УНИВЕРЗИТЕТ У БЕОГРАДУ
МАТЕМАТИЧКИ ФАКУЛТЕТ

Урош Познан

РАЗВОЈ СКАЛАБИЛНИХ ВЕБ АПЛИКАЦИЈА УПОТРЕБОМ ТЕХНОЛОГИЈА AMQP И DOCKER

мастер рад

Београд, 2023.

Ментор:

др Александар Картељ, доцент
Универзитет у Београду, Математички факултет

Чланови комисије:

др Владимир Филиповић, редовни професор
Универзитет у Београду, Математички факултет

др Иван Чукић, доцент
Универзитет у Београду, Математички факултет

Датум одбране:

Садржај

Глава 1

Увод и мотивација

Када говоримо о развоју веба и веб апликација, не може се изоставити концепт скалабилности. Његове карактеристике и изазове које са собом носи и повлачи, при развоју софтвера за веб. Такође, када говоримо о модерном развоју веб апликација и њиховој сврси, морамо имати увид у то како и зашто се користе веб апликације. Наиме, свака веб апликација има неку улогу и служи за то да сваком свом кориснику пружи одређену врсту услуге (енгл. *service*) – или му пружа одређену информацију или му користи као алат у обављању неког посла. Самим тим, при самом развоју, неопходно је одабрати технологије које ће на најбољи начин одговорити захтевима система и тиме пружити максималну ефикасност и удобност корисницима.

Циљ рада је приказ поменутих технологија, анализа њиховог утицаја на брзину и скалабилност веб апликација. У оквиру мастер рада, биће развијена веб апликација у развојном оквиру *Spring* програмског језика *Java* уз коришћење *RabbitMQ*-а и *Docker*-а која за циљ има демонстрацију употребе поменутих технологија и њихових карактеристика. Рад ће приказати коришћење апликације у реалном дистрибуираном окружењу кроз примену у купопродаји улазница за музичке и друге догађаје. Најпре ћемо се упознати са појмовима који фигуришу у наслову овог рада и оних које ћемо најчешће спомињати, а након тога ћемо се упознати са технологијама.

Појмови:

- **Веб апликација** је софтвер који се извршава на веб серверу, а корисник јој приступа помоћу веб прегледача, за разлику од *Desktop* апликација које су морале бити инсталиране локално на машини како би могле да се користе.

- **Скалабилност** је способност система да поднесе велики пораст обима послова, без угрожавања функционалности и поузданости система.
- **(Микро)сервис** је компонента софтверског система која може „самостално" да пружа одређене целовите услуге.

1.1 Скалабилност и веб апликације

Другим речима, можда уопштеније, скалабилност представља могућност раста, а да се задрже сва основна својства, особине и фукције онога о чему се говори.

Ниво скалабилности

Ниво скалабилности система одређује се на основу:

1. *Количине административно-оперативних послова, неопходних ради прилагођавања порасту обима послова.*
   - Скалабилан – ако је довољно повезати нове уређаје (сервере, процесоре, дискове), без додатних административних послова.
   - Умерено скалабилан – ако је потребно предузимати одређене промене конфигурације, као што је мануелна репликација софтвера и/или података.
   - Није скалабилан – ако је потребно мењати архитектуру система.
2. *Количине развојних послова, потребних ради одржавања функционалности система при повећаном оптерећењу.*
   - Скалабилан колико и његова најмање скалабилна компонента.
   - Високо скалабилан – ако нису потребне никакве развојне активности ради прилагођавања повишеном обиму послова тј. архитектура је добро прилагођена високом обиму послова.
   - Умерено скалабилан – ако су потребне само мање модификације.
   - Слабо скалабилан – ако је потребно мењати архитектуру софтвера.

3. *Очекиваног трајања периода неактивности система током прилагођавања високом нивоу оптерећења.*

   - Ниска скалабилност – ако је за додавање нових уређаја (процесора, дискова, сервера) потребно искључити систем.
   - Умерена скалабилност – ако је при додавању нових уређаја потребно да се на кратко (неколико секунди или минута) искључи систем, док се изврши преконфигурисање мреже.
   - Висока скалабилност – ако се нови уређаји и софтверске компоненте могу додавати без искључивања система.

Мерење скалабилности

Трајање извршавања посла је време за које n уређаја (процесора, чворова) изврши посао – у ознаци *T(n)*. Релативно убрзање система је мера смањивања трајања извршавања послова при повећавању рачунарске снаге у ознаци $S(n) = T(1)/T(n)$. Пропусност система је максималан број послова одређене сложености који се могу извршити у јединици времена. Пропусност система са n уређаја – у ознаци $X_{max}(n)$. Скалирање (релативни капацитет) је мера повећања броја послова који могу да се изврше у јединици времена – у ознаци $C(n) = X_{max}(n)/X_{max}(1)$

Категорије скалабилности према релативном капацитету:

1. *Линеарна скалабилност* – релативни капацитет је једнак броју уређаја $C(n) = n$; достижна када се послови добро паралелизују, на пример, велики број малих захтева који су међусобно независни.

2. *Суб-линеарна скалабилност* – релативни капацитет је мањи од броја уређаја $C(n) < n$; када паралелизација захтева додатне ресурсе, комуникацију међу процесима, дељиве ресурсе.

3. *Супер-линеарна скалабилност* – релативни капацитет је већи од броја уређаја $C(n) > n$. Наизглед немогућа, али у неким случајевима достижна и реална. Додавање посматраног уређаја често додаје и друге ресурсе, на пример, када се додаје сервер, додаје се и меморија.

Разликујемо вертикално и хоризонтално скалирање.

Вертикално скалирање

Остварује се додавањем нових ресурса постојећим чворовима система – на пример, додавање процесора, дискова или меморије постојећим чворовима. Очекивано је веће трајање периода неактивности, нижи ниво административних послова (скоро никакав), архитектура је једноставнија и ефикаснија. Преко неке границе постаје далеко скупље решење од хоризонталног скалирања.

Има смисла до неких граница и за одређене проблеме. Купити јачи рачунар, често је једноставније и јефтиније него развијати нови софтвер. Централизоване базе података су много лаке за пројектовање, имплементирање и одржавање.

Хоризонтално скалирање

Остварује се додавањем нових чворова система – на пример, додавање нових сервера, или нови систем за складиштење података. Очекивано трајање периода неактивности је краће, већи је обим и сложенији су административни послови, софтверска архитектура је сложенија. Ипак, после неке границе постаје далеко економичније решење од вертикалног скалирања, скоро да не постоји горња граница у хоризонталном скалирању. Архитектура софтвера је сложенија и виша је цена за мале конфигурације, али је веома исплативо за велике системе.

При развоју веб апликација, као и при развоју било којих других система, јасно је да се тежи да у сваком могућем сегменту постоји висока скалабилност. Међутим, то није увек могуће. У овом раду биће показано како избор технологија утиче на скалабилност, односно како технологије о којима се говори, позитивно или негативно утичу на скалабилност веб апликација. Идеално би било да сваки систем може бити високо скалабилан, међутим, од захтева система који је постављен зависи које ће се технологије и архитектура користити. Самим тим комплетна поставка и одређени избори који се праве морају бити одговарајући, како би се постигла максимална искористивост и перформантност система.

Важност скалабилности

У данашње време, када је више него икада „време једнако новац", људи желе информације одмах. Како би корисник био задовољан, за њега не сме да постоји чекање на веб страницу, форму или слику да се учита. Ако апликација није конципирана и израђена на адекватан начин, те не може да се носи са великим оптерећењем у виду повећаног броја корисника, убрзо неће ни имати кориснике. Скалабилност је фундаментална за сваку веб апликацију. Њено разумевање и уклапање концепата скалабилности је врло битна ствар у којој многе апликације оману, јер иста није имплементирана на добар начин.

1.2 Микросервисна архитектура

Микросервиси – механизам за креирање децентрализованог система и лаганих (енгл. *lightweight*) протокола који су модуларни у свом дизајну и који омогућују одржавање и надоградњу без угрожавања читаве апликације. Микросервисна архитектура је резултат еволуције софтверске индустрије са циљем постизања максималне флексибилности и проширивости. Модуларизацијом се целине апликације издвајају као посебне и такве целине се називају микросервиси. На нивоу архитектуре (а не технологије) постиже се висок ниво уопштавања и флексибилности, нпр. у случају промене начина комуникације, мења се само компонента за комуникацију. Микросервисвна архитектура је архитектурална парадигма која се примарно односи на дизајн софтвера, а не на технологију. Апликација тада представља композицију и груписање лабаво повезаних, уско дефинисаних и на стандардима утемељених микросервиса у веће и сложеније целине.

Сервиси су: повезиви, слабо спрегнути, дефинисани интерфејсима независним од имплементације, корисник сервиса зависи само од интерфејса, промовише се флексибилност при изменама имплементације, међусобно сарађују (интероперабилни), расположиви независно од имплементације или протокола преноса, корисници и пружаоци сервиса могу бити имплементирани на различитим платформама, комуникациони протоколи сервиса треба да буду компатибилни са различитим платформама.

Микросервиси представљају концепт апликације. Они омогућују да се унутрашњост једне апликације разбије на мање делове који могу независно један од другог да функционишу, да буду мењани, скалирани и администрирани.

Овакав тип архитектуре је настао, а и стицао све већу и већу популарност са напретком виртуализације, рачунарства у облаку (енгл. *cloud computing*), агилног развоја и *DevOps*-а.

Највише предности микросервиса у овом контексту произилази из слабе спрегнутости компонената, која доноси следеће:

- Продуктивност и агилност програмера – микросервиси омогућују програмерима да имплементирају нове технологије у само једном делу апликације без утицаја на остале делове. Свака компонента може бити измењена, тестирана и испоручена независно од осталих, што знатно убрзава циклусе итерација.

- Скалабилност – микросервиси могу максимално да искористе скалабилност у облаку (енгл. *cloud scalability*); свака компонента може бити скалирана независно од других, тако да пружа максимално добар одговор изазову највећег радног оптерећења и то са најбољом могућом искористивошћу свих ресурса.

- Флексибилност – захваљујући раздвојености, квар једног микросервиса не утиче на рад других; сервис може да ради у складу са својом доступношћу без утицаја на друге компоненте апликације.

Микросервиси обично имају следеће карактеристике:

- Имају сопствену базу података и модел;

- Комуницирају међусобно путем REST API-ја, токовима догађаја (енгл. *event streaming*) или брокера порука (енгл. *message brokers*);

- Организовани су по способности обављања неког посла.

Предности најлакше могу бити уочене кроз следеће пословне и организационе бољитке које доносе:

- Лакша измена кода;

- Тимови могу да користе различите технологије за различите компоненте;

- Компоненте могу да буду скалиране независно и том приликом смањујемо цену потребе да се скалирају све компоненте једне апликације због једне функционалности која трпи велико оптерећење и потребно јој је скалирање.

Још један начин да се лакше схвати шта су микросервиси, је да се покаже шта они нису. Најлакше их је упоредити са монолитном архитектуром и са сервисно оријентисаном архитектуром.

Разлика између микросервисне и монолитне архитектуре је та да микросервиси чине јединствену апликацију из више мањих делова, док монолит представља једну велику уско спрегнуту апликацију. Зато је све чешће раздвајање некада монолитних апликација у више микросервиса.

Разлика између микросервиса и сервисно оријентисане архитектуре је мало мање очигледна, али најлакше ју је уочити у контектсу обима. Префикс „микро" у микросервисима односи се на грануларност интерних компонената, што значи да оне морају бити знатно мање од онога што SOA[1] обично буде. Компоненте услуга унутар микросервиса углавном имају једну сврху и ту једну ствар раде заиста добро. С друге стране, у SOA сервиси обично укључују много више пословних функционалности, а често се имплементирају као комплетни подсистеми. (Team, 2021)

1.3 AMQP

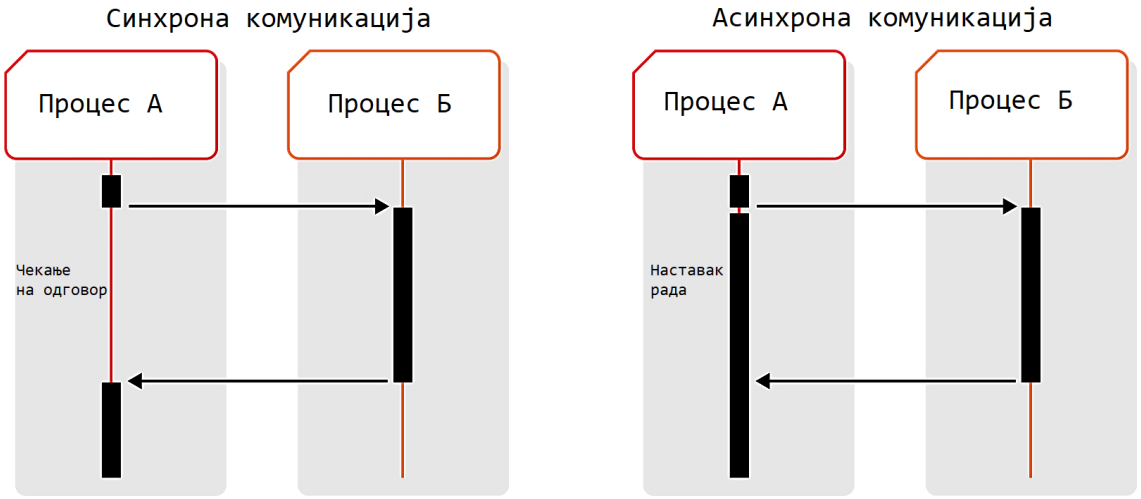
Један од два главна појма који фигуришу у наслову овог рада је AMQP. У претходном одељку када је писано о микросевисима, речено је да је комуникација између њих неопходна. Јер, да би апликација функционисала, микросервиси морају бити интероперабилни, самим ти непрестано морају бити у мрежној комуникацији и размењивати поруке. Ту на сцену ступа AMQP.

