



УНИВЕРЗИТЕТ У НИШУ
ЕЛЕКТРОНСКИ ФАКУЛТЕТ



УПОТРЕБА LARAVELA И ELOQUENT ОБЈЕКТНО РЕЛАЦИОНОГ МАПЕРА ЗА ПРИСТУП ПОДАЦИМА

Дипломски рад

Студијски програм: Електротехника и рачунарство

Модул: Рачунарство и информатика

Студент:

Никола Митић, бр. инд. 16228

Ментор:

Проф. др Александар Станимировић

Ниш, септембар 2025. год.

Универзитет у Нишу

Електронски факултет

**УПОТРЕБА LARAVELA И ELOQUENT ОБЈЕКТНО РЕЛАЦИОНОГ
МАПЕРА ЗА ПРИСТУП ПОДАЦИМА**

**USE OF LARAVEL AND ELOQUENT OBJECT RELATIONAL MAPPER
FOR DATA ACCESS**

Дипломски рад

Студијски програм: Електротехника и рачунарство

Модул: Рачунарство и информатика

Студент: Никола Митић, бр. инд. 16228

Ментор: Проф. др Александар Станимировић

Задатак: Упознати се са појмом и основним концептима коришћења технологије објектно релационих мапера за приступ подацима у релационим базама података. Детаљно проучити Eloquent ORM као типичан пример објектно релационог окружења развијеног за потребе РНР програмског језика. У практичном делу рада, коришћењем РНР програмског језика и Eloquent ORM окружења, развити прототип апликације за управљање такси удружењем.

Комисија за оцену и одбрану:

Датум пријаве рада: _____ 1. _____

Датум предаје рада: _____ 2. _____

Датум одбране рада: _____ 3. _____

УПОТРЕБА LARAVEL И ELOQUENT ОБЈЕКТНО РЕЛАЦИОНОГ МАПЕРА ЗА ПРИСТУП ПОДАЦИМА

САЖЕТАК

Савремено пословање носи са собом велику количину података, како би се имплементирала софтверска решења, креирају се базе података са великим бројем табела и генерално великом количином података. У раду се представљају технологије које се користе у позадини оваквог система. Највише је објашњена употреба *backend* система, његова функционалност заснована на чувању, читању и обради тих података у релационој бази података као и адекватном начину представљања истих остатку система.

Представљен је принцип превођења података базе података у објектно релационе моделе. Извођењем овог поступка добијамо структуру података којом знамо како да манипулишемо и вршимо даљу припрему за употребу. Затим се пружа детаљни опис система који приказује имплементацију жељеног решења и процес креирања истог. Финално долази се и до приказа рада саме апликације, односно демонстрације жељеног решења на примеру у реалности.

Главни циљ рада је имплементација модерних технологија које омогућују лак приступ и обраду комплексних података. Коришћењем радног окружења *Laravel* на *backend*-у који омогућава једноставну обраду захтева и повезивање са *MySQL* базом података добијамо систем који лако и брзо чита и обрађује све податке коришћењем објектно релационог мапера, *Eloquent ORM*-а. Имплементацијом *frontend*-а у радном окружењу *Angular* добијамо систем који се покреће у сваком веб претраживачу са прилагодљивим приказом. Овај систем остварује директну интеракцију *frontend*-а и *backend*-а лако вршећи модификацију података. Пренос података додатно се може обогатити имплементацијом *WebSocket*-а који омогућују пренос података у реалном времену. На основу пренесених података, коришћењем система догађаја, и имплементацијом *Google Maps API*-а омогућује се приказ кретања корисника на мапи.

Употребом *Eloquent* релационог мапера, подаци су повезани у комплексну структуру, обрађени у радном окружењу *Laravel* и приказани прилагодљивим дизајном коришћењем радног окружења *Angular*. Постигнута је аутоматизација управљања системом и пружена једноставна употреба кроз веб претраживач.

Кључне речи: подаци, савремено, приступ подацима, мапирање података, пренос података у реалном времену, повезивање.

USE OF LARAVEL AND ELOQUENT OBJECT RELATIONAL MAPPER FOR DATA ACCESS

ABSTRACT

Modern business operations involve handling vast quantities of data. To implement software solutions that support such operations, databases with numerous tables and huge quantities of data are created. This paper presents the technologies used behind such systems. Emphasis is placed on the use of the backend system, whose functionality is based on storing, reading, and processing data in a relational database, as well as on presenting that data appropriately to other parts of the system.

The process of converting database data into object-relational models is explained. Through this process, a well-structured dataset is obtained—one that can be efficiently manipulated and further prepared for use. A detailed system overview is then provided, illustrating the implementation of the intended solution and the development process. Finally, the functionality of the application is demonstrated through a real-world example.

The main goal of this paper is to implement modern technologies that enable efficient access to and process complex data. By using Laravel framework on the backend—which facilitates simple request handling and integration with a MySQL database—the system efficiently processes data through the object relational mapper, Eloquent ORM. Implementing the frontend in Angular framework results in a responsive system that runs in any web browser. The system enables direct interaction between the frontend and backend, allowing for seamless data exchange. Real-time data transfer can also be achieved through the implementation of WebSockets. Based on the transmitted data, and with the use of an event-driven system and the Google Maps API, it is possible to display user movement on a map.

Using the Eloquent relational mapper, data is linked into a complex structure, processed in Laravel framework, and presented through Angular framework in a responsive design. This implementation achieves automation of system management and ensures easy usage through a web browser.

Keywords: data, modern, data access, data mapping, real-time data transfer, connectivity.

САДРЖАЈ

| | |
|--|----|
| 1. УВОД | 8 |
| 2. <i>PHP</i> ПРОГРАМСКИ ЈЕЗИК И РАД СА БАЗАМА ПОДАТАКА..... | 9 |
| 2.2. <i>PHP</i> ПРОГРАМСКИ ЈЕЗИК | 9 |
| 2.3. Релационе базе података | 10 |
| 3. РАДНО ОКРУЖЕЊЕ <i>LARAVEL</i> | 13 |
| 3.2. Опште..... | 13 |
| 3.3. Рутирање | 14 |
| 3.4. Валидација | 15 |
| 3.5. <i>MVC</i> архитектура | 16 |
| 3.6. <i>Redis</i> | 17 |
| 3.7. <i>Homestead</i> | 18 |
| 4. ОБЈЕКТНО РЕЛАЦИОНИ МАПЕР <i>ELOQUENT ORM</i> | 20 |
| 4.2. Историја | 20 |
| 4.3. Опште..... | 21 |
| 4.4. Мапирање модела и његова употреба..... | 22 |
| 5. ИМПЛЕМЕНТАЦИЈА АПЛИКАЦИЈЕ..... | 25 |
| 5.2. Архитектура апликације..... | 25 |
| 5.3. <i>MySQL</i> | 26 |
| 5.4. Радно окружење <i>Angular</i> | 26 |
| 5.5. <i>Web Socket</i> | 28 |
| 5.6. <i>WebStorm</i> | 30 |
| 5.7. <i>PhpStorm</i> | 30 |
| 5.8. Опис имплементације базе података | 30 |
| 5.9. Имплементација серверског дела..... | 33 |
| 5.9.1. Објектно релациони мапер <i>Eloquent ORM</i> | 37 |
| 5.10. Имплементација клијентског дела | 40 |
| 5.10.1. Приказ мапе | 43 |
| 6. РАД АПЛИКАЦИЈЕ | 46 |
| 6.2. Опис апликације..... | 46 |
| 6.3. Профил корисника апликације | 47 |
| 6.4. Случајеви коришћења..... | 47 |
| 7. ЗАКЉУЧАК | 59 |

| | |
|--------------------|----|
| 8. ЛИТЕРАТУРА..... | 60 |
|--------------------|----|

1. УВОД

Једна од главних карактеристика савременог живота јесте непрестано увећање дигиталног отиска који свака особа креира. Као резултат, потребно је изборити се са све већом количином података, са често веома комплексном повезаношћу. У сврху лакше обраде ових података развијене су бројне технологије и алати.

Један од најпознатијих и најраспрострањенијих програмских језика који се користи за израду веб апликација јесте *PHP* програмски језик. Овај програмски језик пружа једноставну имплементацију и добру документацију. Поред тога, карактеристика *PHP* програмског језика је и постојање великог броја радних окружења односно *framework*-а. Једно од најпознатијих окружења за развој веб апликација је радно окружење *Laravel*.

Комуникација апликације са базом података један је од главних карактеристика радног окружења *Laravel*. Повезивање апликације са релационим базама податка и разумевање садржаја истих постиже се објектно релационим маперима (ORM). Радно окружење *Laravel* са *PHP* програмским језиком нам пружа *Eloquent ORM*.

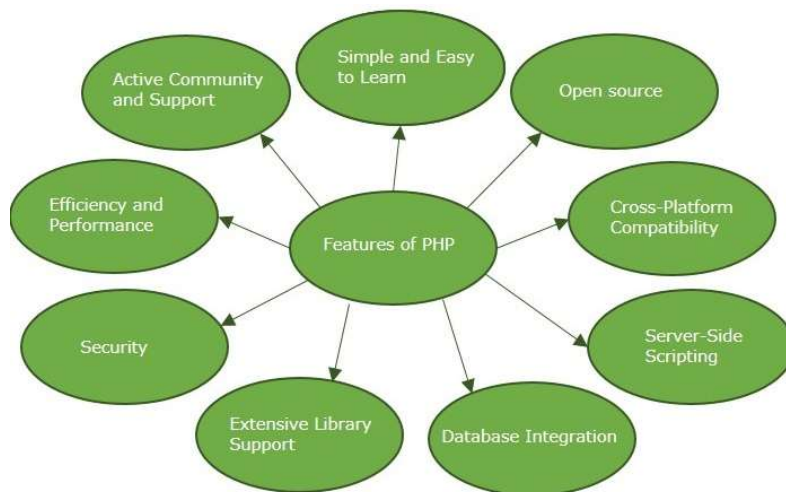
У практичном делу рада биће имплементиран прототип веб апликације за такси удружења. Такав савремени софтвер би омогућио лако поручивање и прихватање вожњи, праћење вожње у реалном времену и праћење историје корисникових захтева. На овај начин врши се практични приказ употребе савремених технологија за обраду, чување и приказ података у реалном времену. За имплементацију овакве апликације биће коришћен *PHP* програмски језик кроз радно окружење *Laravel* са својим *Eloquent ORM*-ом, релациона база података *MySQL*, нерелациона база података *Redis* и *TypeScript* програмски језик кроз радно окружење *Angular*. Одабир конкретно ових технологија базиран је на њиховој одличној документацији, честој примени у савременим софтверским решењима, што додатно олакшава одржавање и развој апликација, лакоћи имплементације и брзини обраде и приказа резултата крајњем кориснику.

Рад који следи у наставку је структуриран на следећи начин. У другом поглављу дато је детаљно објашњење подршке за рад са базама података у *PHP* програмском језику. У трећем поглављу детаљно је описано радно окружење *Laravel* које се користи за развој *backend* дела веб апликација. У четвртном поглављу описан је *Eloquent* објектно релациони мапер који се користи за рад са подацима. Пето поглавље садржи објашњења технологија које су додатно коришћене приликом израде практичног дела овог дипломског рада. Осим тога у петом поглављу описана је и структура имплементираних апликација и различити имплементациони детаљи. У шестом поглављу налази се приказ употребе имплементираних веб апликација. Седмо поглавље садржи закључак, а осмо поглавље списак коришћене литературе.

2. PHP ПРОГРАМСКИ ЈЕЗИК И РАД СА БАЗАМА ПОДАТАКА

2.2. PHP ПРОГРАМСКИ ЈЕЗИК

PHP програмски језик је почео као мали *open source* пројекат који је временом порастао до светски најпознатијег програмског језика за израду веб апликација. Иницијална идеја именовања овог програмског језика била је скраћеница за *Personal Home Page*, међутим, развојем и унапређењем могућности овог програмског језика долази се и до усложњавања самог назива те данас назив *PHP* представља рекурзивну скраћеницу за *PHP: Hypertext Preprocessor*. Развој програмског језика испраћен је дефинисањем 8 главних верзија, са сваком верзијом језик постаје све комплекснији и моћнији. Развој *PHP* програмског језика додатно подстиче *open source* карактеристика. Неке од његових функционалности можемо видети на слици 1.



Слика 1 – PHP функционалности [1]

Овај програмски језик је скриптни програмски језик. То значи да се његов код не компајлира унапред, већ се редом извршава команда за командом. Команде се раздвајају знаком „;“. Дефинисање имена променљивих је *case-sensitive*, односно навођењем *\$firstparam* и *\$firstParam* указује се на две различите променљиве. Имена класа, интерфејса, функција и др. су *case-insensitive*, односно инстанцирањем *new MyClass()* и *new MYCLASS()* креирамо инстанцу исте класе. У зависности од потреба употребе, *PHP* програмски језик се може користити за командно извршавање наредби или се може интерпретирати и извршавати на серверу. За потребе израде веб апликација, користи се други приступ, односно серверско извршавање кода. Сва логика написаног кода налази се у текстуалним датотекама смештеним на серверима. Позивањем руте на којој се налази *php* скрипта, долази до покретања извршења команди. Сервери могу бити било које софтверске платформе јер *PHP* програмски језик подржава извршење на свим најпознатијим оперативним системима. [2]

Усложивањем захтева за израду веб сајтова, напредовало се са развојем програмског језика. Како писање оваквих кодова изискује константу употребу истих основних функционалности, долази се до развоја и употребе разних радних окружења односно *framework*-а. Они пружају могућност брзе употребе већ написаних функционалности смештених у библиотеке. На овај начин олакшава се развој апликација, брзина израде и лакоћа одржавања. Нека од најпознатијих развојних окружења су:

- *Laravel* – најпознатије окружење за развој *PHP* апликација са највећом базом корисника. Пружа одличну документацију у виду текстуалних објашњења и примера али и добро документованих видео материјала. Користи *MVC* архитектуру;
- *Symfony* – најстарије окружење које опстаје и развија се са сваком *PHP* верзијом. Користи се за апликације израђене за потребе великих организација попут корпорација. Користи *MVC* архитектуру. Логика имплементирања *PHP* функционалности дефинисана је *brick-by-brick* системом. Овај систем представља поделу кода у *bundle*-е, односно пакете. Самим тим, у зависности од потребе система, користе се само неопходни пакети, а то даље узрокује израду великих апликације које заузимају мање меморије од других сличних апликација развијених у осталим окружењима; [3]
- *CodeIgniter* – најмање развојно окружење програмског језика *PHP*. Генерално се користи за мање и средње апликације, али у случају комплекснијих апликација и потребе за додатним комплекснијим библиотекама, оне се могу динамично додати и проширити радно окружење. Користи *MVC* архитектуру. Веома лако се повезује са базом података [4];

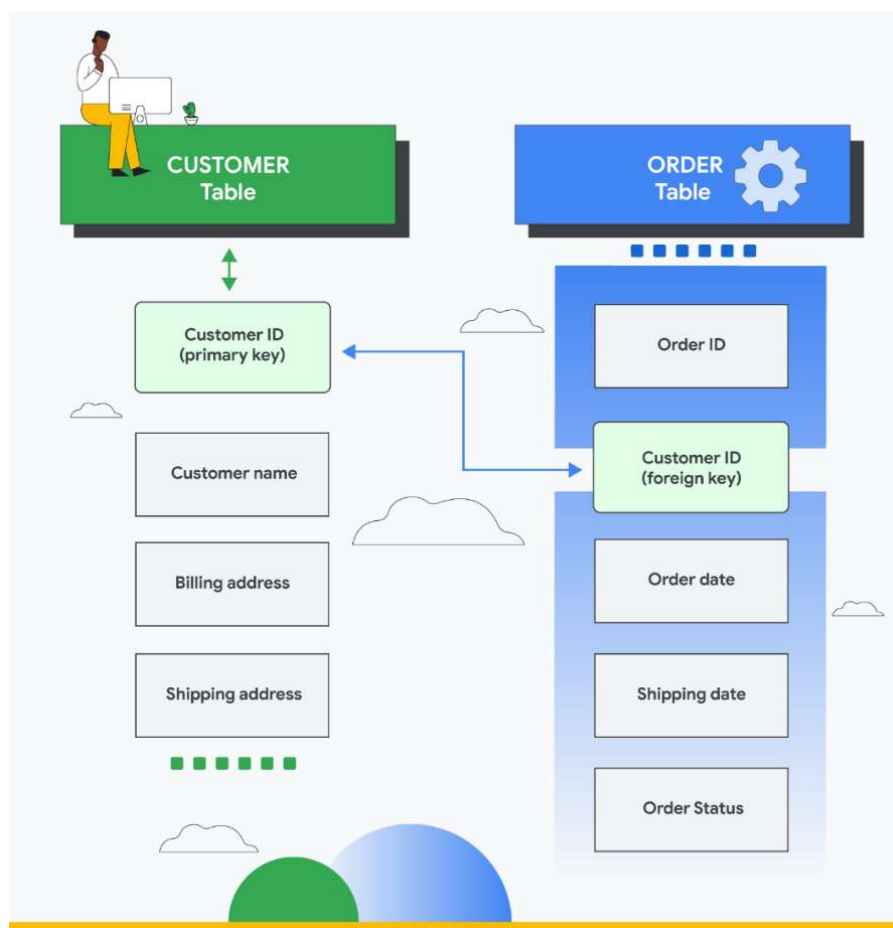
Програмски језик *PHP* се најчешће користи за израду веб апликација из разлога лаке интеграције са *HTML* кодом. Програмски језик *PHP* се са својом скриптном ознаком `<?php ... ?>` лако додаје у било коју *HTML* конструкцију, додајући додатне могућности обраде и приказа података. Одлична подршка за лако повезивање са базама података омогућује лако креирање страница попуњених динамично генерисаним подацима из табела база података. Употребом *framework*-а резултати обраде *PHP* кода осим генерисања *HTML* страница, могу бити и генерисање текстуалних и *JSON* форматираних података, генерисање *PDF* докумената, извршавање акција попут слања *e-mail*-ова, и друго. [5]

2.3. Релационе базе података

Релационе базе података представљају начин организовања података у табеле. Свака табела представља засебан скуп сличних ентитета из реалног света. Колоне табела дефинишу особине којима се описују ентитети реалног света, а сваки ред представља опис по једног ентитета. Ред уписа ентитета у табелу, односно редослед уношења редова није битан. Сваки ентитет може се јединствено идентификовати, неки од ентитета чак и на више начина. Издвајањем атрибута или минималног скупа атрибута који имају јединствену вредност за све појаве одређеног типа ентитета дефинише се кључ. Сваки ред може имати више кључева, називамо их кључеви кандидати. Један од њих, чија вредност никада није недефинисана, односно никада није *null*, и који најбоље идентификује тај ентитет

представља примарни кључ. [6] По аутоматизму, примарни кључ већине релационих база података је редни број уписа у табелу, али може бити било која јединствена вредност. Данас се све чешће користи *UUID* као примарни кључ. *UUID* је 32-карактерна вредност генерисана комбинацијом цифара од 0 до 9 и латиничних слова од А до F у формату 8-4-4-4-12. Како је то 128-битна вредност, сваки систем је може генерисати, а вероватноћа генерисања истог кључа статистички је близу нуле. [7]

Како многе ентитете због своје комплексности није једноставно објаснити једном изјавом и како многа објашњења укључују и додатне информације о другим ентитетима, тако и њихово представљање у бази података захтева креирање додатних описних или повезаних табела. Табеле се у релационом систему повезују коришћењем примарног кључа. Смештањем примарног кључа директно у другу табелу или у посредничку табелу, у ред са којим се логички може повезати, ствара се веза у релационој бази података. Овај кључ, смештен у другу табелу, води се под именом страни кључ јер он јединствено дефинише везу са ентитетом друге табеле. На овај начин стварају се различити типови веза у релационој бази података. Овакав систем омогућује детаљно и структурирано чување комплексних података из реалног света.



