfony se temelji na korištenju ponovno iskoristivih dijelova i pruža najbolju modularnost. Yii koristi MVC obrazac, nema unaprijed definiran alat za predloške i također omogućava korištenje ponovno iskoristivih dijelova, no ne pruža modularnost kao Symfony. Laravel također koristi MVC obrazac i donosi vrlo kvalitetan alat za predloške koji se ističe od ostalih, ali nema jednako dobar pristup modularnosti kao druga dva razvojna okruženja. Symfony se pokazao kao najbolji izbor za razvoj vrlo zahtjevnih projekata zbog načina rukovanja kompliciranim situacijama, iako izbor bilo kojeg drugog razvojnog okruženja neće biti loša odluka. U usporedbi sa Symfony-em Laravel pruža podršku za manji broj baza podataka, dok Symfony pruža podršku za čak njih četrnaest.

Prednosti Laravela

Laravel se ističe po opširnoj i kvalitetnoj dokumentaciji. Više od 90% pitanja koja korisnik može postaviti su već odgovorena u službenoj dokumentaciji, a za sve ostalo postoji podrška od strane samih korisnika na forumima. Jedan od razvojnih programera koji koriste Laravel, Jeffrey Way održava web stranicu "Laracasts" ([2]) koja donosi brojne praktične vodiče u obliku videa i slika. Postoje brojni forumi koji se baziraju isključivo na temi Laravela, a zajednica koja broji više stotina tisuća korisnika raste svakim danom.

Korisnike najviše privlači njegova jednostavnost. Laravel je pristupačan neiskusnim programerima načinom kojim je pisan, kod je vrlo intuitivan i struktura novostvorenog projekta je osmišljena na način da ne zatrpa korisnika manje važnim dijelovima aplikacije ili onima koje neće koristiti/mijenjati. Baza od koje polazi svaka aplikacija u Laravelu je dovoljno opširna da pruži skoro sve osnovne funkcije koje bi aplikacija trebala pružati, a opet dovoljno općenita za posluživanje različitih vrsta aplikacija. Vrlo je fleksibilan, za svaku zadanu opciju aplikacije koju postavlja omogućuje jednostavnu izmjenu koja neće narušiti rad aplikacije i omogućava korištenje više od 9000 dodatnih paketa. Kasnije održavanje je vrlo jednostavno s obzirom na danu arhitekturu i omogućava istovremeni rad više razvojnih programera.

Laravel donosi vrlo jednostavno praćenje kontrole toka uz pomoć posebnih skripti za rute i povezivanja kontrole sa upravljačima. Razdvaja logiku posluživanja različitih HTTP zahtjeva i pojednostavljuje obrađivanje zahtjeva.

Dolazi sa kvalitetnom podrškom za testiranje svih dijelova aplikacije. Probleme sa numeriranjem stranica Laravel maksimalno pojednostavljuje zamjenom standardne, ručne implementacije sa automatiziranim metodama ugrađenim u razvojno okruženje.

Poglavlje 5

Razvoj aplikacije u Laravelu

5.1 Instalacija i pokretanje projekta

Za instalaciju Laravela potrebno je da poslužitelj zadovoljava sljedeće uvjete:

- PHP >= 5.6.4
- OpenSSL PHP ekstenzija
- PDO PHP ekstenzija
- Mbstring PHP ekstenzija
- Tokenizer PHP ekstenzija
- XML PHP ekstenzija

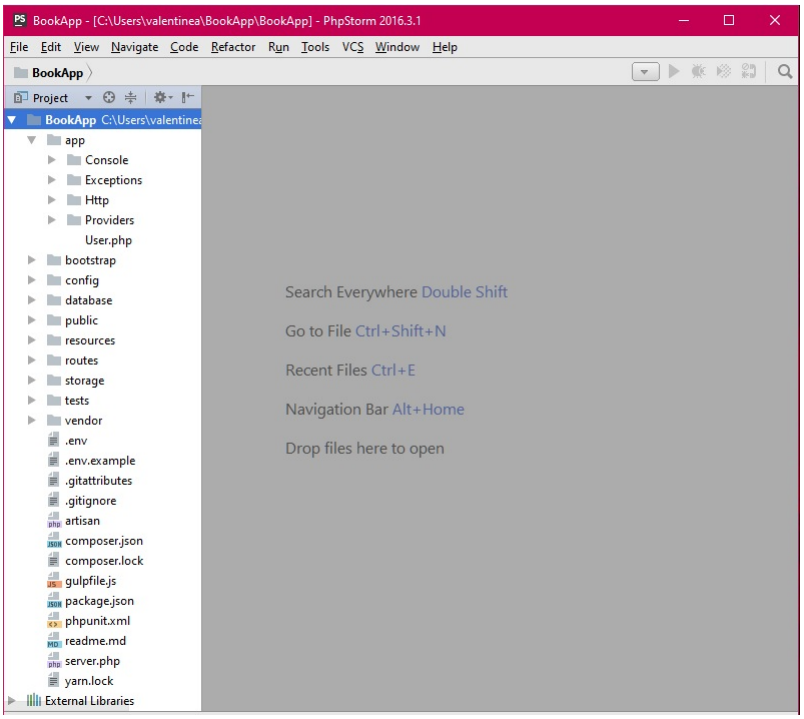
Laravel koristi Composer za upravljanje svojim zavisnostima. U naredbenom retku (Command Prompt/Terminal) uz pomoć Composer-a dohvaćamo instalater Laravela i automatski se provodi njegova instalacija:

```
composer global require "laravel/installer"
```

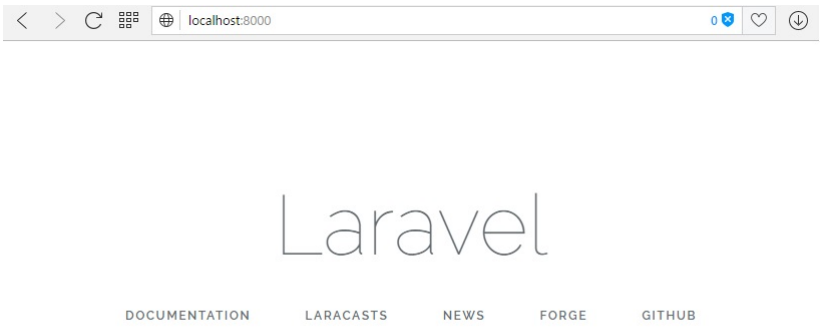
Nakon što se instalacija završi, stvaranje novog projekta se provodi korištenjem naredbe ”laravel new” uz specificiranje direktorija u koji se projekt smješta.

```
mkdir ImeDirektorija
cd ImeDirektorija
laravel new NoviProjekt
```

Uz pomoć Composera, automatski se instaliraju svi potrebni dijelovi i stvoren je novi ”prazan” projekt, odnosno osnovni dijelovi aplikacije.



Slika 5.1: Novostvorena aplikacija



Slika 5.2: Izgled novostvorene aplikacije

Za pokretanje poslužitelja na kojemu će se odvijati naša aplikacija, koristimo Artisan naredbu ”php artisan serve”. Aplikacija će se posluživati na web lokaciji ”http://localhost:8000”.

5.2 Tok kontrole (routing)

Za prosljeđivanje kontrole koristi se klasa Route. U najjednostavnijem slučaju ruta će prihvatiti URI (*Uniform Resource Identifier* [3]) i anonimnu funkciju.

Unutar mape routes nalazimo tri php skripte: ”api.php”, ”console.php”, ”web.php” gdje su smještene sve rute Laravela, a učitava ih samostalno razvojno okruženje.

Rute smještamo u grupe kao na primjer web middleware grupa ili grupa određenog prostora imena (*namespace*). To omogućuje dijeljenje atributa među rutama unutar iste grupe kako bi se smanjilo ponavljanje koda odnosno kako se ne bi moralo kod svakog definiranja rute pisati iste atribute. Grupu ruta definiramo unutar ”routes/api.php” skripte a atribute pišemo u array formatu metode *Route::group()*:

```
Route::group(['middleware' => 'auth'], function () {
    Route::get('/', function () {
    });
    Route::get('user/profile', function () {
     });
});
```

U ovom primjeru rute na glavnu stranicu aplikacije ”http://localhost:8000/” i stranicu ”http://localhost:8000/user/profile” pripadaju grupi ruta koje koriste Auth Middleware.