AMQP (*Advanced Message Queuing Protocol*) је отворени стандард интернет протокол слоја апликације који служи за размену порука. Као сваки систем за размену порука AMQP је протокол за размену порука који се састоји од пошиљаоца (енгл. *publisher*) и примаоца (енгл. *consumer*). Оно у чему је AMQP посебан је то што омогућава одређивање које ће се поруке примати

[1]SOA (енгл. *Service Oriented Architecture*) је предузетнички напор да се стандардизује начин на који сви сервиси комуницирају и како се интегришу један са другим, док је микросервисна архитектура специфична на нивоу апликације.

и одакле, те како се праве компромиси у погледу сигурности, поузданости и перформанси. Глобално признати стандард који у суштини ради на слоју апликације, користи се за развој комуникације између странака: клијента и брокера. Пошиљалац сноси одговорност за генерисање порука док их клијенти прикупљају и администрирају. Улога брокера (енгл. *broker*), као што је то *RabbitMQ*, у целом овом процесу је да осигурају да порука оде директно од пошиљаоца до примаоца. Његове кључне функционалности су рутирање, оријентација порука и стављање у ред/на чекање. Коришћење AMQP-а чини интероперабилност остварeним циљем, који прате другачија конфигурација и инфраструктура. Омогућава програмерима да искористе клијентску библиотеку и брокер, који су усаглашени са протоколом. (TechTarget, 2018)

Када говоримо о AMQP-у, говоримо о асинхроној комуникацији. Када пошиљалац пошаље поруку, он након тога добије потврду о слању те поруке (енгл. *ACKnowledgement*), међутим потврда не садржи ништа више од тога. Након што процес који је послао поруку добије потврду о слању, он наставља да обавља своје друге послове, чиме је конкретан део комуникације, односно слање поруке од пошиљаоца до примаоца, завршен. У зависности од имплементације и бизнис логике имамо случајеве када се очекује повратна порука или када се случај употребе завршава само једносмерним слањем. Међутим, у било ком од два случаја не постоје ресурси који активно чекају одговор и тако троше време. Уколико повратна порука стигне, за то постоји одређени ослушкивач који ће поруку примити и обрадити, независно од почетног дела за слање који је започео комуникацију. Дакле, код асинхроне комуникације, као што је то AMQP, пошиљалац не чека на одговор, након послате поруке. Такође, за синхрону комуникацију, неопходно је да микросервиси који комуницирају буду доступни у исто време, док код асинхроне то није случај. Упоредни однос синхроне и асинхроне комуникације види се на дијаграму секвенци на следећој слици.



Слика 1.1: Упоредни дијаграм асинхроне и синхроне комуникације

Уочавамо да асинхрона комуникација смањује чекање процеса А, јер користи то време за обављање других послова, уместо да чека одговор од процеса Б. Слика служи као илустрација мотивације за рад са AMQP-ом. Није нужно да овакав приступ увек доноси бољитак и да је REST као такав сам по себи лош (чак је и неопходан), већ се истиче да је, за архитектуру апликације о којој се говори, за одређене ствари, много боље користити AMQP. Он представља окосницу комуникације између микросервиса који су развијени у оквиру овог пројекта. Док је за комуникацију између клијента и сервера, наравно, предвиђен REST, јер је свакако и више него подобан за комуникацију између *fronted*-а и *backend*-а.

Упоредни однос AMQP-а и JMS-а

Обично се JMS (*Java Message Service*) користи када и клијент и сервер раде у JVM[2]-у. JMS је први API[3] за размену порука који је достигао широку примену у индустрији. JMS дефинише скуп интерфејса и семантику која омогућује размену порука *Java* апликацијама са другим имплементацијама. JMS је једноставан за коришћење, јер смањује скуп концепата које *Java* програмер треба да научи како би користио системе за размену порука, а у исто време повећава преносивост апликација за размену порука. (Mahmoud, 2004)

[2] *Java Virtual Machine* – *Java* виртуална машина.
[3] *Application Programming Interface* – програмски интерфејс апликације.

У ситуацији када клијент и сервер користе други језик или платформу, јавља се проблем. Тачније, JMS тада није од користи. Уобичајено решење је прелазак на други протокол, као што је AMQP. На поуздан и сигуран начин може ићи као P2P (peer-to-peer, један-на-један), објави/претплати се и друго. Дакле, главна разлика је у томе што је JMS API који имплементирају сервери који су искључиво компатибилни са *Java EE/Jakarta EE*. С друге стране, AMQP није API, већ бинарни протокол који је дизајниран за интероперабилност између различитих произвођача и платформи. Све док је апликација у складу са AMQP-ом, никакве промене на нивоу брокера нису потребне, чак и када се мења или уводи нова платформа у причу. Такође, може се користити и чисти JMS клијент за комуникацију са AMQP протоколом, ако га сервер за размену порука подржава. (Marchioni, 2022)

Предности оваквог приступа

AMQP је решење отвореног кода које омогућује сигурност, поузданост и интероперабилност.

Када се прича о микросервисима, као главна ставка наводи се слаба спрегнутост микросервиса. У прилог томе иде и коришћење AMQP-а. Како је раније поменуто, овакав вид комуникације омогућује да један сервис ради неометано, чак и ако му је други недоступан и да ће се информације које од једног до другог сервиса треба да стигну, пропагирати када други постане доступан. Такође, искористивост ресурса једног сервиса постаје већа, јер немамо процесе који чекају на одговор и тако троше време. Имамо умањено времена чекања.
AMQP доноси следеће:

- Омогућује једноставну P2P комуникацију;
- Шифроване и сигурне трансакције;
- Поруке бивају испоручене када дестинација постане доступна;
- Компатибилан је са свим популарним оперативним системима и програмским језицима;
- Помаже у контролисању оптерећења сервера, јер је број порука које стигну до сервера ограничен бројем процеса који скидају поруке и који се могу експлицитно ограничити.

1.4 RabbitMQ и његове специфичности

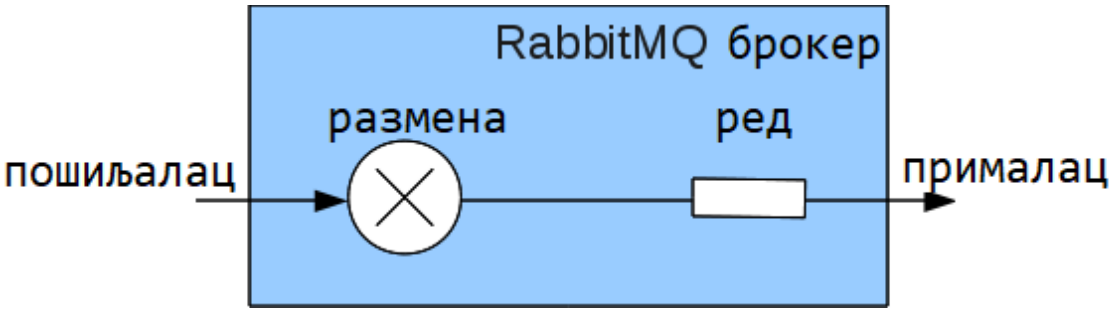
RabbitMQ је лаган, поуздан, скалабилан и преносив брокер. Међутим, за разлику од већине других брокера познатих *Java* програмерима, није заснован на JMS-у. Уместо тога, апликације комуницирају преко протокола који не зависи од платформе.

Размене, редови и везивања

Улога брокера је, као што смо већ поменули, да обезбеди да поруке сигурно стигну од пошиљаоца до правог примаоца. За то, *RabbitMQ*, користи размене и редове.

- **Ред** – редови (енгл. *queues*) су FIFO (*First In First Out*) структуре. Пошиљаоци креирају поруке и гурају их у ове редове, а из редова их примаоци преузимају једну по једну. Ред се понаша као привремено складиште (енгл. *buffer*) које чува поруке које се касније преузимају.
- **Размена** – да би ред примао поруке, мора бити везан за најмање једну размену (енгл. *exchange*).
- **Везивање** – везивање (енгл. *binding*) је релација између реда и размене која се састоји од скупа правила које размена користи да, између осталог, усмерава поруке у редове.
- **Поруке** – поруке (енгл. *messages*) представљају скуп података који се размењује кроз систем. Поруке могу да садрже својства (енгл. *properties*) и заглавља (енгл. *headers*) која могу да их категоришу.
- **Кључ за рутирање (енгл. *routing key*)** – једноставан атрибут поруке која се шаље и који се додаје у заглавље поруке приликом слања.

Као што показује *Слика 1.2.* пошиљалац шаље поруку на именовану размену, док прималац повлачи (енгл. *pull*) поруку са реда или ред гура (енгл. *push*) поруку примаоцу, што зависи од специфичности конфигурације. Првенствено неопходно их је повезати, а то радимо преко самог имена размене. Поруке се не објављују директно на ред, уместо тога, пошиљалац шаље поруке на размену. Размена је одговорна за рутирање порука у различите редове уз



Слика 1.2: Скица RabbitMQ брокера

помоћ везивања и кључева за рутирање. А везивање је, као што је поменуто, спона између реда и размене.
Типови размена:

- **Директна размена** (енгл. *direct exchange*) – испоручује поруке у редове на основу кључа за рутирање порука; порука ће бити испоручена оним редовима који имају кључ за повезивање (енгл. *binding key*) који се тачно поклапа са кључем за рутирање поруке. Уколико не постоји ред са кључем за повезивање који је идентичан кључу за рутирање који порука садржи, та порука се одбацује.

- **Предметна размена** (енгл. *topic exchange*) – рутира поруке на редове у зависности од кључа за рутирање и шаблона за рутирање, који је предодређен специфичним повезивањем. Поруке се прослеђују на један или више редова. Кључ за рутирање мора бити листа речи раздвојених тачком (нпр. *winter.concert.serbia*). Код креирање шаблона за рутирање постоје специјални карактери: * – идентификује било коју реч на специфичној позицији и # – идентификује 0 или више речи. Ове шаблоне креирамо нешто налик регуларним изразима. Примаоци наводе за које поруке су заинтересовани и претплаћују се на поруке које одговарају одређеном шаблону, те ће им управо одговарајуће поруке и бити испоручене. Пошиљалац креира ред, поставља повезивање задатим обрасцем и креира спону са разменом. Подразумевана размена коју AMQP брокери морају да креирају за предметну размену носи назив *„amq.topic”*.

- **Подразумевана размена** (енгл. *default exchange*) – унапред декларисана директна размена без имена, на коју обично упућује празна ниска.

Када се користи подразумевана размена, порука се испоручује у ред са именом једнаким кључу за рутирање поруке. Сваки ред је аутоматски везан за подразумевану размену са кључем за рутирање који је исти као име реда.

- **Размена издвајања** (енгл. *fanout exchange*) – размена копира и усмерава примљену поруку у све редове који су везани за њу без обзира на кључеве рутирања или подударање шаблона као код директне размене и предметне размене. Достављени кључеви ће једноставно бити игнорисани. Ове размене могу бити корисне када исту поруку треба послати на један или више редова са потрошачима који могу да обрађују исту поруку на различите начине. Подразумевана размена коју AMQP брокери морају да креирају за предметну размену носи назив *„amq.fanout”*.

- **Размена заглавља** (енгл. *headers exchange*) – ове размене рутирају порука на основу аргумената који садрже заглавља и опционе вредности. Овакве размене су веома сличне предметним разменама, али се поруке усмеравају на основу вредности заглавља уместо кључева за рутирање. Порука се подудара ако је вредност заглавља једнака вредности наведеној приликом повезивања. Посебан аргумент под називом „*x*-подударање”, додат у везивање између размене и реда, одређује да ли се сва заглавља морају подударати или само једно. Или било које заједничко заглавље између поруке и везивања се рачуна као подударање, или сва заглавља наведена у везивању морају бити присутна у поруци да би се подударала. Својство „*x*–подударање” може имати две различите вредности: *„any”* или *„all”*, где је *„all”* подразумевана вредност. Вредност *„all”* значи да се сви парови заглавља (кључ, вредност) морају подударати, док вредност *„any”* значи да се најмање један од парова заглавља мора подударати. Заглавља се могу конструисати коришћењем ширег опсега типова података, целог броја или нпр. хеша, уместо ниске. Тип размене заглавља (користи се са везујућим аргументом *„any”*), користан је за усмеравање порука које садрже подскуп познатих критеријума. Подразумевана размена коју AMQP брокери морају да креирају за предметну размену носи назив *„amq.headers”*.

- **Размена мртвих писама** (енгл. *dead letter exchange*) – ако се не може пронаћи одговарајући ред за поруку, порука се тихо испушта. *RabbitMQ*

обезбеђује проширење AMQP-а познато као „Размена мртвих писама“, које пружа функционалност за хватање порука које се не могу испоручити.

Редови

Редови у AMQP 0-9-1 моделу су веома слични редовима у другим системима за складиштење порука и задатака, они чувају поруке које конзумирају апликације.
Редови деле нека својства са размена, али имају и нека додатна својства:

- Име;
- Трајност (ред ће преживети поновно покретање/рестарт брокера) – постоје трајни (енгл. *durable*) редови, чији се метаподаци складиште на диску и пролазни (енгл. *transient*) редови, чији се метаподаци чувају у меморији када је то могуће;
- Ексклузивност – користи га само једна веза и ред ће бити избрисан када се та веза затвори;
- Аутоматско брисање – ред који је имао најмање једног потрошача се брише када последњи потрошач откаже претплату;
- Аргументи (опционо; користе их додаци и функције специфичне за брокер као што су TTL[4] поруке, ограничење дужине реда, итд.).

Пре него што се ред може користити, он мора бити декларисан. Декларисање реда ће довести до његовог креирања, ако већ не постоји. Декларација неће имати ефекта ако ред већ постоји и његови атрибути су исти као они у декларацији. Када постојећи атрибути реда нису исти као они у декларацији, биће објављен изузетак на нивоу канала са кодом 406 (*precondition failed*).

