Универзитет у Београду
Математички факултет

Мастер рад

Апликација у стилу микросервиса за праћење аутономних такси-возила у реалном времену

Студент:
Вељко Матић

Ментор:
др Саша Малков

Београд, 2019.

Ментор:

др Саша Малков, ван. проф.
Математички факултет, Универзитет у Београду

Чланови комисије:

др Владимир Филиповић, ван. проф.
Математички факултет, Универзитет у Београду

др Филип Марић, ван. проф
Математички факултет, Универзитет у Београду

*Датум одбране:*_____

Апликација у стилу микросервиса за праћење аутономних такси-возила у реалном времену

Апстракт - Апликација за праћење аутономних такси-возила у реалном времену је веб апликација отвореног кода чија је серверска страна имплементирана у микросервисној архитектури. Апликација користи податке о вожњама аутономних такси-возила у Њујорку. Подаци су саставни део скупа података *NYC Open Data* који садржи све потребне информације о почетку и крају вожње, географским кординатама на основу којих се могу пратити и графички приказати руте, броју путника итд. Апликација је написана коришћењем савременим веб технологијама које су базиране на програмском језику *JavaScript*: серверска страна је написана у радном оквиру *Node.js Express*, док је клијентска страна написана у радном оквиру *React* са складиштем *Redux*. Информације о корисницима се чувају у нерелационој бази *MongoDB*, а информације о вожњама у бази *Redis*. За комуникацију у реалном времену коришћена је библиотека *socket.io*, а пошто је комуникација између миркосервиса условљена појавом догађаја (енг. *event driven* архитектура) коришћен је брокер порука *RabbitMQ*.

Application in microservices style for tracking autonomous taxi-vehicles in real time

Abstract - Application for tracking autonomous taxi-vehicles in real time is a open source web application with the server side part bult in microservice architecture. The application uses data about missions (drivings) of autonomous taxi-vehicles in New York. Used data are part of NYC Open Data datasets which contain all needed information about start and end of driving, geographical coordinates on which we can track and graphicly show routes, number of passengers etc. Application was written in modern web technologies which are based on programming language JavaScript: server side was written in Node.js Express framework, and client side was written in React framework with Redux state management. Information about users are kept in nonrelation database MongoDB, information about missions (drivings) in Redis database. Socket.io library is used for communication in real time, and because communication between microservices are conditioned by the occurrence of the event (event driven architecture), we use a RabbitMQ message broker.

Садржај

1. Увод

Према многима аутономна возила су будућност превоза људи. Пошто су аутономна возила још у развитку неопходне су апликације за праћење аутономних вожњи. Алгоритми вештачке интелигенције који управљају овим возилима, анализирају вожње и руте имају за сврху да унапреде те апликације. Државе које предњаче у развијању аутономних возила су поред САД-а Холандија, Јапан и Сингапур [1]. Апликација за праћење аутономних такси-возила има за циљ да омогући поновно сагледавање кретања свих ових возила, препознавање погрешних рута и неисправних вожњи. Апликација ради у реалном времену користећи савремене веб технологије које су базиране на програмском језику *JavaScript*. Серверска страна је написана у радном оквиру *Node.js Express*, док је клијентска страна написана у библиотеци *React* са складиштем *Redux*. Серверска страна апликације урађена је у стилу микросервиса. За комуникацију између микросервиса коришћен је брокер порука *RabbitMQ.* Апликација користи податке о вожњама аутономних такси-возила у Њујорку. Подаци су саставни део скупа података *NYC Open Data* који садржи све потребне информације о почетку и крају вожње, географским кординатама на основу којих се могу пратити и графички приказати руте, броју путника итд. Апликација приказује те податке у реалном времену и омогућава сагледавање успешности алгоритама за управљање возилима, као и за саму услугу превоза.

2. Технологије

2.1. Микросервисна архитектура

У наставку ће бити описан настанак, потреба и мане микросервисне архитектуре.

2.1.1 Настанак микросервиса и њихова дефиниција

Од настанка рачунарства постоји потреба за сталним унапређењем начина развоја софтвера, поготово у случају великих система. Нови принципи и концепти развоја софтвера настају посматрањем претходних, усвајањем нових технологија и посматрањем како утичу на процес производње софтвера.

Битни концепти за развој микросервиса су пројектовање софтвера према домену проблема (енг. *Domain-driven Design*), континуално испоручивање (енг. *Continuous Delivery*), виртуелизација (енг. *Virtualization*), аутоматизација инфраструктуре (енг. *Infrastructure Automation*), мали аутономни тимови (енг. *Small autonomous teams*) и скалабилни системи (енг. *Systems at scale*) [2].

Пројектовање софтвера подразумева да је главни фокус пројекта на моделу домена и логици домена. Комплексни дизајн мора бити базиран на моделу домена и мора постојати сарадња између техничких стручњака и стручњака домена проблема, са циљем да итеративно креирају концептуални модел чија је сврха да реши одређене проблеме из домена [3].

Континуално испоручивање је приступ развоју софтвера у коме тимови производе нове верзије у кратким циклусима, чиме се омогућује да се софтвер може пустити у рад у било ком тренутку [3].

Виртуелизација скрива физичке карактеристике рачунарске платформе од корисника, приказујући искључиво апстрактну рачунарску платформу [5][6]. Више нема потребе управљати се старим моделом *„један сервер – једна апликација“*. Могуће је имати више сервера са различитим оперативним системима, тако да се сви покрећу на истој хардверској платформи. Сваки сервер се може посматрати као посебни ентитет, тј. као посебна машина. Отказ једног таквог ентитета нема утицаја на рад главне машине, платформе за виртуелизацију, нити на рад осталих ентитета са којима дели ресурсе.

Аутоматизација инфраструктуре је процес стварања скрипти за подешавање окружења, што подразумева инсталирање оперативног система, инсталирање и конфигурација сервера на инстанцама, конфигурисање начина како инстанце и софтвер комуницирају између себе, и друго. Захваљујући скриптама за подешавање окружења иста конфигурација може да се примени на један чвор или на хиљаде њих. Другим речима, аутоматизација рада инфраструктуре се огледа у описивању инфраструктуре и њене

конфигурације једном или више скрипти тако да се окружење може ископирати на начин који минимизује могућност грешке [7].

Систем је скалабилан ако је ефикасан и практичан при употреби ресурса у ситуацији када се сусреће са великим оптерећењем, на пример, код великог скупа података, великог броја корисника, или великог броја чворова који учествују у дистрибуираном систему [8]. Постоји хоризонтална и вертикална скалабилност када се говори о самом хардверу. Вертикална скалабилност је када се апликација смештена на једном серверу, а на повећан проток се реагује тако што се серверу додаје меморија, јачи процесор, нова језгра или додатни хард диск. Хоризонтална скалабилност је идеалније решење, посебно за велике системе. Додавањем нових чворова систем наставља да ради као до сада само са новим играчем (чвором) у тиму. Чвор представља један сервер.

Захваљујући пројектовању софтвера према домену проблема дошло се до бољих начина да се дизајнирају системи и до схватања важности да се реалан свет огледа у рачунарском коду. Захваљујући континуалном испоручивању софтвер се пушта у рад ефективније и ефикасније, и усталио се принцип да се сваки стабилни и потпуно тестирани кôд третира као потенцијални кандидат за пуштање у рад. Платформе за виртуелизацију омогућавају да се по потреби обезбеде и промене величине рачунара, док аутоматизација инфраструктуре омогућује да се лако скалирају. Неке велике успешне организације као Амазон (енг. *Amazon*) и Гугл (енг. *Google*) су развиле и подржале концепт да мали тимови имају пуну контролу над животним циклусом сервиса које развијају. Компанија Нетфликс (енг. *Netflix*) је пре више година поделила са заједницом начине како да се креирају скалабилни системи, што је било тешко замисливо пре само десетак година.

Микросервиси нису настали независно, већ као одговор на реалне потребе у области развоја софтвера. Пројектовање софтвера према домену проблема, континуално испоручивање, виртуелизација, аутоматизација инфраструктуре, мали аутономни тимови, скалабилни системи су корени микросервиса.

Једна дефиниција микросервиса би била: *"Микросервиси су мали аутономни сервиси који раде заједно"* [2].

Микросервиси постоје захваљујући свему што је било пре њих. Многе организације су уочиле да уз микросервисну архитектуру брже испоручују софтвер, као и да се брже адаптирају на нове технологије. Микросервиси омогућавају више слободе и бржи одговор на неизбежне промене које се стално дешавају у развоју софтвера.

2.1.2. Карактеристике микросервиса

Основне карактеристике микросервиса су мала величина, фокусираност и независност.

2.1.2.1. Величина микросервиса и фокусираност

Додавањем сваке нове функционалности програмски кôд расте, и самим тим, када год треба направити промене у коду или поправити грешку, све је теже детектовати део кода који треба исправити, поготово у случају када постоје сличне функционалности.

Софтверски систем се назива "монолитним" када је при дизајну примењена монолитна архитектура, таква да су сви аспекти који су фукнционално различити (нпр. унос и приказ података, процесирање података, обрађивање грешака и кориснички интерфејс) спојени, уместо да се састоје од архитектурално раздвојених целина [9]. У монолитном систему је важно да настојимо да кôд буде кохезивнији тако што правимо апстракције или модуле. Кохезија (настојање да сав сродан кôд буде груписан заједно) је јако битан концепт и код микросервиса. Роберт Мартин који је дао дефиницију принципа искључиве одговорности (енг. *Single Responsibility Principle*) гласи: *"Треба ɡрȕписаᵭи све шᵭо се мења са исᵭим узроком, и раздвојиᵭи све шᵭо се мења са различиᵭим узроцима"* [10]. Код микросервиса овај приступ се примењује при формирању независних сервиса. Јако је битно да се пази да ограничења сервиса буду заснована на ограничењима проблема који се решава. Самим тим за сваку функционалност је јасно где се налази програмски кôд. Сходно чињеници да се пази на ограничење сервиса, па тиме и ограничење реалног проблема, смањује се шанса да сервис постане превише велик и тако се избегавају сви потенцијални проблеми који могу из тога да изникну.
Величина микросервиса није јасно дефинисана јер није могуће прецизирати број линија програмског кода, па се може рећи да величина кода зависи од конкретних случајева.

Још један битан фактор при схватању колико мали треба да буде микросервис је однос сервиса према структури тима. Ако је целокупан програмски кôд решења превелик да би се њиме бавио мали тим, логично је да има потребе да се разбије на ситније целине. Битно је да сервис буде што мањи чиме се повећавају предности, али и мане микросервисне архитектуре. Што се више смањује сервис то више вреде погодности које доносе међузависност, али такође се дешава да се повећава и комплексност која се појављује услед повећања броја сервиса. У циљу бољег управљања овим комплексностима тежи се све мањим и мањим сервисима.

2.1.2.2. Независност

Сваки микросервис је независан. Један микросервис може бити изолован на платформи у облику сервиса (енг. *Platform as a Service - PAAS*) или може бити један процес оперативног система. Иако ова изолација ствара додатне трошкове, резултујућа једноставност олакшава дизајн дистрибуираног система и новије технологије омогућавају да се пређе преко изазова који произлазе из оваквог приступа. У литератури која се бави микросервисима, саветује се да сервиси буду способни да се мењају независно једни од других и да могу да се пуштају у рад без потребе да се траже измене корисника сервиса.

Ако је превише дељења података, сервиси постају превише упарени са интерним репрезентацијама података. Ово смањује независност јер прављење измена захтева додатно консултовање са корисницима сервиса. Сервиси откривају интерфејс за програмирање апликација - АПИ (енг. *application programming interface - API*) и

комуницирају преко истих. АПИ треба да буде независан од технологије за програмирање, што омогућава да се микросервиси пишу у различитим програмским језицима.