Slično, možemo definirati grupu ruta koje pripadaju određenom prostoru imena:

```
Route::group(['namespace' => 'Admin'], function () {
    // rute upravljaca unutar App\Http\Controllers\Admin prostora imena
});
```

Može se definirati i grupa ruta sa zajedničkim prefiksom u URI-u:

```
Route::group(['prefix' => 'admin'], function () {
    Route::get('users', function () {
        // http://localhost:8000/admin/users URL
    });
});
```

U ovom primjeru rute koje se definiraju u toj grupi imaju zajednički prefiks ”http://localhost:8000/admin/”.

[3] niz znakova koji predstavljaju ime ili izvor na Internetu

Definiranje

U skripti "routes/web.php" se definiraju rute vezane za web sučelje i pripadaju web middleware grupi. Koriste se primjerice za upravljanje sesijama i zaštitom protiv CSRF-a[4]. U skripti "routes/api.php" se nalaze rute bez stanja i pripadaju api middleware grupi.

Pogledi se nalaze u direktoriju resources unutar mape views. Možemo definirati dodatne mape unutar views po potrebi. Tada se traženi pogled resources/views/prviFolder/pogled.blade.php može dobiti unutar koda na sljedeći način: "return view(prviFolder.pogled);".

Primjer 5.2.1. *Kada korisnik zatraži posjećivanje "localhost:8000/homepage" pokreće se HTTP GET zahtjev te je akcija određena unutar funkcije "function()". U ovom slučaju aplikacija odgovara prikazivanjem pogleda "welcome".*

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

Moguće je definirati sljedeće rute:
- Route::get($uri, $odgovor);
- Route::post($uri, $odgovor);
- Route::put($uri, $odgovor);
- Route::patch($uri, $odgovor);
- Route::delete($uri, $odgovor);
- Route::options($uri, $odgovor);

U slučaju da nam je potrebna ruta koja bi mogla odgovoriti na više HTTP poziva, koristimo posebnu rutu "match" i unutar zagrada definiramo na koje pozive može odgovoriti. Ako želimo da ruta odgovara na sve HTTP pozive, tada koristimo rutu "any":
- Route::match(['get', 'post'], '/', function () {});
- Route::any('foo', function () {});

Određeni parametri ili objekti se mogu dohvaćati putem URI-ja unutar rute, a mjesto na kojemu ga dohvaćamo označavamo unutar zagrada "{objekt}" općenitim nazivom "objekt".

Primjer 5.2.2. *Unutar pogleda "popisUsera.blade.php" klikom na ul objekt se prosljeđuje kontrola na "localhost:8000/users/*" gdje se na mjesto "*" postavlja svojstvo "userId"*

[4] *Cross-Site Request Forgery* je oblik napada gdje zlonamjerna web stranica, e-pošta, blog, poruka ili program uzrokuje nepoželjne akcije u korisnikovom pregledniku

*određenog člana tablice "users". Pripadajuća ruta unutar "routes/web.php" skripte će pozvati definiranu funkciju i vratiti ispis "Trazeni id korisnika: *".*

popisUsera.blade.php

```
<div>
    @foreach($users as $user)
    <ul> <a href="/users/{{ $user->id }}" style="float:right"> Ime Usera
        </a> </ul>
    @endforeach
</div>
```

routes/web.php

```
Route::get('/users/{userId}', function($userId){
    return 'Trazeni id korisnika:'.$userId;
});
```

U kompliciranijim slučajevima koristimo se metodama definiranim u upravljačima. U prijašnjem primjeru se zahtjev obrađivao u istoj skripti, dok se kod korištenja upravljača poziva određena klasa za obradu rute.

Primjer 5.2.3. *Unutar pogleda "popisUsera.blade.php" klikom na ul objekt prosljeđuje se kontrola na "localhost:8000/users/*" gdje se na mjesto "*" postavlja svojstvo "userId" određenog člana tablice "users". Pripadajuća ruta unutar "routes/web.php" skripte će pozvati metodu show iz klase UsersController. Unutar metode show definirano je da se kao odgovor vraća pogled "views/users/show.blade.php".*

popisUsera.blade.php

```
<div>
    @foreach($users as $user)
    <ul> <a href="/users/{{ $user->id }}" style="float:right"> Ime Usera
        </a> </ul>
    @endforeach
</div>
```

routes/web.php

```
Route::get('/users/{user}', 'UsersController@show');
```

Http/Controllers/Auth/UsersController.php

```
public function show(User $user){
        return view('users.show', compact('user'));
    }
```

Naziv općenite varijable unutar rute (Eloquent varijable: $user) pritom mora biti istog naziva kao varijabla ({user}) koju prima metoda pripadajućeg upravljača (ili funkcija). Laravel će automatski unijeti Eloquent model ukoliko su nazivi ispravni i prepoznati objekt iz tablice ”users” po prosljeđenom jedinstvenom svojstvu ”id”. Ukoliko takav objekt u bazi podataka ne postoji vratit će se ”404” HTTP odgovor. To se zove ”Route model binding”.

Kontroliranje parametara

Parametri koji se šalju putem ruta se mogu kontrolirati i regulirati vrlo intuitivno na istom mjestu gdje je definirana i ruta . Ponekad je parametar samo opcionalan, a ne i nužan za prosljeđivanje kontrole. Tada se uz općeniti naziv parametra dodaje znak ”?” i definiramo zadanu vrijednost u slučaju da nije proslijeđen niti jedan parametar.

```
Route::get('user/{name?}', function ($name = null) {
    return $name;
});

Route::get('user/{name?}', function ($name = 'Ivan') {
    return $name;
});
```

Kada želimo da se parametar ili parametri pokoravaju zadanom formatu, koristimo metodu *where* koja se dodaje na kraj rute uz standardne matematičke izraze poput $[0-1]+$ što označava niz znakova nula i jedinica. Restrikcije za jedan parametar, ukoliko ih je više, se nadodaju jedna na drugu uz razdvajanje znakom ”|”.

```
Route::get('user/{name}', function ($name) {
    return $name;
})->where('name', '[A-Za-z]+|max:20');

Route::get('user/{id}/{name}', function ($id, $name) {
    //
})->where(['id' => '[0-9]+', 'name' => '[A-Za-z]+|max:20']);
```

Kako bismo izbjegli ponovno prepisavanje istih uvjeta na određene parametre, možemo definirati općenito pravilo unutar *boot* metode u ”Providers/RouteServiceProvider.php”

skripti:

```
class RouteServiceProvider extends ServiceProvider
{ ...

    public function boot()
    {
        Route::pattern('name', '[A-Za-z]+|max:25');

        parent::boot();
    }
    ...
}
```

Svaki idući put kada ruta prima parametar tog naziva, automatski se provjerava odgovara li parametar zadanim pravilima definiranima u metodi *boot()*. Funkcija koja slijedi će se izvoditi samo u slučaju ispravnog parametra.

5.3 Upravljački dio (*controllers*)

Kao što je spomenuto, upravljački dio je odgovoran za prihvaćanje i upravljanje unosom podataka i zahtjeva korisnika. Umjesto raspisivanja logike unutar ruta, definiramo da rute pozivaju odgovarajuće klase Controller koje će obraditi zahtjev. Nalazimo ih u App/Http/Controllers mapi. Controller možemo jednostavno stvoriti koristeći naredbeni redak i naredbu ”php artisan make:controller NazivControllera”

Svaki upravljač proširuje glavnu klasu Controller koja dolazi sa Laravelom. Laravel tada stvara klasu jednostavnog izgleda:

```
<?php

namespace App\Http\Controllers;
use App\Http\Controllers\Controller;

class BasicController extends Controller
{
    //
}
```

Metode koje obrađuju zahtjeve se definiraju unutar te klase, a pozivaju se unutar rute na mjestu gdje bi inače pisali funkciju tako da se prvo piše unutar kojeg upravljača se nalazi

metoda i uz oznaku "@" se definira koja se metoda poziva:

```
//upravljac
class BasicController extends Controller
{
    public function metoda(String $podatak){
        return view('pogled');
    }
}
//ruta
Route::get('poziv/{podatak}', 'BasicController@metoda');
```

Ukoliko je potrebno obraditi neke podatke, parametri poslani ruti unutar zagrada "{ }" se također automatski prosljeđuju metodi u danom upravljaču.