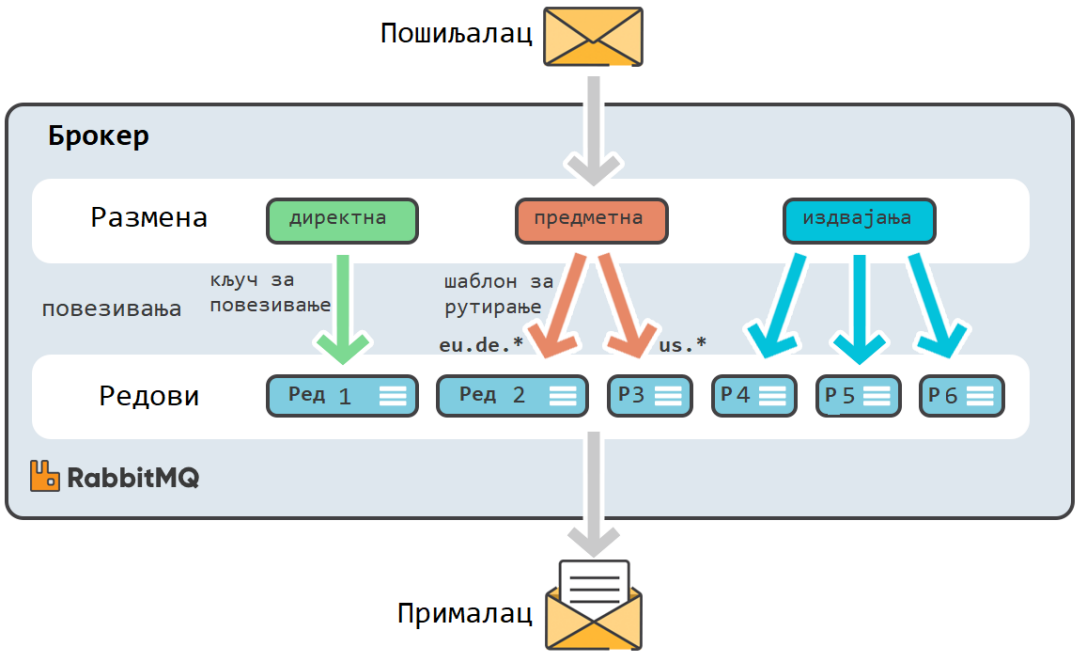
Везивања

Везивања су правила која размене користе (између осталог) за усмеравање порука у редове. Да би размени било наложено да усмерава поруке у одређени

[4] *Time To Live* – животни век поруке; након истека овог времена од тренутка креирања, иста ће бити обрисана.

ред, ред мора бити везан за размену. Везивања могу имати опциони атрибут кључа рутирања који користе неки типови размене. Сврха кључа за рутирање је да изабере одређене поруке објављене на размену, које ће се усмерити у везани ред. Другим речима, кључ за рутирање делује као филтер.

Пакети који се шаљу могу бити било који објекти. Међутим, то је обично облик неке текстуалне поруке. И то у *XML* (*eXtensible Markup Language*) или JSON (*Java Script Object Notation*) формату. Сврха система за размену порука је да размењује овакве поруке у хетерогеним системима на стандардан начин.



Слика 1.3: Скица *RabbitMQ* брокера и типова размена

1.5 Docker

Када говоримо о олакшањима код саме испоруке (енгл. *deployment*), покретања, одржавања, али и развоја апликација са микросервисном архитектуром, ту главну улогу игра *Docker*. *Docker* користи контејнере, у којима је спакован софтвер, библиотеке и конфигурационе датотеке. Служи томе да би се механизмима везаним за испоручивање и извршавање, а који се тичу

мреже, меморије, улаза-излаза и права приступа, управљало изван саме апликације на униформан начин. Такође ова технологија применом виртуализације омогућава једноставно и ефикасно покретање апликација на различитим локацијама.

„Али то је радило на мојој машини“ – колико пута сте чули ово од колега или сте и сами некоме упутили ову реченицу. Рекао бих сигурно више од једном. *Docker*-ом стајемо на пут овом и многим другим проблемима, и олакшавамо живот сваком програмеру. *Docker* је алат дизајниран да олакша стварање, примену и покретање апликација помоћу контејнера. *Docker* енкапсулира мискросервис у контејнер који се може мењати, одржавати и испоручивати, независно од остатка система. Контејнер омогућује програмеру да запакује апликацију са свим деловима који су јој потребни, као што су библиотеке и друге зависности, и да је испоручи (енгл. *deploy*) као један пакет. Чинећи то, програмер може бити убеђен да ће се апликација покретати на било којој другој машини, без обзира на специфична подешавања која машина, на којој се контејнер покреће, може имати и која се могу разликовати од машине на којој је код писан и тестиран. На неки начин *Docker* је као виртуална машина. Али, за разлику од виртуалне машине, *Docker* контејнери користе идентично *Linux* језгро (енгл. *kernel*) као и систем на којем раде, за разлику од правих вирталних машина, које имају своје језгро. Такође за покретање контејнера неопходне су само ствари којих већ нема на *host* машини, што значајно смањује величину апликације. (RedHat, 2023)

Слика 1.4: Упоредни приказ физичке, виртуалне машине и контејнера

Архитектура *Docker*-а

Подељена је у три различите компоненте и све су потребне за подешавање *Docker* окружења.

То су:

- Клијент (*Docker client*) – комуницира са *Docker* хостом слањем *cli* команде коју *Docker* може да разуме.
- Хост (енгл. *Docker host*) – платформа извршава захтеве који стижу од клијента и може бити на компјутеру или у облаку (енгл. *cloud*).
- Регистар (енгл. *Docker registry*) – чува *Docker* слике (енгл. *image*).

Када кроз клијента (терминал) затражите одређену слику, било да она постоји или не, клијент одлази до хоста и упућује захтев. Ако је слика

доступна, демон нит (енгл. *daemon thread*)[5] из хоста покреће слику, а ако слика не постоји, демон одлази у регистар и покушава да је пронађе и да је преузме.

Регистар

Регистар је место где се објављују контејнери који се направе, где ће затим бити регистровани. Свака нова слика добија нову ознаку у складу са верзијом која је похрањена у регистар. На *Docker*-у је могуће имати више верзија исте слике. Ако се посети *Docker hub*, може се пронаћи мноштво верзија исте слике. Регистар је место одакле *Docker* хост повлачи слике. Слике које су покренуте називају се контејнери. Слике су замрзнута верзија копије апликације (енгл. *snapshot*).

Када се покрене слика, она добија меморију и координате са језгром, тада слика постаје контејнер (*image* → *container*). Може бити покренут различит број контејнера од исте слике. Регистар је као *github*[6], али за слике. Такође, могуће је направити и локални регистар.

Слика

Docker слика је датотека само за читање (енгл. *readonly*) која служи да произведе *Docker* контејнер. Може се причати о *Docker* слици као о објекту који садржи систем датотека оперативног система (енгл. *OS filesystem*), апликацију и све зависности те апликације тј. садржи све што је потребно да би се нека апликација покренула. Слика представља, условно речено, шаблон за креирање виртуалне машине. Шаблон виртуалне машине је заправо заустављена виртуелна машина, односно, прецизније, заустављен контејнер.

Слика представља гомилу слабо повезаних слојева, при чему сваки слој садржи једну или више датотека. Све *Docker* слике почињу основним слојем, а сваком новом променом и додавањем садржаја, нови слојеви се додају на врх претходног слоја. Па тако више различитих *Docker* слика може да има идентичне слојеве, на пример исти оперативни систем на ком почивају

[5] Демон нит је она која би требало да пружа општу услугу у позадини све док програм ради, али није део суштине програма. Стога, када се заврше све нити које нису демон нити, програм се прекида. (Karim, 2019)

[6] `https://github.com`

(*Ubuntu 20:04 image* нпр.) или исту верзију запаковане *Java*-е коју ће апликација користити при покретању. Смисао овога је да не бисмо на хост машини дуплирали исте ствари које могу користити различите слике. Иако се састоји од више слојева, ми саму слику посматрамо као јединствени објекат.

Контејнер

Контејнер је покренута инстанца слике или виртуалан приватни сервер. Контејнери садрже оперативни систем, датотеке генерисане од стране корисника и метаподатке. Контејнери су изграђени од слика, али имају слој за читање и писање на врху, па су, за разлику од слика, интерактивни и могу да складиште податке. Ово значи да у њима могу бити покренуте апликације. Након сваког извршења неке апликације у *Docker* контејнеру, може се сачувати снимак резултујућег стања и од њега може бити направљена нова слика.

Контејнери су попут виртуалних машина, међутим доста једноставнији и бржи. Не треба мешати контејнере и виртуалне машине, иако су наизглед врло слични. Разлика је лако уочљива на слици 1.4. Заједничко им је то што им треба *host* где ће се покренути, међутим једна виртуална машина има свој оперативни систем, док више контејнера користи један заједнички оперативни систем и његове ресурсе. Самим тим лакше их је креирати, покренути и одржавати од виртуалних машина. *Docker* омогућује да се апликације могу покренути у контејнеру на било којој платформи и искључује могућност конфликта између локалног окружења и окружења у ком се извршава програм (контејнер). (Poulton, 2020)

Глава 2

Архитектура развијеног система

Апликација *EvEnter* развијена као део овог рада је веб апликација и састоји се од клијентске стране (енгл. *client-side/frontend*) и серверске стране (енгл. *server-side/backend*). Цела апликација, као што је и раније поменуто, конципирана је и почива на микросервисној архитектури. Клијенту је изложен API, преко ког комуницира са серверским делом система. Сам *backend* део система организован је и састоји се од три микросервиса. Сваки микросервис има своју јединствену, уоквирену и недвосмислену улогу у систему. Сваки од наведена три сервиса имплементиран је и организован као вишеслојна *Java Spring* апликација.

Java Spring Framework је популаран оквир отвореног кода за креирање самосталних апликација производног нивоа које се покрећу на *Java* виртуалној машини. *Java Spring Boot* је алат који чини развој веб апликација и микросервиса са радним оквиром *Spring* бржим и лакшим кроз аутоконфигурацију, поуздан приступ конфигурацији и могућност креирања самосталних апликација. Ове функције раде заједно да би пружиле алат који омогућује да се подеси апликација заснована на *Spring*-у са минималном конфигурацијом и подешавањем.
Уопштено говорећи, слојеви сваког појединачног микросервиса су следећи:

- Презентациони слој (енгл. *presentation layer*) – обрађује HTTP захтеве, преводи JSON параметре у објекте и потврђује аутентичност захтева и преноси их на пословни слој.

- Слој пословне логике (енгл. *business layer*) – овде се налази сва пословна логика; састоји се од сервисних класа и користи услуге које пружају

слојеви приступа подацима.

- Слој постојаности (енгл. *persistence layer*) – садржи сву логику око чувања података и преводи пословне објекте у редове у бази података и обратно.

- Слој базе података (енгл. *database layer*) – CRUD (*Create Read Update Delete*) операције над базом података.

2.1 Предности овакве архитектуре

Микросервисна архитектура је постала најбоља алтернатива сервисно оријентисаној архитектури и монолитној архитектури. Сервиси су слабо повезани да би се развијали, тестирали, одржавали и испоручивали независно. Испоручивање веб апликација које користе монолитну архитектуру је тежак задатак због чврсто повезаних компонената. Микросервиси решавају овај проблем раздвајањем апликације на више појединачних компонената/сервиса/апликација. Даље, поједностављује повезивање између сервисних компонената. Сваки сервис се развија, испоручује и живи за себе. Сваки сервис поседује API преко ког комуницира са остатком система, али је као апликација независтан и има јединствену уоквирену улогу.

Као што је то већ поменуто, сервиси су имплементирани тако да комуницирају асинхроно и то преко система за размену порука који користи брокер и заснован је на AMQP протоколу. Посебно наводим коришћење брокера, јер постоје слични системи за размену порука који не користе посреднике тј. брокере, већ комуницирају директно. Пошиљалац шаље поруку посреднику порука, а посредник је испоручује примаоцу. Важна предност коришћења брокера порука је то што пошиљалац не мора да зна мрежну локацију примаоца. Још једна предност је то што посредник порука привремено складишти поруке док прималац не буде у стању да их обради, те тако не морају и пошиљалац и прималац у исто време да буду активни и доступни.

2.2 Потенцијални проблеми које овакав дизајн решава

Следи листа ствари које су побољшане изабраним приступом и архитектуром. Односно наведени су бенефити које су оваква имплементација и избор технологија донели, уз пратећа објашњења – који проблеми су тим путем избегнути.

Слаба спрегнутост – коришћењем брокера избегава се увођење неког механизма за откривање (енгл. *discovery mechanism*) који би морао да води рачуна о томе која је мрежна локација сервиса, колико је инстанци покренуто и читавој логици преусмеравања порука (којој инстанци на којој дестинацији треба проследити одређени захтев). То би могло да се одради и на нивоу апликације: да се сервиси сами региструју на сервисе за откривање, и да се ту направи одређена логика, међутим то захтева више посла[1].

Доступност – брокер складишти поруке интерно. У овом конкретном случају, нпр. корисници су у могућности да резервишу своје карте, чак и када *Wallet* сервис није доступан и не може се извршити конкретна трансакција куповине карте. Карта ће бити купљена када сервис постане доступан и када се захтев обради. Да овде користимо REST комуникацију, морали бисмо након резервације да имамо неки механизам поновног покушаја (енгл. *retry*) који би покушавао да спроведе куповину карте до краја, односно да на одређене временске размаке покушава да успостави комуникацију са *Wallet* сервисом, све док то не буде могуће. Исти случај је и са *Payment* сервисом, који би морао да покушава да спроведе уплате изнова и изнова, све док *Wallet* не буде доступан. Оваквом имплементацијом смо такође постигли да се захтеви чувају на механизму брокера и да ће бити обрађени када сервис буде доступан. Нема потребе за неком додатном логиком и коришћењем неког механизма поновног покушаја.

Већа отпорност и управљање грешкама – сценарији квара микросервиса постају незгодни када се разматрају трансакције које су у току. Платформе за размену порука, као што је то *RabbitMQ*, гарантују испоруку порука без губљења података чак и у тренуцима катастрофалних проблема са радом система и са мрежом.

Једноставан за скалирање – како апликације и предузећа расту, мо-

[1]За овакву поставку може се користити *Eureka*.

гућност повећања капацитета постаје једна од најважнијих предности овакве архитектуре. Пошто је сваки сервис мали и обавља само један задатак, сваки сервис би требало да буде у могућности да расте или да се смањује по потреби. Оваква архитектура знатно олакшава скалирање, а такође и одређивање микросервиса који је уско грло (енгл. *bottleneck*), где би додавање додатних инстанци побољшало перформансе, уместо слепог повећавања ресурса свих сервиса (на пример, ово је лако утврдити ако се, прегледом на брокеру, уочи да постоји гомилање порука на неком реду).