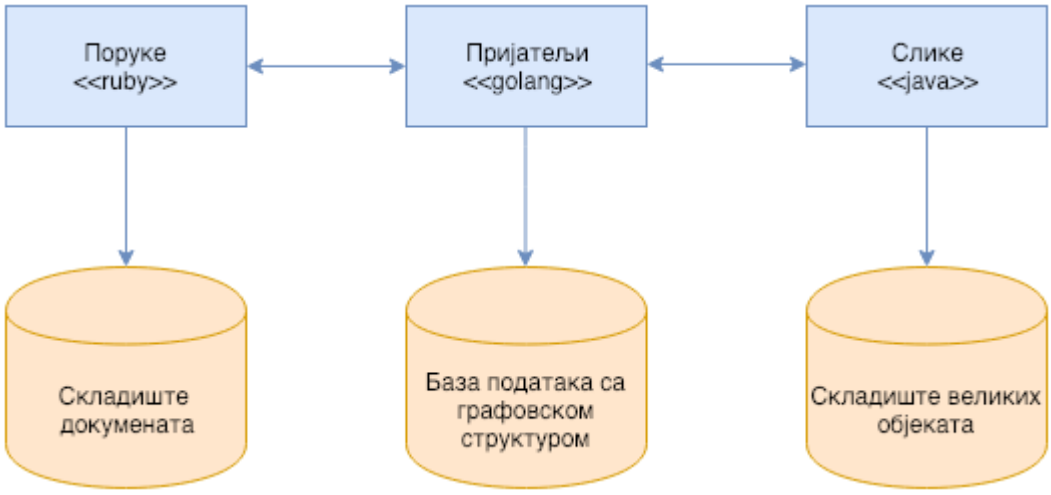
2.1.3. Предности коришћења микросервиса

Основне предности коришћења микросервиса су:

1. Технолошка хетерогеност
2. Одрживост система при престанку рада микросервиса
3. Скалирање
4. Једноставност процеса пуштања софтвера у рад
5. Организација рада
6. Могућност искоришћења микросервиса за различите сврхе
7. Оптимизација за лаку замену

2.1.3.1. Технолошка хетерогеност

Систем сачињен од више независних сервиса који међусобно комуницирају омогућује да се за сваки користи другачија технологија. Самим тим се могу бирати алати који најбоље одговарају проблему који се решава. Ако треба да се унапреди ефикасност једног дела система, могуће је искористити жељену технологију да би се постигла жељена ефикасност. Такође је могуће да начин чувања података буде различит за различите делове система. На пример, за друштвену мрежу, све везе између корисника могу да се сачувају у бази података са графовском структуром, а поруке корисника на мрежи могу да се чувају у складишту докумената, док се велики објекти (енг. *Binary Large Object - BLOB*) као слике чувају у складишту великих објеката. Овиме се постиже хетерогена структура која је за наведени пример приказана на слици 1.



Слика 1. *Хетерогена архитектура*

Микросервиси омогућавају да новије технологије брже уђу у употребу. Једна од највећих препрека при коришћењу нових технологија је ризик који долази са њима. Код монолитних апликација, ако се испробава нови програмски језик, база података, и слично, свака промена има утицај на велики део остатак система. Међутим, код система који се

састоји од различитих сервиса, постоји више места где може да се испроба нова технологија. На пример, може да се изабере сервис са најмањим ризиком, самим тим и најмањим негативним утицајем на систем и на њему се употреби нова технологија. Ако дође до неких нежељених догађаја, минимизован је утицај на систем. Могућност да се брзо усвајају нове технологије је постала велика предност за многе организације.

Још један од битнијих аргумената против испробавања нових технологија у монолитним системима је величина. Испробавање нових технологија код монолитног система захтева писање комплетног система из почетка што због његове величине најчешће није исплативо и могућност појаве грешке је велика. Код микросервисне архитектуре поновно писање једног микросервиса не захтева велико време због његове величине и могућност појаве грешке је мања. Најчешћи разлог за поновно писање микросервиса на нове технологије су лоше перформансе микросервиса условљене лимитом технологије која се користи.

Коришћење више технологија не долази без додатних трошкова. Неке организације се труде да ограниче избор програмских језика. На пример, Нетфликс (енг. *Netflix*) и Твитер (енг. *Twitter*) користе углавном Јава виртуелну машину - JBM (енг. *Java Virtual Machine - JVM*) као платформу пошто имају јако добро разумевање поузданости и перформанси система. Интерно развијају библиотеке и алате за JBM што олакшава скалирање, али отежава рад сервиса и клијената који нису базирани на Јави. Ни Твитер ни Нетфликс не користе искључиво један скуп технологија за све своје послове [2].

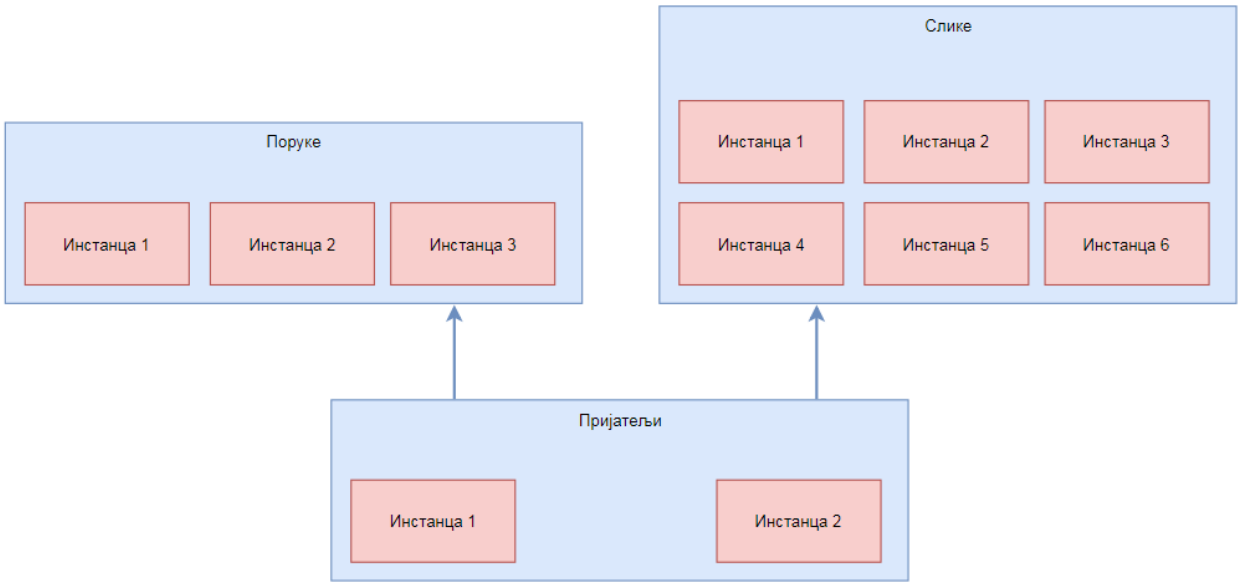
2.1.3.2. Одрживост система при престанку рада микросервиса

Кључан концепт у одрживости система се назива изолациона преграда (енг. *bulkhead*). Ако једна компонента система престане да ради, проблем се може изоловати и неће се пренети на остатак систем, што омогућава да остатак система настави са радом. У монолитној архитектури, ако сервис престане са радом, све престаје са радом, па се проблем решава коришћењем више инстанци апликације. Микросервисна архитектура се гради тако да је отпорна на престанак рада једног или више микросервиса. У случају престанка рада једне или више инстанци микросервиса, функционалност система деградира. Да би микросервисни системи добро искористили одрживост, морају се разумети нови извори потенцијалних проблема код дистрибуираних система. Код дистрибуираних система често долази до престанка рада сервиса. Да би се добро имплементирали микросервиси, битно је знати шта се дешава у овим ситуацијама, како ће остатак система да се понаша у случају престанка рада сервиса и да ли ће имати утицај на финалног корисника софтвера, и ако да, какав утицај. Ово су све кључни детаљи код дизајнирања једног микросервисног система.

2.1.3.3. Скалирање

Код великих монолитних сервиса све расте заједно. Један мали део система може имати проблеме са ефикасношћу, али ако је то понашање закључано у огромну монолитну апликацију, све мора да се скалира заједно. Са поделом апликације на микросервисе се добија да могу да се скалирају само они сервиси који имају потребе за скалирањем, без

потребе да се мењају други делови система. То омогућава да се сервиси који се не мењају извршавају на мањем хардверу, слабије снаге, као што је приказано на слици 2.



Слика 2. *Скалирање микросервиса*

Гилт (енг. *Gilt*), продавац одеће преко интернета, је усвојио микросервисе баш из овог разлога. Компанија је почела развој своје монолитне интернет апликације 2007. године користећи технологију *Rails*, и тако је 2009. године дошло до тога да интернет апликација више није била способна да издржи оптерећење великог броја посета. Главни систем расцепкан је на мање делове, Гилт је успео да унапреди начин борбе са великим ударима у броју посета, и данас имају преко 450 микросервиса, од којих се сваки извршава на више одвојених рачунара.

Када се користе системи који по потреби могу да се конфигуришу, као на пример Амазон Веб Сервиси (енг. *Amazon Web Services*), могуће је применити скалирање по потреби на делове којима је то потребно, што омогућује да се ефикасније контролишу трошкови. Јако често се дешава да избор архитектура буде повезан са уштедом новца.

2.1.3.4. Једноставност процеса пуштања софтвера у рад

Када се у монолитној апликацији од преко милион линија кода догоди измена у једној линији, цела апликација мора да се компајлира и мора опет да се прође кроз процес пуштања у рад. Сам процес поновног пуштања у рад може бити ризичан. У пракси се често чека да се скупи више измена пре пуштања у рад нове верзије апликације. То самим тим значи да свака следећа измена доноси велики број промена, односно што је дужи период између два пуштања у рад апликације већа је шанса појаве грешака. Код микросервиса је ово много боље уређено пошто може да се направи измена на једном сервису и да се само он поново пусти у рад, без потребе да се утиче на остатак система. Ако се појави проблем, лако се изолује на нивоу сервиса и брзо може да се врати у последње стабилно стање док се не реши новонастали проблем. Такође, то значи да корисници брже могу да дођу до

нових функционалности. Ово је један од главних разлога што компаније као Амазон и Нетфликс користе ову архитектуру.

2.1.3.5. Организација рада

Монолитна архитектура и организација људи у велике тимове су се у пракси показали као неефикасни. Код дистрибуираних тимова организација постаје још тежа. Организација у мање тимове који раде на мањим деловима програмског кода је много ефикаснија. Микросервиси дозвољавају да се архитектура боље прилагоди потребама организације, да се смањи број људи који ради на једном делу програмског кода тако да се постигне добар однос између величине тима и продуктивности. Такође, могуће је преносити власништво над сервисима између тимова како би се тимови упознали са комплетним системом.

2.1.3.6. Могућност искоришћења микросервиса за различите сврхе

Једна од најбитнијих очекиваних предности дистрибуираних система и сервисно оријентисане архитектуре је могућност поновног искоришћења функционалности. Код микросервиса функционалности могу бити искоришћене на различите начине за различите потребе, што је посебно битно када се узме у обзир како крајњи корисници користе софтвер. Смисао је да микросервиси омогућавају фунцкионалности које могу бити прилагођене различитим решењима, било за веб или мобилне апликације. Потребно је смислити нове начине тако да апликација може бити и интернет апликација и стандардна апликација и мобилна апликација.