Слика 2 – Пример релационе базе података [8]

Пример са слике 2 представља саму срж идеје која стоји иза креирања релационих база података, међутим релационе базе као такве се користе у много комплекснијим системима са доста више атрибута и повезаности међу табелама. Пружа се могућност вишеструког начина повезивања и прибављања груписаних и филтрираних података по жељи корисника. За рад са оваквим подацима користе се управљачки системи релационих база података (*RDBMS*) као што су *MySQL*, *PostgreSQL*, *MariaDB*, *Microsoft SQL Server* и *Oracle Database*.

Модел релационе базе података настао је 1970их година са циљем замене хијерархијске структуре организације података. Хијерархијска структура података представљала је приказ података у нивоима по припадности. На овај начин вршио се вишеструки унос истих података. Овакав систем имао је велику редундантност података, грешке приликом вишеструких уноса, грешке приликом измене тих података јер су захтевале исту измену на више места и велику комплексност прибављања података. Како би се избегле мане оваквог система креиран је нови систем. Релациони модел не захтева вишеструки унос истих података већ креира једну табелу која се може повезати са више других, а не само са хијерархијском табелом изнад ње. На овај начин избегнута је потреба за константном реорганизацијом табела. [8] Коначно, овакав систем показао се доста ефикаснији у самом одржавању али и прибављању података и извршењу измена истих.

Тако креиране базе података и њихови подаци даље се мапирају како би се употребили у апликацијама направљеним објектно оријентисаним језицима. Објектно релационо мапирање преводи табеле у објекте, а атрибуте табела у атрибуте објекта. На овај начин генерише се објектни преглед података, штеди се време и олакшава рад креирањем предефинисаних упита ка бази и повећава сигурност спречавањем *SQL injection* напада на базу података односно напада на базу података који се извршава слањем упита ка бази са циљем вршења недозвољене измене или прибављања података. [9]

3. РАДНО ОКРУЖЕЊЕ *LARAVEL*

3.2. Опште

Радо окружење *Laravel* је *framework* који користи *PHP* програмски језик. Направљен је са циљем олакшања развоја веб апликација, без потребе да се све увек развија од самог почетка. Омогућава креирање апликација са мноштвом већ имплементираних делова како би се лакше фокусирао на имплементацију решења проблема који су специфично потребни за апликацију. Неке од ствари које ово развојно окружење пружа су аутентификација, *middleware* који на основу дефинисаних правила проверава право приступа подацима, рутирање, рад са базом података, итд. [10] Генерално, ово није први *PHP framework*, већ је настао као потреба и жеља за даљим развојем и унапређењем. Како су многе функционалности недостајале у *Codeigniter*-у, *Taylor Otwell* решио је да започне пут развоја новог радног окружења. Новокреирано радно окружење под именом *Laravel* је објављено 9. јуна 2011. године. Прва верзија одмах је дошла са уграђеном аутентификацијом, *Eloquent ORM*-ом за рад са базама података, моделима и њиховим везама, подршком за вишејезичност система, механизмом за једноставно креирање рута и радом са кешом, сесијама и другим стварима. Овакав систем донео је многе погодности и предности рада са *PHP* програмским језиком, те је врло брзо настављено са његовим развојем. У првих шест месеци се већ дошло до нове верзије радног окружења, а данас имам чак дванаест верзија. Развој радног окружења *Laravel* је доста окренут његовим корисницима и њиховим потребама. Временом се кроз верзије дошло до развија многих погодности. Неке од успутних тачака развоја су контролери, *Blade* странице, *MVC* архитектура, тестови, *Artisan* командни интерфејс, миграције база података, подела система у пакете који се контролишу *Composer*-ом и многе друге. Радно окружење достигло је препознатљивост по својим предностима да користи добро познате обрасце за развој софтвера, по томе што је по креирању апликације одмах спремно за даљи развој и употребу, по томе што је на почетку све подешено по већ установљеним конвенцијама али и даље пружа могућност лаке измене конфигурација, по својој одличној структури докумената и по многим другим карактеристикама. [11]

Artisan, као једна од великих предности употребе радног окружења *Laravel*, представља његов командни интерфејс, односно *CLI*. Овај командни интерфејс доноси моћан сет команди које олакшавају процес развоја апликација, аутоматизују разне акције и побољшавају продуктивност. Најпознатије и најкоришћеније команде су ***make*** и ***migrate***. Команда ***make*** се углавном користи за креирање скелета модела, контролера, миграција и других компоненти, чиме се убрзава процес развоја апликације и избегава процес писања истог основног кода сваке компоненте. Додатно, ова команда пружа могућност развоја самог *Artisan*-а креирањем нових, ручно направљених, команди. Овакве команде представљају специјално развијену логику потребну за конкретне системе и оне се могу покретати на исти начин као и све остале команде. Команда ***migrate*** се користи за олакшани рад са базом података омогућујући извршење, поништавање и проверу миграционих докумената који служе за дефинисање изгледа базе података. Поред употребе ових команди, *Artisan* доноси додатне могућности управљања базом података, покретања тестова, управљања кешом и *HTML* ресурсима, као и многе друге погодности. [12] Додатно, овај командни интерфејс

заједно са већ инсталираним пакетом *Laravel Tinker* омогућава интеракцију са целим системом кроз командни интерфејс. Оваква комбинација може омогућити тестирање али и праву измену података у систему, односно њено покретање омогућује лак преглед података и функционалности система без потребе постојања *frontend*-а.

Како ниједна веб или мобилна апликације не може бити израђена само употребом *PHP* програмског језика, овај *framework*, иако је углавном окренут развоју *backend* дела апликације, ипак пружа могућност целокупног развоја веб апликација. Ово се углавном постиже коришћењем *Laravel Blade* компоненти у самом *framework*-у. Предност њихове употребе је то што је већина основних шаблона за израду *frontend* дела већ имплементирана у *framework*-у са додатком *CSS*-а и *JS*-а. Додатно, у случају потребе за модификацијом ових компоненти, оне се могу *publish*-овати и прилагодити специјалним потребама. Овакве компоненте се одлично уклапају у *Laravel*-ову *MVC* архитектуру, конкретно у *V (Views)* део. Након рутирања и обраде података врши се припрема приказа података кориснику. *Blade* долази као шаблонски систем који омогућава лако читање и приказ података обрађених у радном окружењу *Laravel*, умањује потребу за понављањем истог кода коришћењем услова, петљи и секција. Међутим, из искуства, у раду са *Single Page Applications*, скраћено *SPA*, препоручује се његова употреба само као моћног *API* сервиса. Оваква употреба омогућује јасну поделу кода апликације на *frontend* и *backend*. Радном окружењу *Laravel* се на овај начин оставља обрада података и имплементација логике апликације. Док се, на пример, радном окружењу *Angular* оставља дефинисање корисничког интерфејса и модерног дизајна уз што мање чекања на одговор и што мање освежавања апликације.

3.3. Рутирање

Рутирање је веома битан део развојног окружења *Laravel*. Представља могућност комуникације корисника са системом путем *HTTP* захтева. У зависности од тога да ли је резултат који се рутом враћа приказ веб странице или обрађени податак, руте се деле у два документа унутар *routes* директоријума и то на *web.php* и *api.php*. Креирање рута је почетни задатак приликом имплементирања нових функционалности сваког *Laravel* система и то се постиже употребом *Route* фасаде која омогућава употребу *Router* класе са предефинисаним методама. Најчешће употребљене методе су заправо *HTTP* методе:

- GET – користи се за прибављање података,
- POST – користи се за креирање и упис нових података у базу података,
- PUT – користи се за измену постојећих података у бази података,
- DELETE – користи се за брисање података. [13]

Дефинисање рута води у креирање веома добро структурираног система. Свака рута води до одговарајуће функционалности имплементиране у систему. Процес рутирања у многоме зависи од самог дефинисања руте. Креирање структуре руте са овим методама веома је битан процес. Руте се могу филтрирати специјално дефинисаним функцијама. Ове функције долазе заједно са радним окружењем, а додатно је могуће креирати и нове по потреби. Филтерима се постиже основни концепт контроле приступа, односно дефинисања права приступа корисника подацима и обради истих. Такође, ради избегавања грешке у писању и ради смањивања редундантности кода, руте се могу груписати коришћењем истих

префикса. Руте могу имати различите параметре, они могу бити обавезни или опциони и такође може их бити и више у само једној рути. На основу тога, зна се које податке ће имплементирана функција користити. [14] У случају прибављања података неког модела, добро је пратити конвенцију креирања руте, односно начина прослеђивања параметара. Коришћењем имена модела, рута се треба креирати тако да име модела прати идентификациони број, док је за резултат обраде потребно упутити на функцију унутар контролера који припада том моделу. Правилном конструкцијом руте, прослеђивањем идентификационог броја ентитета тог модела, радно окружење *Laravel* ће само прибавити цео објекат и проследити га функцији на обраду. Овај процес познат је под именом *Route Model Binding* и веома је користан јер скраћује и олакшава процес имплементације жељене функционалности. [15] На слици 3 приказан је пример употребе *Route Model Binding*-а.

```
use App\Models\User;

Route::get('/users/{user}', function (User $user) {
    return $user->email;
});
```

Слика 3 – Пример употребе *Route Model Binding*-а

3.4. Валидација

Осим прослеђивања идентификационог броја или другог уникатног податка ради проналажења инстанце ентитета у бази података, руте малтене увек са собом носе додатне податке. Ови подаци представљају *request data*, односно податке захтева. Овакве податке систем не би требао тек тако обрађивати јер се руте система на тај начин могу злоупотребити или неправилно искористити. Због тога се уводи систем валидације.

Validator класа радног окружења *Laravel* састоји се од бројних валидационих функционалности којима се обрађују прослеђени подаци који могу потицати од неке попуњене форме или бити било који тип података који се шаље на *backend*. Валидатором се дефинишу правила којим се, на пример, провера да ли су сви неопходни подаци прослеђени, да ли су одговарајућег типа, да ли није случајно прослеђен неки забрањени податак или недозвољена комбинација података и слично. Ова правила представљају кључ вредност парове раздвојене „:“ карактером. Сваки кључ представља име податка, а вредност скуп правила које тај податак мора да поштује. Скуп правила, односно вредност, може се дефинисати навођењем правила које се раздвајају „|“ карактером или као низ правила раздвојених запетом унутар угластих заграда. На слици 4 приказан је пример једне валидационе функције.


```
$request->validate([
    'title' => 'required|unique:posts|max:255',
    'author.name' => 'required',
    'author.description' => 'required',
]);
```

Слика 4 – Пример валидације са угњежденим подацима

У случају да подаци не прођу валидацију, вратиће се грешка са *error* атрибутом који ће садржати све информације о невалидности прослеђених података. Оваква информација може се приказати кориснику, а у случају валидације форме често ће се форма поново попунити прослеђеним подацима, али ће овог пута уз те податке стајати и информација о невалидности истих. Док се у супротном наставља са даљим извршењем. [16]

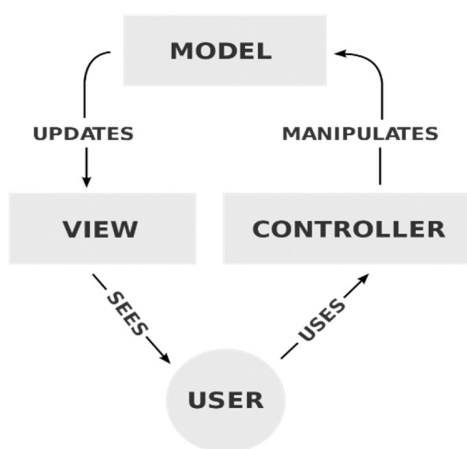
Оваква валидација може се имплементирати директно у контролеру, али ради боље прегледности радно окружење *Laravel* омогућава креирање валидационих класа. Ове класе проширују *FormRequest* класу, те додатно, на свој начин дефинишу *authorize* и *rules* функције. Овим функцијама постиже се специјализована провера да ли корисник који шаље захтев заправо има право приступа тој рути и дефинише се низ валидационих правила. Додатно, ова класа пружа могућност предефинисања и *messages* функције. *Messages* функцијом одређују се валидационе поруке које се враћају кориснику као информација о невалидности у случају да прослеђени подаци не пролазе валидациона правила. [17]

3.5. MVC архитектура

Апликације радног окружења *Laravel* имају по аутоматизму исту, специфично дефинисану структуру директоријума. У овој структури јасно се издвајају директоријуми *models*, *views* и *controllers*. По самој архитектури директоријума види се да се у радном окружењу *Laravel* користи *model view controller (MVC)* архитектура. Ова архитектура омогућава бржи развој апликације, лако кориговање система и повећање скалабилности истог. Додатно, можда и најбитнија, карактеристика ове архитектуре је прегледност имплементирања. Постиге се раздвојеност између имплементација информација о систему, корисниковим акцијама са тим информацијама и визуалне презентације информација. [18] Илустрацију примене архитектуре могуће је видети на слици 5. Како само име каже, овом архитектуром је развојни процес подељен у три главна дела:

- *M* представља скраћеницу за *model*, што заправо представља руковођење подацима. Начин употребе, повезаност и обада података дефинишу се у оваквим класама. Модели представљају спону између контролера и базе података. У њима се може дефинисати које податке можемо читати, уписивати, типски трансформисати, како ће се везе међу подацима тј. моделима у систему посматрати и прибављати.

- *V* представља скраћеницу за *view*, односно репрезентацију података на корисничком интерфејсу. Овај део постиже се коришћењем *HTML*-а, *CSS*-а и *JavaScript*-а. Поглед добија све податке обрађене од стране контролера и врши њихов приказ у прегледном формату који одговара људском оку.
- *C* представља скраћеницу за *controller*, најкомплекснији део ове архитектуре. Сви захтеви за обрадом или прибављањем података преко рутера долазе до контролера и ту се догађа најбитнији део *backend* логики. Пратећи добру праксу, сав код који представља обраду или директну комуникацију са моделом издваја се у сервисе који се позивају из контролера. Контролер може обрадити податке, форматирати их на жељени начин и вратити у *JSON* формату или тако припремљене проследити погледу и вратити целу компоненту за приказивање. [19]



Слика 5 – MVC архитектура у примени [10]

3.6. Redis

Осим рада са структурираним релационим базама података радно окружење *Laravel* омогућује и употребу нерелационих база података. Како само име каже, овакве базе података омогућују упис неструктурираних података по принципу кључ вредност односа. Веома су лаке за подешавање и често се могу користити за кеширање привремених вредности. Такође, оне у потпуности могу заменити употребу релационих база података уколико је то потребно за одређени тип пословања система. У случају рада са подацима које је потребно брзо прибавити али не и дуго задржати у систему, као и када подаци нису обликовани по структури објеката, овај систем чувања података је одличан. [20] Генерално, *Redis* који је можда и најкоришћенији тип ове врсте базе података, доста се користи у хибридном раду са стандардним базама података.