Иста порука на више сервиса – уколико постоји потреба да се једна порука (захтев, одговор или једноставна информација) пошаље на више сервиса, брокер у томе увелико помаже. Коришћењем *publish/subscribe* механизма пошиљалац (енгл. *publisher*) шаље поруку на ред, а сви заинтересовани примаоци (енгл. *subscriber*) скидају своју копију поруке. Када би се у овом случају користила REST комуникација, морала би свака порука појединачно да буде послата сваком заинтересованом сервису, што би представљало додатну компликацију. Овим приступом, брокер води рачуна о томе и његове функционалности омогућују већу флексибилност и једноставнију имплементацију. (Garvie, 2022)

Као и увек у програмирању, скоро свако решење, како има предности, тако поседује и мане. И увек се, приликом одабира одређеног решења, у овом случају архитектуре, балансира између добрих и лоших страна (енгл. *trade-off*).
Мане изабране архитектуре развијеног система су следеће:

- Додатна оперативна сложеност – систем за размену порука је још једна компонента система која мора бити инсталирана, конфигурисана и којом се управља. Потребна је додатна инсталација и конфигурација самог брокера.

- Потенцијално уско грло у перформансама – постоји ризик да посредник порука може бити уско грло у перформансама. На срећу, многи модерни брокери порука су дизајнирани да сами по себи буду веома скалабилни.

- Појединачна тачка квара (енгл. *single point of failure*) – од суштинског је значаја да посредник порука буде стално доступан, у супротном ће то утицати на поузданост система. На срећу, већина модерних брокера је дизајнирана да буде веома доступна.

- Несвакидашњи начин програмирања – неко ко није навикао да ради на апликацијама овог типа и да користи било шта друго осим REST-а, имаће додатан изазов да се прилагоди и уђе у логику и праксе овог начина програмирања.

- Потребно је водити рачуна о акцијама поништавања (енгл. *rollback actions*) – ово представља изазов код микросервисне архитектуре уопште. Ако се деси да је неки део посла/дистрибуиране трансакције[2] само делимично одрађен, а затим се догодила нека грешка, долази се у ситуацију да су на неком од микросервиса остали неконзистентни подаци. Тада је потребно пажљиво имплементирати акцију поништавања трансакције која податаке на свим сервисима доводи у исправно стање.

2.3 Опис апликације

Апликација **EvEnter** најједноставније речено представља систем за куповину карата за разне догађаје *online*, преко веб платформе. Састоји се од *frontend*-а и *backend*-а, који је организован у три микросервиса.

Сервиси

- **PaymentService** – служи као адаптер ка свим спољним пружаоцима платних услуга; тренутно је имплементирана комуникација са само једним платним пружаоцем услуга, али у случају ширења апликације и интеграције са другим пружаоцима, овај сервис је место где ће се те нове имплементације налазити. Сервис путем изложеног API-ја прихвата захтеве за уплату новца и даље их шаље на обраду у овом систему. Уплате се, затим, шаљу *Wallet* сервису, који обрађује те захтеве и увећава биланс корисника.

- **Wallet** – садржи тренутна стања рачуна корисника и трансакције које су утицале на увећање/умањење стања. Сервис прихвата захтеве за увећање/умањење биланса новца корисника, након обраде захтева враћа одговарајући одговор заинтересованим сервисима. Такође, увек је доступан

[2]Микросервисна апликација у свом правом контексту, је дистрибуирани систем. Дистрибуирана трансакција је она која се дистрибуира/протеже на више сервиса, а резултати акција сваког од микросервиса у том ланцу заједно довршава целу једну дистрибуирану трансакцију. (Deshpande, 2020)

да врати појединачно стање рачуна за сваког корисника у систему, како би исти имали увид у сопствени новац.

- **EventTicket** – садржи релевантне податке о свим догађајима у систему, свим улазницама и њиховој вези са корисницима (уколико је карта купљена/резервисана, зна се коме припада). Служи за креирање и администрацију догађаја, садржи податке о корисницима након регистрације. Сва логика око резервације карата од стране корисника креће одавде.

Основни случајеви употребе

У наставку следи списак свих случајева употребе развијених у оквиру апликације *EvEnter*.

- **Регистрација корисника**

  1. Отварање веб странице.
  2. Попуњавање форме за регистрацију.
  3. Информација се чува у базу на *EventTicket* сервису.
  4. Информација о новорегистрованом кориснику се шаље на ред.
  5. Поруку скида *PaymentService* (како би након пристигле уплате могао да повеже уплату са корисником преко идентификатора додељеног кориснику при регистрацији).

- **Уплата новца на корисников налог**

  1. Корисник је уплатио новац у банци, а у позив на број на уплатници је уписао свој ЈМБГ.
  2. Прзжалац платне услуге шаље захтев *PaymentService*-у, он га обрађује.
  3. Појединачну уплату повезује са корисником преко ЈМБГ-а.
  4. *PaymentService* снима уплату у почетном статусу у базу.
  5. *PaymentService* шаље захтев за уплату на ред ка *Wallet* сервису.
  6. *Wallet* сервис прихвата захтев за уплату.
     6.1. Уколико корисниково стање постоји на *Wallet* сервису, увећава биланс корисника.

6.2. Уколико корисник не постоји, креира се његово стање са пристиглим износом.

7. *Wallet* шаље поруку о успешности уплате на ред *PaymentService*-у.
8. У зависности од одговора који стигне са *Wallet* сервиса, *PaymentService* ажурира трансакцију уплате и пребацује је у финални статус.

- **Креирање догађаја**

  Постуслов: нови догађај је креиран и могуће је купити карту за њега

  1. Запослени се логује.
  2. Запослени отвара форму за креирање новог догађаја и уноси потребне податке.
  3. Запослени потврђује своју акцију на форми кликом на одговарајуће дугме за потврду.
  4. На *EventTicket* сервису се нови догађај снима у базу, као и расположиве карте за тај догађај.
  5. Корисницима је сада новокреирани догађај видљив на сајту и улазнице за тај догађај су им расположиве за куповину.

- **Отказивање догађаја**

  1. Запослени врши избор догађаја који жели да откаже.
  2. Запослени означава догађај као отказан и потврђује своју акцију.
  3. Све продате карте за тај догађај враћају се у статус „резервисан" на *EventTicket* сервису.
  4. Ка *Wallet*-у се шаљу захтеви за увећање налога корисника у износу купљених карата за сваког корисника појединачно.
  5. Након успешне трансакције на *Wallet*-у, шаљу се подаци о тим трансакцијама преосталим сервисима.
  6. Након одговора од *Wallet*-а, карта се на *EventTicket* сервису пребацује у статус „отказан".

- **Куповина карте**

1. На веб страни корисник врши одабир догађаја и карата које жели да купи.
2. Кликом на дугме за потврду, карте које је одабрао снимају се у статусу „резервисан" у бази *EventTicket* сервиса и приказује се порука кориснику да је карта резервисана.
3. Шаље се са *EventTicket* сервиса захтев за скидање новца са рачуна у износу укупне суме карата *Wallet* сервису заједно са информацијом о идентификатору корисника.
4. *Wallet* прихвата захтев за умањење биланса корисника.
   4.1. Умањује биланс корисника, уколико корисник постоји, односно ако постоји његово стање.
   4.2. Уколико корисник не постоји или уколико нема довољно новца на његовом рачуну, не врши куповину.
5. *Wallet* шаље поруку о успешности трансакције ка преосталим микросервисима.
6. *EventTicket* сервис прихвата одговор од *Wallet*-а са реда.
   6.1. Након успешног одговора од *Wallet*-а, карте се пребацују у финални статус у бази *EventTicket* сервиса, након чега се корисник обавештава поруком, а генерисана карта му се шаље на мејл. Случај употребе се завршава.
   6.2. Након неуспешног одговора од *Wallet*-а, кориснику излази обавештење на веб страни да ће му карта бити отказана уколико је у наредна два сата не плати.
      i. Уколико време није истекло, а на билансу корисника се појавио износ довољан за куповину карте, случај употребе се наставља на 4.
      ii. Након истека времена, а услед неплаћања, карта се враћа у расположиво стање и није више резервисана. Случај употребе се завршава.

Глава 3

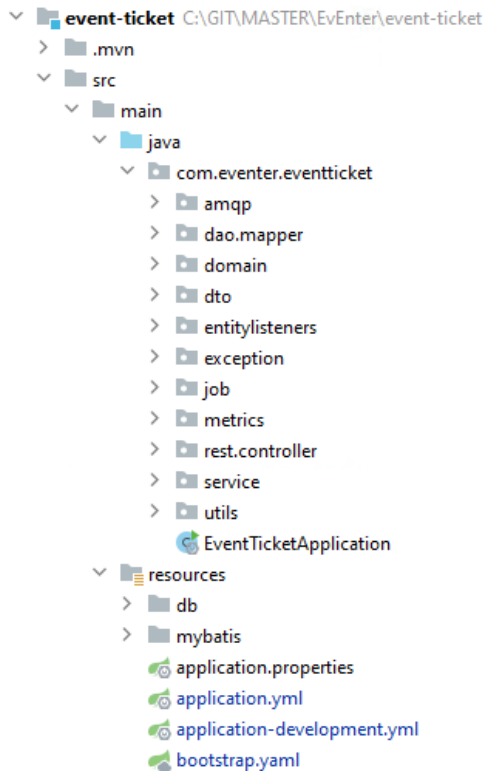
Имплементација система

Као што је то претходно поменуто, од значаја за овај рад је *backend* део *EvEnter* апликације. Комплетан *backend* је тако постављен да се састоји од 3 микросервиса који комуницирају користећи AMQP протокол. Апликација *EvEnter* је апликација отвореног кода и доступна је на *GitHub* репозиторијуму на адреси `https://github.com/upoznat/EvEnter`. Део који се тиче *Docker*-а за покретање контејнера налази се у репозиторијуму `https://github.com/upoznat/docker-compose-eventer`.

3.1 Технологије и структура апликације

Сваки микросервис конципиран је на идентичан начин. Израђен је као *Spring* апликација. Тако постоје делови који су задужени управо за комуникацију и везу са брокером и изгледају идентично на сваком микросервису. Дизајн микросервиса је вишеслојни.

У пакетима *rest.controller* (прихвата захтеве са *frontend*-а) и *amqp* (задужен за комуникацију са другим микросервисима) налазе се класе које припадају презентационом слоју, највишем слој апликације који прима захтеве, конвертује JSON у *Java* објекте (и обратно) и прослеђује их сервисном слоју. У пакету *service* налазе се класе које припадају сервисном слоју, односно слоју пословне логике, у ком се налази пословна логика апликације. У пакету *dao.mapper* налазе се класе које одговарају слоју података и у њима се налазе методе које се позивају за манипулисање подацима директно над базом података. Пакет *resources.mybatis* садржи *.xml* датотеке у којима су одрађена пресликавања и написани упити над базом података, који се позивају из



Слика 3.1: Структура Spring Boot пројекта једног мискросервиса

претходнопоменутих класа.

3.2 Класе

Следи списак класа са појашњењима и детаљима. Класе које ће бити наведене су оне које се издвајају као специфичне за комуникацију међу микросервисима.

Ослушкивач (енгл. *Listener*)

Класе ослушкивачи изгледају овако. На слици је пример једне такве класе из једног сервиса, међутим слично изгледају и остале, одабран је произвољан пример без умањења општости. Свака класа ослушкивач означена је *Spring @Component* анотацијом. *Spring Application Context* је место где *Spring* држи инстанце објеката за које је идентификовао да се њима управља и дистрибуира их аутоматски. Ови објекти се зову зрна (енгл. *beans*). Управљање

зрнима и могућност за уметање зависности (енгл. *Dependency Injection*) су неке од главних карактеристика *Spring*-а.

Уметање зависности представља шаблон архитектуре софтвера чији је главни принцип да одвоји имплементацију објекта од креирања других објеката од којих тај објекат зависи и на које се ослања. Ово је специфичнија верзија онога што зовемо инверзијом контроле. Уметање зависности уклања чврсту везу између објеката, омогућавајући и објектима и апликацијама које их користе да буду флексибилнији. Олакшава креирање лабаво повезаних објеката и њихових зависности. (Kanjilal, 2021)

Инверзија контроле (енгл. *Inversion of Control*) је принцип који промовише лабаву везу између компонената апликације. Главна предност је то што класе више не зависе једна од друге. Класа која се користи више неће зависити од класе која је користи и промене у једној неће утицати на другу. Користећи принцип инверзије контроле *Spring* прикупља инстанце зрна из апликације и користи их у одговарајуће време. Без потребе да се експлицитно подешавају и инстанцирају ти објекти. *@Component* је анотација која омогућује *Spring*-у да аутоматски пронађе зрна која су направљена. Другим речима, без писања експлицитног кода, *Spring* скенира апликацију тражећи класе означене са *@Component*, инстанцира их, убацује све потребне зависности у њих и убацује (енгл. *inject*) ове класе где год да је то потребно.