Организације полако одбацују уско размишљање о каналима употребе и прелазе на више свеобухватне концепте рада са крајњим корисницима. Из тог разлога је неопходно смислити нове архитектуре које могу да прате тај тренд.

2.1.3.7. Оптимизација за лаку замену

Велике организације често имају велике системе који су стари по двадесет и више година и они су од виталног су значаја за рад компаније. Углавном су написани у језику који се више не користи свакодневно као што је Кобол (енг. *Cobol*) и раде на хардверу који је дотрајао пре десетак година. Ови системи опстају и нису модернизовани зато што су превелики и јако је велик ризик да се мењају. Када су појединачни сервиси мали много је јефтиније заменити их са бољом имплементацијом или чак обрисати. Пошто су микросервиси често сличне величне, препреке за поновно писање или брисање сервиса у потпуности су веома мале. Тимови који користе микросервисни приступ комотно мењају сервисе кад је потребно или их елеминишу кад више немају потребе за њима.

2.1.4. Неке мане микросервиса

Код микросервиса треба обратити пажњу да се не користе за сваки сценарио јер нису примењиви за све.

Микросервиси имају све комплексности као и други дистрибуирани системи. Свако ко долази из света монолитних система мора да унапреди своје знање о процесу пуштања апликације у рад, тестирања и надзора да би упознао све до сад наведене предности. Такође, такви корисници морају да промене начин размишљања како се системи скалирају и да обезбеде одрживост система при престанку рада неких делова.

Други проблем који може да се појави код рада са микросервисима је недовољно разумевање домена проблема. Што је мање разумевање теже је пронаћи права ограничења за сервис. Као што је наведено раније, погрешне процене код ограничења сервиса могу резултирати са много промена у комуникацији између сервиса, што је јако скупа операција. Тако да је јако битно да се одвоји време за добро упознавање домена проблема да би се исправно одредила ограничења сервиса.

Тежина изазова рада са микросервисима се повећава скалирањем. Ако се одржавање ради ручно биће изводљиво пазити на један до два сервиса, али већ негде између пет и десет долази до проблема [2]. Стари начин надзора, кад се надгледају рад процесорске јединице и меморије, биће добар за неколико сервиса, али постаје све тежи са додавањем нових колаборација између сервиса.

Зато се у литератури препоручује добро упознавање са доменом проблема, унапређивање претходних знања и добро дизајнирање система.

2.2. Апликације за рад у реалном времену

У наставку биће објашњено шта су апликације које раде у реалном времену и подела тих апликација по последицама.

2.2.1. О апликацијама за рад у реалном времену

Апликације за рад у реалном времену су апликативни програми који функционишу у времеском оквиру тако да крајњи корисник има утисак као да се извршава моментално [11]. Апликације за рад у реалном времену морају да гарантују извршавање у ограниченом временском року који се често назива “крајњи рок” [12].

Неке од примена апликација за рад у реалном времену су [11]: програми за пренос видео конференције, односно, пренос видео сигнала уживо, пренос аудио сигнала преко интернет протокола (енг. *VoIP voice over Internet Protocol*), видео игрице које се играју преко интернета, новчане трансакције, критични делови система као на пример АБС систем код возила који омогућава да код кочења не дође до блокирања точкова чиме би возило проклизало [13], итд.

Систем за рад у реалном времену се описује као “*систем који контролише окружење тако што прима податке, обрађује их и враћа резултате довољно брзо да утиче на окружење у том тренутку*” [14]. Исправност ових типова система зависи од временског аспекта колико и од функционалног. Одговори у реалном времену се често очекују да буду у милисекнудама, а неретко у микросекундама. Систем за који није наведено да

функционише у реалном времену не може да гарантује да ће одговор бити у оквиру одређеног временског рока, али је могуће да се предвиди очекивано време одговора.

2.2.2. Подела апликација (система) за рад у реалном времену по последицама

Апликације (системи) за рад у реалном времену (као и њихови крајњи рокови) се деле по последицама које настају ако систем не успе да изврши операцију пре истека очекиваног крајњег рока:

- тврди (тешке последице) - неуспех да се операција изврши пре крајњег рока доводи до тоталног колапса система,
- меки (лакше последице) - корисност резултата опада ако се испоручи после крајњег рока, самим тим обарајући и квалитет услуге.

Самим тим циљ тврдих система за рад у реалном времену је да обезбеди да се испуне сви крајњи рокови. За меке системе за рад у реалном времену циљ је да се постигне одређени подскуп крајњих рокова да би се оптимизовао неки критеријум специфичан за апликацију. Критеријум који се оптимизује конкретно зависи од апликације. Неки од чешћих случајева су ситуације када се максимизује број крајњих рокова који треба да се постигну, или кад се минимизује кашњење операција или кад се максимизује количина операција који треба да се изврше пре својих крајњих рокова.

Тврди системи за рад у реалном времену се користе када је неопходно да се гарантује да ће одређени случај да се догоди у оквиру строгог крајњег рока. Гаранција је потребна код система код којих долази до великог губитка ако се не реагује у оквиру временског рока, поготово код случајева када се може доћи до физичких оштећења или опасности по живот. Систем за контролу нукларне централе је један пример тврдог система за рад у реалном времену зато што у случају да сигнал закасни може доћи до огромних катастрофа.

Меки системи за рад у реалном времену се углавном користе да се реши питање конкурентног приступа и потребе да се повезани системи који се мењају ажурирају тако да су сви упознати са насталим променама у другим системима. Софтвер који одржава и ажурира времена полетања и слетања авиона је један пример меког система за рад у реалном времену. Времена слетања и полетања треба да буду тачна у неком разумном оквиру за крајње кориснике, али могу да функционишу са кашњењем од више секунди. Софтвери који преносе аудио-видео сигнал уживо су у већини случајева још један пример меких система за рад у реалном времену. У случају да дође до кршења рокова резултат је губитак квалитета преноса аудио-видео сигнала, али и поред тога систем може да настави са радом и може да се опорави у будућности, користећи предикцију и методологије поновног конфигураисања [14].

2.3. Веб технологије

Апликација је написана коришћењем савремених веб технологија. Клијентска страна написана је у библиотеци *React* са складиштем *Redux*, док је серверска страна написана у радном оквиру *Node.js Express*.

2.3.1. О библиотеци *React*

React (познат и као *React.js* или *ReactJS*) је библиотека *JavaScript-a* која се користи за креирање корисничких интерфејса [15]. Свака страна је састављена од *React* компоненти, а кретање кроз странице је могуће захваљујући делу библиотеке за рутирање. Библиотека *React* користи се за једностраничне (енг. *single-page*) или за мобилне апликације (*React Native*). Апликације направљене у овом програмском оквиру углавном захтевају коришћење додатних библиотека за управљање глобалним стањем (*Redux*), рутирање и интеракцију са АПИ-јем [16]. *React* је креирао Џордан Волк, софтверски инжињер компаније *Facebook*, угледајући се на *XHP - HTML* компоненту оквира за *PHP* програмски језик.

Најважније карактеристике библиотеке *React* су коришћење виртуелног објектног модела докумената и коришћење JSX језика (JavaScript XML).

DOM (енг. *Document Object Model)* је објектни модел докумената и представља АПИ којим се *JavaScript* програмима омогућава да комуницирају са репрезентацијом ових докумената и елемената у меморији прегледача. Једна од карактеристика библиотеке *React* је коришћење виртуалног *DOM.* Библиотека *React* креира кеш структуре података у меморији, израчунава актуелне разлике, а затим ефикасно ажурира прегледачев *DOM* [17]. Ово омогућава програмерима да пишу кôд као да се цела страна освежава при свакој промени, док библиотека *React* приказује само подкомпоненте које су заиста промењене.

На пример, компоненте корпе за куповину могу бити написане тако да прикажу целокупан садржај корпе при било којој промени података. Уколико подкомпоненте производа немају промене у подацима, биће употребљен кеширани приказ. Ово значи да ће релативно споре пуне измене у прегледачевом *DOM*-у бити избегнуте. Поред тога, уколико се промени број производа, подкомпоненте производа ће бити приказане; крајњи приказ се може разликовати у само једном члану и само тај члан ће бити измењен у DOM-у.

JSX је синтаксно проширење *JavaScript-a* које омогућава лако навођење *HTML*-а и коришћење *HTML* ознака. Компоненте библиотеке *React* су обично записане у JSX-у. JSX се користи са библиотеком *React* да бисмо описали како кориснички интерфејс треба да изгледа. JSX подсећа на шаблонски језик, али за разлику од њих долази са свим JavaScript функционалностима. На примеру 1. је приказана *JavaScript* класа која наслеђује *React* класу *React.Component*. React метода *render()* коришћена у овом примеру је једина нужна метода *React* компоненте. *React* користи враћену вредност те методе како би закључио шта треба приказати на екрану апликације. У овом примеру креирања компоненти резултат *render()* методе не изгледа као традиционални *JavaScript* - тај део кода је написан помоћу JSX-а.

```
var React = require('react');
var ReactDom = require('react-dom');
class App extends React.Component {
    render() {
        return (
            <div>
                Hello World
            </div>
        )
    }
}
```

Пример 1. JSX

Код библиотеке *React* логика превођења је уско везана за логику корисничког интерфјеса: како се обрађују догађаји, како се стање мења током времена, и како се подаци припремају за приказивање.

Уместо вештачког одвајања технологија, тако што се *HTML* ознака и логика стављају у одвојене датотеке, библиотека *React* користи благо везане јединице које се називају компоненте и које садрже и *HTML* ознаке и логику. Библиотека *React* не подразумева коришћење *JSX*, али је већина људи користи *JSX* због лакшег визуелног рада са корисиничким интерфејсом унутар *JavaScript* кода.

Компоненте не могу директно да промене било које својство које им је прослеђено, али им може бити прослеђена функција са повратним позивом да промени вредности. Овај механизам је изражен као "својства иду доле; акције горе".

На пример, компонентна корпа за куповину може да укључује више компоненти производа. Визуелизација производа користи само својства која су му прослеђена и то не може да утиче на укупну вредност колица за куповину. Међутим, производ може бити прослеђен функцији као аргумент која ће се позвати ако се активира дугме "обриши овај производ" и тада та функција утиче на укупан рачун.

2.3.2. *Redux*

Redux је библиотека *JavaScript-a* отвореног кода за управљање стањем апликације [18]. Најчешће се користи са оквирима као што су *React* и *Angular* за изградњу корисничког интерфејса. Библиотека *Redux* је инспирисана архитектуром *Flux* компаније *Facebook*, и креирали су је Ден Абрамов и Андреј Кларк.

Redux је мала библиотека са једноставним и ограниченим АПИ-јем дизајнираним да буде предвидљив контејнер за стање апликације. Функционише на сличан начин као функција редуковања (концепт функционалног програмирања). Настала је под утицајем функционалног језика *Elm* [19].

Библиотека *Redux* доноси следеће предности апликацијама:

1) Предвидљивост исхода - увек постоји једно глобано стање, без конфузије о томе како да се синхронизује тренутно стање са акцијама и другим деловима апликације.
2) Одрживост - имајући предвидљив исход и строгу структуру, кôд се лакше одржава.
3) Организација - библиотека *Redux* је строжија око тога како треба организовати кôд што чини кодове конзистентнијим и лакшим за рад са тимом.
4) Приказивање са сервера - библиотека *Redux* омогућава приказивање података који долазе са сервера.
5) Алатке за развој - програмери могу пратити све што се дешава у апликацији у реалном времену, од активности до промена стања
6) Заједница и екосистем - ово је велики плус када се учи или користи било која библиотека или оквир. Имати заједницу иза библиотеке *Redux*-а чини је још привлачнијом за коришћење.
7) Лако тестирање - прво правило писања кода који се лако тестира је писање малих функција које раде само једно и независно. *Redux*-ов кôд су углавном функције које су управо мале, чисте и изоловане.