Ponekad definiramo upravljače koji imaju samo jednu metodu. Tada nije potrebno navoditi koja se metoda poziva unutar rute već toj metodi dajemo naziv "__invoke":

```
class Profile extends Controller
{
    public function __invoke($id)
    {
        return view('user', ['user' => User::findOrFail($id)]);
    }
}
```

```
Route::get('user/{id}', 'Profile');
```

Međusoftver se može dodijeliti rutama upravljača na sljedeća dva načina; kod definiranja rute:

```
Route::get('poziv', 'BasicController@metoda')->middleware('auth');
```

i unutar klase upravljača gdje imamo veću kontrolu jer se može dodijeliti međusoftver i samo određenim metodama:

```
class BasicController extends Controller
{
    public function metoda(String $podatak){
        return view('pogled');
    }
```

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

        $this->middleware('log')->only('index');

        $this->middleware('subscribed')->except('store');
    }
}
```

Laravel uvelike olakšava upravljanje izvorima podataka uz pomoć upravljača. Uz samo jednu naredbu "php artisan make:controller NameController –resource" stvorit će upravljač koji već ima potrebne metode za obradu podataka. Ako je potrebno upravljati primjerice slikama koje aplikacija sprema, definirali bi:

```
php artisan make:controller PhotoController --resource
```

Definirani upravljač ima sljedeće metode definirane za akcije:

```
GET          /photos               ->    photos.index
GET          /photos/create        ->    photos.create
POST         /photos/store        ->    photos.store
GET          /photos/{photo}       ->    photos.show
GET          /photos/{photo}/edit  ->    photos.edit
PUT/PATCH    /photos/{photo}       ->    photos.update
DELETE       /photos/{photo}       ->    photos.destroy
```

Uz jednu deklaraciju rute dobivamo višestruke definicije ruta za brojne akcije kao što su u primjeru iznad definirane:

```
Route::resource('photos', 'PhotoController');
```

No ako ne želimo sve akcije automatski definirane, tada jednostavno dodajemo uvjet "only" ili "except" i listu akcija za koje uvjet vrijedi, u deklaraciju:

```
Route::resource('photos', 'PhotoController', ['only' => [
    'index', 'show']]
);
Route::resource('photos', 'PhotoController', ['except' => [
    'edit', 'update', 'destroy']]
);
```

Naravno, ukoliko ne želimo da metode imaju zadana imena ili želimo drugi naziv pa-

rametara, moguće je zaobići zadane vrijednosti tako da dodamo u rutu uvjete ”names” za izmjenu naziva metoda i ”parameters” za izmjenu naziva parametara:

```
Route::resource('photos', 'PhotoController', ['names' => [
    'create' => 'photos.build']]
);

Route::resource('users', 'AdminController', ['parameters' => [
    'user' => 'admin_user']]
);
```

Izmjena naziva parametara se tada veže i na poziv URI-a. Tako na primjer, više se ne bi pozivalo ”localhost:8000/users/user” već bi se pozivalo ”localhost:8000/users/{admin_user}”.

Ako nam je potrebno više metoda od definiranih, tada ih dodajemo ručno u upravljač, a poziv moramo definirati posebno prije deklaracije ”Route::resource”:

```
Route::get('photos/novo', 'PhotoController@novaMetoda');

Route::resource('photos', 'PhotoController');
```

5.4 Baza podataka

Laravel olakšava komunikaciju sa bazom podataka koristeći Eloquent ORM. Podržava korištenje sljedećih baza podataka:

- MySQL
- SQLite
- SQL Server
- Postgres

Zadana baza podataka koja se koristi je MySQL, no po potrebi se može promijeniti. Sve informacije vezane za povezivanje sa bazom podataka definiramo unutar datoteke ”config/database.php”:

```
<?php

return [
    'fetch' => PDO::FETCH_OBJ,
    'default' => env('DB_CONNECTION', 'mysql'), //definicija baze podataka
    ...
```

U istoj datoteci su deklarirani svi potrebni podaci za svaki od četiri izbora baze podataka na temelju najčešće korištenih podataka za iste. Sve podatke možemo po potrebi mijenjati:

config/database.php (nastavak koda)

```
'connections' => [

        'sqlite' => [
            'driver' => 'sqlite',
            'database' => env('DB_DATABASE',
                database_path('database.sqlite')),
            'prefix' => '',
        ],
        'mysql' => [
            'driver' => 'mysql',
            'host' => env('DB_HOST', '127.0.0.1'),
            'port' => env('DB_PORT', '3306'),
            'database' => env('DB_DATABASE', 'forge'),
            'username' => env('DB_USERNAME', 'forge'),
            'password' => env('DB_PASSWORD', ''),
            'charset' => 'utf8',
            'collation' => 'utf8_unicode_ci',
            'prefix' => '',
            'strict' => true,
            'engine' => null,
        ],
```

Za manje zahtjevne aplikacije preporučljivo je koristiti SQLite. SQLite je relacijska baza podataka prikladna za korištenje kod web stranica sa malom do srednjom količinom dnevnog prometa (do 100 000 klikova) i ne zahtijeva mnogo (ili uopće) administracije.

Nova SQLite baza podataka se može ručno stvoriti tako da se stvori nova datoteka pod imenom ”database.sqlite” unutar mape database ili koristeći naredbu ”touch database/database.sqlite” u naredbenom retku. Tada je potrebno urediti datoteku ”.env” tako da pokazuje na istu bazu podataka: .env

```
...
DB_CONNECTION=sqlite
DB_FILE=database.sqlite
DB_DATABASE=/absolute/path/to/database.sqlite
...
```

Ključna riječ ”DB” dozvoljava metode za sljedeće upite: *insert, select, update, delete,*

statement. Tada je moguće koristiti upite za bazu podataka u bilo kojem obliku, standardnim upitima za danu bazu ili koristeći Eloquent ORM.

Prvi argument metode je SQL upit na bazu podataka, a drugi predstavlja parametar koji se veže na upit i štiti protiv SQL injekcija [5].

Select metoda vraća array rezultata u obliku PHP StdClass objekta koji se mogu koristiti na već nama poznat način:

```
$users = DB::connection('mysql')->select('select * from users where active
    = ?', [1]);
foreach ($users as $user) {
    echo $user->name;
}
```

Update i *delete* metode utječu na postojeće podatke u bazi i vraćaju broj redaka koji su danim upitom bili mijenjani. Neki upiti na bazu podataka ne vraćaju odgovor odnosno ne vraćaju nikakvu vrijednost. U tom slučaju koristimo metodu *statement*:

```
DB::statement('drop table users');
```

Laravel nam omogućava "istovremeno" korištenje više vrsta baza podataka. Na mjestima na kojima koristimo određenu vrstu potrebno je to deklarirati uz pomoć metode *connection*:

```
$users = DB::connection('mysql')->select('select * from users where active
    = ?', [1]);
$results = DB::connection('mysql')->select('select * from users where id =
    :id', ['id' => 1]); //ekvivalentno naredbi iznad
$photos = DB::connection('sqlite')->insert('insert into photos (id, title)
    values (?,?)', [1, 'Sunset']);
```

Posebno, u modelima se dodatno korištenje druge vrste baze podataka deklarira svojstvom "$connection":

```
class ModelZaTablicu extends Model
{
    protected $connection = 'baza_podataka';
}
```

[5] jedan od čestih propusta u sigurnosti informacijskih sustava gdje se podaci mijenjaju, dodaju ili brišu

Upiti na bazu podataka

Za rad sa tablicom podataka koristi se metoda *table* na intuitivan način. Na tu metodu se mogu vezati drugi upiti koji se tiču tablice kao na primjer dohvat podatka, uvjet na podatak i slično. Podatke dobivamo koristeći metode *get, pluck, first* i *all*, a uvjete definiramo u metodi *where*.

Get metoda vraća Illuminate\Support\Collection kolekciju svih podataka u obliku PHP StdClass objekta koji odgovaraju danom upitu, a pojedinim podacima pristupamo na već poznate načine:

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

foreach ($users as $user) {
    echo $user->name;
}

$users = DB::table('users')->select('name', 'email')->get();
```

Jednostavni oblik metode *where* zahtijeva tri argumenta: ime stupca u tablici, operator koji je podržan u korištenoj bazi podataka, vrijednost s kojom uspoređujemo prvi argument. Ako želimo dohvatiti podatke koji su ekvivalentni nekoj vrijednosti, tada možemo izostaviti drugi argument jer se pretpostavlja operator izjednačavanja. Često je korisno birati između mogućih podataka pa za to postoji metoda *orWhere* koja se dodaje iza pojave metode *where* i argumenti su definirano kao metodi *where*. Moguće je također koristiti više operatora odjednom:

```
$users = DB::table('users')->where([
    ['active', '=', '1'],
    ['age', '>', '30'],
    ])->orWhere('employed', '1')
    ->get();
```

Za specifično definiranje uvjeta postoji veliki izbor metoda poput: *whereBetween/whereNotBetween, whereIn/whereNotIn, whereNull/whereNotNull, whereDate/whereMonth/whereDay/whereYear, whereDate / whereMonth / whereDay / whereYear*...