Метода која је задужена за скидање порука са реда, означава се анотацијом *@RabbitListener*. Ова анотација, налази се у пакету који је увезен у пројекат путем *Maven*-а и то додавањем следеће зависности (енгл. *dependency*).

```
<dependency>
    <groupId>org.springframework.amqp</groupId>
    <artifactId>spring-rabbit</artifactId>
</dependency>
```

Код 3.1: Приказ *Maven* зависности за *RabbitMQ*

Као параметар анотације наводи се име реда за који се везује. Улазни аргумент методе која је анотирана је, у ствари, порука која је преузета са реда, десеријализована и претворена у објекат задатог типа. Након прихватања поруке и десеријализације исте, улази се у тело методе, а порука нестаје са реда. На приложеном парчету кода види се метода која се везује за ослушкивач на један ред, међутим, у овој класи налазе се све методе које се везују за

било који ред.

```
@Slf4j
@Component
public class PaymentRequestListener {

    @Autowired
    UserTransactionService userTransactionService;

    @Autowired
    ResponseSender sender;

    @Async("payin-request-executor")
    @RabbitListener(queues = Queues.WALLET_PAYIN_REQUEST_QUEUE)
    public void receiveWalletPayinRequest(UserRequestDTO pmt) {
        log.info("Recieved Message From RabbitMQ: " + pmt);
        try {
            PaymentResponseDTO resp =
                userTransactionService.processPaymentRequest(pmt);
            sender.sendPayinResponse(resp);
        } catch (Exception e) {
            log.info("Error while processing payment:{}", e.getCause());
            throw new PaymentTransactionProcessingException(e.getCause());
        }
    } ...
```

Код 3.2: Ослушкивач класа и веза на један ред

Такође, користи се анотација *@Async* која омогућује да се извршавање методе коју означава издвоји у посебну нит (енгл. *thread*). Као параметар овој анотацији задаје се извршилац (енгл. *Thread Pool Executor*) који се може посебно конфигурисати и помоћу ког имамо већу контролу, те се може дефинисати одређено понашање, нпр. број нити које су стављене на располагање и коришћене за извршење ове методе. Још један разлог зашто је искоришћена, су примери из праксе када није коришћена ова анотација, те је само једна нит била задужена за скидање порука са реда и бивала ослобођена тек на крају извршења методе. Тако, у случају када постоји одређени проблем, нит која скида поруку са реда у даљем току извршења методе се, на пример, блокира

над базом података или чека неки други ресурс у свом извршавању. Тада бива прекинуто скидање порука са реда, док се блокирана нит не ослободи.

Ако не би био издвојен посебан скуп нити, односно ако би све нити програма биле стављене на располагање било ком процесу у програму, опет би могло да дође до проблема. Уколико постоји неки део апликације који „једе“ ресурсе, они бивају искоришћени само на једном месту. Док други делови, који би могли да функционишу како треба, немају потребне ресурсе за свој рад (нпр. скидање порука са једног реда због проблема над једном табелом заузме све нити, па тако није могуће скидање порука са неког другог реда и обрада других захтева који би прошли и успешно били обрађени). Стога, имплементирано је на овај начин како се не би дошло у ситуацију да превелики број порука на једном реду угрози функционисање читаве апликације и заузме све расположиве нити.

Пошиљалац (енгл. *Sender*)

```
@Slf4j
@Component
public class RequestSender {

    @Autowired
    private AmqpTemplate rabbitTemplate;

    @Async("payin-request-sending-executor")
    void sendBuyTicketRequest(WalletRequest message) {
        rabbitTemplate.convertAndSend(Queues.WALLET_BUY_TICKET_REQUEST_QUEUE,
            message);
    } ...
}
```

Код 3.3: Пошиљалац класа и веза на један ред

Пошиљалац класа садржи методе које служе искључиво за слање порука на одређени ред. Ова компонента (зрно, класа) се умеће и користи у методама из којих се врши слање порука на редове. У ову класу је уметнута инстанца класе *AmqpTemplate* која представља директну везу са брокером на који се повезује. У наставку се може видети како се ово зрно креира и са којим

параметрима, односно како се врши повезивање на одређени брокер.

AMQP конфигурација

```
@Configuration
public class MessagingConfiguration {

    @Autowired
    RabbitMQProperties properties;

    @Bean
    Queue buyTicketRequestQueue() {
        return new Queue(WALLET_BUY_TICKET_REQUEST_QUEUE, false);
    }

    @Bean
    Binding bindingTicketRequest() {
        return
            BindingBuilder.bind(buyTicketRequestQueue()).to(walletExchange());
    }

    @Bean
    FanoutExchange walletExchange() {
        return new FanoutExchange(WALLET_EXCHANGE);
    } ...
}
```

Код 3.4: *MessagingConfiguration* класа и зрна за повезивање реда и размене

У приложеном исечку кода види се класа која је анотирана са *@Configuration* што значи да се у овој класи налазе методе чије су повратне вредности објекти класа који ће постати, односно бити регистровани као зрна. Па су тако и методе означене *@Bean* анотацијама. Метода *buyTicketRequestQueue()* креира зрно типа *Queue*, односно креира ред *wallet_ buy_ ticket_ request_ queue* на који ће се слати захтеви за куповину карата од *EventTicket* ка *Wallet* сервису. Метода *walletExchange()* креира одговарајућу размену за коју ће ред бити везан. Метода *bindingTicketRequest()* повезује претходнопоменути ред и размену.

```
@Bean
    public ConnectionFactory connectionFactory
        (RabbitConnectionFactoryBean rabbitConnectionFactory) {
        CachingConnectionFactory cachingConnectionFactory =
               new CachingConnectionFactory(rabbitConnectionFactory
                                            .getRabbitConnectionFactory());
        cachingConnectionFactory.setChannelCacheSize(10);
        cachingConnectionFactory.setCacheMode(CachingConnectionFactory
                                            .CacheMode.CHANNEL);
        return cachingConnectionFactory;
    }

    @Bean
    public RabbitConnectionFactoryBean rabbitConnectionFactory() {
        RabbitConnectionFactoryBean connectionFactory = new
            RabbitConnectionFactoryBean();
        connectionFactory.setHost(properties.getHost());
        connectionFactory.setUsername(properties.getUsername());
        connectionFactory.setPassword(properties.getPassword());

        return connectionFactory;
    }

    @Bean
    public AmqpTemplate template(ConnectionFactory connectionFactory,
        MessageConverter converter){
        final RabbitTemplate rabbitTemplate =
               new RabbitTemplate(connectionFactory);
        rabbitTemplate.setMessageConverter(converter);
        return rabbitTemplate;
    } ...
}
```

Код 3.5: *MessagingConfiguration* класа и зрна за повезивање на брокер

rabbitConnectionFactory() прави везу на брокер користећи параметре које ишчитава из *property* датотеке. Ова конекција се користи као улазни аргумент

методе *template()* која креира објекат који ће се директно користити за слање порука на дефинисани брокер, односно за ред који буде одабран и направљен.

3.3 Покретање апликације као Docker контејнера

Циљ је да се *Spring* апликација која је описана покрене као *Docker* контејнер. Прво је потребно креирање слике од тренутне верзије апликације. Неопходно је да се у *pom.xml* датотеци у оквиру <*build*> ознаке дефинише име резултујућег *.jar*-а под ознаком <finalName>.

```
    <properties>
        <SERVICE_NAME>event-ticket</SERVICE_NAME>
    </properties>
    <build>
        <finalName>${SERVICE_NAME}</finalName>
        <plugins>
            <plugin>
                <groupId>org.springframework.boot</groupId>
                <artifactId>spring-boot-maven-plugin</artifactId>
            </plugin>
        </plugins>
    </build>
</project> ...
```

Код 3.6: Део *pom* датотеке неопходан за креирање слике

Узима се за пример *EventTicket* сервис, јер је већ описан део детаља његове имплементације (напомена: остала два сервиса конфигуришу се на исти начин). Након стандардне *maven clean install* команде за прикупљање недостајућих зависности и *build*-а пројекта, резултујући *jar* налази се у *target* директоријуму апликације и то под именом које је претходно дефинисано – *event-ticket.jar*

У кореном директоријуму пројекта креира се *Dockerfile*, то је датотека у којој ће бити дефинисано на који начин се креира *Docker* слика. У *FROM* делу задаје се назив постојеће *Docker* слике са *Docker hub*-а на коју ће се наслонити креирани микросервис. У *ADD* делу задаје се релативна путања

Dockerfile

```
1  FROM openjdk:11
2  ADD /target/event-ticket.jar event-ticket.jar
3  EXPOSE 8085
4  ENTRYPOINT ["java", "-jar", "event-ticket.jar", "-Dspring.profiles.active=development"]
```

Слика 3.2: *Dockerfile*

од места на ком се извршава *docker* команда до *.jar*-а који ће се користити, други аргумент је ново име поменутог *.jar*-а; *EXPOSE* представља *port* на ком ће се изложити апликација унутар контејнера; *ENTRYPOINT* представља команде и пратеће аргументе који ће се извршавати приликом покретања контејнера. Након овако креираног *Dockerfile*-а, из командне линије потребно је позиционирати се у директоријум (један ниво изнад *target* директоријума) и следећом командом креирати слику *docker build -f Dockerfile -t event-ticket .*

upoznat@UPoznan-nov: /mnt

```
upoznat@UPoznan-nov:/mnt/c/GIT/MASTER/EvEnter/event-ticket$ docker build -f Dockerfile -t event-ticket .
Sending build context to Docker daemon  85.42MB
Step 1/4 : FROM openjdk:11
 ---> 47a932d998b7
Step 2/4 : ADD /target/event-ticket.jar event-ticket.jar
 ---> Using cache
 ---> 63bd172c95b5
Step 3/4 : EXPOSE 8085
 ---> Using cache
 ---> c2f8964ba36f
Step 4/4 : ENTRYPOINT ["java", "-jar", "event-ticket.jar", "-Dspring.profiles.active=development"]
 ---> Using cache
 ---> 20aff3960f10
Successfully built 20aff3960f10
Successfully tagged event-ticket:latest
upoznat@UPoznan-nov:/mnt/c/GIT/MASTER/EvEnter/event-ticket$ docker run -it --network=eventer-net -p 8085:8085 event-ticket
```

Слика 3.3: Изградња слике од резултујућег *.jar*-а апликације

Након извршене команде креирана је слика под називом *event-ticket*. Покретање креиране слике ради се командом

docker run -it –network=eventer-net -p 8085:8085 event-ticket

Овом командом и параметрима покренута је слика, повезана је на *Docker* мрежу *eventer-net*, пресликан је локални порт машине 8085 да преусмерава на исти порт у контејнеру. Након тога контејнер је покренут, а самим тим и сервис.

Docker-compose

Уколико је почетна конфигурација сервиса увек иста, рецимо ако се користи за развој у локалном окружењу, па и у продукцији, згодно је користити *docker compose*. *Compose* омогућује да се сачува конфигурација неопходна за прављење слике и за покретање контејнера, као и свих осталих зависности неопходних за креирање и покретање. У наставку може се видети пример изгледа датотеке која се користи за покретање *RabbitMQ* брокера у локалном окружењу. Такође, на исти начин, конфигуришу се и остале датотеке, за покретање сервиса и за базу података. За сваки од сервиса, којих има 3, за базу и за брокер постоје посебни контејнери, укупно 5. Могуће их је спојити све у једну *docker-compose.yml* датотеку, са дефинисањем међусобних веза, те их све заједно покренути само једном командом. У току развоја, згодно је одвојено покретати све сервисе, али генерално одлука је на програмеру на који начин ће докеризовати свој систем.

```
version: "3.2"
services:
  rabbitmq:
    image: rabbitmq:3-management-alpine
    container_name: 'rabbitmq'
    ports:
        - 5672:5672
        - 15672:15672
    volumes:
        - ~/.docker-conf/rabbitmq/data/:/home/upoznat/rabbitmq/lib
        - ~/.docker-conf/rabbitmq/log/:/home/upoznat/rabbitmq/log
    networks:
        - eventer-net

networks:
 eventer-net
  driver: bridge
```

Код 3.7: *docker-compose.yaml* датотека за покретање *RabbitMQ* контејнера

У претходном исечку приказан је садржај *docker-compose.yaml* датотеке. Следе објашњења сваке ознаке која се налази у поменутој датотеци, редом.

version – верзија *docker compose*-а која се користи;
services – наводе се редом имена сервиса који ће бити покренути овом датотеком, где се за сваки наводи конфигурација;
image – назив слике која се повлачи са *docker hub*-а;
container_name – под којим именом се покреће контејнер;
ports – пресликавање портова са локалне машине ка одговарајућим портовима у оквиру контејнера и то у запису *порт са локалне машине*:*порт у оквиру контејнера*;
volumes – пресликавање директоријума са локалне машине ка одговарајућим директоријумима у оквиру контејнера и то у запису *директоријум са локалне машине*:*директоријум у оквиру контејнера*;
networks – *Docker* мрежа коју контејнер користи; поље под овим називом као главном ознаком, а не у оквиру *services*, представља дефинисање *Docker* мреже, која се на овај начин прави и која ће моћи да се користи.

Након креирања ове датотеке, потребно је позиционирати се из терминала у директоријум где се датотека налази и једноставном командом *docker-compose up* клонирати слику на локалну машину и покренути контејнер према задатој конфигурацији.

```
upoznat@UPoznan-nov: /mnt
upoznat@UPoznan-nov:/mnt/c/GIT/MASTER/docker-compose-eventer/rabbitmq$ docker-compose up -d
Pulling rabbitmq (rabbitmq:3-management-alpine)...
3-management-alpine: Pulling from library/rabbitmq
8a49fdb3b6a5: Pull complete
3ee385a09527: Pull complete
2ab56bd8d7d2: Pull complete
23c069ac05d1: Pull complete
c04f4bd6dbc0: Pull complete
54daf963c22d: Pull complete
290b8aa7054e: Pull complete
265e7f0d0352: Pull complete
812d8297a792: Pull complete
037e246423c7: Pull complete
2e868ee61678: Pull complete
Digest: sha256:16d7da8ea63c31484f754b7d6781654420b7b54cd45e9360e06af67bdacd0526
Status: Downloaded newer image for rabbitmq:3-management-alpine
Creating rabbitmq ... done
```

Слика 3.4: *Docker compose* за покретање *RabbitMQ*-а

Глава 4

Перформансе развијеног система

Одабране су одређене метрике које ће бити мера скалабилности и извршена је анализа понашања апликације и параметара који се прате при различитом оптерећењу апликације и повећању броја конкурентних корисника. За симулирање вештачког оптерећења апликације и саобраћаја од мноштва корисника користи се *Apache*-јев алат *JMeter*[1]. Он пружа могућност да се изведе тест оптерећења (енгл. *load testing*) апликације, поседује мноштво опција и конфигурација којима се може што реалније симулирати понашање у продукцији и велики број конкурентних корисника за задато понашање.