2.3.3. Платформа *Node.js*

Node.js је вишеплатформско *JavaScript* радно окружење отвореног кода за извршавање *JavaScript* кода на серверској страни. Историјски гледано *JavaScript* је примарно коришћен на клијентској страни, где су скрипте написане у *JavaScript* коду биле уграђиване у *HTML* странице, како би се извршиле на клијентској страни у веб прегледачу. Платформа *Node.js* омогућава да се *JavaScript* користи за скрипте на серверској страни које омогућавају да се садржај динамичних веб страница генерише на серверу пре него што се пошаље до веб прегледача корисника. Због тога је *Node.js* постао један од основних елемената парадигме "*JavaScript свуда*" [20], јер омогућава униформисање развоја веб апликација у једном програмском језику, без потребе да се за скрипте на серверској страни користи други програмски језик.

Иако је *.js* конвенционална екстензија за *JavaScript* кôд, назив "*Node.js*" се не односи ни на једну датотеку у овом контексту и само представља назив производа. Платформа *Node.js* поседује архитектуру базирану на догађајима која је способна да обавља асинхроне улазе/излазе. Овакав избор архитектуре омогућава оптимизацију пропусности и скалабилности у веб апликацијама са много улазно-излазних операција, као и за веб апликације у реалном времену [21].

Node.js дистрибуирани пројекат води *Node.js* задужбина [22], уз подршку Линукс Фондације и њиховог програма колаборативних пројеката [23]. Неке од компанија које користе *Node.js* су: *GoDaddy, Groupon, IBM, LinkedIn*, *Microsoft*, *Netflix, PayPal, Rakuten, SAP, Tuenti, Voxer, Wallmart, Jaxy* и *Cisco*.

Express.js је најчешће коришћен серверски оквир за *Node.js* [26]. *Express.js* је дизајниран за израду једностраничих, вишестраничних и хибридних веб апликација [24]. Аутор

оригиналног програма *TJ Holowaychuk* је био инспирисан софтверским сервером *Sinatra*, што значи да је релативно мали са много могућности за надоградњу. *Express.*js је позадински део *MEAN* скупа технологија, заједно са *Mongo DB* системом за рад са базама података и *Angular.js* оквиром.

2.3.4. RabbitMQ

RabbitMQ је софтвер отвореног типа за управљање порукама. Концепт рада *RabbitMQ* је сличан пошти: када ставимо писмо у поштанско сандуче можемо бити сигурни да ће га поштар однети до примаоца. На сличан начин, *RabbitMQ* је поштанско сандуче, пошта и поштар. Главна разлика између *RabbitMQ* и поште је коришћење бинарних БЛОБ података (порука) за прихватање, чување и слање уместо папира.

Слање порука назива се производња. Програм који шаље поруке је произвођач. Ред је име за поштанску кутију која се налази унутар *RabbitMQ*. Иако поруке пролазе кроз *RabbitMQ* и апликацију, оне се могу чувати само у реду. Ред је ограничен на радну меморију и величину диска корисника, у суштини је то велики бафер за поруке. Многи произвођачи могу слати поруке преко једног истог реда и многи потрошачи могу покушати да примају податке из једног истог реда.

Потрошња представља пријем поруке. Потрошач је програм који углавном чека да прими поруке. Треба имати на уму да произвођач, потрошач и ред (бафер порука) не морају да се налазе на истом домаћину; заправо у већини апликација они се и не налазе.

На слици 3, "*ПР*" је произвођач, док је "*ПО*" потрошач. Кутија у средини је ред - бафер порука коју *RabbitMQ* може да чува у име потрошача [25].

Слика 3. *Слање порука брокера RabbitMQ*

2.3.5. *Redis*

Redis (енг. *Remote Dictionary Server*) је складиште података у меморији, користи се као база података, за кеширање или као боркер порука. О њеној растућој популарности довољно говори то да се *Redis* скоро увек користи као једно од решења када се ради реинжењеринг сајтова са великим посетама [27]. Данас се може срести у апликацијама као што су *Instagram, Pinterest, Tumblr, StackOverflow, Disqus, Guardian, Github, Blizzard* и многим

другима. Потпуно је бесплатна, написана у језику *ANSI C* у свега двадесетак хиљада редова. Написао ју је италијан Салваторе Санфилипо. Подржава главни-подређени (енг. *Master-slave*) репликацију, а оно чему је аутор посветио пуну пажњу у новој верзији је кластеровање и подршка извршењу *Lua* скрипти над *Redis* сервером.

Главне предности базе *Redis*:

- брзина извршавања,
- изутно је лагана за учење (једноставна конфигурација и покретање),
- активан развој (свакодневно се развија и надограђује),
- омогућава главни-подређени репликацију што доприноси скалабилности,
- атомичне операције.

Redis не треба схватити као обично кључ/вредност (енг. key/value) складиште података као што је *Memcache*. *Redis* јесте то али је и значајно више – то је заправо сервер структура података. Структура података је начин на који се податак записује у рачунар. Разумевајући структуре у *Redis-у*, како раде, које команде су над њима могуће и који подаци се могу чувати у њима, кључно је за разумевање самог *Redis-а*. *Redis* је сервер следећих структура података:

- Стрингови (енг. *Strings*) – представљају основне структуре кључ/вредност приступа. Кључеви могу садржати било шта од података величине до 512 МБ. Дакле кључ или вредност може да буде нпр. садржај неког ЈПГ (енг. *JPG*) фајла, ЈСОН (енг. *JSON*) објекат, обичан текст, и друго.
- Скупови (енг. *Sets*) – представљају неуређене колекције јединствених елемената. Дакле *Redis* не дозвољава понављање елемената у скуповима што значи да програмери не морају да воде рачуна о томе да ли неки елемент већ постоји у скупу или не – *Redis* то одрађује уместо њих. Програмери позивају основне команде за додавање, избацивање, пресек, унију и разлику елемената у скуповима.
- Сортирани скупови (енг. *Sorted sets*) – представљају уређене колекције јединствених елемената. Максималан број елемената је укупно 2^64 - 1 елемената по скупу. Подржане су команде над скуповима као и код обичних скупова и уз то, омогућене су команде помоћу којих се лако долази до података као што су првих десет елемената по резултату, ранг елемента на основу резултата, елементи који имају резултат између одређених вредности и друго.
- *Redis* хешеви (енг. *Redis Hashes*) – представљају заправо асоцијативне низове. Дакле *Redis* хешеви су хешеви који представљају мапе имена поља и њихових текстуалних вредности. Хеш се користи када је потребна структура слична табелама у базама података. Потребно је нагласити да хешеви не заузимају пуно места тако да се милиони објеката могу сачувати на релативно малом простору. Основне операције убацивања у хеш и читања из хеша су константне сложености. Као и скупови, хешеви су имплементирани као речници. Речници у *Redis-у* су имплементирани као хеш табеле користећи хеш функцију *MurmurHash2* и расту инкрементално. Колизија се решава уланчавањем. За мање хешеве користи се посебна врста структуре података *ziplist*. У суштини то је посебно кодирана двоструко повезана листа која је оптимизована за уштеду времена. *Ziplist* се такође користи за оптимизацију мањих сортираних скупова и листи. Хеш када је спљоштен на такву

листу изгледа као [кључ1, вредност1, кључ2, вредност2, ...]. Хешеви са неколико кључева могу бити ефикасно упаковани у линеарни низ са гарантованом константном сложеношћу за убацивање и читање. Очигледно је да се ово не може одржати са повећавањем хеша. Како расте хеш, он се конвертује у стандардну структуру података речник како би се задржала константа сложеност операција убацивања и читања из хеша али се губи уштеда простора.

- Листе (енг. *Lists*) – представљају листе стрингова. Елементи су сортирани по редоследу убацивања у листу. У суштини листе се могу посматрати као низови. Елементи се могу додавати на почетак (главу) и на крај (реп) листе. Максималан број елемената је укупно 2^64 - 1 елемената по листи. Елементи се убацују у листу у константном времену, односно исто је времена потребно када се убацује елемент у листу од 10 елемената и у листу од 10 милиона елемената. Листе врло брзо враћају резултате као што су првих *n* елемената са почетка (главе), односно са краја (репа), док су спорије ако покушате да приступите средини веома велике листе јер је та операција линеарне сложености [26].

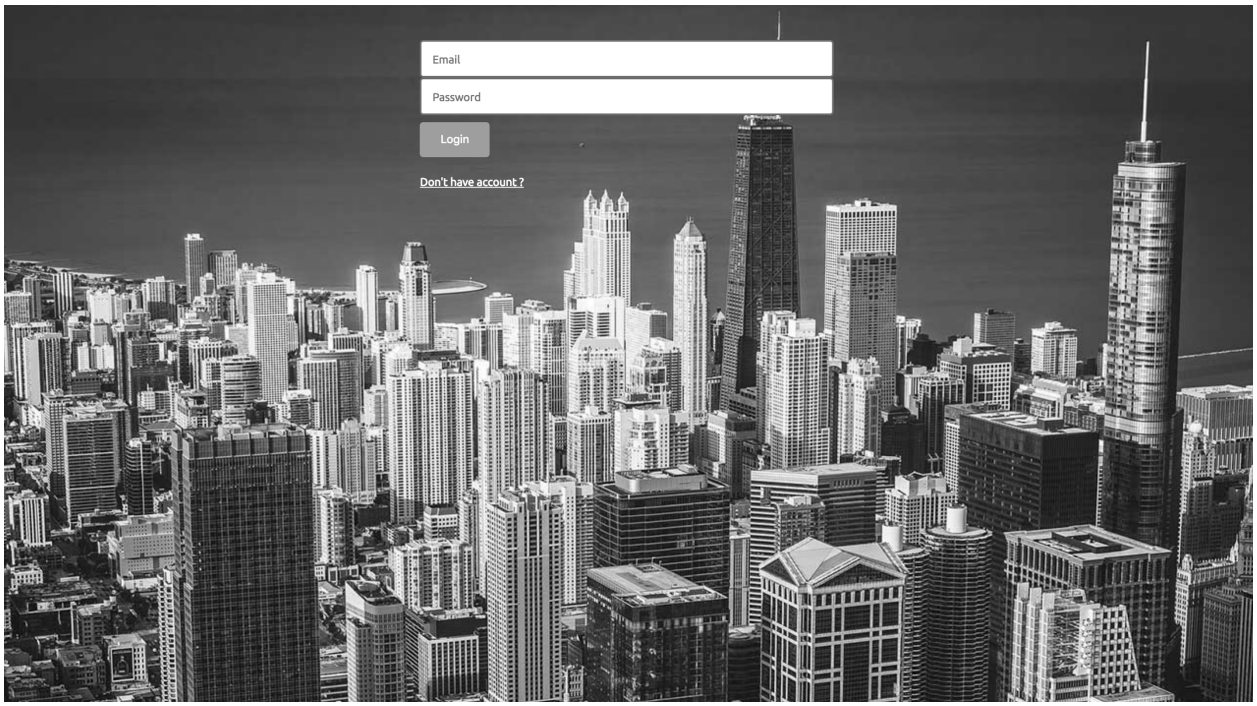
3. Архитектура апликације

3.1. Опис апликације

Циљ апликације је приказивање рута аутономних такси-возила на Гугл мапи тачним редоследом којим су се остваривале, приказивање броја путника који су транспортовани, укупно растојање пређено од стране аутономних такси возила у метрима и просечна пређено растојање у метрима. Апликације је отвореног кода и доступна је на *GitHub* репозиторијуму на адреси https://github.com/veljkomatic/master_rad. Сама апликација се састоји од клијентске стране написане у *React-у* са *Redux-ом,* серверске стране која је израђена у стилу микросервиса у технологији *Node.js* са оквиром *Express*. Апликација користи податке о вожњама аутономних такси-возила у Њујорку. Подаци су саставни део скупа података *NYC Open Data* који садржи све потребне информације о почетку и крају вожње, географским кординатама почетне и крајње тачке на основу којих се могу пратити и графички приказати руте, броју путника итд. Апликација користи те податке и омогућава поновно сагледавање свих остварених вожњи.