Redis представља добро развијени систем чувања различитих врста података у кључ вредност односу. Један кључ може имати вредност у виду текста, листе, сета података или хеш табеле. Чување различитих типова вредности имплементирано је различитим

специјално генерисаним функцијама које могу да раде са тим типом података. Примере чувања и читања података можемо видети на слици 6.

```
use Illuminate\Support\Facades\Redis;

Redis::set('demo_type', 'string');
Redis::get('demo_type');

Redis::sadd('set', json_encode($demoObject1));
Redis::sadd('set', json_encode($demoObject2));
Redis::smembers('set');
```

Слика 6 – Пример додавања и читања текстуалног података и сета података

Овако сачувани подаци најчешће се користе за:

- кеширање вредности чије ће прибављање бити веома брзо потребно. Овакав начин употребе се често користи са *Cache* фасадом радног окружења *Laravel* подешавањем *CACHE_DRIVER* атрибута конфигурационог документа на вредност *redis*,
- чување распореда *Job*-ова за извршење у случају синхроног или одложеног извршења, подешавањем *QUEUE_CONNECTION* атрибута конфигурационог документа на вредност *redis*,
- пренос података у реалном времену између клијентског и серверског дела система, углавном приликом обавештавања клијентске стране о променама које се дешавају на серверској страни. [21]

3.7. Homestead

Homestead је развијен како би се избегло локално подешавање система програмера за рад у развојном окружењу *Laravel*. Он представља употребу виртуелне машине на којој се налази окружење потпуно опремљено свим сервисима за развој *Laravel* апликације, односно пружа кориснику на употребу *PHP* програмски језик, *MySQL* релациону базу података, *Redis* нерелациону базу података, *Node* пакет менаџер и друге сервисе. Може се покренути на *Windows*, *Mac* или *Linux* системима. Да би се покренуло, потребно је на рачунару подесити *VirtualBox* 6.1.x и *Vagrant*. Пре самог покретања, потребно је дефинисати конфигурациони документ. Овим документом *Homestead* добија сазнања како да преслика податке са локалног система на виртуелну машину, које портове за комуникацију да отвори и на којим веб адресама да покрене апликацију. [22] На слици 7 приказан је пример конфигурационог документа.

```

---
ip: "192.168.56.56"
memory: 2048
cpus: 2
provider: virtualbox

authorize: ~/.ssh/id_rsa.pub

keys:
  - ~/.ssh/id_rsa

folders:
  - map: ~/code
    to: /home/vagrant/code

sites:
  - map: homestead.test
    to: /home/vagrant/code/public

databases:
  - homestead

features:
  - mariadb: false
  - postgresql: false
  - ohmyzsh: false
  - webdriver: false
  - influxdb: false

services:
  - enabled:
    - "mysql"

```

```

#   - disabled:
#     - "postgresql@11-main"

# ports:
#   - send: 33060 # MySQL/MariaDB
#     to: 3306
#   - send: 4040
#     to: 4040
#   - send: 54320 # PostgreSQL
#     to: 5432
#   - send: 8025 # Mailpit
#     to: 8025
#   - send: 9600
#     to: 9600
#   - send: 27017
#     to: 27017

```

Слика 7 – Изглед Homestead.example.yaml конфигурационог документа

4. ОБЈЕКТНО РЕЛАЦИОНИ МАПЕР *ELOQUENT ORM*

4.2. Историја

Један од најважнијих задатака радног окружења *Laravel* је прикупљање, обрада и чување података. Током развоја радног окружења долазило је до различитих начина извршења ових задатака. Испрва, користио се најједноставнији, али уједно и најкомплекснији и најнепрегледнији начин, а то је писање чистог *SQL* кода. Иако је овај приступ најбржи у смислу времена потребног за извршење жељене акције, само писање и касније разумевање наредби није најбоље могуће решење. Овакав приступ се данас користи у ретким случајевима, како би се избегло креирање грешака приликом писања самих наредби. Подаци прибављени на овај начин представљају низ података. Потом, долази до увођења *Query Builder*-а. Овај метод рада представља унапређену верзију писања захтева ка бази података. Овако написани захтеви доста су ближи командама *PHP* програмског језика. Коришћењем предефинисаних наредби, остварује се веза са базом података, односно са директно жељеном табелом. Даљим навођењем помоћних функција врши се филтрирање, прибављање или измена података. Подаци прибављени на овај начин представљају колекције података. Коначно, применом *Active Record* обрасца долази се до дефинисања *Eloquent ORM*-а. Овом методом рада постиже се потпуно пресликавање података из базе података на објекте *PHP* програмског језика. Приступ и обрада ових података дефинишу се у моделима. Подаци прибављени на овај начин представљају објекте података. Наведени начини прибављања података приказани су на слици 8.

```
// raw SQL
$users = DB::select('SELECT * FROM users WHERE active = ? ORDER BY name', [1]);

// Query Builder
$users = DB::table('users')->where('active', 1)->orderBy('name')->get();

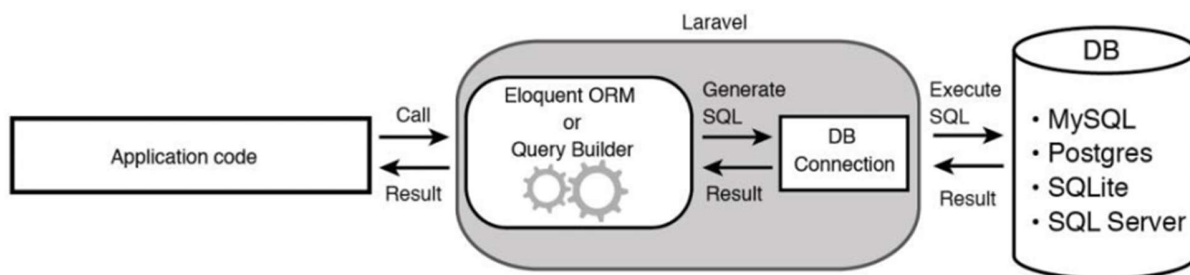
// Eloquent ORM
$users = User::where('active', 1)->orderBy('name')->get();
```

Слика 8 – Пример различитог начина прибављања података

Сам пример имплементације ових метода прибављања података показује како лакоћа имплементирања расте у смеру од чистог *SQL*-а, преко *Query Builder*-а, све до *Eloquent ORM*-а. Посматрајући сва три метода и њихове карактеристике, као и потребе имплементације програмер доноси одлуку о томе шта је најпотребније за његов систем. Иако је *Eloquent ORM* најједноставнији, можда није увек и најбољи одабир. Те због тога кроз сам развој, увек је остао у употреби и претходно имплементирани начин рада са подацима.

4.3. Опште

Како је већ наведено, коришћењем *Active Record* обрасца врши се имплементација објектно релационог мапера или ти *Eloquent ORM*-а. Он служи за повезивање података апликације са подацима из базе података, што се илустровано може видети на слици 9. Свака табела базе података посматра се као класа модела од које се креира објекат. Све колоне табела посматрају се као атрибути објекта, који се такође дефинишу у моделу. Како се мапирање ових података врши у моделу, ово заправо представља праву примену *MVC* архитектуре. [23] Без употребе ове технологије, свака комуникација са базом података би била доста комплекснији и дуготрајнији задатак. Како је сама идеја постојања радног окружења олакшање развоја апликација, тако је исто и код употребе објектно релационог мапера. Повећањем комплексности софтвера, дошло је до повећавања количине података, па сходно томе и бројем табела у бази података, те више ни једно софтверско решење не ради са једноцифреним бројем табела у бази података. Имплементација обраде података на старе начине постаје све компликованија. Долазећи са сетом већ унапред дефинисаних инструкција, објектно релационо мапирање остварује веома брз приступ подацима. Осим директног приступа подацима, остварује се брз и лак приступ и подацима који су у релацији са тренутно прибављеним моделом.



Слика 9 – *Eloquent ORM* принцип рада [24]

Имплементација радног окружења *Laravel* пропраћена је дефинисањем конвенција, те исто важи и код објектно релационог мапирања. Мапер на основу креиране класе модела дефинише аутоматско име табеле, као име модела само у множини. Ова конвенција, као и све остале, постоји по аутоматизму при креирању, али је увек могуће изменити је специјалним дефинисањем у моделу. [25]

Рад са подацима постиже се веома једноставно. Дефинисање приступне команде почиње именом модела, пропраћен двоструком двотачком и конкретном методом или низом метода. Као и код *Query Builder*-а, ове методе могуће је надовезивати једну на другу све док њихово извршавање има смисла у целисти.

Када говоримо о приступу подацима, све рекорде једне табеле, односно све инстанце модела могуће је прибавити коришћењем методе *all()* или методе *get()* у случају када се директно везује на модел без метода за филтрирање. Модел је могуће филтрирати методама *find(\$id)* или *where('column', 'value')* уз додатак метода *get()* или *first()*. На овај начин, метода *find*, функционише по дефинисаној конвенцији тражећи у бази јединствени идентификатор, тј. главни кључ. Метода *where* примениће жељени филтер, а зависно од пропратне методе за

прибављање, добићемо један или више ентитета у резултату. Додатно, како постоје везе између модела, односно табела, навођењем команде *with('model_name')* одлучујемо се за *eager loading*, тј., не навођењем одлучујемо се за *lazy loading*. Ова два типа прибављања података разликују се по томе што ће се у случају *eager loading*-а сви подаци модела и сви подаци његовог релационо повезаног модела одмах прибавити уз мањи број упита ка бази података. Са друге стране, *lazy loading*, ће се иницијално брже извршити, али ће се касније, приликом приступа релационо повезаним подацима, при сваком приступу извршити нови упит ка бази података. Ово указује да треба бити обазрив приликом дефинисања начина читавања, односно унапред треба размислити о потребним подацима. Одабиром типа прибављања избегава се нагомилавање података у меморији система, односно избегава се оптерећења система $N+1$ упитима ка бази података.

Осим приступа, на исти начин могу се вршити дефинисања команди креирања, измене или брисања података. У случају креирања, коришћењем методе *create(['column1' => 'value', 'column2' => 'value'])*, пратећи унапред дефинисане вредности *fillable* поља модела, аутоматски се креира и у бази података чува жељени запис. У случају измене података, након прибављања жељеног ентитета, позива се метода *update(['column' => 'new_value'])* којом се дефинишу промене жељених атрибута и по аутоматизму долази до чувања тих измена у бази података. За брисање података, након прибављања ентитета и навођења методе *delete()*, долази до брисања података из базе података. [26]

Коришћење ових предефинисаних команди заправо представља употребу наредби програмског језика *PHP* које ће радно окружење *Laravel* у позадини превести у *SQL* наредбе. Слично важи и за повратни процес, извршиће се мапирање прибављених података у објекте. Када се креирани објекат сачува, радно окружење *Laravel* ће у позадини ову акцију извршити уписом података у нови ред табеле у бази података. Потом, у случају прибављања података, пратећи дефинисане конвенције, ред података из базе података у систему ће се представити као објекат. Свака измена података у том реду у бази података биће директно пресликана на прибављени објекат. [24] Додатно, како би се одржала могућност праћења историје података, свака од измена над подацима у бази података биће пропраћена аутоматским акцијама записивања временског тренутака у којем се десила измена. Овај процес аутоматског бележења другачије се назива *automatic timestamps* чиме се постиже најједноставнији вид праћења измена над подацима у систему. [26] *Eloquent* је такође у радно окружење *Laravel* донео могућност привидног брисања података. Ово се постиже извршавањем *soft delete* процеса. Овим процесом, подаци се не бришу стварно из базе података. Ред остаје у табели док се на њему заправо врши *update* метода којом се у поље *deleted_at* уписује тренутак позива овог процеса. На овај начин, такав ред биће изузет од свих акција прибављања и измена. Овакав податак могуће је „вратити“ једноставним поништавањем вредности поља *deleted_at*.

4.4. Мапирање модела и његова употреба

Како је сваки ентитет базе података у радном окружењу *Laravel* представљен као класа, преко *Eloquent* модела морају се вршити дефинисања начина употребе и повезивања

ових података. Нека од најчешће употребљених функционалности модела су вршења дефинисања:

- *fillable* – поља која се могу уносити и мењати у бази података. Овим се упис података у базу података ограничава само на наведене атрибуте, односно колоне. Ова поља ће се подразумевано користити приликом масовног додељивања вредности у случају чувања из контролера или у случају приступа истим,
- *hidden* – поља која је могуће уписати, мењати али не и читати. Често се користе за аутентификациона поља односно лозинку и токене,
- *casts* – поља чију вредност у бази података чувамо у једном типу података, а онда у овој секцији дефинишемо преображавање у други тип података,
- *appends* – поља чије се вредности не чувају у бази података већ се могу израчунати на основу вредности других података. Углавном представљају комплексни, изведени, податак, а такви се у бази података не чувају,
- *table* – име табеле са којом се врши мапирање тог модела, често у пракси ово можемо и изоставити јер ће *framework* сам препознати о којој табели се ради на основу самог именовања модела праћењем конвенције,
- дефинисање повезаности модела са његовом фабриком за креирање нових инстанци – начин креирања оваквих инстанци и могућност аутоматске модификације креираних података, [27]
- итд.

Eloquent модели пружају могућност дефинисање релационих односа ка другим моделима, односно табелама. Везе које се у овим моделима могу дефинисати су:

- *one to one* – основни тип везе међу подацима у коме је очигледна припадност једног модела другом. У власничком моделу веза се дефинише методом *hasOne(ModelName::class)* док се у припадајућем моделу веза дефинише методом *belongsTo(ModelName::class)*. Ове методе позиваће се у функцијама које носе име једнине модела са којим се повезују,
- *one to many* – веома сличан тип везе првом типу са проширењем повезаности једног модела ка више других, односно један модел поседује више њих. У овом случају за дефинисање веза користе се методе *hasMany(ModelName::class)* и *belongsTo(ModelName::class)*. Ове методе позиваће се у функцијама које носе име редом множине и једнине модела са којим се повезују,
- *many to many* – тип везе којим се постиже приказивање вишеструке обостране припадности међу моделима. У оба модела користе се методе *belongsToMany(ModelName::class)* док се у бази података додатно креира посредничка табела за праћење релације познатија као пивот табела. Ове методе позиваће се у функцијама које носе име множине модела са којим се повезују,
- *polymorphic relation* – тип везе који омогућује припадност једног модела ка више типова различитих модела. Па ће тако припадајући модел користити методу *morphTo()* без наглашавања специфичног модела у оквиру функције која носи име свог модела уз суфикс *able*, док ће власнички модели имати

методе *morphOne/Many(ModelName::class, 'modelable')* унутар функције која носи име јединице/множине модела са којим се повезује.,

- додатно могу се дефинисати и комплексне посредничке везе након дефинисања основних типова веза коришћењем унакрсног везивања чиме се постиже *has one through* и *has many through*. [28]

Креирање веза међу моделима омогућава олакшани приступ широком спектру података. Уграђене *Eloquent* методе уз прибављање једне инстанце модела, могу прибавити и све жељене повезане моделе. Овакве методе повећавају квалитет написаног кода и спречавају настајање бројних грешака у писању комплексних *SQL* упита.

Након имплементације мапирања *Eloquent*-а у класама модела, треба разумети да се ово мапирање и његови модели користе широм целог *backend*-а и налазе примену и у другим деловима имплементације. Говорећи о овим имплементацијама имамо:

- фабрике – класе којима се наслеђивањем апстрактне класе *Factory* додељује могућност креирања лажних података објекта пратећи мапирану структуру објекта креирану моделом,
- *seeder*-е – класе којима се наслеђивањем апстрактне класе *Seeder* додељује могућност аутоматског уписивања прослеђених података у базу података. Прослеђени подаци углавном се генеришу употребом већ дефинисаних фабрика,
- миграције – класе којима се наслеђивањем апстрактне класе *Migration* и на основу идеје о структури података додељује могућност повезивања са базом података и дефинисања имена и типова поља у бази података, њихових веза са другим објектима, начин њиховог креирања, мењања или брисања,
- *api* руте – скуп рута које заправо представљају инстанце функционалности класа. Позивом функција класа, у случају правилног именовања рута, користи се *model binding* механизам. Њиме се инстанце модела повезују са идентификаторима из рута, те се функцијама на обраду дају инстанце објекта оног типа као што је у моделу дефинисано [29]
- контролере – класе које од рута примају објекте модела и прослеђују на даљу обраду испуњавајући *CRUD* и друге захтевније захтеве,
- *event*-е – класе којима се прате промене над мапираним објектима података и даље се примењују друге зависне акције, често праћене *listener* класама
- *middleware* – правило приступа одређеним ресурсима провером поређења модела корисника који захтева приступ и модела корисника коме је приступ одобрен,
- итд.

5. ИМПЛЕМЕНТАЦИЈА АПЛИКАЦИЈЕ

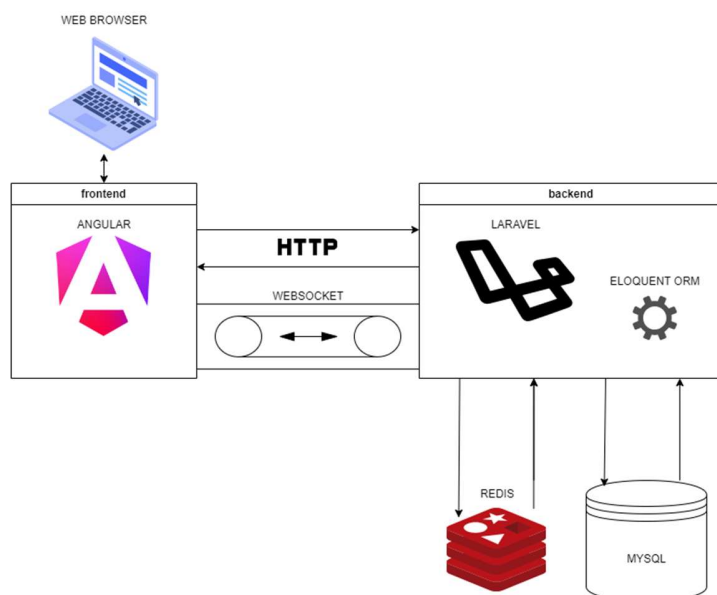
У овом поглављу биће детаљно описана архитектура и имплементација проучених технологија са циљем доказивања предности употребе ових технологија на реалном проблему. Поголавље се технички ослања на већ објашњену терминологију те се неће наводити поновна објашњења.

Код веб апликације налази се на два *GitHub* репозиторијума. Код је по имплементацији подељен на *web* и *api* део:

- [taxi-api](#) [30]
- [taxi-web](#) [31]

5.2. Архитектура апликације

За потребе објашњења проучених технологија, односно практичног дела дипломског рада, креирана је веб апликација ЕлФак такси. Апликација представља класичан пример клијент сервер апликације. Апликација омогућава наручивање, прихватање и праћење вожње, као и управљање корисничким профилем. Апликација је намењена свим људима који желе да добију услуге брзог транспорта, као и људима који би лако и брзо зарадили возећи такси, док постоји надређени менаџер који прати њихово пословање. Самим тим апликација је морала бити дизајнирана тако да њена употреба буде што једноставнија, али и да приступ истој буде брз. Из тих разлога креирана је веб апликација са прилагодљивим дизајном која се може приказати у сваком веб претраживачу. Креирана је у радним окружењима *Angular* и *Laravel* са коришћењем *MySQL* и *Redis* база података. Приказ ове архитектуре дат је на слици 10.