За прикупљање метрика користи се *Prometheus*[2]. *Prometheus* се, такође, покреће као контејнер, конфигурисан је тако да прикупља податке који су му изложени на поменутим трима микросервисима кроз унапред одређене ресурсе. *Prometheus* у задатом такту упућује REST позиве ка изложеним сервисима, као одговор добија све расположиве метрике сервиса и складиште их у своју базу података временских серија (енгл. *time-series database*). База података временских серија је софтверски систем који је оптимизован за складиштење и дохватања података као временских серија, које представљају скуп повезаних парова временских тренутака и вредности. Временске ознаке пружају критичан контекст за сваку од тачака података у томе како су оне повезане са другима. (Hazelcast, 2021).

За графички приказ поменутих метрика прикупљених преко *Prometheus*-а,

[1] `https://jmeter.apache.org/`
[2] `https://prometheus.io/`

користи се *Grafana*[3]. *Grafana* је мулти-платформска веб апликација отвореног кода за веб која служи за аналитику и интерактивну визуализацију. Обезбеђује дијаграме, графиконе, могућност за креирање обавештења у случају критичних вредности одређених виталних параметара сервиса. Може се лако инсталирати коришћењем *Docker(Compose)*-а, повезује се на извор података на основу којих креира графике. (Anna, 2019)
У конкретном случају, извор података је управо *Prometheus*.

4.1 Одзив – одбијање захтева

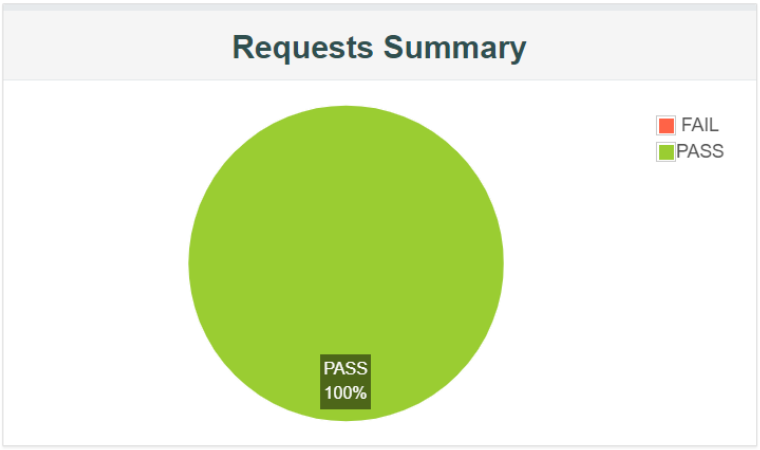
Овде ћемо видети какав је одзив (енгл. *responsiveness*) тј. до које крајње границе апликација може да функционише нормално док не почне да одбија све новодолазеће захтеве. Односно када је број корисника којима је могуће вратити одговор максималан, где би сваки следећи корисник при истом оптерећењу добијао неку грешку као немогућност одговора од апликације. Када се исцрпе ресурси апликације, она својим интерним механизмом блокира нове захтеве, јер нема ресурсе да их опслужи.

Пратиће се стање апликације и уочити на ком броју конкурентних корисника апликација престаје да ради како треба, односно где се као одговор добија велики број *timeout*-а или изузетака при повезивању (енгл.*connection exception*) који сугеришу да сервер више не прихвата захтеве. Додатно, поставља се временски параметар обраде захтева као ограничење. Односно сматраће се грешком сваки упућен захтев за чији је одговор од сервера тј. обраду, потребно дуже од 10 секунди. Наравно ово је опциони параметар у тестирању, време чекања може да се продужи, у зависности од тога колико је дозвољено пустити корисника да чека на добијање одговора. У реалном сценарију рада апликације у продукцији, 10 секунди је можда и много. Ово све зависи од природе апликације и случаја употребе које сваки захтев задовољава.

Почиње се са малим бројем конкурентних корисника и повећава се број, док се не дође до крајњих граница. На слици 4.1 види се проценат успешних одговора када је број конкурентних корисника *50, 100* затим *200*.

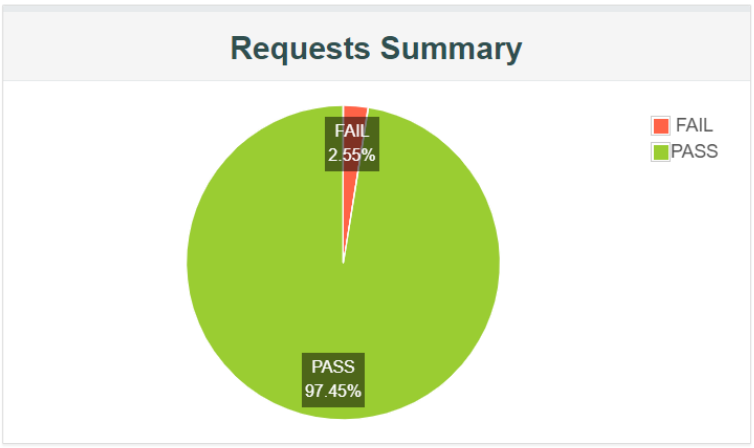
Наредна слика је проценат када 450 конкурентних корисника сваке секунде упућују истовремено захтеве ка сервису. Види се да са 450 конкурент-

[3] `https://grafana.com/`



Слика 4.1: Проценат успешности – 50, 100, 200 корисника; 1 инстанца

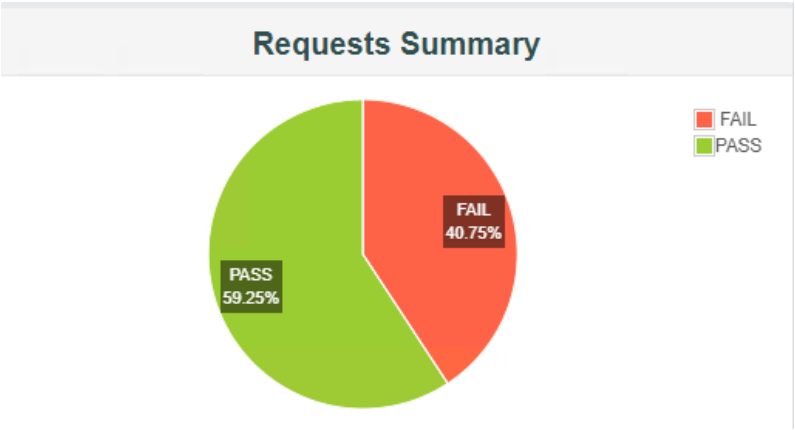
них корисника и једном инстанцом, аликација функционише добро, али на граници. Веома мали проценат негативних одговора. Статистика искључиво негативних одговора је таква да 52.63% од укупног броја грешака настаје услед одбијене конекције, јер сервиси више нису могли да прихвате захтеве. Односно, да би једноставан *retry* на овај ресурс вероватно прошао. Док је остатак, тј. 47.37% проглашен грешком. Тврдња која је задата, да је ограничено време чекања на 10 секунди, даје грешку, иако је суштински одговор враћен.



Слика 4.2: Проценат успешности – 450 корисника; 1 инстанца

Ако се повећа број конкурентних корисника на *600*, види се да је проценат

грешке много већи, односно да апликација не може да поднесе оволики број захтева. Успешност је 59.25%. Овим закључком остаје се на броју од *450* корисника које апликација може да опслужи при нормалном раду.

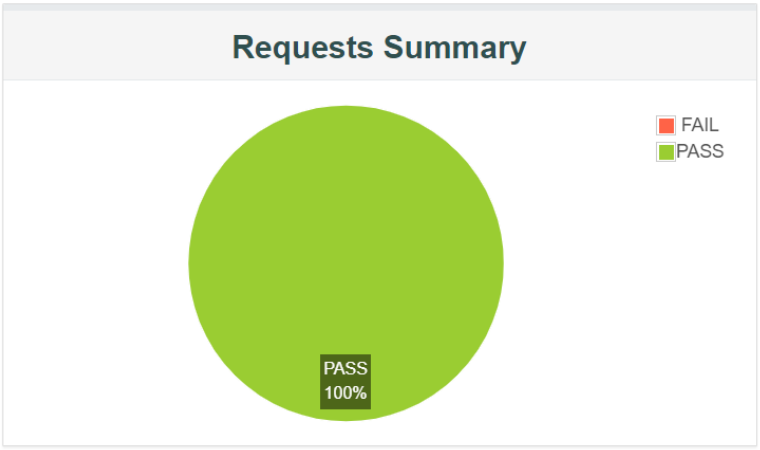


Слика 4.3: Проценат успешности враћених одговора – 600 корисника; 1 инстанца

4.2 Хоризонтално скалирање

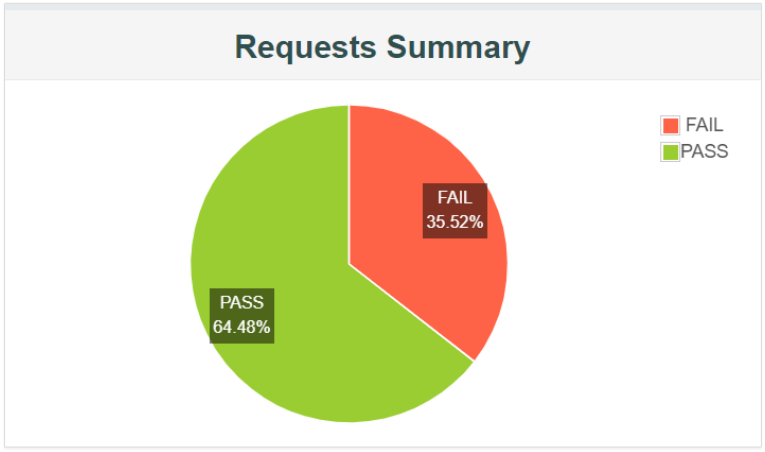
Као што је већ познато, хоризонтално скалирање апликације значи додавање још једне инстанце у циљу задржавања перформантности система, а услед повећаног оптерећења сервиса. Додаје се још једна инстанца сервиса и задаје се дупло веће оптерећење од укупног оптерећења које је једна инстанца успела да подржи, односно сада је број конкурентних корисника 900. Овде се опет види предност *Docker*-а, јер у пар минута може да се покрене нова инстанца сервиса при идентичној конфигурацији као први пут. Једина разлика је што је потребно да се изложи други порт локалне машине, јер је покретањем претходног контејнера заузет порт 8085. Следи статистика.

Закључак је да сервис ради очекивано уз максималан број корисника који може да опслужу у тренутној поставци. Додавањем још једне инстанце практично је дуплиран број конкурентних корисника и перформансе система су побољшане. Успешност враћених одговора је 100%.



Слика 4.4: Проценат успешности враћених одговора – 900 корисника; 2 инстанце

Покушава се са даљим скалирањем. Додаје се још једна инстанца, те сада постоје три; при том је број конкурентних корисника повећан на 1350 и врши се тестирање. Резултати су следећи.



Слика 4.5: Проценат успешности враћених одговора – 1350 корисника; 3 инстанце

За некога можда неочекивано, а са друге стране резултат, који је добијен у овом окружењу и тренутној поставци, је сасвим на месту. Извлачи се закључак да у одређеној поставци, окружењу, са специфичним случајевима употребе и стањем ресурса који су дати на располагање апликацији, хоризонтално скалирање не може да се врши у недоглед. Ако је разлог због ког се

врши хоризонтално скалирање, односно уско грло апликације, недовољан број нити апликације које може да се креира како би био опслужен одређени број корсника, онда је то у реду. И то је управо постигнуто увећањем броја инстанци са једне, на две. Међутим, у овом случају уско грло је база података. У току теста, база је више пута била на врхунцу свог капацитета перформантности и контејнер је више пута поново покренут. И то из два разлога. Ресурси којима приступају корисници су дељени. Просто, инфраструктура и модел података је такав, да у неким тренуцима, за одређене захтеве покушава се приступ истим редовима у табели. Односно долази се у ситуацију да више различитих корсника чека на ажурирање истог реда у бази и ту се ствара застој (енгл. *row lock contention*). Ово се дешава када један корисник ажурира или брише ред, који друга сесија жели да ажурира или избрише. (Carr, 2023) Други разлог је, јер су ограничени хардверски ресурси, односно ограничена меморија која је дата на располагање бази података. А која у овом случају, не може да поднесе толики притисак. Једноставно контејнер долази до тога да је преоптерећен и да не може да функционише како треба.

Наравно, у пракси и продукционим окружењима, ситуација је другачија. Конкретно база података је идеално на другом серверу, у другом *Docker* кластеру, на другој машини, са много већим ресурсима од оних које има у инфраструктури у којој се врши ово тестирање. Ситуација у којој се спроводи тест је таква да су сви контејнери покренути у истом кластеру, на истој машини, која нема баш завидне ресурсе. Наравно, ово је тест ради демонстрације скалирања и окружење постављено за сврху овог рада на просечном рачунару који се користи за рад и развој, не и као сервер за продукционо окружење апликације са великим бројем корисника.

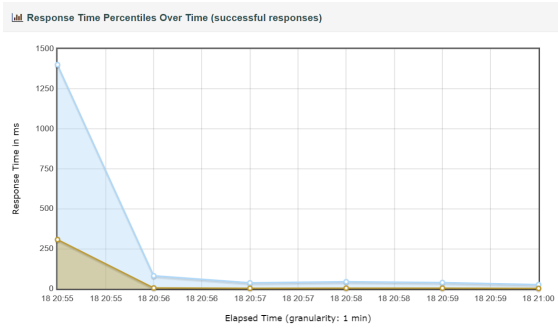
4.3 Брзина обраде захтева

Следећи критеријум, који се може видети на следећем графику су времена обраде захтева. Надовезује се на прву секцију ове главе, где је било узето у обзир време обраде. Даље, редом су приказани графици за идентичне случајеве као у претходне две главе, а то су:

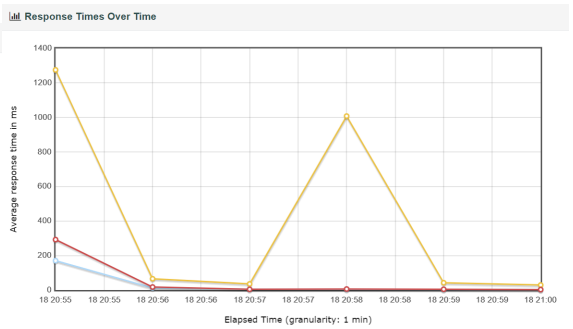
1. број корисника < 200, 1 инстанца;
2. број корисника $= 450$, 1 инстанца;

3. број корисника = 900, 2 инстанце;

4. број корисника = 1350, 3 инстанце.