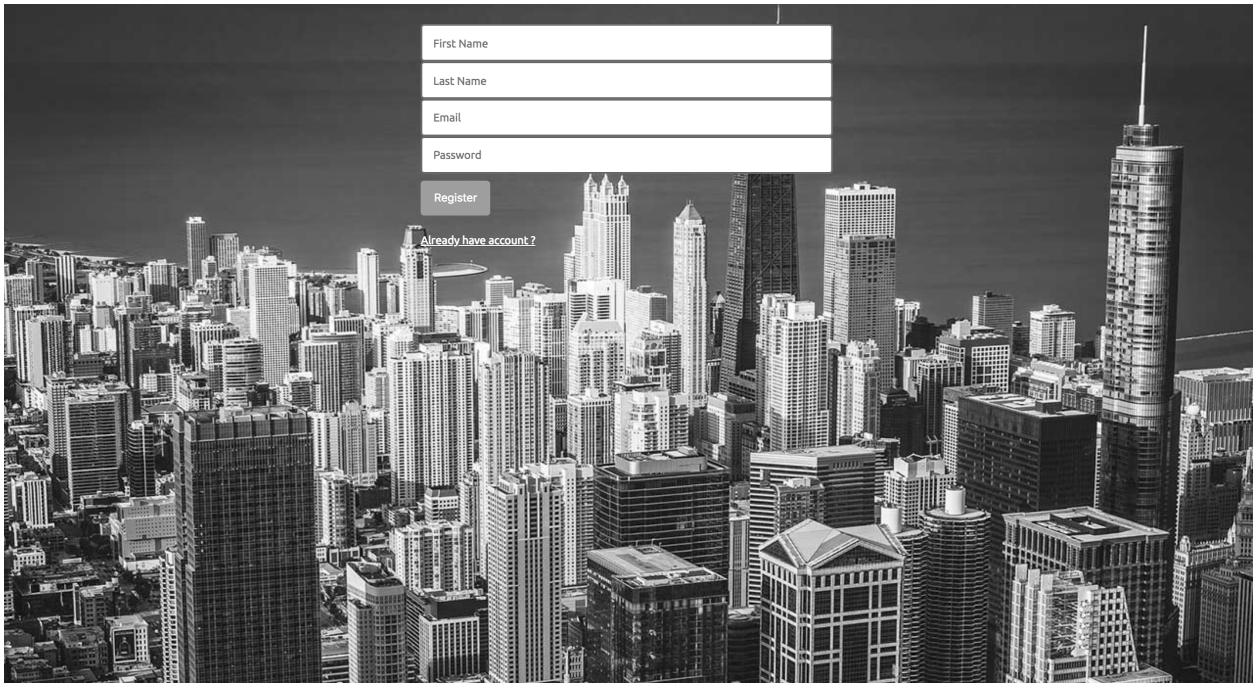
3.2. Клијентска страна апликације

Клијентска страна се састоји од три стране: страна за пријављивање, страна за регистрацију и страна са Гугл мапом и горе наведеним метрикама.
Страна за пријављивање се састоји од два поља за унос корисникове *e-mail* адресе и шифре корисника, дугмета за пријављивање и линка ка страници за регистрацију у случају да корисник нема налог (слика 5).



Слика 5. *Страница за пријављивање*

Страна за регистрацију се састоји од поља за унос имена, презимена, корисникове *e-mail* адресе и шифре корисника, линка ка страници за пријављивање у случају да корисник има налог и дугмета за регистрацију (слика 6).



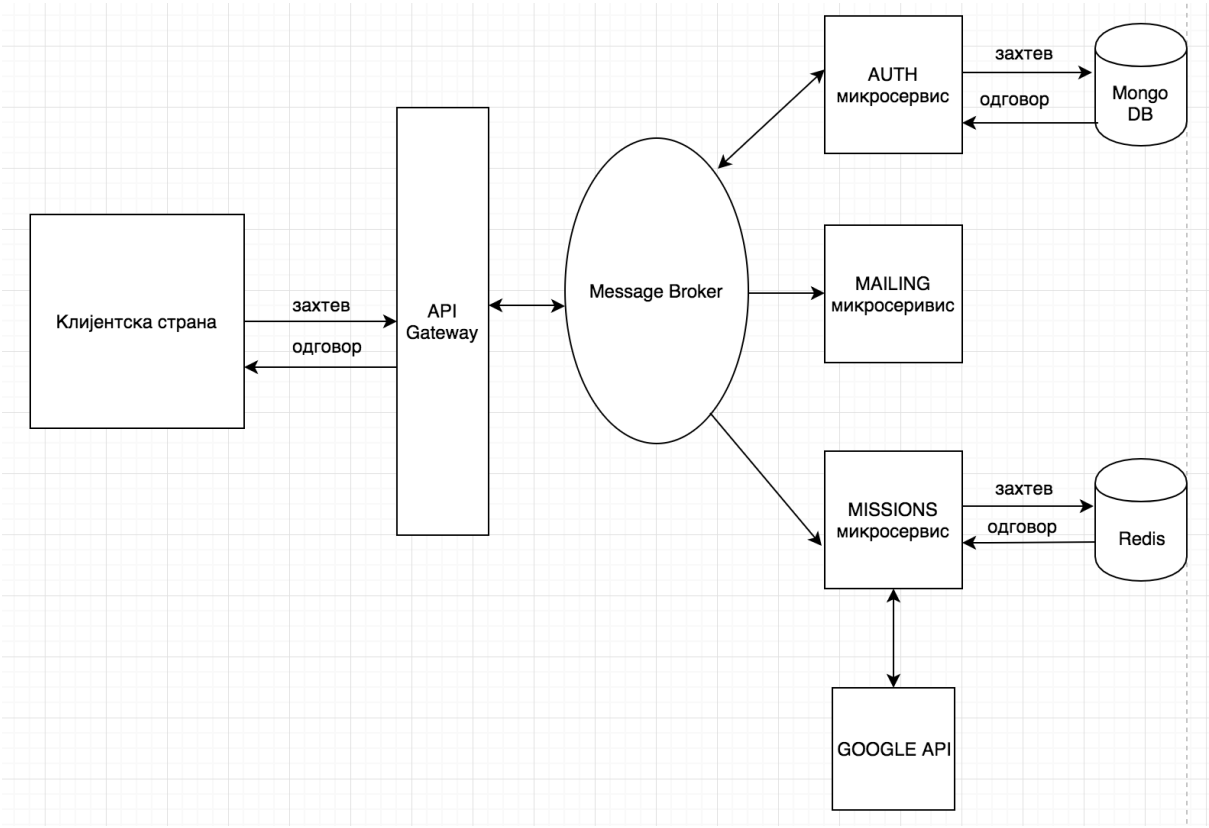
Слика 6. *Страна за регистрацију*

Након успешне регистрације или пријављивања корисник се навигира на страницу за приказивање рута аутономних такси-возила у реалном времену. Руте се исцртавају на Гугл мапи (слика 7).

Слика 7. *Руте аутономних такси-возила са метрикама*

3.3. Серверска страна апликације

Серверска страна апликације урађена је у стилу микросервиса. Архитектура серверске стране састоји се од *API Gateway*-а и три микросервиса: микросервис *Auth*, микросервис *Mailing*, микросервис *Missions*. Микросервис *Auth* се бави аутентикацином корисника, микросервис *Mailing* се бави слањем електронске поште и микросервис *Missions* се бави проналаском вожња аутономних такси возила које почињу и које завршавају у реалном времену. Комуникација се одвија тако што клијентска страна шаље захтев серверској страни, тај захтев долази до *API Gateway*-а и даље се пропагира брокеру порука на који су миркосервиси претплаћени (слика 4).



Слика 4. *Комуникација између клијентске стране и серверске стране*

4. Имплементација

У наставку рада биће описана имплементација како клијентске тако и серверске стране апликације.

4.1. Страна за пријављивање

Након уноса корисникове *e-mail* адресе и шифре корисника, кликом на дугме за пријављивање подаци се шаљу на сервер. АПИ клијент коришћен за слање података на сервер је библиотека *Axios*. Пошто је библиотека *React* базиран на компонентама, за интеракцију са АПИ-јем је коришћен *Redux*. Кликом на дугме за пријављивање окида се *Redux* акција. *Redux* акција је омогућена помоћу специјалне библитеке *redux-thunk*. *Redux-thunk* је средњи слој *Redux*-а који нам дозвољава да пишемо акције креирања које враћају функцију уместо акције. *Redux-thunk* се може користити за одлагање раније акције, или отпреме акције ако су испуњени посебни услови. Унутрашња функција прима методе главног стања *dispatch* и *getState* као параметре. Следи кôд акције за пријављивање корисника на апликацију.

```
export const loginUser = (email, password) => async (dispatch) => {
    dispatch({
        type: actionTypes.LOGIN_USER
```

```
    });
    const response = await Network.post({
        url: 'login',
        data: {
           email,
           password
        }
    });
    if (response.error) {
        return dispatch({
           type: actionTypes.LOGIN_USER_FAIL,
           error: response.error
       });
    }
    localStorage.setItem('authToken', response.token || '');
    localStorage.setItem('refreshToken', response.refreshToken || '');
    dispatch({
        type: actionTypes.LOGIN_USER_SUCCESS,
        payload: response.user
    });
   return {
        result: true
   };
};
```

Позивањем методе главног стања *dispatch* са параметрима као што је тип акције, подаци se враћају са сервера ако је позив успешан, или је дошло до грешке. У супротном зове се други део *Redux-a*, а то је *Reducer*. *Reducer* се састоји од иницијалног стања и логике мењања стања у случају одређене акције. Кôд испод приказује *Reducer* који обрађује акције које су се десиле.

```
import { actionTypes } from '../actionCreators/types';
const INITIAL_STATE = {
    user: {},
    error: '',
    loading: true
};

export default (state = INITIAL_STATE, action) => {
    switch (action.type) {
    case actionTypes.LOGIN_USER:
         return {
                ...state,
               error: '',
               loading: true
          };
    case actionTypes.LOGIN_USER_SUCCESS:
               return {
               ...state,
```

```
            error: '',
            loading: false,
            user: action.payload
        };
    case actionTypes.LOGIN_USER_FAIL:
        return {
              ...INITIAL_STATE,
              loading: false,
              error: action.error
        };
    case actionTypes.REGISTER_USER:
              return {
            ...state,
            loading: true,
            error: ''
        };
    case actionTypes.REGISTER_USER_SUCCESS:
        return {
            ...state,
            error: '',
            loading: false,
            user: action.payload
        };
    case actionTypes.REGISTER_USER_FAIL:
        return {
            ...INITIAL_STATE,
            loading: false,
            error: action.error
        };
    case actionTypes.LOGOUT_USER:
        return {
            ...INITIAL_STATE,
            loading: true
        };
    case actionTypes.LOGOUT_USER_SUCCESS:
        return {
            ...INITIAL_STATE,
            loading: false
        };
    case actionTypes.LOGOUT_USER_FAIL:
        return {
            ...INITIAL_STATE,
            loading: false
        };
    default:
        return state;
    }
};
```

У случају успешног корака за пријављивање корисник прелази на страницу са Гугл мапом.