Слика 10 – Приказ архитектуре веб апликације

5.3. MySQL

MySQL је најпознатији *open source* управљачки систем релационих база података. Како сам опис каже, користи се за креирање и управљање релационим базама података. Због своје карактеристике *open source* платформе, сами корисници по наилажењу на још увек нерешене проблеме имају могућност имплементације свог решења и дељења истог са осталим корисницима, што представља начин раста и развоја овог система. Осим ове карактеристике, систем је веома лак за употребу што му додатно иде на руку. Омогућава рад разноврсних система због варијације чувања количине података, од минималних података неког ресторана, до великих комплексних записа података бизнис компанија. [32]

Најбитније својство MySQL базе података је употреба *ACID* трансакција. Овај акроним представља атомичност, конзистентност, изолованост и издржљивост. Оваквим атрибутима описује се процес извршења једне радње над подацима у бази података. Таква радња назива се трансакција података. Трансакције су атомичне јер се њихово извршење дозвољава само у целости. Уколико неку од радњи трансакције није било могуће извршити, трансакција ће се поништити. Поништавањем трансакције никаква измена неће бити сачувана у бази података, макар до поништавања дошло и на крају извршења трансакције. Оне су конзистентне јер њихово извршавање омогућава упис једино валидних података, пратећи дефинисану структуру типа података и међусобног односа истих. Трансакције су и изоловане, односно не дозвољавају конкурентност модификације података у истом тренутку. Самим тим захтеви се преводе у синхроне процесе, спречавајући да се подаци система доведу у невалидно стање. Коначно, оне су и издржљиве односно гарантују очување уписаних података неvezано за стање система у случају пада истог. [33]

5.4. Радно окружење Angular

Angular је *frontend* радно окружење које користи *TypeScript* програмски језик. Креиран је са истом идејом као и радно окружење *Laravel*, односно тежи да олакша израду комплексних веб апликација без потребе да се све развија од почетка. Апликације овако развијене имплементирани су на *MVC* архитектури, о којој је већ писано.

Дефинисање и понашање елемената апликација одређује се декораторима. Декоратори пружају начин за додавање додатних функционалности и мета податке у елементе апликације. Најчешће се користе за дефинисање компоненти, сервиса, директива, *pipe*-ова и другог. Најбитнији декоратори су:

- класни декоратори – дефинишу понашање компоненти апликације. Компонента је основни елемент сваке апликације, данас се апликације генеришу од самосталних компоненти без потребе за родитељски модулom. На слици 11 је приказана таква компонента.

```
@Component({
  selector: "app-demo",
  template: "./demo.component.html",
  styleUrls: "./demo.component.sxss",
  standalone: true,
})
class DemoComponent {...}
```

Слика 11 – Пример самосталне компоненте

- декоратори својства – дефинишу понашање атрибута компоненти апликације. На пример, на овај начин могу се дефинисати улазни атрибути и пратити њихове измене које се дешавају у родитељској компоненти, али и дефинисати емитере промена ка родитељској компоненти. На слици 12 је приказан пример дефинисања једног улазног атрибута и излазне функције.

```
@Input() title = '';
@Output() signalChange: EventEmitter<{ title: string}> = new EventEmitter();
```

Слика 12 – Пример примања вредности и враћања акције

- декоратори параметра – омогућују вршење *Dependency Injection*-а односно убацивања зависности. На тај начин постиже се извлачење комплексних логика у класе сервиса које се коришћењем оваквих декоратора лако укључују у све компоненте система. [34] На слици 13 је приказан пример учитавања једне овакве сервисне класе.

```
@Injectable({ providedIn: 'root' })
export class HelperService { ... }

@Component({
  selector: "app-demo",
  template: "./demo.component.html",
  styleUrls: "./demo.component.sxss",
  standalone: true,
})
class DemoComponent {
  private helperService = inject(HelperService);
  ...
}
```

Слика 13 – Пример дефинисања и убацивања помоћног сервиса

Посматрање података дефинисано је двосмерним повезивањем. Ажурност података на приказу постиже се посматрањем везе између приказаног податка на прегледу и атрибута у класи компоненте. Ова веза омогућује приказ промена у реалном времену без потребе за освежавањем странице. [35] Додатно унапређивање у преношењу ажурности података постигнуто је сигнаlima. Сигнали представљају нов концепт у дефинисању података. Понашају се као омотач око вредности податка који у случају измене вредности податка сигнализира остатку апликације нову вредност. Користе се за обавештавање међу компонентама, израчунавање нових вредности изведених атрибута након промена основних вредности и за извршење додатних функционалности након измена неких података. Самим тим уочавамо сигнале за упис, израчунате сигнале и ефекте. [36]

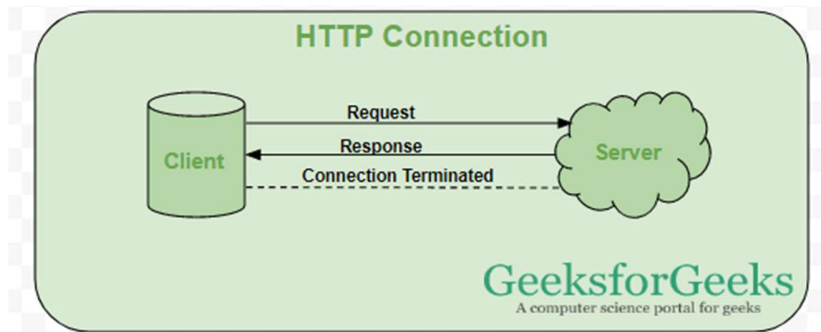
Апликације развијене у радном окружењу *Angular* су заправо апликације са једном страницом односно *SPA*. Овакав приступ омогућује иницијално приказивање главне компоненте која представља корен система. На овој компоненти ређају се остале компоненте. Како се сви системи заправо састоје од више различитих страница, најбитније је заправо употребити *router-outlet* у овој компоненти. На овај начин радно окружење *Angular* ће само смењивати приказ жељених компоненти у главној компоненти на основу руте веб претраживача без регенерисања главе компоненте. Овакав приступ у многоме побољшава перформансе система. [37] На слици 14 је приказан изглед оваквих компоненти.

```
<div>
  <app-header></app-header>
  <router-outlet></router-outlet>
  <app-footer></app-footer>
</div>
```

Слика 14 – Пример главне компоненте *SPA*

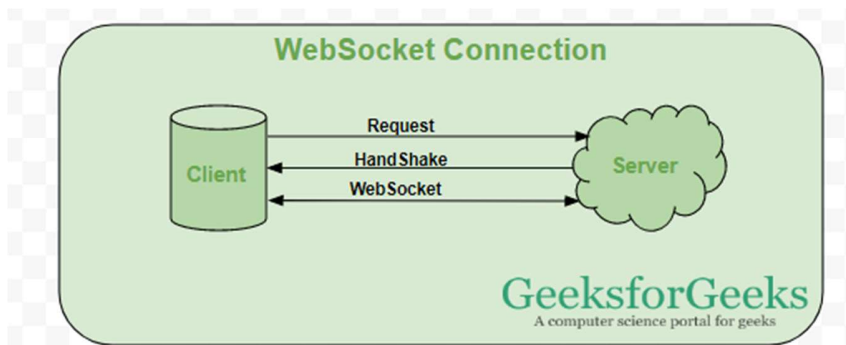
5.5. *Web Socket*

Са развојем апликација долази се до ситуација када слање *HTTP* захтева за прибављање података није довољна функционалност. Овакав вид комуникације је приказан на слици 15. Често, нови и захтевнији системи очекују неки вид обавештења приликом измена података у позадини или у току процеса у ком сами нису учествовали. Иницијална идеја константног позивања *backend*-а није добро решење, зато су осмишљени *WebSocket*-и. Уместо слања једносмерних захтева са *http://*, односно *https://*, чији одговор клијент очекује након позива, у овим случајевима користи се двосмерни канал са *ws://*, односно *wss://*, ознаком.



Слика 15 – Приказ процеса слања HTTP захтева [38]

Web Socket-и пружају двосмерну комуникацију у систему, омогућавајући интеракцију *backend*-а са *frontend*-ом без постојања константних упитних захтева. Како би се комуникација остварила, користи се *handshake* метода којом клијент и сервер потврђују жељену комуникацију и креирају се комуникациони канали. Овако креиране канале обавезно је затворити када више није потребно ослушкивати податке или пре прекидања рада апликације. Сваки канал носи своје идентификационо име како би апликације могле да ослушкују поруке на том каналу. Поруке на каналу дефинисане су различитим типом догађаја, и преносе жељене податке смештене у *JSON*-у. На слици 16 је приказан процес креирања канала *Web Socket*-а.



Слика 16 – Приказ процеса слања *WebSocket* захтева [38]

Зависно од потреба и технолошке позадине апликације, треба разумети да коришћење *WebSocket*-а не може и не треба заменити *HTTP* конекцију. Пожељно је користити их приликом прибављања података у реалном времену, израде апликације за комуникацију, израде апликације за праћење кретања или слично. У осталим случајевима пожељно је придржавати се традиционалног начина прибављања података јер креирање и одржавање канала комуникације није корисно за апликације које имају само захтеве за повремено прибављање или ажурирање података без интеракције са другим корисницима. [38]

5.6. *WebStorm*

WebStorm је развојно окружење погодно за развој *frontend* дела веб и мобилних апликација. Омогућава једноставни развој оваквих апликација коришћењем технологија као што су *JavaScript*, *React*, *TypeScript*, *Angular*, *Vue*, *HTML*, *CSS*.

Омогућава кориснику да одмах започне са кодирањем без потребе да инсталира и подешава додатке. *WebStorm* укључује све што је иницијално потребно за развој апликација у *JavaScript* и *TypeScript* програмским језицима. Оставља могућност за каснију персонализацију са мноштвом различитих додатака и подешавања. [39]

Од додатака са верзијом 2024.2.3 у имплементацији се користе *ESLint* и *Prettier* ради лакшег и бржег декорисања и форматирања кода како би се постигла униформност при развоју на било ком рачунару.

5.7. *PhpStorm*

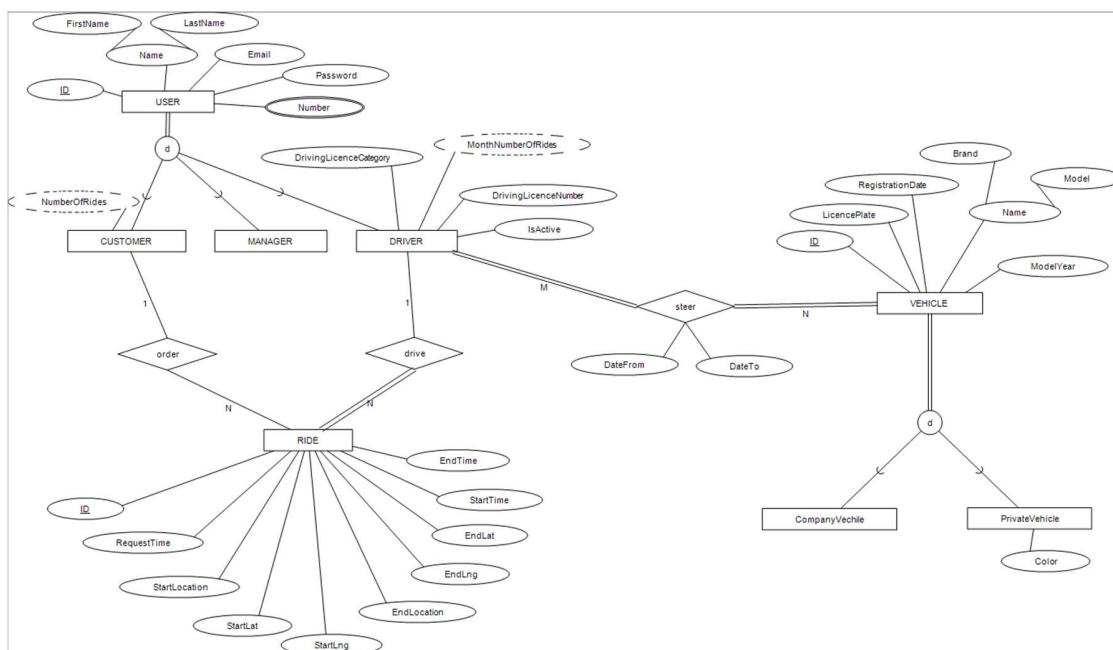
PhpStorm је развојно окружење погодно за развој *backend* дела веб и мобилних апликација. Омогућава једноставни развој оваквих апликација коришћењем технологија као што су *PHP*, *Laravel*, *Blade*, *Symfony*, *JavaScript*, *SQL*.

У случају развоја апликације на *Windows* оперативном систему користи се са виртуелном машином. Кориснички интерфејс додатно је обogaћен лаком и брзом претрагом докумената у пројекту као и њиховог садржаја са могућношћу дефинисања опсега претраге.

У имплементацији коришћена је верзија 2024.2.3. Такође, велика предност овог окружења је и додаток *Database* који ради по истом принципу као и засебни систем *DataGrip*. Коришћење овог додатка у имплементацији је омогућило брз и лак увид у тренутно стање базе података, као и доста опција за манипулисање истом.

5.8. Опис имплементације базе података

Најбитнији део израде практичног дела дипломског рада представља процес дефинисања изгледа жељене базе података. Анализом задатака, долази се до издвајања свих могућих ентитета и њихових атрибута. Затим приступа се анализи ентитета, проналажењу њихових сличности и дефинисању евентуалних надкласа издвајањем заједничких атрибута. Следећи корак представља разумевање односа ентитета у задатку, односно успостављање веза међу њима. Битно је водити рачуна о обавезности учествовања ентитета у вези као и квантитативном учинку. Ради разумевања изгледа базе података, најпре се креира *EER* дијаграм који је приказан на слици 17.



Слика 17 – EER дијаграм ЕлФак такси базе података

Посматрањем дијаграма, а затим и применом правила превођења *EER* модела у релациони модел добијамо базу података са табелама:

- CUSTOMER – садржи све податке о кориснику типа муштерија,
- DRIVER – садржи све податке о кориснику типа возач,
- MANAGER – садржи све податке о кориснику типа менаџер,

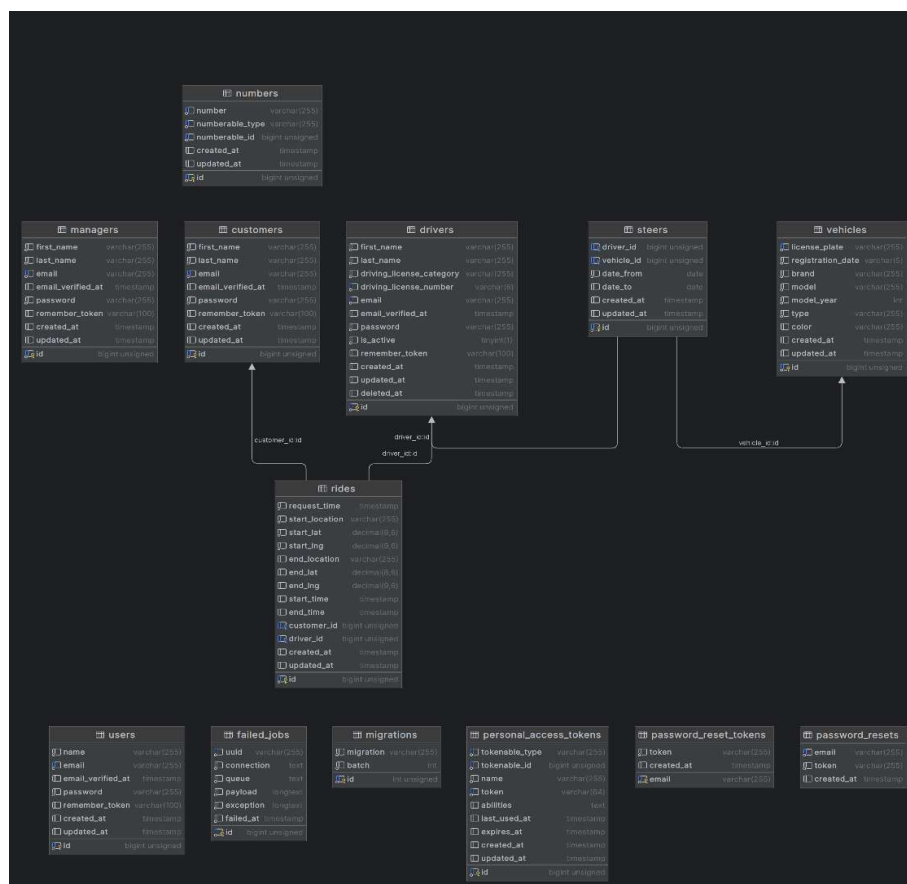
У превођењу надкласе *User* и подкласа *Customer*, *Driver* и *Manager*, коришћен је други начин превођења овог типа ентитета, односно креиране су табеле за подкласе са свим атрибутима надкласе. Ово је касније довело до комплексније логике регистрација и пријављивања на систем, односно потребе за ручном имплементацијом ових функционалности, без могућности употребе већ постојећих, имплементираних од стране радног окружења. А самим тим и до интегрисања специјалне логике за проверу аутентификационог статуса корисника употребом нових *middleware*-а у провери што се види на слици 18.

```
public function authCheck(Request $request): JsonResponse
{
    $type = $request->query('type', 'web');
    if (auth()->guard($type)->check()) {
        $user = auth()->guard($type)->user();
        return response()->json(["user" => $user, "type" => $type]);
    }
    return response()->json(NULL, 401);
}
```

Слика 18 – Пример имплементације провере аутентификационог статуса корисника

- **NUMBERS** – садржи податке о бројевима телефона корисника апликације, као и њихов тип и идентификациони број. Нема јасно дефинисану релацију, јер применом *morphs* типа везе, омогућујемо креирање припадности броја телефона било ком ентитету базе података,
- **RIDE** – садржи све податке о возњи, од креирања захтева, преко управљања истим, до завршетка возње. Представља најупотребљивију табелу система јер константо долази до ажурирања података исте,
- **VEHICLE** – садржи податке о свим возилима у систему. Коришћењем трећег начина превођења ентитета у односу надкласа подкласа, добијамо једну табелу са свим подацима надкласе, типом унесеног возила и свим подацима подкласа где су са *NULL* означена поља која тренутном уносу не одговарају,
- **STEERS** – садржи историју односа управљања возилом од стране возача. Због везе типа више према више, креирана је нова засебна табела у којој се налазе страни кључеви возача и возила, како би се одредила њихова повезаност. Додатно памте се атрибути везе, односно период управљања возилом од стране возача, како би било могуће пратити историју управљања.