Za dohvaćanje samo jednog retka u tablici dovoljno je definirati uvjete na metodu *table* i pozvati metodu *first*. No ako tražimo samo jedan podatak iz jednog reda tablice, tada umjesto metode *first* koristimo metodu *value* i definiramo koji element tablice tražimo:

```
$user = DB::table('users')->where('name', 'Ivan')->first();

echo $user->name;

$email = DB::table('users')->where('name', 'Ivan')->value('email');

echo $email;
```

Pluck metoda služi za dohvaćanje skupa podataka iz jednog stupca tablice:

```
$emails = DB::table('users')->pluck('email');

foreach ($emails as $email) {
    echo $email;
}
```

Ukoliko je potrebno obaviti neku akciju nad velikim skupom podataka (nekoliko tisuća), tada je zgodno koristiti metodu *chunk* koja u više navrata dohvaća određeni broj podataka koji definiramo i obrađuje dio po dio. Tako je moguće prividno ubrzati proces obrade podataka. Ukoliko želimo zaustaviti proces nakon prvih nekoliko rezultata, tada jednostavno vratimo vrijednost ”false”.

```
DB::table('users')->orderBy('name')->chunk(100, function ($users) {
    foreach ($users as $user) {
        echo $user->name;
    }
    //   return false;
});
```

Navedimo još neke korisne metode:
- count : računa broj rezultata
- max : vraća rezultat s najvećom vrijednosti
- min : vraća rezultat s najmanjom vrijednosti
- avg : vraća prosječnu vrijednost rezultata
- sum : vraća zbroj rezultata

Rezultatima se može manipulirati standardnim metodama iz SQL-a poput orderBy, latest/oldest, inRandomOrder (za nasumično razvrstavanje), groupBy/having, skip/take:

```
$users = DB::table('users')
              ->orderBy('name', 'desc')
              ->get();  //skup korisnika sortiranih silazno po imenu

$user = DB::table('users')
              ->latest()
              ->first();

$nasumicniUser = DB::table('users')
              ->inRandomOrder()
              ->first();  //skup korisnika nasumicno poredanih

$users = DB::table('users')
              ->groupBy('id')
              ->having('id', '>', 100)
              ->get();  //skup korisnika s id-om vecim od 100
```

Umetanje i izmjena podataka se obavlja metodama *insert* i *update*. *Insert* metoda prima array sa nazivima stupca i vrijednosti koja mu se pridodaje operatorom "=>". Jednako tako metoda *update* prima array vrijednosti koje se mijenjaju:

```
DB::table('users')->insert(
    ['name' => 'Ivan', 'email' => 'ivan@example.com']
);

DB::table('users')
          ->where('id', 1)
          ->update(['email' => 'ivan@mail.com']);
```

Podatke brišemo iz baze podataka uz pomoć metode *delete*. Ovisno o podatku ili podacima koje želimo izbrisati, prvo podatak pronalazimo metodama opisanim prije i dodamo na to metodu *delete*. U slučaju da se želi izbrisati cijela tablica podataka, koristi se metoda *truncate*:

```
DB::table('users')->delete();
DB::table('users')->truncate();
DB::table('users')->where('surname', 'horvat')->delete();
```

Migracije

Migracije se koriste kao oblik kontroliranja baze podataka. Za stvaranje tablice koriste se migracije tako da se u naredbenom retku provodi naredba:

```
php artisan make:migration create_tablica --create = nazivTablice
```

To će stvoriti datoteku naziva ”year_month_day_hour_minute_second_create_tablica” u direktoriju database/migrations u kojem je definirana tablica ”nazivTablice”. Laravel dodaje datum u naziv datoteke kako bi pratio vremena stvaranja migracija. Zadani izgled migracije je klasa koja proširuje klasu Migration i sastoji se od funkcija ”up()” i ”down()”:

```
<?php

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

class CreateTablica extends Migration
{
    public function up()
    {
        Schema::create('tablica', function (Blueprint $table) {
            $table->increments('id');
            $table->string('nekiNoviStupac');
            $table->timestamps();
        });
    }
    public function down()
    {
        Schema::dropIfExists('notes');
    }
}
```

Zatim se provodi naredba ”php artisan migrate” kako bi se tablica pokrenula.

Metoda *up* služi za dodavanje novih tablica, stupaca i indeksa u bazu podataka. Metoda *down* služi za poništavanje svih akcija koje je provela metoda *up*. Za stvaranje tablice koristi se Laravel Schema pročelje (*Illuminate\Support\Facades*). Samu tablicu možemo mijenjati, tj. dodavati, brisati i mijenjati stupce (naziv, tip, ograničenja) u bilo kojem trenutku. Naravno, ukoliko smo mijenjali tablicu nakon njenog stvaranja i unošenja podataka, potrebno je ostatak baze podataka prilagoditi promjenama i ponovno je migrirati. Ponovno definiranje migracije se pokreće provodeći naredbu ”rollback” ili ”reset” za ponovno defi-

niranje svih migracija:

```
php artisan migrate:rollback

php artisan migrate:rollback --step=2

php artisan migrate:reset

php artisan migrate:refresh
```

Naredba rollback će ponovno definirati zadnju definiranu migraciju, a dodajući "−−step = x" ponovno će se definirati x (broj) zadnje definiranih migracija. Također, naredba "refresh" će redefinirati sve migracije i provesti naredbu "migrate" za sve njih time ponovno stvarajući čitavu bazu podataka.

Tablicu stvaramo koristeći metodu *create* Laravel pročelja Schema koja prima argumente: naziv tablice, funkcija koja prima Blueprint objekt. Nakon što tablici dodamo željene stupce pokrećemo metodu *migrate*. Postojeću tablicu podataka lako preimenujemo ili obrišemo sljedećim metodama:

```
Schema::rename($nazivPrije, $nazivPoslije); //preimenovanje
Schema::drop('tablica'); //brisanje
```

Pročelje Schema prihvaća velik broj tipova podataka kao primjerice binary, char, boolean, double, dateTime, increments, integer, string, text... Svakom definiranom stupcu možemo dodati uvjete odnosno svojstva koja želimo da imaju. Jednostavno se na kraj definiranog stupca nadoda željeno svojstvo ili niz svojsva:

```
public function up()
    {
        Schema::create('tablica', function (Blueprint $table) {
            $table->increments('id');
            $table->string('nekiNoviStupac')->default($vrijednost);
            $table->integer('nekiBroj')->nullable()->unique();
            $table->timestamps();
        });
    }
```

Relacije

Tablice u bazi podataka su često povezane jedna s drugom nekom relacijom. Eloquent podržava sljedeće relacije:

- jedan naprama jedan
- jedan naprama više
- više naprama više
- jedan naprama više kroz atribut
- polimorfne relacije
- više naprama više polimorfnih relacija

Pojedinu relaciju definiramo kao funkciju unutar Eloquent modela i uobičajena je praksa dati joj naziv tablice s kojom je povezan. Primjerice, ako neki korisnik ima više fotografija, tada bi unutar klase modela User definirali relaciju jedan naprama više:

```
public function photos(){
    return $this->hasMany(Photo::class);
}
```

Jedan naprama jedan

Najjednostavnija relacija je upravo jedan naprama jedan. Jedan element tablice odnosno jedan model pripada točno jednom elementu druge tablice odnosno modelu i obratno. Definiramo ju koristeći metodu *hasOne*. Unutar metode koja definira relaciju upisujemo s kojim modelom je povezan taj model.

```
<?php
namespace App;

use Illuminate\Database\Eloquent\Model;

class User extends Model
{...
public function pin(){
    return $this->hasOne('App\Pin'); //ili Pin::class
}
...
```