4.2. Страна за регистрацију

Кликом на дугме *Register* окида се *Redux* акција за регистрацију корисника која у зависности од успешности мења иницијално стање у *Redux Reducer-y.* Ако је акција била успешна корисник иде на страницу са Гугл мапом.

4.3. Страна за приказивање рута

За примање података у реалном времену искоришћена је библиотека за сокете *socket.io*. Пре исцртавања компоненте позива се *Redux* акција која отвара сокет комуникацију са сервером. Кôд испод приказује акцију subscribeToMissions која користи сокет комуникацију.

```
import { actionTypes } from './types';
import openSocket from 'socket.io-client';
const socket = openSocket('http://localhost:3100');

export const subscribeToMissions = () => dispatch => {
    socket.emit('missionsSubscribe', localStorage.getItem('authToken'));

    socket.on('startingMission', (startingMission) => {
        dispatch({ type: actionTypes.NEW_STARTING_MISSION, payload:
startingMission });
    });
    socket.on('finishedMission', (finishedMission) => {
        dispatch({ type: actionTypes.FINISHED_MISSION, payload:
finishedMission });
    });
}
```

Емитује се догађај *missionsSubscribe* са токеном за аутентикацију, након тога се чекају догађаји *startingMission* и *finishedMission*. Када се деси неки од ових догађаја позива се метода продавнице *dispatch* са типом акције која одговара догађају који се десио, као и подацима враћеним са сервера.

4.4. *API Gateway*

API Gateway се налази између клијентске стране и серверске стране и представља део са којим комуницирају све клијентске стране ако их има више. Проблем који *API Gateway* решава је како клијентска страна приступа индивидуалном микросервису. *API Gateway* обрађује захтеве на један од два начина. Неки захтеви се једноставно рутирају ка једном миркосервису, док се неки распоређују ка више микросервиса. Комуникација са микросервиса се врши или преко *Rest* позива или брокера порука као што је на пример *RabbitMQ* ili *Kafka* или чак преко *RPC-a* (енг. *Remote Procedure Call*) као на пример *SOAP*.

У овој апликацији за комуникацију са микросервисима користи се брокер порука и то *RabbitMQ. API Gateway* апликација се састоји од два АПИ слоја који комуницира са брокером порука и апликативним слојем који врши проверу токена. На захтев за пријављивање

корисника који стиже са клијентске стране формира се порука о захтеву у АПИ слоју која се преко брокера порука прослеђује микросервисима. Када се захтев обради од стране једног или више микросервиса, одговор се шаље клијенту. Кôд који следи представља обраду захтева за пријављивање.

```
router.post(LOGIN.url, async (req, res, next) => {
  try {
        logger.info(`Send to ${LOGIN.name} queue`);
        await channel.assertQueue(LOGIN.name);
        await channel.sendToQueue(LOGIN.name,
                Buffer.from(JSON.stringify(req.body), 'utf8'));
        await channel.assertQueue(LOGIN.consume);
        channel.consume(LOGIN.consume, (msg) => {
                logger.info(`Consume ${LOGIN.consume}`);
                const result = JSON.parse(msg.content);
                logger.info(
                `{${LOGIN.consume} Result: ${JSON.stringify(result)}`
                );
                if(result.error) {
                   const { error } = result;
                   return res.status(error.httpStatus).send({ error
                     error.message
                    });
                }
                res.send(result);
        });
      } catch(e) {
         next(e);
      }
})
```

Захтев за регистрацију корисника послат од стране клијента обавештава микросервисе путем брокера порука да се захтев за регистрацију догодио. Када се захтев обради одговор се шаље клијенту. Кôд који следи представља обраду захтева за регистрацију.

```
router.post(REGISTER.url, async (req, res, next) => {
        try {
            logger.info(`Send to ${REGISTER.name} queue`);
            await channel.assertQueue(REGISTER.name);
            await channel.sendToQueue(REGISTER.name,
               Buffer.from(JSON.stringify(req.body), 'utf8'));
            await channel.assertQueue(REGISTER.consume);
            channel.consume(REGISTER.consume, (msg) => {
                logger.info(`Consume ${REGISTER.consume}`);
                const result = JSON.parse(msg.content);
                logger.info(`${REGISTER.consume} Result:
${JSON.stringify(result)}`);
                if(result.error) {
```

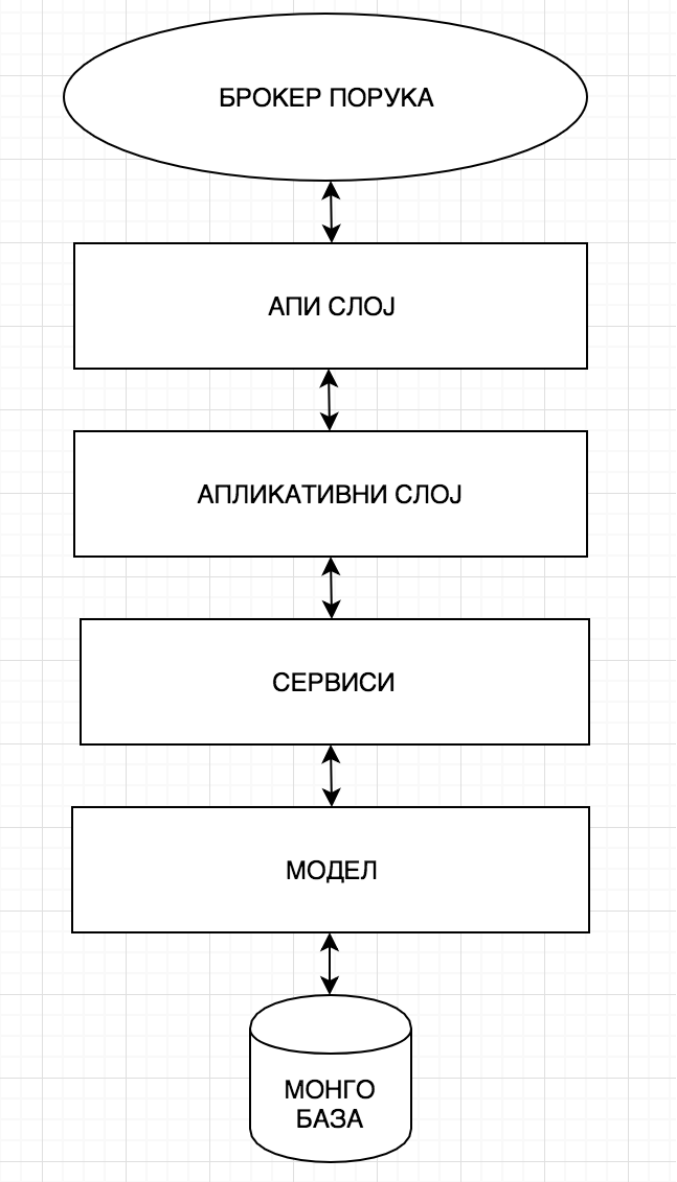
```
                    const { error } = result;
                    return res.status(error.httpStatus).send({ error:
error.message });
                }
                res.send(result);
        });
        } catch(e) {
            next(e);
        }
});
```

Комункација клијентске стране са серверском страном врши се преко сокета у реалном времену. Клијентска страна емитује догађај *missionsSubscribe* са токеном за аутентикацију АПИ слој у *API Gateway* ослушкује овај догађај и када се деси позива се апликативни слој и провера токена за аутентикацију. Уколико токен није исправан клијенту се враћа порука да токен није валидан са статус кодом 401. Уколико је токен валидан обавештавају се микросервиси преко брокера порука да се догађај *missionsSubscribe* десио. Како пристижу обрађени захтеви о вожњама тако се они путем сокет конекције шаљу клијенту. Кôд који следи представља сокет конекцију са клијентом и обраду захтева који преко ње стижу.

```
module.exports = ({ io, channel }) => {
   io.on('connection', async (socket) => {
       await channel.assertQueue('missionsSubscribe');
       await channel.assertQueue('finishedMission');
       await channel.assertQueue('startingMission');
       logger.info('Finished assertQueues: missionsSubscribe,
finishedMission, startingMission');
       socket.on('missionsSubscribe', async (token) => {
           try {
               authentication.validUser(token);
               logger.info('Send to queue missionsSubscribe');
               await channel.sendToQueue('missionsSubscribe');
               channel.consume('startingMission', (msg) => {
                   logger.info('Consume startingMission');
                   socket.emit('startingMission', JSON.parse(msg.content))
               });
               channel.consume('finishedMission', (msg) => {
                   logger.info('Consume finishedMission');
                   socket.emit('finishedMission', JSON.parse(msg.content))
               });
           } catch(e) {
               logger.error('Missions Controller API Gateway, Error: ', e);
           }
       });
   });
};
```

4.5. Микросервис *Auth*

Микросервис *Auth* комуницира са брокером порука и обрађује захтеве везане за аутентикацију, а то је пријављивање корисника и регистрација новог корисника. Такође комуницира и са микросервисом *Mailing* преко брокера порука. Архитектура микросервис *Auth* (слика 8) се састоји од АПИ слоја (контролера) у коме се врши обрада порука из брокера порука, апликативног слоја који санитизира захтев из брокера порука, такође и санитизирање одговора који се шаље брокеру порука. Санитизирање захтева представља проверу формата захтева, и проверу заглаља и тела захтева, док санитизирање одговора служи за избацивање непотребних података из тела одговора. Сервиси служе за обраду захтева, позивање модела и креирање токена за аутентикацију и *refresh* токена. Аутентикација прати *OAuth 2.0* стандард. *OAuth 2.0* подразумева креирање токена за аутентикацију и *refresh* токена. Токен за аутентикацију истиче за 72 сата, након чега токен постаје неисправан. *Refresh* токен служи за поновно креирање исправног токена за аутентикацију. *Refresh* токен обично нема време истицања. У модел слоју се креира *mongoose* модел података са корисничком схемом. *Mongoose* је оквир *JavaScript-a* који се обично користи у *Node.js* апликацијама са *MongoDB* базом података. *Mongoose* дефинише објекте са строго типизираном схемом која је мапирана на *MongoDB* документ. *Mongoose* пружа додатне функционалности око креирања и рада са схемама.