Овакво креирана база података се визуелно може приказати шемом, као што је приказано на слици 19. Додатно, како би радно окружење *Laravel* управљао овом базом податка, потребна је одређена конфигурација, а део исте је приказан на слици 20.



Слика 19 – MySQL шема релационе базе података


```
DB_CONNECTION=mysql
DB_HOST=127.0.0.1
DB_PORT=3306
DB_DATABASE=taxidb
DB_USERNAME=homestead
DB_PASSWORD=secret
```

Слика 20 – Део конфигурације *.env* документа везан за комуникацију са базом података

5.9. Имплементација серверског дела

Серверски део апликације имплементиран је у радном окружењу *Laravel* 11 са употребом *PHP* 8.2 програмског језика. Пре самог почетка имплементирања апликације, добро је осмишљена архитектура исте, затим је на основу задатка креиран дијаграм базе података и преведен у релациони модел, а онда су пре самог почетка имплементације подешени радно окружење *Laravel* и рачунар уз помоћ виртуелне машине и *Homestead*-а. Део конфигурационог *Laravel* документа приказан је на слици 21.

```
APP_NAME=ElfakTaxi
APP_ENV=local
APP_KEY=base64:po2SuFHsnVC8Gxyso2caUs33H5hs8b05IJy13i0jNQs=
APP_DEBUG=true
APP_URL=http://localhost
```

Слика 21 – Део имплементације *.env* документа који је основни за сваку *Laravel* апликацију

Радном окружењу *Laravel* је за потребе тестирања приликом развоја додат пакет *laravel/tinker*. Овај пакет се покреће у конзоли и омогућује испробавање имплементиране апликације. Креирањем нових објеката користећи имплементиране модуле, можемо испробати имплементиране функционалности. Сва креирања, измене и брисања која се дешавају унутар *tinker*-а користила су конекцију са базом података, те се тиме мењају реални подаци система.

За потребе пријављивања и регистрација на систем додат је пакет *laravel/sanctum*. Овај пакет који користи технологију провере валидности приступа корисника провером његових токена. Токени се у *HTTP* захтеву прослеђују у *headers* секцији. Након инсталације пакета потребно је прецизно подесити конфигурациони *config/sanctum.php* документ. Њиме се подешава очекивани клијентски домен, како систем не би захтеве корисника сматрао неауторизованим. Такође, подешава се и конфигурациони *config/cors.php* документ, како систем не би захтеве клијента посматрао као потенцијалне нападе. На слици 22 је приказан део конфигурационог документа *cors*, а на сликама 23 и 24 део конфигурације *sanctum*-а.

```
<?php

return [
    'paths' => ['api/*', 'sanctum/csrf-cookie', 'login', 'logout', 'register',
'password/reset'],
    ...
    'supports_credentials' => true,
];
```

Слика 22 – Део имплементација *config/cors.php*

```
<?php

use Laravel\Sanctum\Sanctum;

return [
    'stateful' => explode(',', env('SANCTUM_STATEFUL_DOMAINS', sprintf(
        '%s%s',
        'localhost,localhost:3000,127.0.0.1,127.0.0.1:8000,::1',
        Sanctum::currentApplicationUrlWithPort()
    ))),
    'guard' => ['web'],
    ...
];
```

Слика 23 – Део имплементација *config/sanctum.php*

```
SESSION_DRIVER=cookie
SESSION_LIFETIME=120
SESSION_DOMAIN=localhost
SANCTUM_STATEFUL_DOMAINS=http://localhost:4200
COOKIE_SAME_SITE_POLICY=strict
```

Слика 24 – Део имплементације *.env* документа везан за *sanctum*

За потребе чувања краткорочно потребних података у радном окружењу *Laravel* додат је *Redis*. Обзиром да се ради о малој количини једноставнијих података који се никада не нагомилавају, без потребе подешавања *Redis* сервера искоришћен је пакет *predis/predis* који представља библиотеку *PHP* програмског језика. Додатно, како је за обављање неких обрада података потребно сачекати одређени временски период додата је употреба *Laravel Job*-ова чије се распоређивање за извршење такође врши коришћењем *Redis*-а. За употребу *Redis*-а било је потребно креирати конфигурацију у главном конфигурационом документу, део ове конфигурације приказан је на слици 25. На слици 26 је приказан један од начина употребе *Redis*-а.

```
QUEUE_CONNECTION=redis
REDIS_CLIENT=predis
REDIS_HOST=127.0.0.1
REDIS_PASSWORD=null
REDIS_PORT=6379
```

Слика 25 – Део конфигурације *.env* документа везан за *Job* и *config/database.php* документ

```
$redisKey = "ride:$rideId:driver-location";
Redis::sadd($redisKey, json_encode(['lat' => $request->get('lat'), 'lng' =>
$request->get('lng'), 'driver_id' => $driver->id]));
```

Слика 26 – Пример додавања података у *Redis* низ

За потребе додатне комуникације клијената у систему, односно преношења података у реалном времену тј. додељивања возње најближем кориснику и приказа кретања такси возила на клијентској мапи, додат је и пакет *laravel/verrb*. Овим пакетом омогућено је слање података специјално креираним каналима. Након конфигурације слања података у *config/broadcasting.php* и *config/verrb.php* датотекама, потребно је креирати класе *Laravel event*-а који се проширују *ShouldBroadcast* интерфејсом. Свака од ових класа дефинише које податке добија приликом слања поруке на канал, дефинише како ће проследити те податке, на ком каналу ће их послати и под којим именом. На сликама 27, 28 и 29 приказана је конфигурација и начин употребе.

```
<?php
return [
    'default' => env('BROADCAST_CONNECTION', 'null'),
    'connections' => [
        'verrb' => [
            'driver' => 'verrb',
            'key' => env('REVERB_APP_KEY'),
            'secret' => env('REVERB_APP_SECRET'),
            'app_id' => env('REVERB_APP_ID'),
            'options' => [
                'host' => env('REVERB_HOST'),
                'port' => env('REVERB_PORT', 443),
                'scheme' => env('REVERB_SCHEME', 'https'),
                'useTLS' => env('REVERB_SCHEME', 'https') === 'https',
            ],
            'client_options' => [],
        ], ...
    ],
];
```

Слика 27 – Део имплементације *config/broadcasting.php*

```
BROADCAST_CONNECTION=reverb
REVERB_APP_ID=583497
REVERB_APP_KEY=ifplsmoecxg4mk1efqa5
REVERB_APP_SECRET=8s6rxweoxjlk2v3znate
REVERB_HOST=localhost
REVERB_PORT=8080
REVERB_SCHEME=http
```

Слика 28 – Део конфигурације .env документа везан за config/reverb.php документ

```
<?php

namespace App\Events;

use ...

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

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

    public function broadcastOn(): Channel
    {
        return new Channel('drivers');
    }

    public function broadcastAs(): string
    {
        return 'ride-requested';
    }

    public function broadcastWith(): array
    {
        return ['ride' => $this->ride];
    }
}
```

Слика 29 – Пример имплементације једног Laravel event-a са reverb-ом

Генерално, апликација је дефинисана тако да у *routes/api.php* документу можемо уочити дефинисање свих *API* рута. Руте су подељене по смисленим целинама, а додатно у зависности од логике приступа подацима посебно заштићене *middleware*-ом који ће све *HTTP* захтеве проверити пре приступа контролеру. Након приступа контролеру, пре самог почетка извршавања кода, долази до валидације прослеђених података. Конкретно у задатку генерисане су засебне валидационе класе за сваки од *API* захтева. Затим, следе позиви помоћних сервисних функција које врше обраду података и комуникацију са базом података. Помоћне сервисне функције налазе се у сервисним класама које имплементирају жељене сервисне интерфејсе. Ови интерфејси се у контролеру прослеђују убацивањем зависности (преко *DI*), док је њихова инстанца регистрована у систему по *singleton* обрасцу који се брине о томе да класа има само једну инстанцу, док пружа глобални приступ истој.

5.9.1. Објектно релациони мапер *Eloquent ORM*

Коришћењем *Eloquent*-а извршено је мапирање табела релационе базе података на објекте који се користе у систему. Сваки од ентитета базе података, односно свака табела базе података, у систему је представљен као класа модела. *Eloquent*-ом се постигло дефинисање поља за поуну, изведених поља и поља за преображавање типа вредности. Дефинисан је назив табеле са којом је потребно извршити мапирање, као и дефинисање релација и начина учествовања модела у истој. Пример имплементације једног од модела система дат је на слици 30.

```
<?php

namespace App\Models;

use ...

class Ride extends Model
{
    use HasFactory;

    protected $fillable = [
        'request_time',
        'start_location',
        'end_location',
        'start_time',
        'end_time',
        'customer_id',
        'driver_id',
    ];
}
```

```

protected $casts = [
    'request_time' => 'datetime',
    'start_time' => 'datetime',
    'end_time' => 'datetime',
];

public function customer(): BelongsTo
{
    return $this->belongsTo(Customer::class);
}

public function driver(): BelongsTo
{
    return $this->belongsTo(Driver::class);
}
}

```

Слика 30 – Пример имплементације Ride модела

Поред дефинисања изгледа модела, објектно релациони мапер омогућио је дефинисање миграционих датотека. Миграционим датотекама програмски дефинишемо изглед базе података. Миграционе датотеке омогућују дефинисање типова вредности атрибута, подразумеваних вредности у случају када њихово присуство није обавезно, веза ка другим табелама и понашање у случају брисања или измене вредности у повезаним табелама. Имплементације једне овакве датотеке приказана је на слици 31.

```

<?php

use ...

return new class extends Migration {
    public function up(): void
    {
        Schema::create('rides', function (Blueprint $table) {
            $table->id();
            $table->timestamp('request_time');
            $table->string('start_location');
            $table->decimal('start_lat', 8, 6);
            $table->decimal('start_lng', 9, 6);
            $table->string('end_location')->nullable();
            $table->decimal('end_lat', 8, 6)->nullable();
            $table->decimal('end_lng', 9, 6)->nullable();
            $table->timestamp('start_time')->nullable();
            $table->timestamp('end_time')->nullable();
            $table->foreignId('customer_id')->nullable()
                ->constrained()->cascadeOnUpdate()
                ->nullOnDelete();
        });
    }
};

```

```

        $table->foreignId('driver_id')->nullable()
            ->constrained()->cascadeOnUpdate()
            ->nullOnDelete();
        $table->timestamps();
    });
}

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

};

```

Слика 31 – Пример имплементације миграције *Ride* модела односно *rides* табеле

Овако дефинисаним мапером, лако смо добили могућност креирања класних фабрика које прате правила дефинисања модела. У фабрици се додатно дефинише које врсте вредности желимо на местима атрибута. Затим се позивањем инстанце фабрике у класи *DatabaseSeeder* омогућује аутоматско креирање тестних података. Ови подаци, уписују се у креирану релациону базу података. Овај приступ додатно олакшава тестирање имплементираних релационих база података и објектно релационог мапера пружајући јединствене тестне податке без потребе ручног уноса. На слици 32 је приказана имплементација једне од фабрика овог система.

```

<?php

namespace Database\Factories;
use ...

class SteerFactory extends Factory
{
    protected $model = Steer::class;

    public function definition(): array
    {
        return [
            'driver_id' => Driver::factory(),
            'vehicle_id' => Vehicle::factory(),
            'date_from' => $this->faker->date(),
            'date_to' => null,
        ];
    }
}

```

Слика 32 – Пример имплементације фабрике за креирање тестних података

5.10. Имплементација клијентског дела

Клијентски део апликације имплементиран је у радном окружењу *Angular* 18. На самом почетку имплементирања апликације, добро је осмишљена навигациона структура исте. Потом је праћењем објектно релационог мапера извешено креирање модела и на самом *frontend*-у како би апликација знала које податке може очекивати приликом читања одговора *backend*-а. Како би се та комуникација преноса података и остварила, дефинисан је конфигурациони документ, приказан на слици 33.

```
export const environment = {
  production: false,
  API_URL: 'http://localhost:8000/api',
  API_DOMAIN: 'http://localhost:8000',
  REVERB_APP_KEY: 'ifplsmodcxg4mk1efqa5',
  REVERB_APP_SECRET: '8s6rxweoxjlk2v3znate',
  REVERB_WS_HOST: 'localhost',
  REVERB_WS_PORT: '8080',
  GOOGLE_API_KEY: 'AIzaSyDbkojycfNgFWqv_Zw8ectCFTlY-WtpgoQ',
};
```

Слика 33 – Глобална конфигурација која дефинише параметре комуникације

Радном окружењу *Angular* су за потребе израде прилагодљивог дизајна за приказ апликације на паметном телефону и таблету додати пакети *@ng-bootstrap/ng-bootstrap* и *bootstrap*. Ови пакети омогућили су употребу предефинисаних стилски класа којима се апликација различито приказује у зависности од презентационог уређаја. Овакав дизајн постигнут је дефинисањем *div* елемента са класом *row* који у себи има друге градивне елементе дефинисане класама *col-1*, *col-2*, па све до *col-12* и *col-auto*. Овај принцип ради по систему да један ред може имати 12 јединица.

За потребе чувања и употребе података, коришћен је нови приступ коришћења података сигнаlima. Након прибављања кључних података о пријављеном кориснику, исти се чувају у *signalStore*, прибављају се дефинисаним функцијама прибављања и ажурирају дефинисаним функцијама промене вредности. *Store* је дефинисан на глобалном нивоу и у свим независним компонентама коришћен је убацивањем зависности. Како се меморија овог система губи са сваки освежавањем странице веб претраживача додатно је употребљена меморија веб претраживача у коју се уписују све кључне вредности тренутног система. *Store* овог система приказан је на слици 34.


```
export type AuthState = {
  user: Customer | Driver | null;
  type: string | undefined;
  token?: string;
  session?: string;
  isLoading: boolean;
};

const initialState: AuthState = {
  user: null,
  type: undefined,
  token: undefined,
  session: undefined,
  isLoading: false,
};

export const AuthStore = signalStore(
  { providedIn: 'root' },
  withState(initialState),
  withMethods((store) => ({
    setToken: (token: string) => {
      localStorage.setItem('token', token);
      patchState(store, { token });
    },
    setSession: (session: string) => {
      localStorage.setItem('session', session);
      patchState(store, { session });
    },
    setAuthUser: (user: Customer | Driver, type: string) => {
      localStorage.setItem('type', type);
      patchState(store, { user, type });
    },
    setLoading: (isLoading: boolean) => patchState(store, { isLoading }),
    clearAuthUser: () => {
      localStorage.removeItem('token');
      localStorage.removeItem('session');
      localStorage.removeItem('type');
      patchState(store, {
        user: null,
        type: undefined,
        token: undefined,
        session: undefined,
      });
    },
  })),
  {}),
);
```

```

withHooks((store) => ({
  onInit: () => {
    const localStorageToken = LocalStorage.getItem('token');
    const localStorageSession = LocalStorage.getItem('session');
    const localStorageType = LocalStorage.getItem('type');
    patchState(store, {
      ...(localStorageToken && localStorageToken !== '' &&
        { token: localStorageToken }
      ),
      ...(localStorageSession && localStorageSession !== '' &&
        { session: localStorageSession }
      ),
      ...(localStorageType && localStorageType !== '' &&
        { type: localStorageType }
      ),
    });
  },
}));
});
);

```

Слика 34 – Пример имплементације store-a

За потребе додатне комуникације клијената у систему, односно ослушкивање порука послатих са *backend*-a, у радном окружењу *Angular* су додати и пакети *laravel-echo* и *pusher-js*. Имплементација *echo* сервиса приказана је на слици 35. Овим пакетима омогућено је ослушкивање канала са информацијом о новим подацима без потребе за константним слањем *HTTP* захтева. Потребно је имплементирати помоћну сервисну класу која ће инстанцирати *Pusher* и *Echo* и омогућити функционалност отварања и затварања комуникационих канала. Овај сервис је убачен у независне компоненте које имају потребу за оваквим типом комуникације и на основу креираних услова отварају односно затварају ове канале. Пример ослушкивања приказан је на слици 36.

```

...
private initEcho() {
  inject(NgZone).runOutsideAngular(() => {
    if (isPlatformBrowser(this.platformId)) {
      (Window as any).Pusher = Pusher;
      this.echo = new Echo({
        broadcaster: 'reverb',
        key: environment.REVERB_APP_KEY,
        wsHost: environment.REVERB_WS_HOST,
        wsPort: environment.REVERB_WS_PORT,
        forceTLS: false,
        enabledTransports: ['ws'],
      });
    }
  });
}

```

```
listen(channelName: string, eventName: string, callback: (res: any) => void) {
    if (!this.echo) {
        return false;
    }
    return this.echo.listen(channelName, eventName, callback);
}

leave(channelName: string) {
    if (!this.echo) {
        return;
    }
    this.echo.leave(channelName);
}

disconnect() {
    if (!this.echo) {
        return;
    }
    this.echo.disconnect();
}
```

Слика 35 – Пример имплементације echo сервиса

```
private echoService = inject(EchoService);
...
listenToRideAccepted(rideId: number) {
    this.echoService.listen(`rides.${rideId}`, '\\ride-accepted', (res: { ride: Ride
}) => {
        this.toastService.success('Vozač je krenuo ka Vama!');
        this.ride.set(res.ride);
    });
}
```

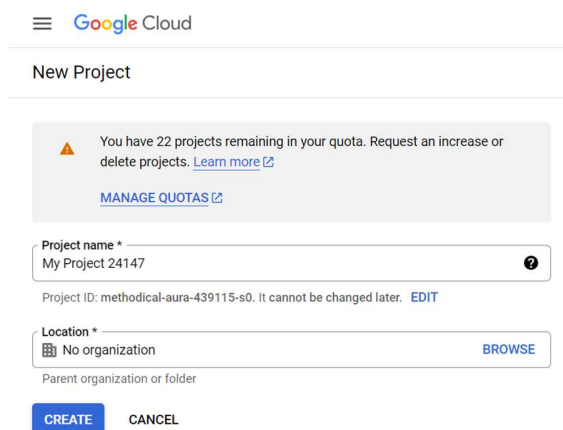
Слика 36 – Пример употребе echo сервиса за комуникацију WebSocket-има

5.10.1. Приказ мапе

Иако је главна тема овог рада рад са релационим базама података, сама имплементација практичног дела дипломског рада базира се око приказа мапе на којој је могуће видети своју локацију, унесене почетне и одредишне тачке, локацију возача и навигацију кретања. Оваква функционалност омогућена је коришћењем *Google Maps JavaScript API* сервиса.