Nakon definiranja relacije, moguće je koristiti dinamička svojstva Eloquenta u svrhu upravljanja podacima odnosno metode za upravljanje podacima u bazi kao da su definirani unutar modela. Eloquent će prepoznati povezanost dva modela po stranom ključu koji definiramo u jednoj od tablica i važno je imenovati ga ”nazivPrvogModela_id”. Primje-

rice, ukoliko za svakog korisnika imamo posebno definiranu tablicu ”racuni”, tada ćemo je definirati:

```
class CreateRacuniTablica extends Migration
{
    public function up()
    {
        Schema::create('racuni', function (Blueprint $table) {
            $table->increments('id');
            $table->integer('user_id')->unsigned()->index(); //model je User
            $table->integer('broj_racuna')->unique();
            $table->timestamps();
        });
    }
    public function down()
    {
        Schema::dropIfExists('notes');
    }
}
```

Naravno, i to se može zaobići tako da kod definiranja relacije jednostavno dodamo još jedan argument naziva atributa tablice za koji želimo da predstavlja poveznicu odnosno primarni ključ. No Eloquent i dalje pretpostavlja da taj strani ključ sadrži ”id” drugog modela, stoga ako ne želimo da poveznicu predstavlja vrijednost ”id, koristimo treći argument kao lokalni ključ:

```
public function pin(){
    return $this->hasOne('App\Pin', 'atributStraniKljuc_id',
        'atributStraniKljuc);
}
```

Pravila o ključevima koji povezuju modele vrijede za sve relacije.
Definiranjem relacije u jednom modelu (User) smo omogućili pristup drugom modelu (Pin) kroz prvi model. Primjerice:

```
$pin_korisnika = $user->pin()->first();

$user->pin()->create(['broj_racuna' => '0123456789']);

$pin = new Pin;
$pin->broj_racuna = '0123456789';
$user->pin()->save($pin);
```

Za obratno djelovanje, potrebno je definirati prigodnu relaciju u drugom modelu koristeći *belongsTo* metodu na isti način:

```
<?php
namespace App;

use Illuminate\Database\Eloquent\Model;

class Pin extends Model
{...
public function user(){
    return $this->belongsTo('App\User'); //ili User::class
}
```

Ista pravila vezana za primarni i lokalni ključ vrijede i u ovom slučaju.

Jedan naprama više

Ova relacija govori da jednom elementu tablice odnosno modelu pripada više elemenata druge tablice odnosno drugih modela. Kao i u relaciji jedan naprama jedan, u oba modela se definira relacija; u modelu kojemu pripada više modela koristimo metodu *hasMany*, a u drugom modelu metodu *belongsTo*:

```
//klasa User kojemu pripada vise fotografija
<?php

namespace App;

use Illuminate\Database\Eloquent\Model;

class User extends Model
{
    public function photos()
    {
        return $this->hasMany('Photo::class');
    }
}
...
//klasa Photo koje pripadaju tocno jednom korisniku User
<?php

namespace App;
```

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

class Photo extends Model
{
    public function user()
    {
        return $this->belongsTo('User::class');
    }
}
...
```

Metode za upite na bazu podataka se koriste intuitivno (objašnjeno) i u ovoj vrsti relacije.

Više naprama više

Nešto kompliciranija relacija je kada istvoremeno jednom elementu jedne tablice pripada više elemenata druge i obratno. U tom slučaju potrebno je definirati treću tablicu koja će sadržavati atribute "id" (u najčešćem slučaju, može to biti i neki drugi atribut) obiju tablica. Za definiranje relacije koristimo metodu *belongsToMany*. Pokažimo to na primjeru gdje jednom korisniku pripada više hobija:

```
<?php

namespace App;
use Illuminate\Database\Eloquent\Model;

class User extends Model
{
    public function hobies()
        return $this->belongsToMany('Hoby::class');
    }
}
//hobi
<?php

namespace App;
use Illuminate\Database\Eloquent\Model;

class Hoby extends Model
{
```

```
    public function users(){
        return $this->belongsToMany('Hoby::class');
    }
}
//hobi
<?php

namespace App;
use Illuminate\Database\Eloquent\Model;

class Hoby extends Model
{
    public function users(){
        return $this->belongsToMany('User::class');
    }
}
```

Svakom elementu tih modela kojemu pristupamo automatski je dodijeljen atribut "pivot" koji služi za tablici "posrednik". Jedan pivot objekt se zadano sastoji samo od ključeva oba modela (primjerice atribut "id"), no mogu se definirati i dodatni atributi (stupci) dodajući metodu *withPivot*:

```
//definiranje relacije
public function hobies(){
return
$this->belongsToMany('Hoby::class')->withPivot('dodatniStupac1',
'dodatniStupac2');
}
//pozivanje metoda
$user = App\User::find(1);
foreach ($user->hobies as $hoby) {
    echo $hoby->pivot->created_at;
}
```

Za više informacija o ostalim relacijama upućujem čitatelja na stranicu: `https://laravel.com/docs/5.3/eloquentrelationships`.

5.5 Forme (validacija i request)

Forme kao i do sada u PHP-u, se definiraju na sljedeći način:

```
<form method="POST" action="/link">
    <input type="hidden" name="_token" value="{{ csrf_token() }}">

        <textarea name="nekiTekst">{{ old('nekiTekst') }}</textarea>

    <button type="submit">Gumb</button>
</form>
```

Ovisno o kojim se podacima radi u formi, definira se metoda GET ili POST i akcija je neka ruta koju definiramo u skripti "routes.php". Svaki puta kada definiramo HTML formu važno je dodati polje "CSRF token" unutar forme kako bi međusoftver za zaštitu provjerio valjanost forme i prihvatio zahtjev. U skripti "routes.php" tada definiramo pripadajuću rutu uz dodatnu mogućnost osim GET i POST, a to su PUT, PATCH i DELETE. HTML forme standardno ne podržavaju te akcije pa je potrebno unutar forme dodati skriveno polje "method_field('akcija')".

Kada mijenjamo podatke u bazi unosom podataka od strane korisnika, u formi definiramo metodu POST i željenu akciju, a u ruti definiramo da se radi o akciji "PATCH".

```
<form method="POST" action="/link/{{ $nekaVrijednost }}">

    {{ method_field('PATCH') }}
    <input type="hidden" name="_token" value="{{ csrf_token() }}">

        <textarea name="nekiTekst"></textarea>

    <button type="submit">Gumb</button>

</form>

//ruta u routes.php skripti
Route::patch('link/{vrijednost}', 'NekiController@nekaMetoda');
```

Elementi forme za unošenje podataka imaju atribut "name" i tu vrijednost koristimo u ostalim skriptama kao ime varijable čija je vrijednost unešeni podatak. Tako će se u prijašnjem primjeru tekst unešen u "textarea" sa nazivom "name='nekiTekst'" dohvaćati koristeći varijablu "$nekiTekst" i vrijednost će joj biti unešeni podaci od korisnika. Podatke također možemo slati ruti putem akcije tako da jednostavno dodamo "{{ $nekaVrijednost }}" naziv varijable u akciju, a naziv te varijable u skripti "routes.php" u pripadajućoj ruti mora imati naziv koji odgovara nazivu u upravljaču.

Za obradu zahtjeva poput unošenja podatka putem forme koristimo klasu Illuminate\Http\Request koju unosimo u upravljaču koji obrađuje zahtjev. Zahtjev će automatski biti poslan odgovarajućoj metodi kroz varijablu "Requests $zahtjev", a zasebne podatke dohvaćamo putem njihovih atributa "name" i metode *input* ili jednostavno "->nazivPodatka".

```
<?php

namespace App\Http\Controllers;
use Illuminate\Http\Request;

class NekiController extends Controller
{
    public function nekaMetoda(Request $zahtjev, TipPodatka $vrijednost){
        $this->validate($request, [
        'nekiTekst' => 'required|max:255',
        'atribut' => 'required',
        ]);
        //svi podaci u formi: request()->all()
        $podaci = $zahtjev->input('nekiTekst'); // ili $zahtjev->nekiTekst;
    }
}
```

Skoro uvijek želimo kontrolirati podatke koje korisnik unosi, a to nam omogućava metoda *validate*. U metodu unosimo parametre "Request $zahtjev" i pravila koja definiramo za svaki od podataka. Metoda će usporediti dobivene podatke i pravila te ukoliko su podaci ispravni, zahtjev prolazi validaciju. Ukoliko neki od podataka nisu ispravni kažemo da validacija nije uspjela te se podaci o pogrešci spremaju u općoj varijabli "$errors" dostupnoj unutar pogleda gdje se nalazi forma.