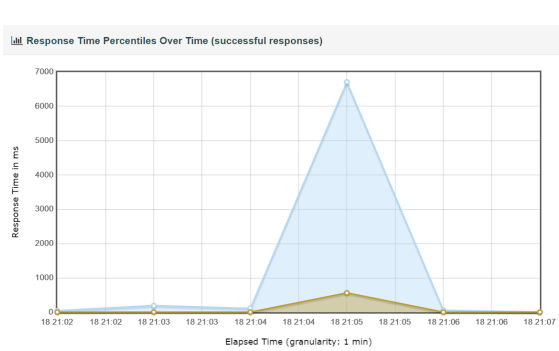


Слика 4.6: 95% одговора

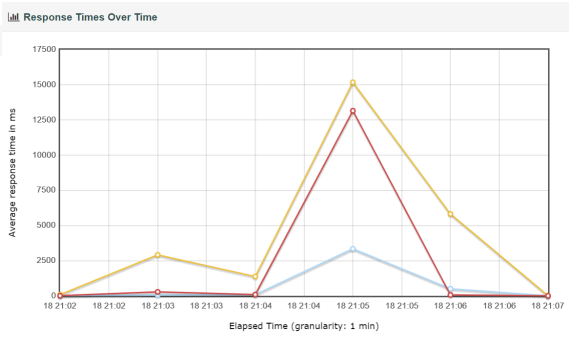


Слика 4.7: Дужина трајања захтева у милисекундама

Слике 4.6 и 4.7 одговарају случају 1. Из приложеног се види да је просечно време испод задате границе и да достиже неку већу вредност у пику који се истиче на графику. Такође, примећује се да је резервација карата позив који одузима највише времена, односно то је најскупља акција (на графику жутом бојом), што је и логично, јер има највише позадинске логике. Остале се мере у милисекундама.



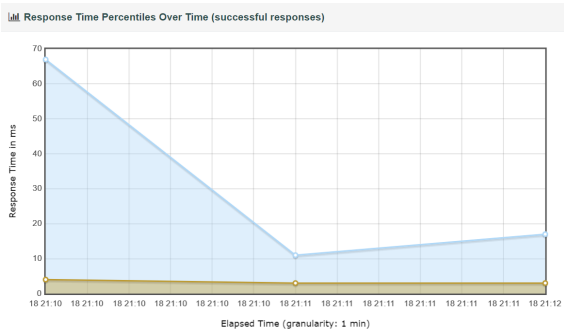
Слика 4.8: 95% одговора



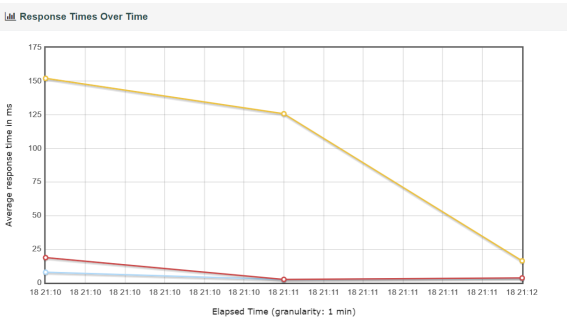
Слика 4.9: Дужина трајања захтева у милисекундама

Слике 4.8 и 4.9 одговарају случају 2. Просечно време било је испод задате границе и достиже већу вредност (прелази 10000 мс, односно 10 секунди) у пику који се истиче на графику. Такође, види се да је резервација карата

позив који одузима највише времена, односно то је најскупља акција (на графику жутом бојом).

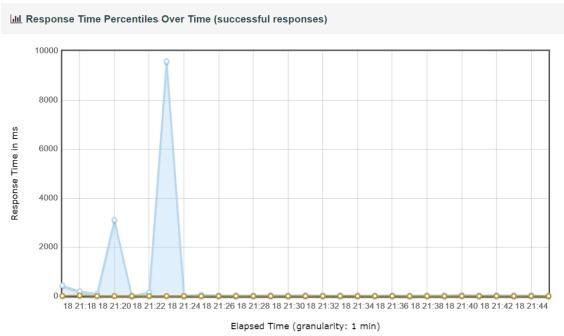


Слика 4.10: 95% одговора

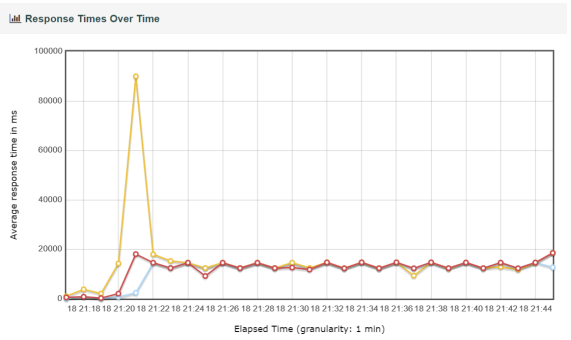


Слика 4.11: Дужина трајања захтева у милисекундама

Слике 4.10 и 4.11 одговарају случају 3. Просечно време, далеко је испод задате границе и да, чак и на највишој тачки на графику не прелази 150 милисекунди, што је изванредан одзив. Далеко најбоља ситуација је у случају 3, са покренуте две инстанце.



Слика 4.12: 95% одговора



Слика 4.13: Дужина трајања захтева у милисекундама

Графици са последње две слике су најмање релевантни, јер постоји велики број грешака и успешност одговора је тек нешто више од 50%. Са првог графика, види се да је код успешних одговора највећи број одговора био близу 10 секунди, и то опет за резервацију карата. А време чекања је у највећем делу између 10 и 20 секунди. Изузеци су почетак теста, са нормалним перформансама и статистиком, и пик, када је време чекања недопустиво велико.

4.4 CPU и Меморија

Пратиће се интерни системски параметри, односно стање машине на којој је покренута апликација. Свакако, приликом креирања *Docker* контејнера, додељује се одређена меморија контејнеру, па тако на јачој машини и са бољим ресурсима, апликација ће моћи да подржи веће оптерећење пре него што оптерети систем до границе пуцања. Тест који се врши је за контејнере којима је додељена почетна конфигурација од 1 GB меморије и 1 CPU-а.
Напомена: у току вршења самог теста, ни једна друга апликација на рачунару није покренута, нити постоје неки текући процеси који утичу на рад, наравно, мимо системских, неопходних.

```
Operating System: Ubuntu 20.04.4 LTS
OSType: linux
Architecture: x86_64
CPUs: 16
Total Memory: 15.52GiB
Name: UPoznan-nov
```

Слика 4.14: docker info

Након извршене команде у терминалу *docker info*, може се видети колико је ресурса додељено *Docker*-у. Тест се врши на рачунару са 15GB RAM меморије и 16 CPU језгара На слици изнад приказани су укупни ресурси, међутим приликом покретања сваки контејнер се може ограничити, што ће и бити урађено.
Memory: 1 GB
CPU: 1 core
Команда којом се покреће контејнер *event-ticket* је
docker run –cpus 1 –memory 1g –restart=always –name event-ticket -d -it –network=eventer-net -p 8085:8085 event-ticket
Такође *wallet* и *payment-service* контејнери, покрећу се са истом конфигурацијом, док су бази података – контејнер *oracle-db* и брокеру – контејнер *rabbitmq* дати неограничени ресурси, тј. нису специфично одређене границе меморије и процесора. Прати се стање процесора приликом повећања оптерећења на цео систем који се доводи до крајњих граница, односно до максималне оптерећености.

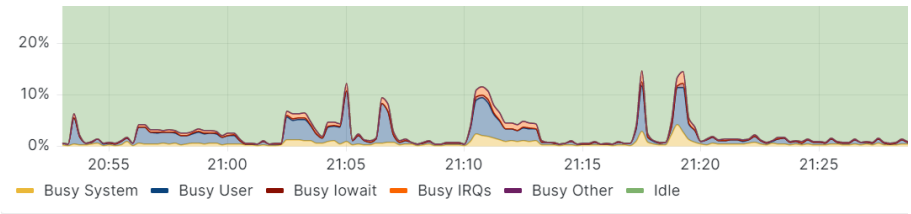
CPU

Након теста, прво са малим оптерећењем, па са повећањем из приложене табеле и графика, види се да је чак и једно језгро процесора неискоришћено од стране сервиса, што доводи до закључка да нема потребе да се врши вертикално скалирање. Односно, нема потребе додавати ресурсе у виду процесора контејнерима у којима апликација ради.

	max	mean
event-ticket	0.119%	0.0648%
payment-service	0.0984%	0.0402%
wallet	0.0303%	0.0143%
rabbitmq	0.0341%	0.0259%
oracle-db	0.255%	0.114%

Табела 4.1: Проценат искоришћености процесора при тесту

Контејнери који су покренути на *Docker* окружењу користе само до 2% процесора и раде без проблема. Чак и при већем оптерећењу не може се оптеретити процесор толико да сервиси не могу да раде неометано или да се угрози читав *Docker* кластер. Зеленом бојом, приказан је проценат процеса који нису заузети.



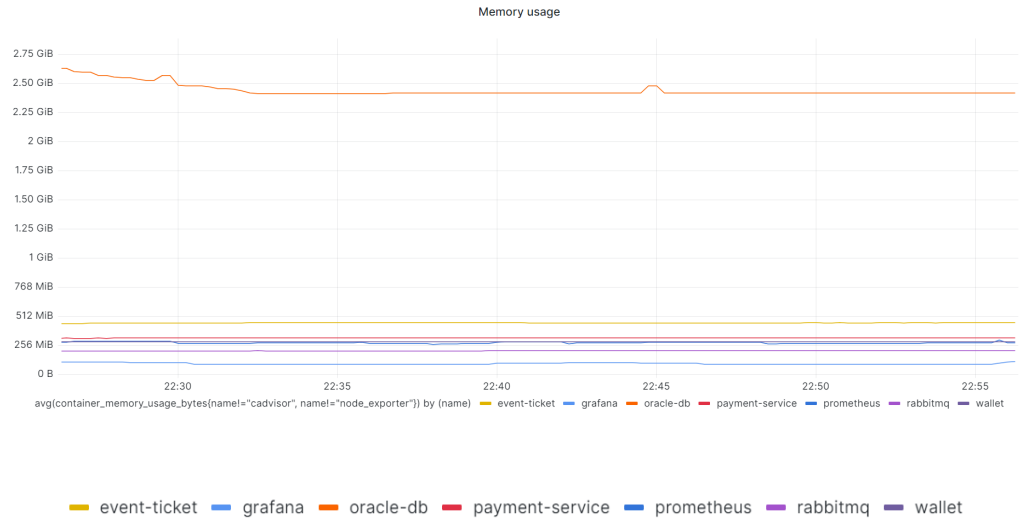
Слика 4.15: Статистика искоришћености процесора према стању процеса

Меморија

Као и у претходном примеру, прати се меморија при минималном оптерећењу, а затим при повећању саобраћаја.

	max	mean
event-ticket	460 MB	457 MB
payment-service	322 MB	322MB
wallet	292 MB	292 MB
rabbitmq	213 MB	211 MB
oracle-db	2.57 GB	2.43 GB

Табела 4.2: Проценат искоришћености RAM меморије при тесту



Слика 4.16: Стање RAM меморије

Са графика и из табеле закључује се да ни један од контејнера, којима је дат само 1GB меморије не изискују више од тога за број корисника који опслужује. Наравно, осим базе података, која има неограничене ресурсе. Па тако, што се тиче процесора и меморије, није потребно вертикално скалирање.

4.5 Промашај странице

Најпопуларнији начин имплементације виртуалне меморије, који се у пракси показао као најефикаснији, је страничење на захтев (енгл. *Demand Paging*). Слично као код страничења, овај приступ дели програм на странице. Приликом покретања програма, у RAM меморију смештају се само оне странице

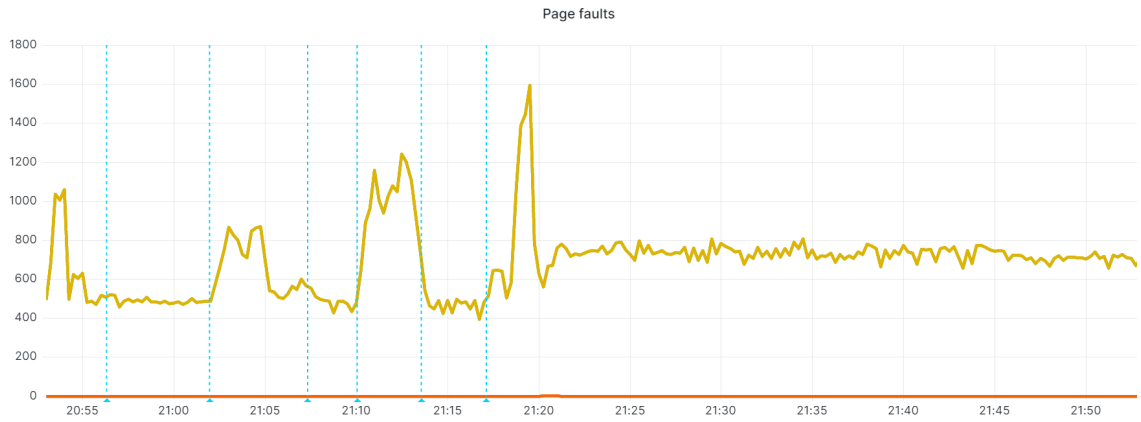
које су неопходне у првом тренутку. Тиме се смањује количина заузете меморије и повећава степен мултипрограмирања. Изворно мултипрограмирање подразумева да се више процеса конкурентно извршава на процесору, тако да се један процес извршава на процесору, све док на ред не дођу улазно-излазне операције када се он прекида и процесор дедељује следећем процесу који на њега чека. *Multitasking* подразумева да је јединица извршавања на процесору посао (енгл. *task*) који не мора нужно да обухвата извршавање процеса на процесору између две улазно-излазне операције. Такође, овај приступ подразумева да се систем организује на сличним принципима као и када је мултипрограмирање у питању, тако што ће процес по извршавању посла ослободити процесор за следећи посао процеса који је на реду.