Процес прибављања овог сервиса је персонализован и мора га сваки власник апликације радити. Процес је приказан на сликама 37, 38 и 39. Потребно је генерисати *Google Cloud* пројекат на *Cloud Console*-и, односно попунити основне податке о пројекту и

додати могућност наплате. *Google* омогућује одређени број тестних захтева, односно у случају прекорачења долази до наплате по сваком захтеву за лоцирање, навигирање, итд. Омогућено је додавање различитих *API*-а који би прошири могућност употребе апликације. По завршетку конфигурације, битно је прибавити *Maps API Key* јер је њега потребно укључити у захтев за учитавање сервиса приликом учитавања апликације, како би се омогућила употреба *Google* сервиса.



Google Cloud

New Project

Warning: You have 22 projects remaining in your quota. Request an increase or delete projects. [Learn more](#)

[MANAGE QUOTAS](#)

Project name *
My Project 24147

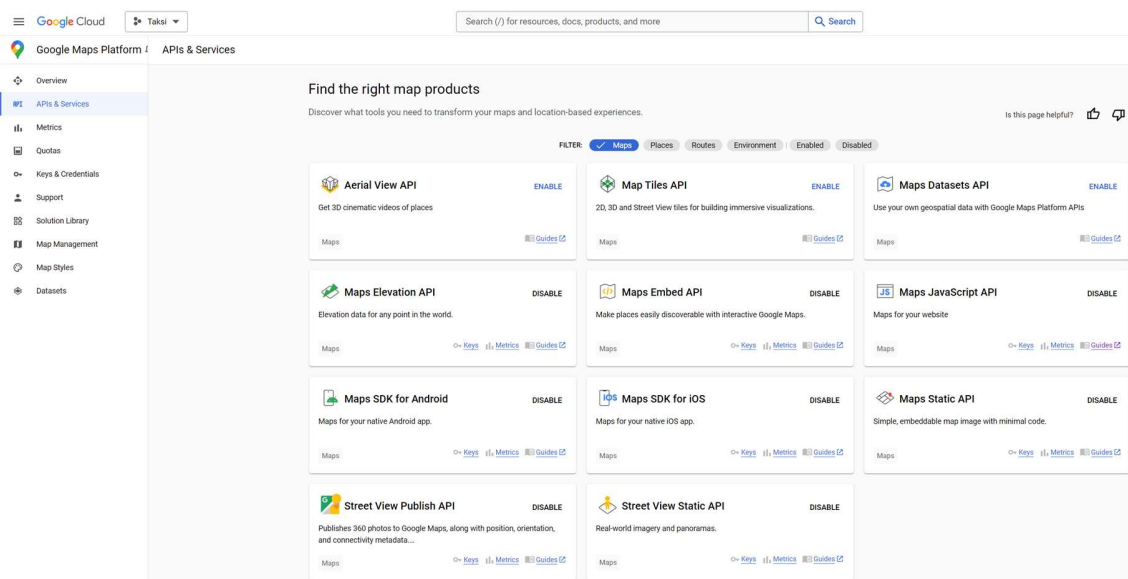
Project ID: methodical-aura-439115-s0. It cannot be changed later. [EDIT](#)

Location *
No organization [BROWSE](#)

Parent organization or folder

[CREATE](#) [CANCEL](#)

Слика 37 – креирање *Google Cloud* пројекта [40]



Google Cloud

Google Maps Platform APIs & Services

Find the right map products

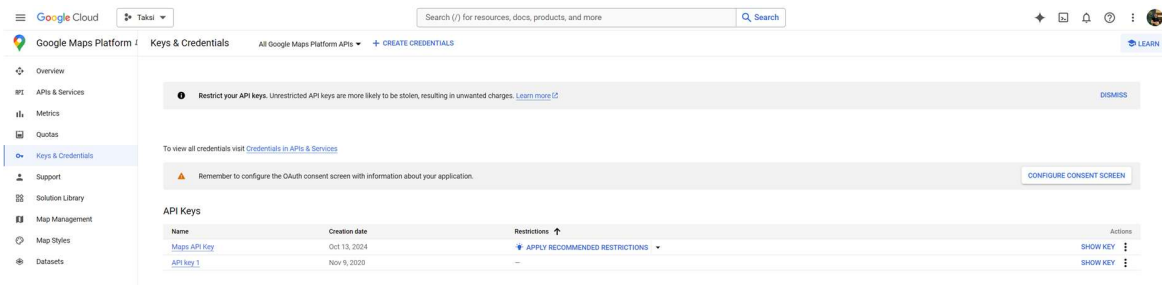
Discover what tools you need to transform your maps and location-based experiences.

Is this page helpful? [Feedback](#)

FILTER: Maps Places Routes Environment Enabled Disabled

| | | |
|--|---|--|
| Aerial View API ENABLE Get 3D cinematic videos of places Maps GUIDES | Map Tiles API ENABLE 2D, 3D and Street View tiles for building immersive visualizations. Maps GUIDES | Maps Datasets API ENABLE Use your own geospatial data with Google Maps Platform APIs Maps GUIDES |
| Maps Elevation API DISABLE Elevation data for any point in the world. Maps Keys Metrics GUIDES | Maps Embed API DISABLE Make places easily discoverable with interactive Google Maps. Maps Keys Metrics GUIDES | Maps JavaScript API DISABLE Maps for your website Maps Keys Metrics GUIDES |
| Maps SDK for Android DISABLE Maps for your native Android app. Maps Keys Metrics GUIDES | Maps SDK for iOS DISABLE Maps for your native iOS app. Maps Keys Metrics GUIDES | Maps Static API DISABLE Simple, embeddable map image with minimal code. Maps Keys Metrics GUIDES |
| Street View Publish API DISABLE Publishes 360 photos to Google Maps, along with position, orientation, and connectivity metadata. Maps Keys Metrics GUIDES | Street View Static API DISABLE Real-world imagery and panoramas. Maps Keys Metrics GUIDES | |

Слика 38 – Додавање *API*-а [40]



Слика 39 – Прибављање конекционих кључева [40]

Након успешног подешавања, приступа се употреби *Google* сервиса. На слици 40 је приказан пример имплементације приказа мапе. У овом раду, извршен је приказ мапе са свим жељеним ознакама и коришћене су методе:

- проналажења назива адресе на основу пружених координата,
- налажења координата на основу пружене адресе, пример имплементације је приказан на слици 41,
- налажења путање кретања на основу прослеђених координата почетка, усputних стајалишта и одредишта.

```
<google-map height="65vh" width="100%" [options]="{
  mapId: 'e547d2444e8aaf6f',
  mapTypeId: 'roadmap',
  zoomControl: true,
  scrollwheel: true,
  gestureHandling: 'cooperative',
  zoom: 15,
  maxZoom: 30,
  minZoom: 12,
  clickableIcons: false,
  streetViewControl: false,
}" (mapClick)="handlePickingLocation($event)"
>
  @if (myLocation && !directionsResult) {
    <map-advanced-marker [content]="icons()[0] || null"
      [position]="{ lat: myLocation.lat, lng: myLocation.lng }"
    />
  }
  @if (driverLocation()) {
    <map-advanced-marker [content]="icons()[3] || null"
      [position]="{ lat: driverLocation().lat, lng: driverLocation().lng }"
    />
  } ... // implementation of other markers
  @if (directionsResult) {
    <map-directions-renderer [directions]="directionsResult" />
  }
</google-map>
```

Слика 40 – Пример имплементације приказа мапе

```

findLatLng(position: string, address: string): void {
    const bounds = new google.maps.LatLngBounds(
        new google.maps.LatLng(this.cityCenter().lat - 0.1, this.cityCenter().lng -
0.1),
        new google.maps.LatLng(this.cityCenter().lat + 0.1, this.cityCenter().lng +
0.1));
    this.geocoder
        .geocode({ address, bounds })
        .then((result) => {
            this.updatePosition(
                result.results[0].geometry.location.lat(),
                result.results[0].geometry.location.lng(),
                result.results[0].formatted_address,
                position
            );
        })
        .catch((error) => {
            console.error(error);
            this.toastService.error('Nismo pronašli željeno mesto na Google Mapi');
        });
}

```

Слика 41 – Пример употребе Google сервиса за одређивање координата адресе

6. РАД АПЛИКАЦИЈЕ

У овом поглављу биће приказан опис и рад веб апликације „ЕлФак такси“, која је осмишљена са идејом унапређења и олакшања свакодневних активности.

6.2. Опис апликације

Ова апликација омогућава муштеријама једноставно и брзо наручивање такси возила, самостално одређивање одредишта путовања и праћење локације возила у реалном времену. Такође, апликације је окренута и самом такси удружењу, односно возачима и менаџерима. Она возачима омогућава преглед нових захтева вожњи, одабир, односно прихватање следеће вожње и навигацију како до муштерије, тако и до одредишта. Док менаџерима пружа увид у пословање такси удружења, односно увид у највредније возаче актуелног месеца, преглед историје вожњи са претрагом и свих возача такође са претрагом.

Све групе корисника, након пријаве у систем, имају могућност прегледа својих профила и историје вожњи. Додатно, муштеријама које не желе да се региструју на систем, и даље је омогућен велики број функционалности система али без историје њихових активности.

Апликација је развијена тако да има прилагодљив веб дизајн како би се идеално приказала на паметном телефону и таблет уређају корисника. Фокус употребе система базиран је на идеји да сами паметни телефони поседују *GPS*. На тај начин се може одредити почетна адреса вожње. Такође, муштерији се оставља могућност измене почетне адресе у случају *GPS* грешке или ако жели да вожњу наручи за другу особу.

6.3. Профил корисника апликације

„ЕлФак такси“ је веб апликација осмишљена са идејом да обједини више такси сервиса и да кориснику олакша саму употребу система. Како су се данас сва такси удружења окренула мобилним апликацијама, овај приступ сам по себи не оптерећује корисника да било шта инсталира на свој паметни уређај. Овој веб апликацији сваки корисник ће лако приступити преко претраживача свог паметног телефона.

Профили корисника које имамо у веб апликацији:

- Муштерија - Свака особа која има приступ интернету и потребу да се брзо транспортује до другог места у граду. Овакви корисници имају могућност прегледа почетне стране, поручивање вожње, пријаву односно регистрацију на систем, могућност прегледа, измене и брисање профила, као и праћења историје вожњи.
- Возачи - Сами такси возачи, односно такси удружења, идеални су корисници ове веб апликације јер ће им било који паметни уређај који већ поседују бити довољан за пословање. Пријавом на систем добијају специјални преглед мапе на којој могу прегледати будуће вожње и управљати одредиштем, почетком и крајем исте.
- Менаџери – Запослени у такси удружењу, задужени за преглед пословања, лако могу остварити овакав увид прегледом веб апликације на било ком уређају јер је развијен приказ за паметне телефоне, таблете и рачунаре. Пријавом на систем добијају специјални табеларни преглед пословања такси удружења.

6.4. Случајеви коришћења

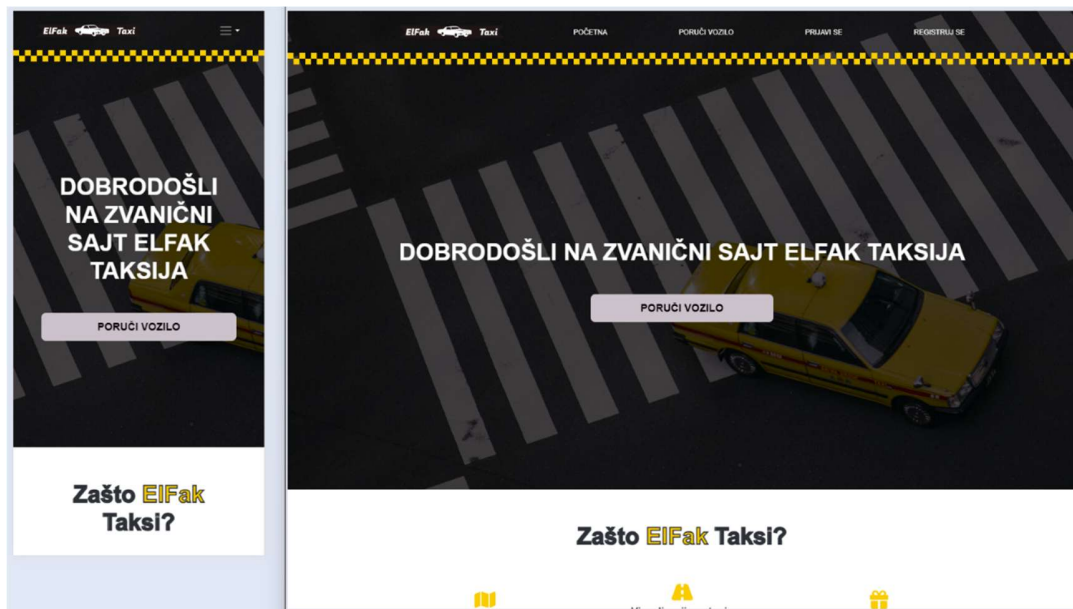
Имплементирани случајеви коришћења веб апликације:

- Приказ информација о веб апликацији
Опис: Приказ информација о веб апликацији којим се кориснику у кратким цртама јасно објашњавају све предности овог система, изглед ове странице је приказан на слици 42
Актери: Било који посетилац веб апликације
Предуслов: Улаз на веб апликацију било којим претраживачем
Основни ток:
 - Корисник приступа веб апликацији

- Приказује се почетна страница са основним информацијама
- Корисник види могућност за даљу навигацију

Изузеци: /

Последице: /



Слика 42 – Почетна страница

- Креирање корисничког профила муштерије

Опис: Приказ регистрационе странице и креирање корисничког профила муштерије, изглед ове странице је приказан на слици 43

Актери: Муштерија

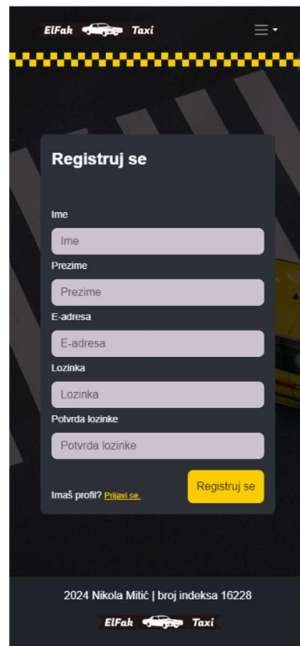
Предуслов: Корисник није већ пријављен на систем и нема профил са истом е-адресом

Основни ток:

- Кликот на навигационо дугме „REGISTUJ SE“ врши се навигација до регистрационе форме
- Унос података
- Кликот на потврдно дугме „Registruj se“ врши се провера валидности унесених података
- У случају исправности података, исти се прослеђују на *backend* и креира се нови профил памћењем података у бази податак
- У случају невалидних података, приказат ће се искачући прозор са садржајем обавештења о грешци приликом попуњавања форме

Изузеци: Невалидан унос неопходних података

Последице: Нови корисник, односно муштерија, пријављен је на систем и приказана му је *dashboard* страница те може поручити такси возило



Слика 43 – Регистрациона страница

- Пријављивање корисника на систем

Опис: Приказ странице за пријављивање на систем, изглед ове странице је приказан на слици 44

Актери: Муштерија, возач или менаџер

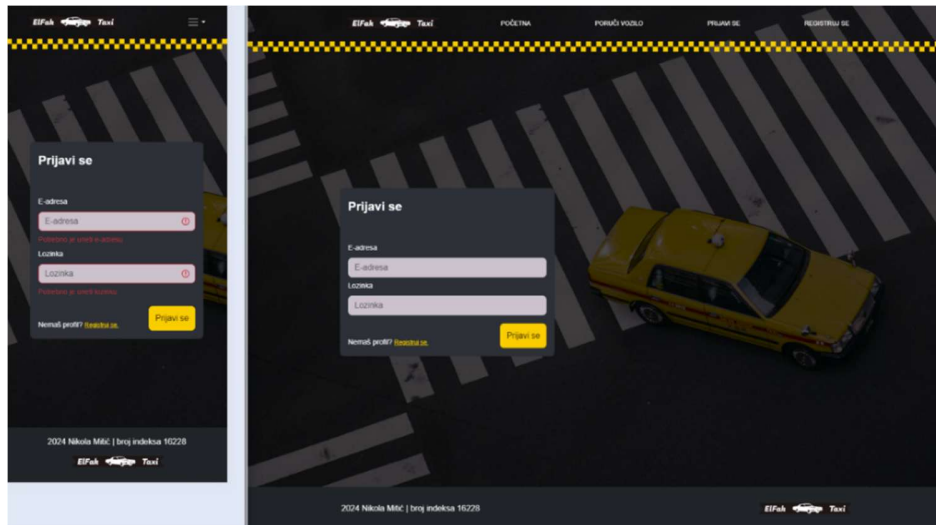
Предуслов: Корисник није већ пријављен на систем и има креиран профил