```
class NekiController extends Controller
{
    public function nekafa(Request $zahtjev, TipPodatka $vrijednost){
        $this->validate($zahtjev, [ 'nekiTekst' => 'required|max:100'])

        $podaci = $zahtjev->input('nekiTekst'); // $zahtjev->nekiTekst;
    }
}

//dohvacanje pogreska ispod forme u pogledu
@if( count($errors) )
    <ul>
```

```
        @foreach($errors->all() as $error)
            <li>{{ $error }}</li>
         @endforeach
    </ul>
@endif
```

U kompliciranim slučajevima moguće je stvoriti "form request", odnosno posebnu klasu koja će biti zadužena za validaciju zahtjeva. Novostvorena klasa se sprema u direktorij app/Http/Requests. Pravila za validaciju smještamo u metodu *rules*:

```
php artisan make:request klasaZaValidaciju //naredbeni redak

//klasa
<?php

namespace App\Http\Requests;
use Illuminate\Foundation\Http\FormRequest;

class klasaZaValidaciju extends FormRequest
{
    public function authorize(){
        return false;  //zadana vrijednost
    }
    public function rules(){
        return [
            'email' => 'required|unique:email|max:255',
            'surname' => 'required|max:50',
        ];
    }
}
```

Korištenje klase za validaciju je jednostavno; u funkciji koja prima zahtjev umjesto Request koristimo klasu i vrijede ista pravila:

```
class NekiController extends Controller
{
    public function nekaMetoda(klasaZaValidaciju $zahtjev, TipPodatka
        $vrijednost){

        $podaci = $zahtjev->input('nekiTekst'); // $zahtjev->nekiTekst;
    }
}
```

U metodi *authorize* provjeravamo ima li korisnik punomoć za mijenjanje podataka. Ukoliko metoda vraća "false", automatski se vraća odgovor o pogrešci "HTTP 403", no ako namjeravamo provjeravati punomoć korisnika u nekom drugom dijelu, tada jednostavno postavimo da metoda vraća "true".

Koristeći Validator pročelje možemo stvoriti vlastiti "validator" odnosno metodu koja će provjeravati valjanost forme bez korištenja "Request". Validator se stvara uz metodu *Validator::make()* koja prima dva argumenta; podaci koje želimo provjeriti (ako uzimamo sve iz forme to je onda "$request->all()"), pravila valjanosti. Varijablu kojoj smo definirali validator provjeravamo sa metodom *fails*:

```
<?php

namespace App\Http\Controllers;
use Validator;
use Illuminate\Http\Request;
use App\Http\Controllers\Controller;

class NekiController extends Controller
{
    public function nekaMetoda(Request $request)
    {
        $validator = Validator::make($request->all(), [
            'nekiTekst' => 'required|max:255',
            'atribut' => 'required',
        ]);

        if ($validator->fails()) { //ako nije prosao provjeru valjanosti
            return redirect('post/create')
                        ->withErrors($validator)
                        ->withInput();
        }
    }
}
```

Pogreške spremljene u varijabli "$errors" su automatski dostupne pogledima nakon prenošenja kontrole sa metodom *redirect* na neki pogled, koristeći metodu *withErrors* koja prihvaća validatore, MessageBag i PHP array.

Poruke koje opisuju pogreške koje su se dogodile u kodu imaju svoje zadane vrijednosti, no možemo stvoriti vlastite poruke ovisno o pravilu na više načina: definiranjem funkcije u klasi za validaciju, dodavanjem varijable u kojoj definiramo poruke kao treći

argument validatora te definiranjem ”custom” poruka.

Unutar klase u skripti ”app/Http/Requests/klasaZaValidaciju.php” definiramo funkciju ”messages” koja vraća poruke dodijeljene pravilima za validaciju:

```
public function messages()
{
    return [
        'email.required' => 'Potrebno je unijetu email adresu!',
        'email.unique' => 'Unesena email adresa vec postoji!',
    ];
}
```

Definiranje i dodavanje varijable koja opisuje poruke o pogreškama kao treći argument:

```
$messages = [
    'required' => 'The :attribute field is required.',
];

$validator = Validator::make($request->all(), [
           'nekiTekst' => 'required|max:255',
           'atribut' => 'required'],
           $messages);
```

Predefiniranje ”custom” poruka unutar ”resources/lang/en/validation.php” skripte:

```
<?php

return [

    'accepted'         => 'The :attribute must be accepted.',
    'active_url'       => 'The :attribute is not a valid URL.',
    ...,
    'string'           => 'The :attribute must be a string.',
    'timezone'         => 'The :attribute must be a valid zone.',
    'unique'           => 'The :attribute has already been taken.',
    'uploaded'         => 'The :attribute failed to upload.',
    'url'              => 'The :attribute format is invalid.',

    'custom' => [
        'email' => [
            'required' => 'Potrebno je unijeti email adresu!',
        ],
```

```
        'attribute-name' => [
            'rule-name' => 'custom-message',
        ],
    ],
```

5.6 Autentikacija i heširanje

Laravel donosi jednostavno gotovo rješenje za autentikaciju korisnika. Prije stvaranja projekta potrebno je provesti naredbu ”php artisan make:auth” u naredbenom retku kako bi se stvorili svi potrebni dijelovi poput tablice za korisnike i lozinke, forme za prijavu, odjavu i jednostavne forme za registraciju, upravljači za te akcije i drugo.

Skripta za konfiguraciju autentikacije se stvara unutar config mape naziva ”config/auth.php” i sadrži opcije za određivanje ponašanja usluga koje donosi Laravel autentikacija poput opcije za ponovno postavljanje lozinke i dohvaćanja korisnika iz baze podataka.

```
<?php

return [
...
    'guards' => [
        'web' => [
            'driver' => 'session',
            'provider' => 'users',
        ],
        'api' => [
            'driver' => 'token',
            'provider' => 'users',
        ],
    ],
...
    'providers' => [
            'users' => [
                'driver' => 'eloquent',
                'model' => App\User::class,
            ],
    ],
    ...
];
```

Autentikacija se najvećim dijelom sastoji od takozvanih ”guards” i ”providers”. Guards ili čuvari definiraju kako se provodi autentikacija korisnika kod obrade zahtjeva, primjerice uz korištenje sesija ili tokena. Svi upravljači autentikacije sadrže korisnički ”provider” ili opskrbljivač. Oni definiraju kako se korisnici dohvaćaju iz baze podataka ili drugih mehanizma skladištenja podataka koje aplikacija koristi. Laravel u tu svrhu koristi Eloquent i standardni graditelj upita na bazu podataka. Ukoliko se ne koristi Eloquent, tada je moguće koristiti drugi upravljač autentikacije za bazu podataka koji koristi Laravelov graditelj upita.

Nakon stvaranja projekta definiran je model User za upravljanje korisnicima. Nakon stvaranja autentikacije stvoreni su pripadni upravljači unutar prostora imena App\Http\Controllers\Auth poput ”RegisterController” koji je zadužen za registraciju korisnika, ”LoginController” koji je zadužen za prijavu korisnika u aplikaciju, ”ForgotPasswordController” koji je zadužen za oporavak lozinke u slučaju da ju je korisnik zaboravio i ”ResetPasswordController” koji je zadužen za ponovno postavljanje lozinke. Njihova zadana definicija je oblikovana na način da bez izmjene može posluživati veliku većinu aplikacija danas. Također se stvaraju pogledi (layout skripte) koji prikazuju forme za prijavu i registraciju te skripta ”resources/views/layouts/app.blade.php” u kojoj se provjerava postoji li prijavljen korisnik ili aplikaciji pristupa gost, koju nasljeđuju ostali pogledi. Definirane su rute potrebne za preusmjeravanje toka kontrole kod prijave, registracije i odjavljivanja korisnika i upravljač ”HomeController” za upravljanje početnom stranicom nakon ulaska u aplikaciju.

Primjer 5.6.1. *Primjeri nekih skripta:*