Слика 8. - Архитектура микросервиса *Auth*

АПИ слој комуницира са брокером порука и у њему се врши обрада захтева за пријављивање и захтева за регистрацију корисника, као и слањем одговора брокеру порука. Поред тога постоји и опција *health* која позива *health* сервис и враћа "здравље" микросервиса. Следи кôд који приказује обраду захтева за пријављивање у АПИ слоју и слање одговора.

```
channel.consume('login', async (msg) => {
        await channel.assertQueue('loginResponse');
        try {
            const ctx = sanatize.login(JSON.parse(msg.content));
            const login = await authService.loginPost(ctx);
            return channel.sendToQueue('loginResponse',
Buffer.from(JSON.stringify(login), 'utf8'));
        } catch(e) {
            const error = errorMap(e);
```

```
            return channel.sendToQueue('loginResponse', Buffer.from(
                JSON.stringify({ error }), 'utf8'));
        }
    });
```

Следи кôд који приказује обраду захтева за регистрацију и слање одговора након обраде.

```
 channel.consume('register', async (msg) => {
        await channel.assertQueue('registerResponse');
        await channel.assertQueue('sendEmail');
        try {
            const ctx = sanatize.register(JSON.parse(msg.content));
            const registered = await authService.registerPost(ctx);
            await channel.sendToQueue('sendEmail',
Buffer.from(JSON.stringify({
                userEmail: registered.user.email,
                subject: 'Welcome',
                fileName: 'welcome_email',
                data: {
                    user: `${registered.user.firstName}
${registered.user.lastName}`
                }
            }), 'utf8'));
            return channel.sendToQueue('registerResponse',
Buffer.from(JSON.stringify(registered), 'utf8'));
        } catch(e) {
            const error = errorMap(e);
            return channel.sendToQueue('registerResponse',
Buffer.from(JSON.stringify({ error }), 'utf8'));
        }
    });
```

АПИ слој позива апликативни слој који санитизира захтев за пријављивање или регистрацију корисника, такође врши санитизирање одговора који се шаље брокеру порука. Кôд који следи приказује санитизирање захтева и одговора у апликативном слоју.

```
module.exports = {
    login: (data) => {
        const { email, password } = data;
        checkEmail(email);
        checkPassword(password);
        return {
            email: email && email.toLowerCase(),
            password
        };
    },
register: (data) => {
    const {
            firstName, lastName, email, password
     } = data;
```

```
    checkEmail(email);
    checkPassword(password);
        checkFirstName(firstName);
        checkLastName(lastName);
     return {
         firstName,
         lastName,
         email: email && email.toLowerCase(),
         password
      };
    },
sanitizeUser: (user) => {
     return {
         id: user._id,
         email: user.email,
         firstName: user.firstName,
         lastName: user.lastName
      };
  }
};
```

У сервисима се врши креирање новог корисника, комуникација са слојем модела података ради чувања новог корисника као и креирање токена. За креирање токена за аутентикацију користи се библиотека *jsonwebtoken*. Приликом обраде захтева за пријављивање креира се поред токена за аутентикацију и *refresh* токен. Кôд који следи служи за креирање токена за аутентикацију и *refresh* токена.

```
const generateAuthToken = async (user) => {
        const token = jwt.sign({ _id: user._id }, keys.secret, {
        expiresIn: 60 * 60 * 72 // expires in 72 hours
        });
        const refreshToken = randtoken.uid(256);
        user.services.refreshToken = refreshToken;
        const updatedUser = await user.save();
        const sanitizedUser = sanitize.sanitizeUser(updatedUser);
        return  {
                token,
                refreshToken,
                user: sanitizedUser
        }
};
```

Обрада захтева за пријављивање врши се у сервису *loginPost* тако што се провери да ли корисник са унетом *e-mail* адресом постоји испод слоја модела података као и да ли је лозинка исправна. Кôд који следи представља комуникацију са слојем модела података.

```
loginPost: async (ctx) => {
     const { email, password } = ctx;
     const hashedPassword = sha256(password);
```

```
     const user = await User.findOne({ email });
     if (!user) {
          throw errorCode.USER_NOT_EXISTS;
      }
      const difference = await bcrypt.compare(hashedPassword,
user.services.password.bcrypt);
      if (!difference) {
          throw errorCode.NOT_ALLOWED
      }
      return generateAuthToken(user);
}
```

Обрада захтева за регистрацију корисника се врши у сервису *registerPost* тако што се превери да ли постоји корисник са унетом *e-mail* адресом, уколико не постоји формира се објекат корисника и шаље се модел слоју који комуницира са *Mongo* базом. Након уноса корисника у базу шаље се одговор брокеру порука који обрађује *API Gateway*. Кôд који следи представља креирање новог корисника као и креирање токена за аутентикацију и *refresh* токена.

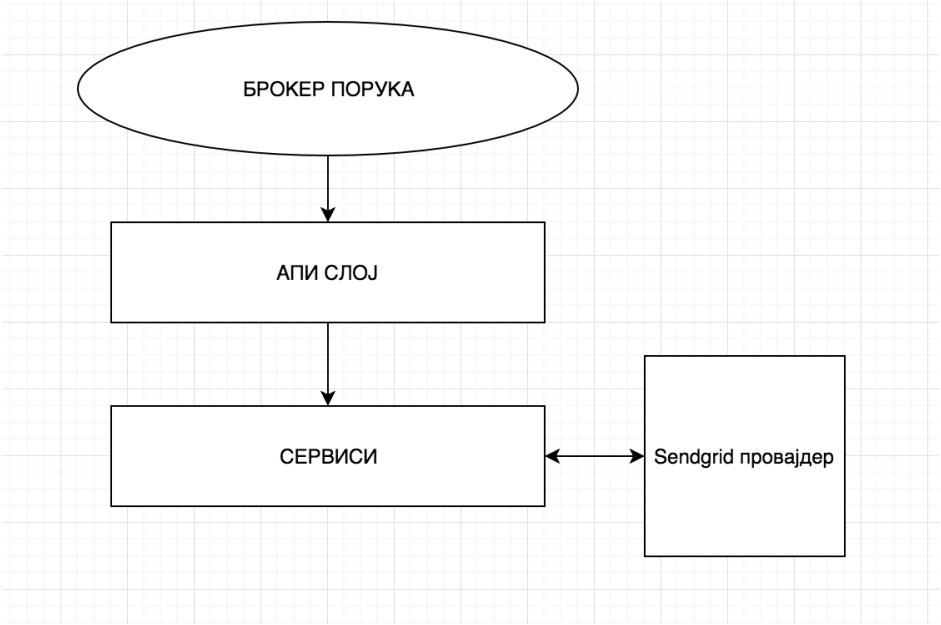
```
registerPost: async (ctx) => {
        const { email, firstName, lastName, password } = ctx;
        const oldUser = await User.findOne({ email });
        if (oldUser) {
            throw errorCode.REGISTER_USER_EXISTS;
        }
        const hashedPassword = bcrypt.hashSync(sha256(password), 10);
        const userToSave = new User({
           email,
           firstName,
           lastName,
           services: {
              password: {
                bcrypt: hashedPassword
              }
           }
        });
        const newUser = await userToSave.save();
        return generateAuthToken(newUser);
}
```

У слоју модела података се креира *User mongoose* модел података са корисничком схемом и представља конекцију са *Mongo* базом података.

4.6. Микросервис *Mailing*

Микросервис *Mailing* служи за слање било какве врсте порука. У апликацији се шаљу само *e-mail* поруке преко провајдера *Sendgrid*. *Sendgrid* је провајдер услуга за размену порука. *Sendgrid* нуди високо оптимизовани сервер који се може користити за слање e-mail порука.

Микросервис *Mailing* се састоји од АПИ слоја и сервиса који комуницира са провајдером *Sendgrid* (слика 9).



Слика 9. *Архитектура микросервиса Mailing*

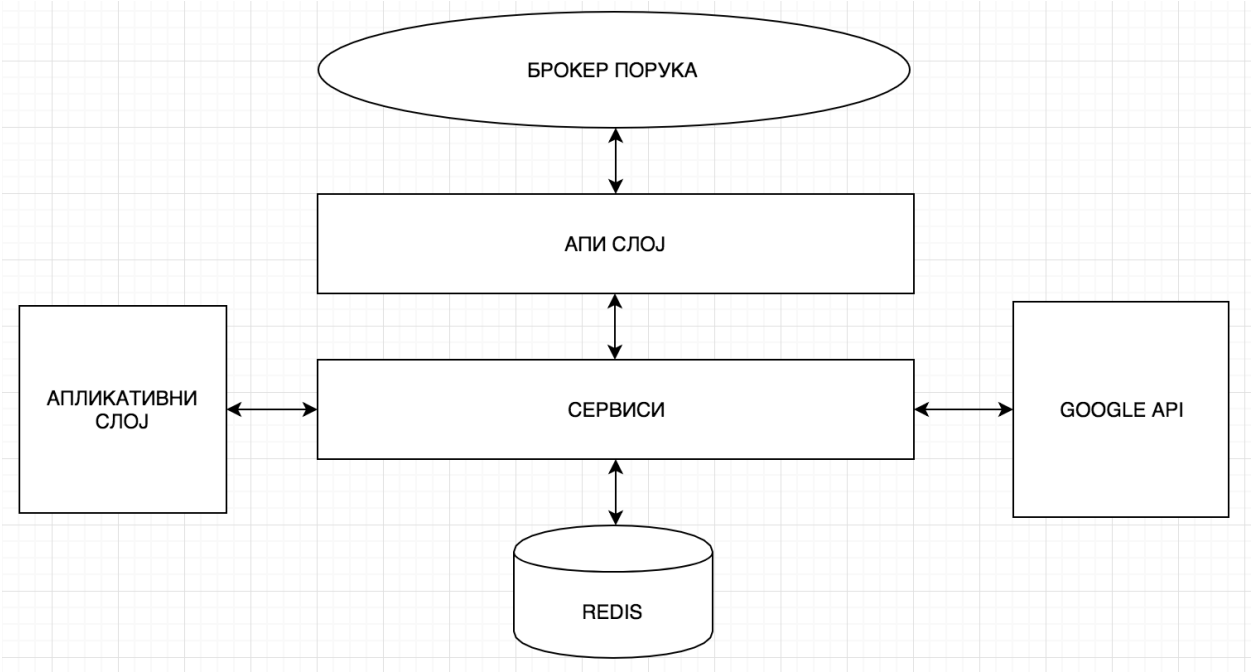
АПИ слој комуницира као и код микросервиса *Auth* са брокером порука и позива сервис који комуницира са провајдером *Sendgrid*. Сервис *sendEmailToUser* прима као параметре *e-mail* адресу, наслов поруке, име фајла, податке и формат фајла чија је подразумевана вредност *.ejs*. Фајл се учитава са диска и приказује се у *html* формату са одређеним подацима, и онда се заједно са насловом поруке и *e-mail-ом* апликације шаље на *e-mail* адресу корисника приликом регистрације. Кôд који следи представља имплементацију *sendEmailToUser* сервиса који комуницира са *Sendgrid* провајдером.

```
const ejs = require('ejs');
const fs = require('fs');
const path = require('path');
const sgMail = require('@sendgrid/mail');
const logger = require('winston');

const keys = require('../config/keys');
sgMail.setApiKey(keys.SENDGRID_API_KEY);

module.exports = {
        sendEmailToUser: (userEmail, subject, fileName, data, extension =
'.ejs') => {
         let renderedHtml;
         return new Promise(async (resolve, reject) => {
            fs.readFile(path.join(__dirname, '/emails/',
`${fileName}${extension}`), 'utf8', async (error, email) => {
         if (error) {
             return reject(error);
         }
         renderedHtml = ejs.render(email, data);
```

```
          const msg = {
               to: userEmail,
               from: keys.email,
               subject: subject,
               html: renderedHtml,
             };
             try {
                 await sgMail.send(msg);
             } catch(e) {
                  logger.error('Error sendEmailToUser - Mailing microservice:
', e);
             }
             resolve()
           });
                });
     }
};
```

4.7. Микросервис *Missions*

Микросервис *Missions* се бави проналаском вожњи аутономних такси возила које треба да почну у реалном времену. Подаци који се користе у овом микросервису су део скупа података *NYC OpenData* из 2016. године. Подаци су сортирани по времену почетка вожње да бисмо их обрађивали оним редом којим се се вожње остваривале и смештени су у базу података *Redis*. Архитектура микросервиса *Missions* се састоји од АПИ слоја, апликативног слоја и сервиса (слика 10).