Основни ток:

- Кликом на навигационо дугме „PRIJAVI SE“ врши се навигација до форме за пријаву на систем
- Унос података
- Кликом на потврдно дугме „Prijavi se“ врши се провера валидности унесених података
- У случају исправности података, исти се прослеђују на *backend* и креира се нови аутентикациони токен
- У случају невалидних података, приказаше се искачући прозор са садржајем обавештења о грешци приликом попуњавања форме

Изузеци: Невалидан унос неопходних података

Последице: Корисник је пријављен на систем и приказана му је *dashboard* страница, а у зависности од типа корисник може поручити односно прихватити или прегледати возњу



Слика 44 – Страница пријаве на систем

- Поручивање возила

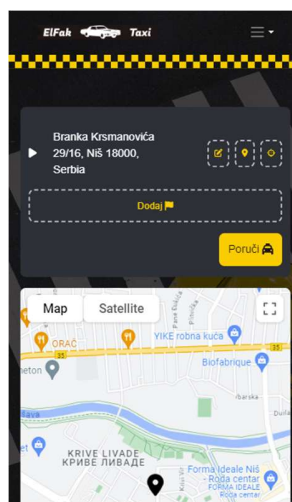
Опис: Уношење почетне и одредишне адресе и поручивање вожње, изглед ове странице је приказан на слици 45

Актери: Муштерија, са или без корисничког профила

Предуслов: Корисник не сме да буде пријављен као возач или менаџер

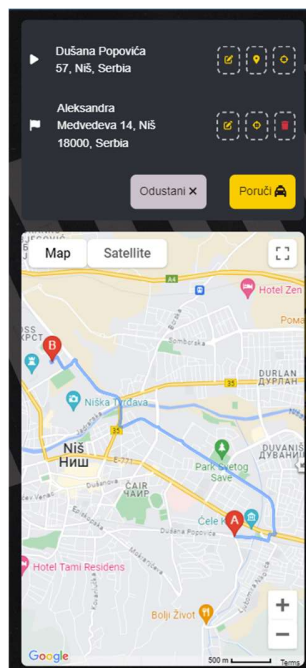
Основни ток:

- Кликом на навигационо дугме „PORUČI VOZILO“ врши се навигација до странице са формом за унос почетне и крајње адресе и приказом мапе града. Муштерија ће иницијално видети своју локацију.
- Унос почетне адресе могућ је уз помоћ GPS лоцирања, отварањем модала за одабир адресе или додиром мапе како би се одабрала тачка
- Унос одредишне адресе могућ је отварањем модала за одабир адресе или додиром мапе како би се одабрала тачка



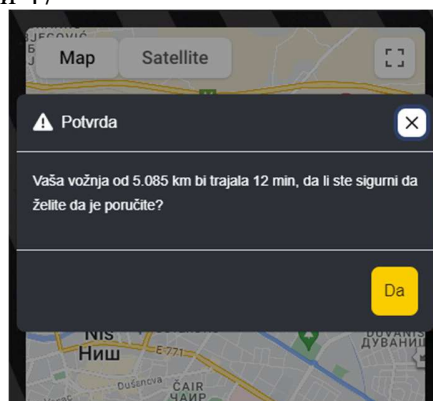
Слика 45 – Страница за поручивање возила

- Након уноса обе адресе, на мапи се илуструје путања, приказано на слици 46



Слика 46 – Приказ илустрације путовања на мапи

- У случају клика на дугме „Odustani“ одустаје се од захтева или у случају клика на дугме „Poruči“ отвара се дијалог са детаљним информацијама о возњи који је потребно прихватити кликом на дугме „Да“ како би се у систем уписала захтевана возња, дијалог је приказан на слици 47



Слика 47 – Приказ потврдног модала

- Систем захтева локацију свих активних возача и одређује најближег те њему првом прослеђује захтев за прихватање возње
- У случају да најближи возач не прихвати возњу у року од 60 секунди, захтев се прослеђује свим возачима

Изузеци: /

Последице: Муштерија је креирала захтев за вожњу, исти је забележен у бази података и прослеђен на прихватање

- Отказивање вожње

Опис: У случају да се муштерија предомислила постоји опција отказивања поручене вожње пре него што возач дође по њу, изглед прегледа овог стања приказан је на слици 48

Актери: Муштерија, са или без корисничког профила

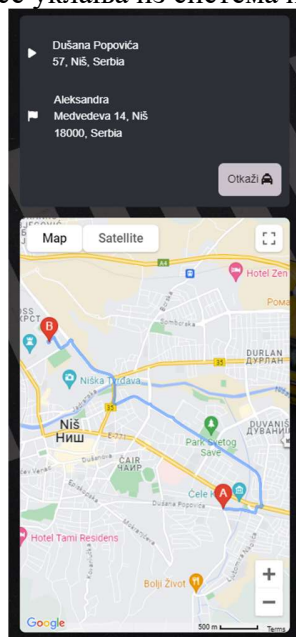
Предуслов: Вожња још увек није започета

Основи ток:

- Кликом на дугме „Откажи“ врши се брисање захтеване вожње у бази података. Уколико нико до тада није прихватио вожњу, овај захтев ће само нестати из листе понуђених возачима. Уколико је неко од возача прихватио вожњу и кренуо ка муштерији, навигација ће се прекинути и добиће информацију о отказивању. Муштерији се приказује почетни подrazумевајући приказ ове странице

Изузеци: Возач је купио корисника и вожња је већ почела

Последице: Вожња се уклања из система и базе података



Слика 48 – Приказ могућности отказивања вожње

- Преглед и управљање вожњом

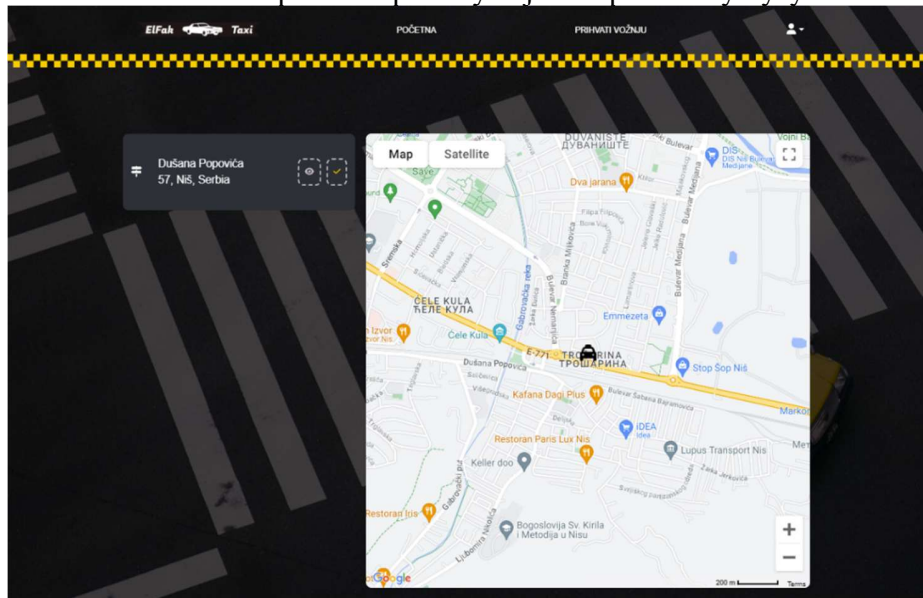
Опис: Возач има могућност прегледа путање вожње, прихватања, мењања одредишта и означавања почетног и крајњег тренутка вожње, изглед странице са овим могућностима је приказан на слици 49

Актери: Возачи

Предуслов: Водач мора да буде пријављен на систем и у систему постоје захтеване возње

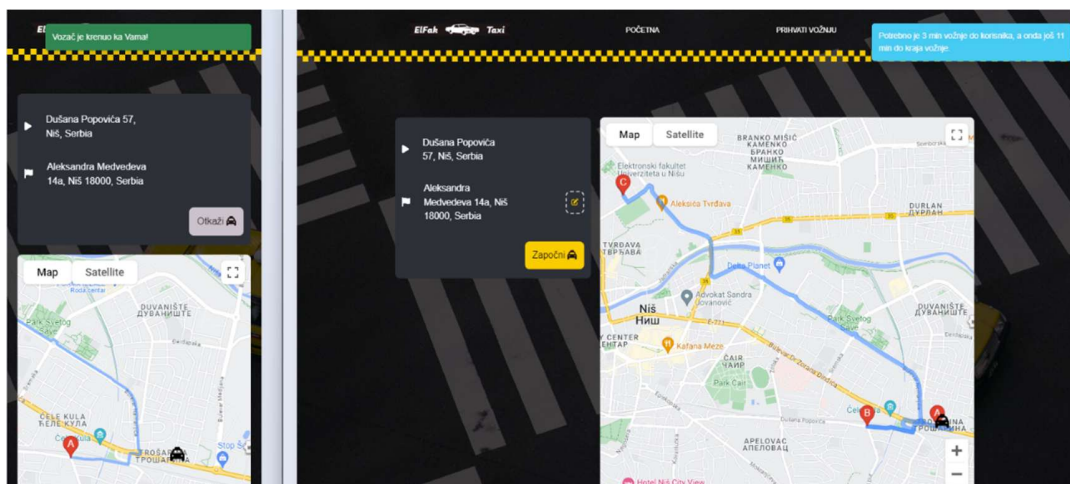
Основни ток:

- Корисник у листи доступних возњи кликом на дугме са иконицом око добија преглед возње на мапи, а у искачућем прозорчету информације о километражи и времену које ће провести у путу



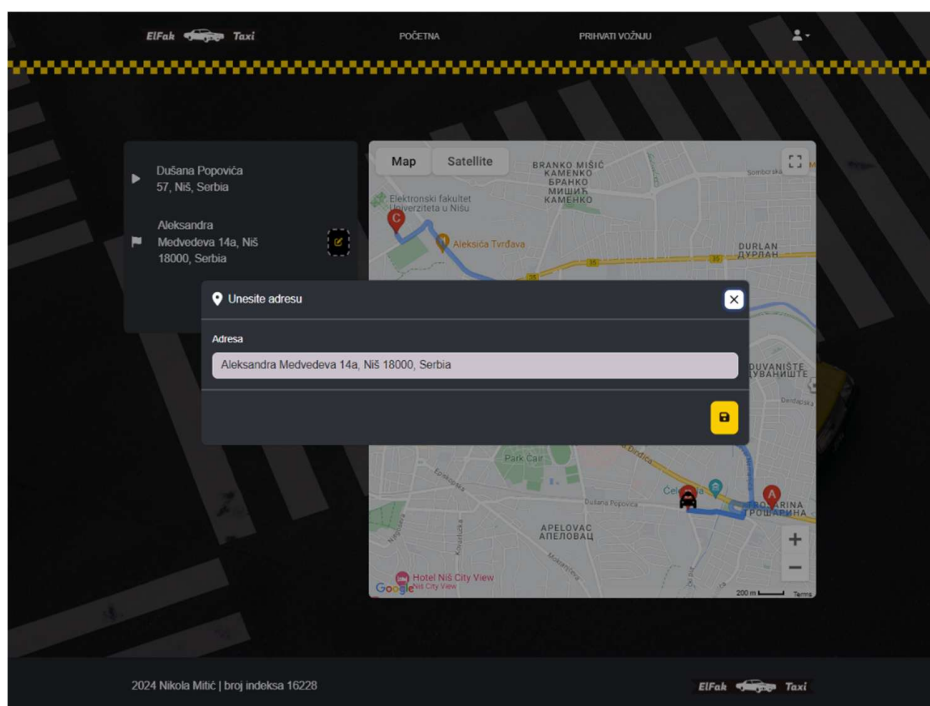
Слика 49 – Страница за прихватање возњи

- Корисник у листи доступних возњи кликом на дугме штикла прихвата возњу, односно систем бележи идентификациону ознаку возача у бази података. Уколико је возач био најближи полазној тачки, добио је предност од 60 секунди за прихватање возње, уколико је ово време протекло сви возачи су добили могућност прихватања ове возње, односно ствара се тркачка ситуација у којој захтевану возњу преузима возач који је најбрже прихвати кликом на дугме са иконицом штикла. Остали корисници возачи остају без исте у понуди захтеваних возњи. Промене које се дешавају на страници муштерије и возача приказане су на слици 50



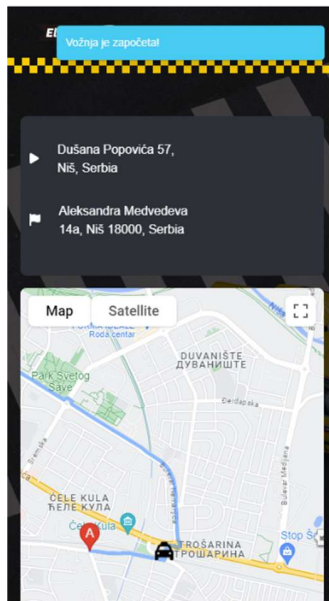
Слика 50 – Приказ прегледа када возач крене по муштерију

- Након што је прихватио возњу корисник може да измени унето одредиште или унесе исто у случају да оно не постоји у систему, ово је приказано на слици 51



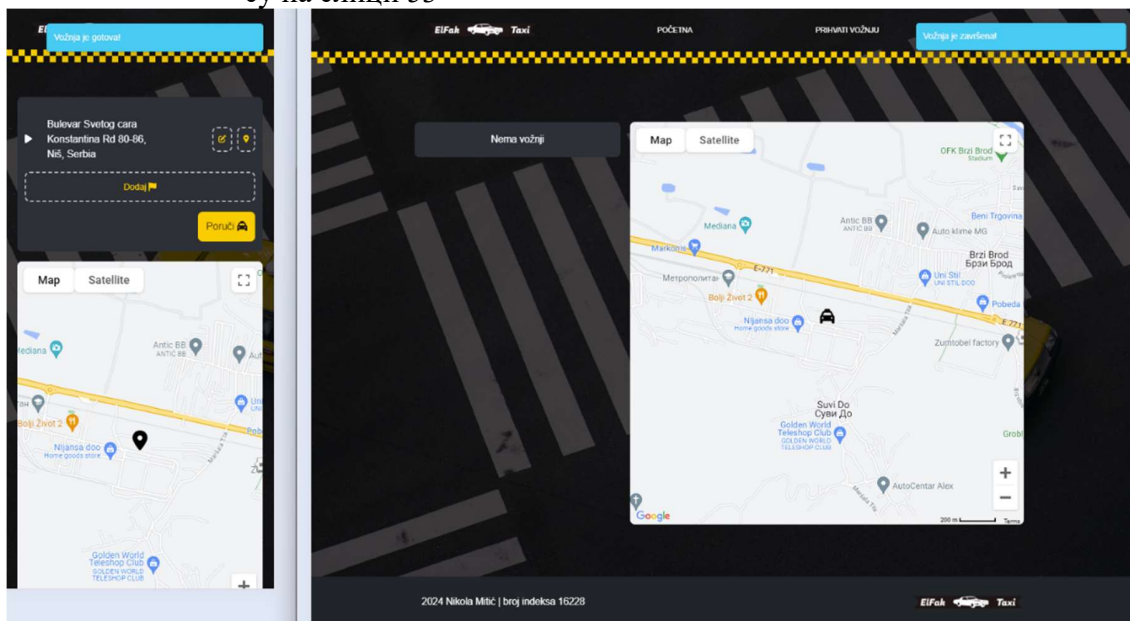
Слика 51 – Приказ корекције завршне адресе возње

- Након доласка до муштерије возач бележи време почетка возње кликом на дугме „Зарошпи“. Муштерија у том тренутку добија нови приказ илустрован на слици 52



Слика 52 – Приказ прегледа муштерије када је возња започета

- Након доласка до одредишта или раније, изузетно у случају договора са муштеријом, возач кликом на дугме „Заврши“ бележи време краја возње у систем. Промене на приказу код муштерије и возача приказане су на слици 53



Слика 53 – Финални приказ завршетка возње

Изузеци: Нема доступних возњи или је муштерија отказала возњу у току пута до исте

Последице: Возња у бази података добија све преостале податке

- Преглед пословања

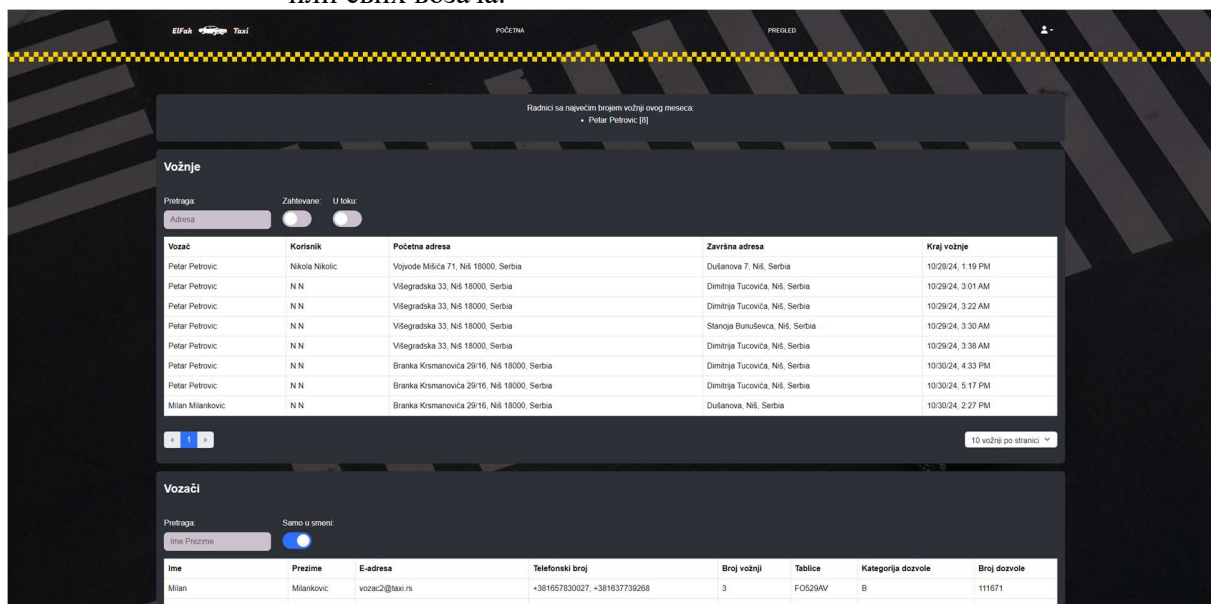
Опис: Пријављени корисник има могућност прегледа свих битних података о пословању такси удружења забележених у систем, изглед ове странице је приказан на слици 54

Актери: Менаџер

Предуслов: Корисник мора бити пријављен на систем

Основни ток:

- Кликот на навигационо дугме „PREGLJED“ отвара се приказ прегледа података о пословању
- Корисник прво може видети листу највреднијих возача овог месеца, са бројем остварених вожњи
- Даље, приказује се историја свих вожњи. Табела приказа садржи информације о муштерији, возачу, одредиштима и времену краја вожње. Корисник податке може претраживати уносом жељеног текста за претрагу у поље за претрагу, може одабрати да ли жели да види и вожње на чекању, као и вожње у току
- Приказује се и списак свих возача који постоје у систему. Табела прилаза садржи основне и контакт информације о возачу, укупан број остварених вожњи и број таблица возила које је тренутно у употреби. Корисник податке може претражити уносом жељеног текста за претрагу у поље за претрагу и одабиром приказа само возача у смени или свих возача.