```
//HomeController
<?php

namespace App\Http\Controllers;
use Illuminate\Http\Request;
use App\Recipe;
use App\User;
use App\Ingredient;
use Illuminate\Support\Facades\Auth;

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

```
    public function index()
    {
        return view('home');
    }
}
//routes.php
Auth::routes(); ...
//upravljac za registraciju korisnika
<?php

namespace App\Http\Controllers\Auth;
use App\User;
use App\Http\Controllers\Controller;
use Illuminate\Support\Facades\Validator;
use Illuminate\Foundation\Auth\RegistersUsers;

class RegisterController extends Controller
{
    use RegistersUsers;

    protected $redirectTo = '/home';

    public function __construct()
    {
        $this->middleware('guest');
    }
    protected function validator(array $data)
    {
        return Validator::make($data, [
            'email' => 'required|email|max:255|unique:users',
            'password' => 'required|min:6|confirmed',
        ]);
    }
    protected function create(array $data)
    {
        return User::create([
            'email' => $data['email'],
            'password' => bcrypt($data['password']),
        ]);
    }
}
```

Pročelje Auth nam omogućava rad sa podacima korisnika koji je prijavljen u aplikaciju. Trenutno prijavljenog korisnika jednostavno dohvaćamo naredbom ”Auth::user()”, a sve podatke o njemu dohvaćamo standardnim metodama. Provjera postoji li prijavljen korisnik se provodi sa ”Auth::check()” što vraća odgovor ”false” ili ”true”. Možemo upravljati ovlastima pristupa određenim rutama korisnicima uz pomoć middleware-a za rute definiranog unutar ”Illuminate\Auth\Middleware \Authenticate”. Dovoljno je kod definiranja rute dodati metodu *middleware()* i odrediti kome je dozvoljen pristup:

```
//routes.php
...
Route::get('/zasticenaRuta', function(){...})->middleware('auth'); //samo
    prijavljeni korisnici mogu pristupiti ovoj ruti
```

Naravno, moguće je prilagoditi autentikaciju vlastitim standardima. Nije nužno koristiti upravljače koje donosi Laravel, možemo ručno definirati vlastite upravljače i klase koji će služiti istoj svrsi.

Pročelje Bcrypt predstavlja jednostavnu sigurnosnu implementaciju za heširanje lozinki korisnika. Upravljači za prijavu i registraciju automatski koriste Bcrypt pri unosu podataka, a sami to možemo iskoristiti na sljedeći način:

```
<?php

namespace App\Http\Controllers;
use Illuminate\Http\Request;
use Illuminate\Support\Facades\Hash;
use App\Http\Controllers\Controller;

class VazanController extends Controller
{
    public function unosLozinke(Request $request){
        $request->user()->fill( [ 'password' =>
            Hash::make($request->newPassword) ] )->save();
    }
}
```

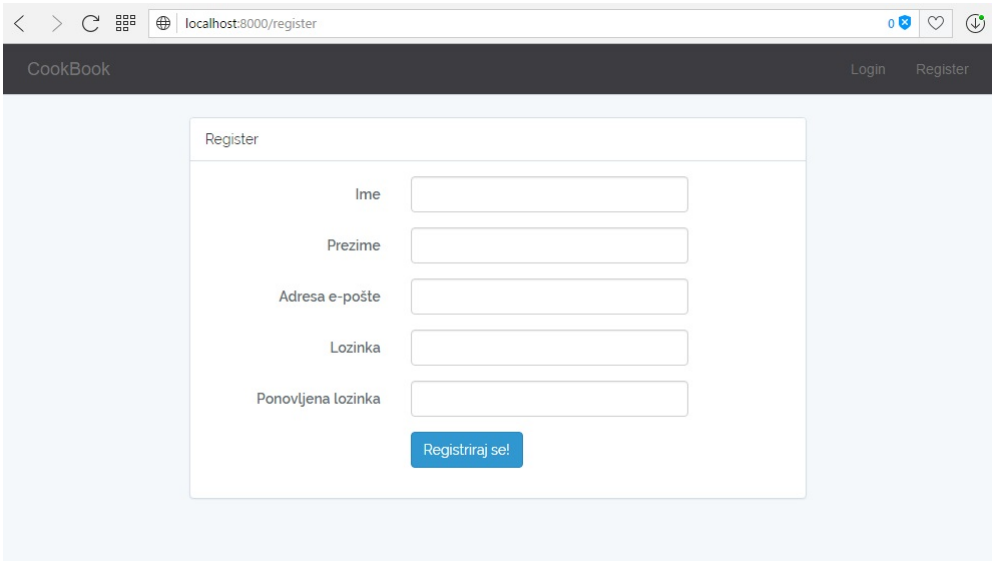
Kod prijave korisnika aplikacija će provjeravati odgovara li lozinka u bazi podataka haširanom unosu koristeći metodu ”Hash::check('unos', $pravaLozinka)”.

5.7 Primjer aplikacije izrađene u Laravelu

Slijedi opis aplikacije razvijene u Laravelu.

Naslovna stranica aplikacije započinje sa jednostavim pogledom koji sadržava linkove na prijavu i registraciju korisnika te link na stranicu ”O aplikaciji”.

Pri registraciji korisnik unosi sljedeće podatke: ime, prezime, adresu e-pošte, lozinku, ponovljenu lozinku. Nakon uspješne registracije korisnika, aplikacija prijavljuje korisnika i prikazuje se naslovna stranica kao i nakon uspješne prijave registriranog korisnika.



Slika 5.3: Prikaz forme za registraciju korisnika

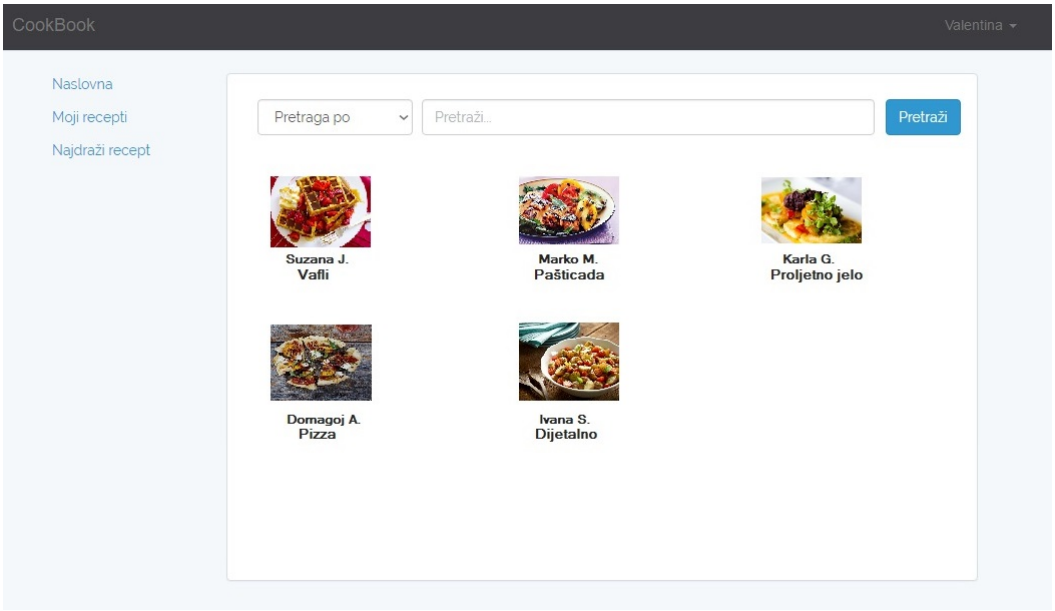
```
//isjecak koda forme za registraciju
<form class="form-horizontal" role="form" method="POST" action="{{
    url('/register') }}">
    {{ csrf_field() }}
    <div class="form-group{{ $errors->has('name') ? ' has-error' : '' }}">
        <label for="name" class="col-md-4 control-label">Ime</label>

        <div class="col-md-6">
            <input id="name" type="text" class="form-control" name="name"
                value="{{ old('name') }}" required autofocus>

            @if ($errors->has('name'))
                <span class="help-block">
                    <strong>{{ $errors->first('name') }}</strong>
                </span>
            @endif
```

```
        </div>
    </div>
    ...
    <div class="form-group">
        <div class="col-md-6 col-md-offset-4">
            <button type="submit" class="btn btn-primary"> Registriraj se!
            </button>
        </div>
    </div>
</form>
```

Na naslovnoj stranici lijevo je smješten izbornik koji omogućuje korisniku pristup sljedećim stranicama: Naslovna, Moji recepti, Najdraži recepti. Prikazano je prvih devet recepata koje aplikacija dohvaća iz baze podataka po datumu njihovog stvaranja (najnovije stvorene recepte) i alat za pretraživanje recepata po nazivu korisnika (ime ili prezime), vrsti jela (kako su korisnici definirali, primjerice ”večera”) ili nazivu recepta.