Слика 10. *Архитектура микросеривса Missions*

АПИ слој као и код других микросервиса комуницира са брокером порука и обрађује захтев *missionsSubscribe*. Након обраде захтева *missionsSubscribe* поставља се интервал који сваке секунде позива сервисе *findStartingMissons* и *findFinishedMissions*. Следи кôд који приказује обраду захтева *missionsSubscribe.*

```
channel.consume('missionsSubscribe', async (msg) => {
     try {
          await channel.assertQueue('finishedMission');
          await channe l.assertQueue('startingMission');
          setInterval(() => {
               missionsServices.findStartingMissons({ channel });
               missionsServices.findFinishedMissions({ channel });
          }, 1000)
     } catch(e) {
          logger.error('Error missions microservice: ', e);
     }
});
```

Апликативни слој обавља санитизирање одговора који се шаље брокеру проука. Сервиси проналазе вожње које почињу и које су се завршиле. Сервис *findStartingMissons* учитава податке о вожњама из *Redis* базе, увећава време за једну секунду и позива функцију која проналази вожње које почињу и провера јесу ли исправне. Проверава се да ли је место поласка исто као и место доласка. Уколико су ови услови испуњени вожња није исправна. Кôд испод приказује функцију која проналази вожње које почињу.

```
const getStartMissions = (greenTaxiTripData) => {
    const startMissionsData = [];
    for(let el of greenTaxiTripData) {
          const pickupDate = new Date(el.lpep_pickup_datetime);
          if(pickupDate.getTime() > date.getTime()) {
             break;
           }
          if((pickupDate.getTime() === date.getTime()) &&
validation.validateMission(el)) {
          startMissionsData.push(el);
           }
   }
   return startMissionsData;
}
```

Уколико су пронађене исправне вожње које почињу, пролази се кроз сваку од њих и позива се сервис који зове Гугл АПИ да би добио податке за исцртавање рута. Да би се послали подаци на клијентску страну који могу да се исцртавају потребно је мапирати податке које је вратио Гугл АПИ. Проблем који настаје овде је тај што Гугл АПИ не враћа координате за исцртавање рута него кодирани *polyline*. *Polyline* представља низ објеката који садрже географску дужину и географску ширину. Пошто је нама Гугл АПИ вратио кодирани *polyline*, он даље треба да се декодира помоћу библитеке *decode-google-map-polyline* да би се добио низ објеката који представљају стварне кординате рута (географску дужину и географску ширину). Када је све завршено позива се апликативни слој који санитизира

податке везане за вожњу и резултат шаље брокеру порука. Кôд испод представља имплементацију сервиса findStartingMissons.

```
findStartingMissons: async ({ channel }) => {
       logger.info('Missions Services - Find Starting Mission And Remove It
From Data');
       const data = await getAsync('greenTaxiTripData');
       const greenTaxiTripData = JSON.parse(data);
       date.setSeconds(date.getSeconds() + 1);
       const newStartingMissions = getStartMissions(greenTaxiTripData);
       if(newStartingMissions.length > 0) {
           logger.info(`Found ${newStartingMissions.length} new starting
missions`)
           const newData =
greenTaxiTripData.slice(newStartingMissions.length);
           newStartingMissions.forEach(async (el) => {
               const response = await
externalServices.GoogleGetRoute(el.Pickup_latitude, el.Pickup_longitude,
el.Dropoff_latitude, el.Dropoff_longitude);
               const googleRoute = mapper.mapGoogleRoute(response);
               const startingMission = Object.assign(
                   {},
                   el,
                   {
                       id: uniqid(),
                       computeRoutes: googleRoute
                   }
               );
               logger.info('Sending to queue startingMission');
               await channel.sendToQueue('startingMission',
Buffer.from(JSON.stringify(sanitize.sanitizeMissionResponse(startingMission
)), 'utf8'));
           });
           client.set('activeMissions',
JSON.stringify(newStartingMissions), redis.print);
           client.set('greenTaxiTripData', JSON.stringify(newData),
redis.print);
       }
   }
```

Након проналаска свих вожњи које почињу, ажурирају се подаци у *Redis* бази везани за све преостале вожње, као и за нове активне вожње. Сервис *findFinishedMissions* проналази све вожње које су завршене. Учитавају се све активне вожње из *Redis* базе, и проналазе се оне које су се завршиле позивањем функције *getFinishedMissions*. Уколико има таквих вожњи пролази се кроз њих, позива се апликативни слој који санитизује податке везане за вожњу и резултат шаље брокеру порука. Након проласка кроз завршене вожње, ажурирају се активне вожње у *Redis* бази. Кôд који следи приказује функцију *getFinishedMissions* која проналази завршене вожње.

```
const getFinishedMissions = (activeMissions) => {
    const finishedMissionsData = [];
    for(let el of activeMissions) {
        const dropoffDate = new Date(el.Lpep_dropoff_datetime);
        if(dropoffDate.getTime() > date.getTime()) {
           break;
        }
        if(dropoffDate.getTime() < date.getTime()) {
           finishedMissionsData.push(el);
        }
    }
    return finishedMissionsData;
}
```

Кôд који следи представља имплементацију сервис *findFinishedMissions* у микросервису *Missions.*

```
findFinishedMissions: async ({ channel }) => {
       logger.info('Missions Services - Find Finished Mission And Remove It
From Active Missions');
       const data = await getAsync('activeMissions');
       const activeMissions = JSON.parse(data);
       const finishedMissions = getFinishedMissions(activeMissions);
       if(finishedMissions.length > 0) {
           logger.info(`Found ${finishedMissions.length} finished
missions`)
           const newData = activeMissions.slice(finishedMissions.length);
           finishedMissions.forEach(async (el) => {
               logger.info('Sending to queue finishedMission');
               await channel.sendToQueue('finishedMission',
Buffer.from(JSON.stringify(sanitize.sanitizeMissionResponse(el)), 'utf8'));
           });
           client.set('activeMissions', JSON.stringify(newData),
redis.print);
       }
   }
```

Поред *findStartingMissons* и *findFinishedMissions* имплементиран је и *GoogleGetRoute* сервис који се позива у *findStartingMissons* сервису. *GoogleGetRoute* сервис комуницира са *Google Api*-јем преко *directions* методе *Google Map* клијента. Да би се искористили добијени подаци и исцртале руте на клијентској страни морају да се мапирају подаци, декодира *polyline* и да се добију координате за руте. Кôд који следи представља имплементацију сервиса *GoogleGetRoute.*

```
const googleMap = require('@google/maps');
const logger = require('winston');

const keys = require('../config/keys');
```

```
const googleMapClient = googleMap.createClient({
   key: keys.googleApiKey,
   Promise: Promise
});

module.exports = {
   GoogleGetRoute: async (originLat, originLong, destinationLat,
destinationLong) => {
       try {
           logger.info(`Google API - Get Directions For Origin:
${originLat},${originLong}, Destination:
${destinationLat},${destinationLong}`);
           const response = await googleMapClient.directions({
               origin: `${originLat},${originLong}`,
               destination: `${destinationLat},${destinationLong}`,
               mode: 'driving',
           }).asPromise();
           return response.json;
       } catch(e) {
           logger.info(`GoogleGetRoute ERROR: ${JSON.stringify(e)}`);
       }
   }
};
```

Након што су послате поруке о започетим и завршеним вожњама брокеру порука, *API Gateway* обрађује те поруке и шаље их преко сокет везе на клијент.

5. Закључак и даљи развој

Апликација у стилу микросервиса за праћење аутономних такси-возила у реалном времену има за циљ могућност поновног сагледавања вожњи аутономних такси-возила са циљем прикупљања података о успешно завршеној вожњи, исправним рутама, невалидним подацима о вожњама, и друго.
Други циљ апликације је да се прикаже и архитектура серверске стране која је урађена у стилу микросервиса. Микросервиси комуницирају путем брокера порука што је познато као архитектура вођена догађајима (енг. event-driven architecture), и написани су тако да се понашају као црна кутија. За брокер порука коришћена је библиотека *RabbitMQ*. Квалитет серверске стране огледа се у увођењу *API Gateway*-а који је део са којим комуницира клијентска страна и који шаље захтеве брокеру порука, и који бивају конзумирани од стране неког од микросервиса задужених за обраду тог захтева. *RabbitMQ* се може конфигурисати да користи *SSL* као додатни безбедносни слој у брокеру порука. По потреби може и да се скалира и до преко пола милиона порука у секунди. Мана коришћења библиотеке *RabbitMQ* је зависност од *Erlang-a* па је потребна и минимална конфигурација.

Приликом имплементације тежило се томе да микросервиси на серверској страни буду имплементирана са што већом конхерентношћу и да буду лабаво везани. Треба још напоменути да је клијентска страна имплементирана коришћењем најсавременијих технологија, библиотеке *React* и *Redux* са циљем да се прикаже њихово коришћење и организација кода.

Предности коршћења *Node.js* и *React* технологија је у коришћењу *JavaScript* програмског језика и за клијент и за сервер. *Node.js* користи асинхрони неблокирајући И / О модел који помаже у конкурентној обради захтева. *Node.js* је једнонитно окружење. Ово се често сматра озбиљним недостатком технологије. У неким случајевима, задатак везан за процесор (разне калкулације) може блокирати петљу догађаја која резултира секундама кашњења за све кориснике *Node.js* веб сервера.

Даљи развој апликације има у виду унапређивање микросервисне архитектуре. Она ће се унапредити убацивањем алата за континуирану интеграцију (енг. *continuous integration)* у сваки микросервис тако да се сваки микросервис засебно испоручује. Такође, план је и повећање покривености кода тестовима у сваком микросервису и прављење глобалног сервиса у коме ће се приказивати сви логови из микросервиса и ”здравље” микросервиса. Још један од начина унапређивања архитектуре микросервиса је смањена функционалност апликације у случају отказивања неког од микросервиса.

Референце

1. KPMG - Autonomous Vehicles Readiness Index
2. Newman, Sam (2015), “Building Microservices”, O’Reilly
3. Evans, Eric (2004). Domain-Driven Design: Tackling Complexity in the Heart of Software
4. Chen, Lianping (2015). "Continuous Delivery: Huge Benefits, but Challenges Too"
5. Turban E., King D., Lee J., Viehland D. (2008). "19". Electronic Commerce A Managerial Perspective (5th ed.)
6. https://www.ibm.com/cloud/learn/virtualization-a-complete-guide
7. Paul Duvall (2012), “Infrastructure Automation”, IBM
8. Laudon Kenneth Craig, Traver Carol Guercio (2008). E-commerce: Business, Technology, Society
9. Rod Stephens (2 March 2015). Beginning Software Engineering
10. Martin, Robert C. (2003). Agile Software Development, Principles, Patterns, and Practices
11. Margaret Rouse. 2018. What is a real time application (RTA)
12. M. Ben-Ari, "Principles of Concurrent and Distributed Programming
13. Krishna Kant (May 2010). Computer-Based Industrial Control
14. Martin, James (1965). Programming Real-time Computer Systems
15. https://thenewstack.io/javascripts-history-and-how-it-led-to-reactjs/
16. https://medium.com/codecademy-engineering/react-router-to-redux-first-router-2fea05c4c2b7
17. https://reactjs.org/docs/refs-and-the-dom.html
18. https://redux.js.org/
19. "Redux – An Introduction". *Smashing Magazine*.
20. https://www.ibm.com/developerworks/community/blogs/gcuomo/entry/javascript_everywhere_and_the_three_amigos?lang=en
21. Laurent Orsini (2013-11-07). "What You Need To Know About Node.js"
22. https://foundation.nodejs.org/
23. https://www.linuxfoundation.org/projects/
24. „Express.js”
25. Case study: How & why to build a consumer app with Node.js. VentureBeat.com.
26. https://www.rabbitmq.com
27. https://redis.io/