Слика 54 – Приказ детаља пословања

Изузеци: /

Последице: Корисник је видео податке такси удружења

- Преглед корисничког профила

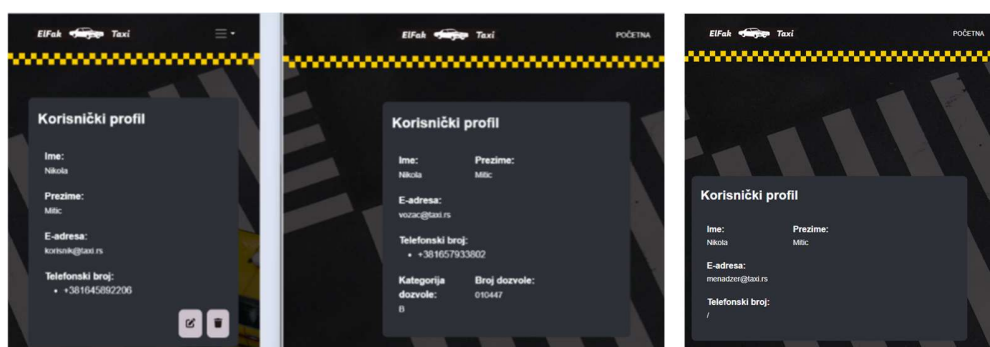
Опис: Пријављени корисник има могућност прегледа својих података забележених у систем, изглед ове странице је приказан на слици 55

Актери: Муштерија, возач или менаџер

Предуслов: Корисник мора бити пријављен на систем

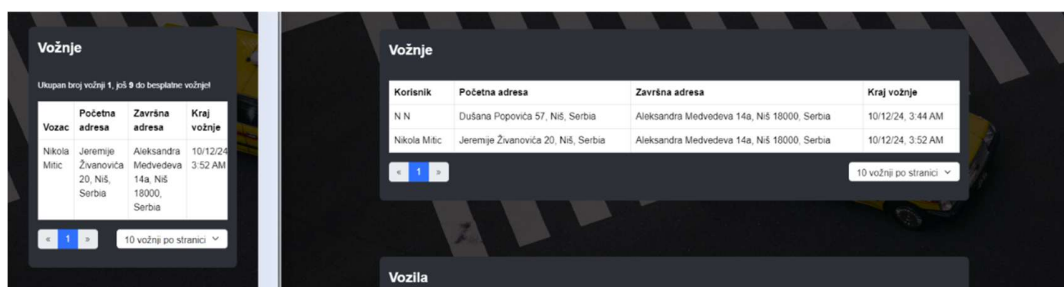
Основни ток:

- Кликом на навигационо дугме „PPROFIL“ отвара се приказ података о пријављеном кориснику
- У случају муштерије, приказује се и навигационо дугме са иконицом оловчице које води до странице са формом за измену података и дугме са иконицом канта за ђубре којом корисник може да бесповратно обрише свој профил



Слика 55 – Приказ детаља профила муштерије, возача и менаџера

- У случају муштерије и возача, приказује се историја такси вожњи. Гледано из угла корисника муштерије, овај део странице садржи информацију у томе која је ово вожња по реду, јер свака регистрована муштерија има право на сваку десету бесплатну вожњу, те је може од возача и захтевати. У случају да је у вожњи партиципирао непријављени корисник као муштерија, у систему неће постојати подаци о њему. Изглед овог приказа види се на слици 56.



Слика 56 – Приказ историја вожњи муштерије и возача

- У случају возача, приказује се и историју управљања возилима. Ова табела пружа детаљнији опис управљаним возилима, као и период управљања њима, она је приказана на слици 57.

| Tip | Brend | Model | Godina proizvodnje | Registracija | Upravlja od | Upravlja do |
|----------|---------|--------|--------------------|--------------|-------------|-------------|
| Privatno | Peugeot | Golf 5 | 1986 | 03-12 | 6/19/10 | 6/25/10 |
| Privatno | Peugeot | Golf 5 | 1986 | 03-12 | 6/5/10 | 6/9/10 |
| Privatno | Peugeot | Golf 5 | 1986 | 03-12 | 7/1/24 | U toku |

Слика 57 – Приказ историје управљања возилима возача

Изузеци: /

Последице: Корисник је видео своје податке, односно муштерија је обрисала свој профил

- Измена корисничког профила

Опис: Измена података за пријаву на систем или личних података, изглед ове стране је приказан на слици 58

Актери: Муштерија

Предуслов: Корисник мора бити пријављен на систем

Основни ток:

- Кликом на навигационо дугме са иконицом оловчице отвара се страница са формом попуњеном свим подацима корисника
- Корисник може изменити своје податке и додати нове, изменити или обрисати старе бројеве телефона
- Кликом на потврдно дугме врши се провера валидности унесених података
- У случају исправности података, исти се прослеђују на *backend* и врши се измена података у бази
- У случају невалидних података, приказаше се искачући прозор са садржајем обавештења о грешци приликом попуњавања форме

Изузеци: /

Последице: Подаци корисника измењени су у бази података

Слика 58 – Страница измене профила муштерије

7. ЗАКЉУЧАК

Теоријским делом рада извршено је истраживање *PHP* програмског језика, радног окружења *Laravel* и његовог објектно релационог мапера *Eloquent ORM*-а, као и основе релационих база података и њихову примену у креирању савремених веб апликација. Изучен је метод објектно релационог мапирања података из релационих база података и организација система по *MVC* архитектури.

У практичном делу рада, имплементиран је прототип веб апликације која служи за управљање такси удружењем. Коришћењем радног окружења *Laravel* и објектно релационог мапера *Eloquent ORM* обрађени су подаци који описују муштерије, возаче, возила и вожње, као и међусобне релације међу набројаним ентитетима. Да би се овако обрађени подаци приказали крајњем кориснику, независно од платформе на којој покреће апликацију, за приказ коришћено је радно окружење *Angular*.

Релациони модел базе податак је омогућио стварање комплексне структуре података која се објектно релационим мапером преноси на даљу обраду и приказ. Иницијално дефинисање одговарајућих релација међу табелама показало се као интересантан изазов који је поставио основу система. Затим, објектно релациони мапер, *Eloquent ORM*, веома је убрзао развој и олакшао рад са базом података. Његова могућност дефинисања модела за даљу обраду података у систему се истакла као највећа предност и олакшица у раду. То је довело до једноставног писања упита за прибављање и измену података, као и до велике брзине рада *backend* система. Међутим, и поред овако добрих предности, морало се водити рачуна о писању упита. Показало се као критично битно водити рачуна о *eager* и *lazy loading*-у, јер погрешно написани упити могу довести до пада перформанси система. Током целог развоја, највише је значила добра документација и велика заједница програмера који користе радно окружење *Laravel*.

Овако имплементирана апликација демонстрирала је употребу изучених технологија. Она преноси у реалност идеју о аутоматизацији управљања такси службом без потребе постојања диспечерског центра. Такође, решава проблем навигације у гужвама и даје сигурност муштеријама. Како је сваки систем могуће унапредити, тако и овде постоје могућности развоја система на више нивое. Ову веб апликацију могуће је додатно проширити додавањем нових функционалности попут имплементације система процене цене вожње, *online* плаћања, додавањем међустаница у вожњи и праћења броја активних возача. Верујем да би овакав систем, чак и без унапређења, у многама додатно унапредио јавни превоз.

8. ЛИТЕРАТУРА

- [1] „PHP - Features,“ [На мрежи]. Available: https://www.tutorialspoint.com/php/php_features.htm. [Последњи приступ 15 Децембар 2024].
- [2] „What is PHP? The PHP Programming Language Meaning Explained,“ [На мрежи]. Available: <https://www.freecodecamp.org/news/what-is-php-the-php-programming-language-meaning-explained/>. [Последњи приступ 15 Децембар 2024].
- [3] „What Is Symfony?,“ [На мрежи]. Available: <https://builtin.com/software-engineering-perspectives/symfony>. [Последњи приступ 15 Децембар 2024].
- [4] „Welcome to CodeIgniter4,“ [На мрежи]. Available: https://www.codeigniter.com/user_guide/intro/index.html. [Последњи приступ 15 Децембар 2024].
- [5] „PHP - Introduction,“ [На мрежи]. Available: https://www.tutorialspoint.com/php/php_introduction.htm. [Последњи приступ 15 Децембар 2024].
- [6] П. Л. Стоименов, „Модели података и пројектовање база података,“ Ниш, 2015/2016.
- [7] A. S. Gillis, „TechTarget,“ [На мрежи]. Available: <https://www.techtarget.com/searchapparchitecture/definition/UUID-Universal-Unique-Identifier>. [Последњи приступ 17 Мај 2025].
- [8] „Google Cloud,“ [На мрежи]. Available: <https://cloud.google.com/learn/what-is-a-relational-database?hl=en>. [Последњи приступ 14 Октобар 2024].
- [9] „What is an ORM – The Meaning of Object Relational Mapping Database Tools,“ [На мрежи]. Available: <https://www.freecodecamp.org/news/what-is-an-orm-the-meaning-of-object-relational-mapping-database-tools/>. [Последњи приступ 15 Децембар 2024].
- [10] „What is Laravel? Explain it like I’m five,“ [На мрежи]. Available: <https://runcloud.io/blog/what-is-laravel>. [Последњи приступ 15 Октобар 2024].
- [11] M. Surguy, „History of Laravel framework,“ у *Laravel - my first framework*, Leanpub, 2014, p. 154.
- [12] „<https://ngonyoku.medium.com/understanding-laravel-artisan-c834aae215fb>,“ [На мрежи]. Available: <https://ngonyoku.medium.com/understanding-laravel-artisan-c834aae215fb>. [Последњи приступ 17 Децембар 2024].
- [13] „Laravel Routing Guide – How Create Route to Call a View,“ [На мрежи]. Available: <https://www.cloudways.com/blog/routing-in-laravel/>. [Последњи приступ 15 Децембар 2024].

- [14] M. Surguy, „Routing,“ у *Laravel - my first framework*, Leanpub, 2014, p. 154.
- [15] „Route Model Binding,“ [На мрежи]. Available: <https://laravel.com/docs/11.x/routing#route-model-binding>. [Последњи приступ 15 Децембар 2024].
- [16] S. McCool, „Validation,“ у *Laravel Starter*, Birmingham, Packt Publishing Ltd., 2012, pp. 41 - 43.
- [17] „Data Validation in Laravel: Convenient and Powerful,“ [На мрежи]. Available: <https://kinsta.com/blog/laravel-validation/>. [Последњи приступ 15 Децембар 2024].
- [18] M. Surguy, „Laravel applications use Model-View-Controller pattern,“ у *Laravel - my first framework*, Leanpub, 2014, p. 154.
- [19] „What is Laravel? A Beginner's Introduction,“ [На мрежи]. Available: <https://www.fastfwd.com/what-is-laravel/>. [Последњи приступ 15 Октобар 2024].
- [20] „SQL vs NoSQL,“ [На мрежи]. Available: <https://www.mcloud.rs/blog/sql-vs-nosql/>. [Последњи приступ 1 новембар 2024].
- [21] „How to Use Redis with Laravel: A Comprehensive Guide,“ [На мрежи]. Available: <https://medium.com/@mohammad.roshandelpoor/how-to-use-redis-with-laravel-a-comprehensive-guide-247cfcac0a68>. [Последњи приступ 1 новембар 2024].
- [22] „Laravel Homestead,“ [На мрежи]. Available: <https://laravel.com/docs/11.x/homestead>. [Последњи приступ 15 Октобар 2024].
- [23] „A Practical Introduction to Laravel Eloquent ORM,“ [На мрежи]. Available: <https://www.digitalocean.com/community/tutorial-series/a-practical-introduction-to-laravel-eloquent-orm#how-to-create-a-one-to-many-relationship-in-laravel-eloquent>. [Последњи приступ 15 Октобар 2024].
- [24] I. Jound и H. Halimi, „Comparison of performance between Raw SQL and Eloquent ORM in Laravel,“ Faculty of Computing Blekinge Institute of Technology, Karlskrona.
- [25] M. Surguy, „Eloquent ORM,“ у *Laravel - my first framework*, Leanpub, 2014, p. 154.
- [26] M. Shukla, „Eloquent ORM in Laravel: Simplifying Database Interactions,“ [На мрежи]. Available: <https://manoj-shu100.medium.com/eloquent-orm-in-laravel-simplifying-database-interactions-b0269942f190>. [Последњи приступ 11 Јануар 2025].
- [27] „Eloquent ORM,“ [На мрежи]. Available: <https://laravel.com/docs/11.x/eloquent>. [Последњи приступ 15 Октобар 2024].
- [28] „Eloquent: Relationships,“ [На мрежи]. Available: <https://laravel.com/docs/11.x/eloquent-relationships>. [Последњи приступ 11 Јануар 2025].

- [29] „Mastering Route Model Binding in Laravel: A Comprehensive Guide.,“ [На мрежи]. Available: <https://medium.com/@moumenalisawe/mastering-route-model-binding-in-laravel-a-comprehensive-guide-d95d3e0327f2>. [Последњи приступ 31 октобар 2024].
- [30] Н. Митић, „taxi-api, GitHub,“ [На мрежи]. Available: <https://github.com/MiticBNikola/taxi-api/>. [Последњи приступ 17 Мај 2025].
- [31] Н. Митић, „taxi-web, GitHub,“ [На мрежи]. Available: <https://github.com/MiticBNikola/taxi-web/>. [Последњи приступ 17 Мај 2025].
- [32] „MySQL: Understanding What It Is and How It’s Used,“ [На мрежи]. Available: <https://www.oracle.com/mysql/what-is-mysql/>. [Последњи приступ 1 новембар 2024].
- [33] „ACID Explained: Atomic, Consistent, Isolated & Durable,“ [На мрежи]. Available: <https://www.bmc.com/blogs/acid-atomic-consistent-isolated-durable/>. [Последњи приступ 1 новембар 2024].
- [34] „What are decorators in Angular?,“ [На мрежи]. Available: <https://www.geeksforgeeks.org/what-are-decorators-in-angular/>. [Последњи приступ 16 Октобар 2024].
- [35] „What is Angular: The story behind the JavaScript framework,“ [На мрежи]. Available: <https://www.altexsoft.com/blog/the-good-and-the-bad-of-angular-development/>. [Последњи приступ 16 Октобар 2024].
- [36] „Angular Signals: A Complete Guide,“ [На мрежи]. Available: <https://kurtwanger40.medium.com/angular-signals-a-complete-guide-04fa33155a46>. [Последњи приступ 16 Октобар 2024].
- [37] „SPA (Single-page application),“ [На мрежи]. Available: <https://developer.mozilla.org/en-US/docs/Glossary/SPA>. [Последњи приступ 16 Октобар 2024].
- [38] „What is web socket and how it is different from the HTTP?,“ [На мрежи]. Available: <https://www.geeksforgeeks.org/what-is-web-socket-and-how-it-is-different-from-the-http/>. [Последњи приступ 16 Октобар 2024].
- [39] „JetBrains,“ [На мрежи]. Available: <https://www.jetbrains.com/webstorm/>. [Последњи приступ 12 Октобар 2024].
- [40] „Google Cloud Console,“ [На мрежи]. Available: <https://console.cloud.google.com>. [Последњи приступ 18 Октобар 2024].
- [41] „2.3. Converting ERD to a relational model,“ [На мрежи]. Available: https://runestone.academy/ns/books/published/practical_db/PART2_DATA_MODELING/03-ERD-to-relational/ERD-to-relational.html. [Последњи приступ 14 Октобар 2024].

- [42] „ER diagrams vs. EER diagrams: What’s the difference?“, [На мрежи]. Available: <https://nulab.com/learn/software-development/er-diagrams-vs-eer-diagrams-whats-the-difference/>. [Последњи приступ 14 Октобар 2024].
- [43] „Extended Entity-Relationship (EE-R) Model,“ [На мрежи]. Available: <https://www.tutorialspoint.com/extended-entity-relationship-ee-r-model>. [Последњи приступ 14 Октобар 2024].
- [44] M. J. U. H. N. J. R. Edy Budiman, „Eloquent Object Relational Mapping Models for Biodiversity Information System,“ Universitas Mulawarman, Samarinda, Indonesia, 2017.