Slika 5.4: Prikaz naslovne stranice prijavljenog korisnika

Na stranici ”Moji recepti” korisniku se prikazuje lista njegovih recepata razvrstanih po datumu stvaranja i omogućeno mu je stvaranje novog recepta klikom na ”Novi recept”. Pri stvaranju novog recepta, korisniku se prikazuje više formi: za unos naziva i vrste recepta,

unos sastojaka te unos glavnog teksta recepta. Sljedeći isječak koda prikazuje metodu *dodajRecept* koja obrađuje dio podataka pri stvaranju novog recepta.

```
<?php

namespace App\Http\Controllers;
use Illuminate\Http\Request;
use App\Recipe;
use App\User;
use App\Ingredient;
use Illuminate\Support\Facades\Auth;
use Illuminate\Support\Facades\DB;

class HomeController extends Controller
{
    ...
    public function mojiRecepti(){
        return view('mojiRecepti');
    }
    public function dodajRecept(Validator1 $request){

        $recept = new Recipe;
        $recept->type = $request->type;
        $recept->title = $request->title;
        $recept->text = "";
        $user = Auth::user();
        $user->recipes()->save($recept);

        $recept = Recipe::where('title', $request->title)->first();

        return view('newRecipe2', compact('recept'));
    }...
}
```

Posebno, na stranici ”Najdraži recepti” korisnik može pregledati recepte koje je otvorio i označio oznakom ”Sviđa mi se”. Svaki prijavljen korisnik je u mogućnosti komentirati svaki recept. Cjelokupni kod aplikacije može se pogledati na linku `https://github.com/valentinea/CookBook`.

Korištenje Laravela je uvelike olakšalo izradu aplikacije. Početna baza aplikacije koju Laravel stvara je smanjila značajnu količinu posla kojeg bi trebalo napraviti. Izmjena zadanih postavki poput tablice podataka o korisnicima ili rad upravljača se jednostavno i brzo odvija bez narušavanja rada ostatka aplikacije.

Poglavlje 6

Zaključak

Kroz ovaj rad smo se upoznali sa pojmom razvojnog okruženja, MVC-a i detaljno predstavili razvojno okruženje Laravel. Definirali smo najosnovnije sastavne dijelove poput modela, pogleda, upravljača, objasnili način rada sa bazom podataka, unos i obradu podataka, obradu korisničkih podataka i tok kontrole. Provjerili smo i pokazali na vlastitom primjeru kako se razvija aplikacija.

Upoznavanje sa Laravelom je teklo vrlo glatko i intuitivno. U vrlo kratkom roku (nekoliko dana) i uz malo truda uspjeli smo stvoriti jednostavnu aplikaciju spremnu za posluživanje korisnika. Modularnost Laravela je bila zadovoljavajuća za potrebe razvoja nezahtjevne web aplikacije. Vrlo intuitivna arhitektura i podjela odgovornosti aplikacije je pomogla brzom snalaženju u radu na aplikaciji. Mnogobrojne funkcionalnosti koje se očekuju od većine web aplikacija već su implementirane za općenite slučajeve bez nepotrebnog dodavanja ili izostavljanja važnih dijelova. Dokumentacija unutar samih kodova je skromna, ali dovoljna za razumijevanje uloge tog dijela koda te eventualno upućivanje korisnika na dodatne izvore znanja. Svojstva za koja se predviđa da bi korisnik htio personalizirati poput izgleda pogleda, korištenja određene baze podataka, registracije korisnika i mnogo više, su lako podesiva. Lako se unose izmjene tih svojstava bez narušavanja logike ili rada cijele aplikacije. Njegova jednostavnost i automatiziranje brojnih osnovnih funkcionalnosti je rezultirala brzim razvojem i ostavila pozitivan dojam.

Bibliografija

[1] *Bootstrap 3 Tutorial*, (2015), http://www.w3schools.com/bootstrap/.

[2] *Laracasts*, https://laracasts.com.

[3] *Laravelbook*, http://laravelbook.com.

[4] *Laravel*, https://laravel.com.

[5] Joel Reyes, *Discussing PHP Frameworks: What, When, Why and Which?*, (2009), http://www.noupe.com/development/discussing-php-frameworks.html.

[6] Lerdorf Tatroe, MacIntyre, *Programming PHP*, 2013, http://www.infoap.utcluj.ro/multi/programming_PHP.pdf.

[7] Editorial Team, *Web Frameworks: Pros And Cons Of Using Frameworks*, (2015), http://1stwebdesigner.com/web-frameworks/.

[8] Peter Wayner, *7 reasons why frameworks are the new programming languages*, (2015), http://www.infoworld.com/article/2902242/application-development/7-reasons-why-frameworks-are-the-new-programming-languages.html.

Sažetak

Laravel je besplatno PHP razvojno okruženje otvorenog koda namijenjeno brzom razvoju web aplikacija. U kratkom vremenu je zadobio pažnju razvojnih programera i postao jedan od najpopularnijih razvojnih okruženja. PHP razvojno okruženje je alat odnosno slojevita struktura koja služi bržem i sigurnijem razvoju web aplikacije. Pruža stabilan kostur aplikacije, smanjuje količinu ponavljajućih dijelova koda, pomaže održati čitljivost i logiku strukture i olakšava kasnije održavanje. Temelji se na najčešće korištenom MVC (*Model-View-Controller*) obrascu arhitekture aplikacije. MVC dijeli aplikaciju u prezentacijski sloj (pogledi), sloj poslovne logike (upravljači) i podaktovni sloj (modeli). Za upravljanje zavisnostima poput biblioteka uz Laravel se koristi alat Composer. Laravel predstavlja vlastiti alat za predloške Blade koji se koristi u izradi pogleda i očituje se u nastavku ".blade.php". Eloquent ORM je poseban alat kojeg donosi Laravel i uvelike olakšava rad sa bazom podataka. Za svaku tablicu koju stvaramo u bazi podataka možemo stvoriti jedan Eloquent Model koji će nam služiti za svu potrebnu komunikaciju sa danom tablicom.

Summary

Laravel is a free of cost, open-source PHP framework intended for rapid development of web applications. In just a short period of time it captured the attention of web developers and became one of the most popular frameworks. PHP framework is a software framework or a well rounded structure designed to support rapid and safe development of web applications. It provides developers with a basic structure for an application, reduces the amount of repetitive coding, ensures clean code and preserves the logic of the structure and makes the maintaining an easy job. It is based on MVC (*Model-View-Controller*), the most popular architecture pattern in computer programming. MVC ensures the separation of presentation (views), logic (controllers) and data. For dependency managment such as libraries, Laravel uses Composer. Laravel provides with its own templating engine called Blade which is used for creating views and is recognizable by the exstension ".blade.php". Eloquent ORM is a special tool brought by Laravel designed to ease working with database. For each table we create in the database we can create a corresponding Eloquent Model which serves as a special query for all comunication with that table.

Životopis

Valentina Dumbović rođena je 22. veljače 1993. godine u Sisku. Školovanje započinje u Osnovnoj školi Mladost u Lekeniku i nastavlja u Općoj gimnaziji u Velikoj Gorici koju završava 2011. godine. Iste godine upisuje sveučilišni studij Matematike na Matematičkom odsjeku Prirodoslovno-matematičkog fakulteta u Zagrebu. 2014. godine završava preddiplomski studij i dobiva titulu sveučilišne prvostupnice matematike. Iste godine upisuje diplomski sveučilišni studij Računarstvo i matematika. Počinje se baviti razvojem web i desktop aplikacija te odlučuje posvetiti temu diplomskog rada razvoju web aplikacija u popularnom razvojnom okruženju Laravel.