Када се нека страница затражи при извршавању процеса, прво се проверава да ли је та страница већ у меморији, па ако није пронађена, проналази се у секундарној меморији и пребацује у радну. Наравно, оваквим приступом се постиже да се у радну меморију не учитавају странице које никада неће бити потребне. До промашаја странице (енгл. *page fault)* неће доћи ако процес никада не покуша да приступи страницама које нису у радној меморији, односно ако се направи добра процена и у радну меморију сместе све потребне странице. У том случају процес ће се извршавати на исти начин као и када су све странице у меморији. У супротном, ако процес покуша да приступи страници која није у меморији, јавља се промашај странице, а затим и прекид. На прекид се реагује и омогућава наставак извршавања процеса доношењам потребне странице у меморију. Када дође до промашаја странице, прво се врши провера да ли тражена страница уопште припада процесу који ју је затражио. Ако то није случај, долази до прекида програма услед грешке при адресирању. Уколико страница припада процесу, онда је потребно учитати је у радну меморију.

Учитавање се изводи у три корака:

- Проналази се слободан меморијски оквир, уколико постоји, или ослобађа.
- Учитава се страница из секундарне у радну меморију.
- Модификује се табела страница тако да бит валидности говори да је страница у радној меморији.

(Marić, 2017)



Слика 4.17: Промашаји страница

На графику су, вертикалним линијама, означене различите секције теста.

1. Покретање апликације, односно покретање свих контејнера (време од почетка графика до 20:55);
2. Број корисника $<$ 200, 1 инстанца (време од 20:55 до 21:02);
3. Број корисника $=$ 450, 1 инстанца (време од 21:02 до 21:07);
4. Број корисника $=$ 900, 2 инстанце (време од 21:10 до 21:15);
5. Број корисника $=$ 1350, 3 инстанце (време од 21:18 до краја графика).

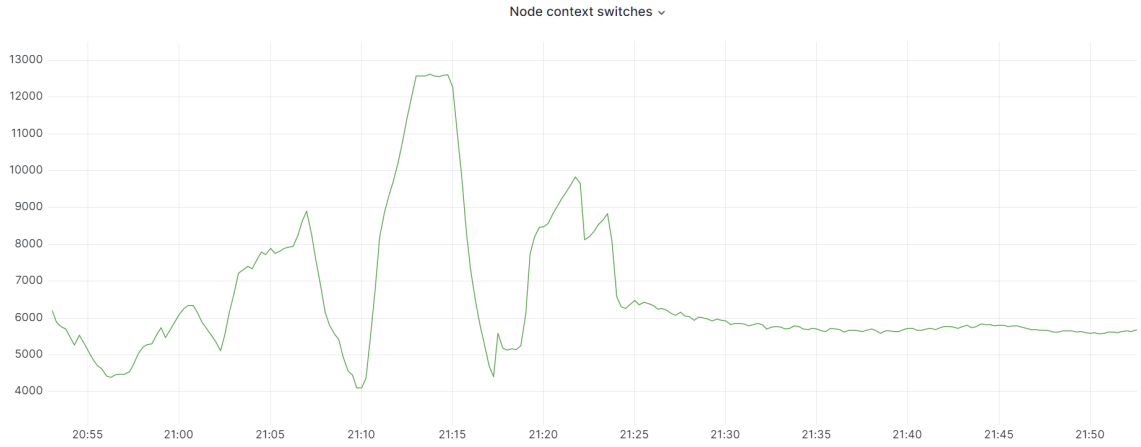
Пауза између делова 3. и 4. и 4. и 5. постоји, јер је остављено сервису да се стабилизује услед повећаног оптерећења. Грешка странице се дешава када процес приступи делу свог виртуалног меморијског простора који не постоји или је заштићен. Прво се може десити ако је процес баговит и покуша да приступи неважећој адреси (тада ће бити послат *SIGSEGV* сигнал, који га обично убија познатом поруком о грешци сегментације *Segmentation fault*). Ово последње се може десити када процес чита из меморијске зоне која је замењена, или која одговара мапираној датотеци: у том случају, језгро ће учитати страницу са диска и пустити CPU да заврши приступ меморији. То се такође може десити када процес уписује у меморијску зону, исто тако,

језгро ће предупредити процес, дуплирати меморијску страницу и наставити операцију писања на сопственој копији странице процеса.

„Главне" (енгл. *major*) грешке, дешавају се када језгро заправо мора да прочита податке са диска. Када само треба да дуплира постојећу страницу или додели празну страницу, то је редовна (или „мања") грешка. На графику тачно и видимо да се број грешака повећао за скоро 100%, односно да се при повећаном оптерећењу, у односу на просек када је сервис под минималним оптерећењем (секција 2.) види скок на графику. Наравно изузима се прва секција, где се види пораст, међутим, то је опет очекивано понашање, јер се контејнери тек покрећу, односно иницијално учитавају у меморију.

Интересантна статистика, међутим, примећује се да се ништа значајно није променило, чак и са овим порастом, односно да није утицало негативно на перформансе апликације тј. целог подсистема и *Docker*-а. Када су покренуте 2 инстанце, види се да се број практично дуплира. Са 3 инстанце, види се да се број скоро па утростручио, међутим, враћа се у нормалу, јер је сервис, одбијајући захтеве почео да трпи мање оптерећење. Тачније, узрок томе је отказивање базе података.

4.6 Време чекања

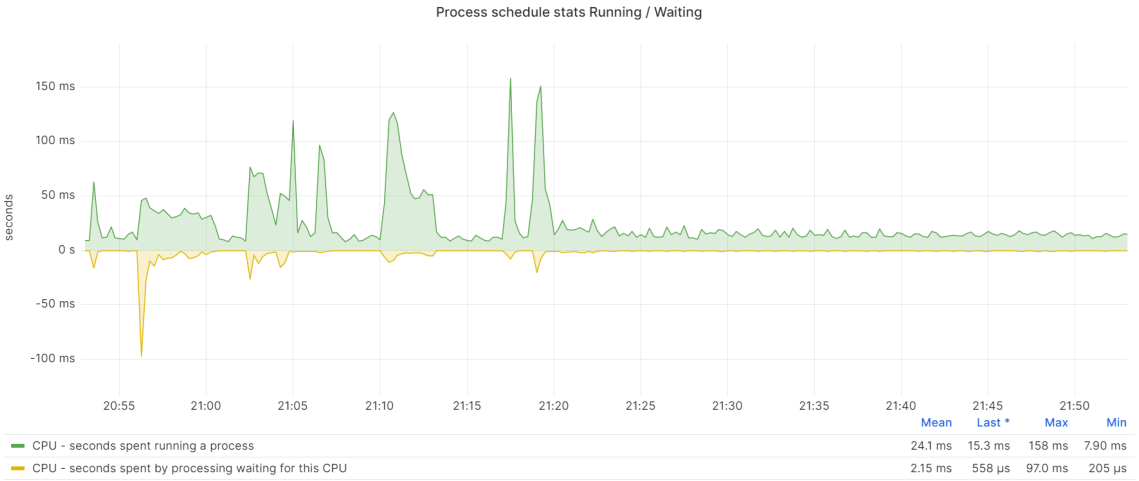


Слика 4.18: Пребацивање контекста

Пребацивање контекста представља поступак којим се процес који се тренутно извршава на процесору прекида, памте се његови параметри, а затим

се уместо њега покреће неки други процес и при томе се учитавају параметри другог процеса. (Marić, 2017)

Промена контекста је описана тако да језгро суспендује извршење једног процеса на процесору и наставља извршавање неког другог процеса који је претходно био суспендован. Пребацивање контекста је потребно за сваки прекид и сваки задатак који планер одабере. Пребацивање контекста може бити због мултитаскинга, руковања прекидима, промене режима корисника и језгра. Стопа прекида ће природно бити висока, ако постоји већи мрежни саобраћај или већи саобраћај на диску. Такође зависи од апликације која с времена на време врши системске позиве. Ако језгра CPU-а нису довољна за руковање оптерећењем нити које је креирала апликација, то ће такође довести до промене контекста. То није разлог за забринутост док се укупне перформансе не покваре. Очекује се да ће CPU извршити промену контекста. График који најближе може да прикаже које је време утрошено на извршавање процеса у односу на време потрошено на чекање. (Brazil, 2020)



Слика 4.19: Време које процеси проводе радећи, односно чекајући

Напомена: вредности испод нуле на y-оси посматрати као апсолутне вредности; на графику су приказане као негативне искључиво због прегледности.

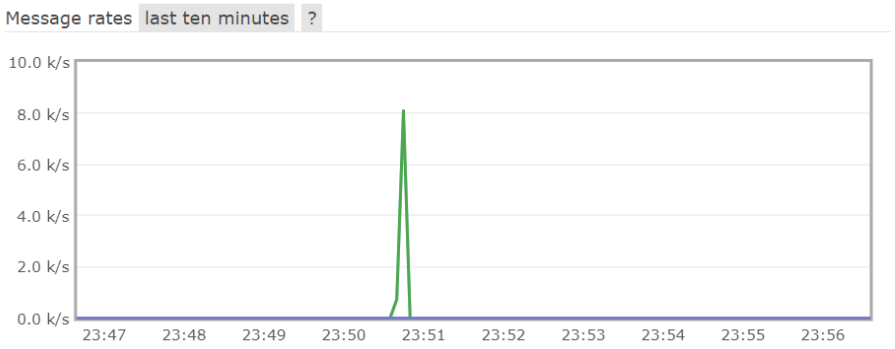
	min	max	mean	current
running	9.33	121	24.1	14.2
waiting	0.252	45.0	2.15	0.495

Табела 4.3: Време које процеси проводе радећи, односно чекајући

Број нити које су тренутно у стању рада (енгл. *running*) – ова метрика представља број нити које активно користе CPU. Број нити које су тренутно у стању „спреман" (енгл. *waiting*) – ова метрика представља број нити које чекају на ресурс или догађај пре него што могу да наставе са извршавањем.

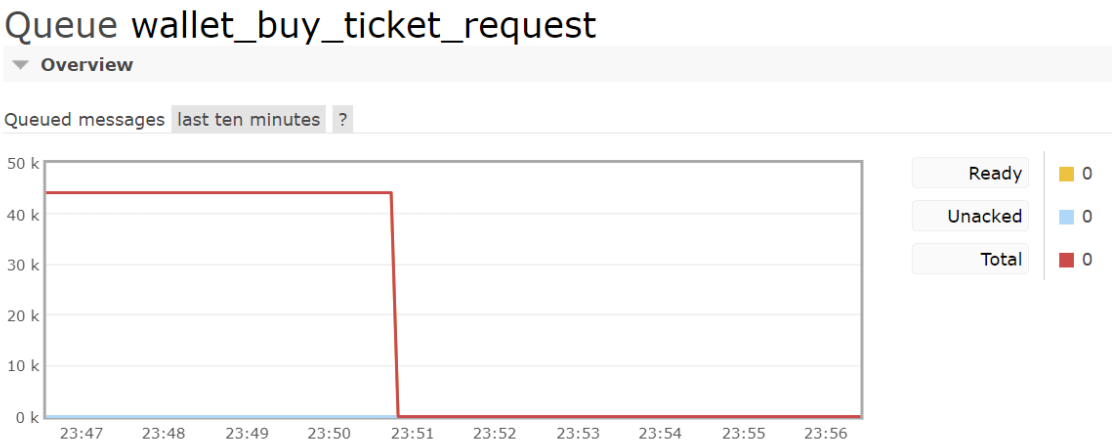
4.7 Брзина обраде порука – RabbitMQ

У наставку биће показано да, као у овом случају када уплата може асинхроно да прође, је оптерећење мање . *Wallet* сервис узима поруке брзином којом може да их обради. Да су захтеви били синхрони, они би после неког времена били одбијани, као што се то видело са бројем конкурентних корисника који захтевају одговор од сервиса одмах. Или би долазило до *timeout*-а, јер је сервис преоптерећен. Овако, поруке се скидају из реда и обрађују одређеном брзином, коју сам сервис може да поднесе. Такође, издвојен је посебан скуп нити (енгл. *thread pool*) са одређеним бројем нити који служи искључиво за обраду ових порука, на тај начин је обезбеђено да се не потроше све нити сервиса и да читав сервис не буде угрожен. Следе графици ситуације када је *Wallet* сервис био заустављен. У реду *wallet_ buy_ ticket_ request_ queue* било је нешто мање од 45 000 порука. Тренуци издвојени на графицима су када је сервис покренут и када је почео да скида и обрађује поруке.



Слика 4.20: Брзина обраде порука

Према званичном опису на графику који је приказан на веб конзоли која показује статистике за *RabbitMQ*, види се да је приказана брзина којом се скидају поруке из реда, односно када корисник пријави да је поруку прихватио. Тај број достиже 8000 порука у секунди.

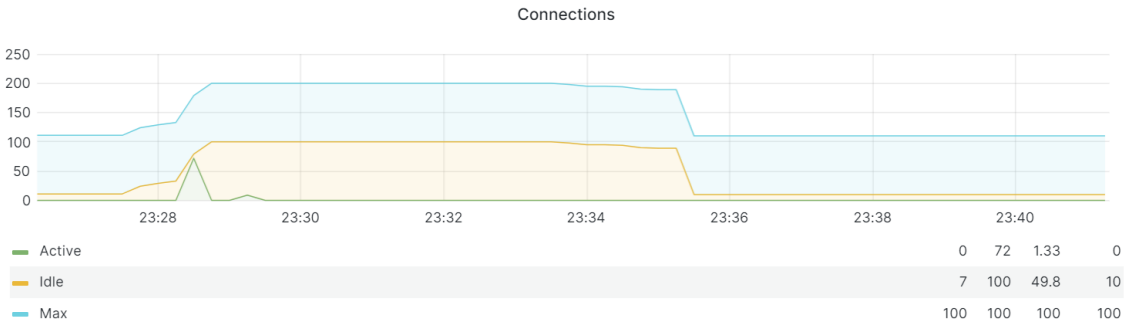


Слика 4.21: Број порука у реду *wallet_buy_ticket_request*

На овом графику означен је укупан број порука који се налазио у реду, пре него што је сервис почео да их скида. Као што се уочава, око 40 хиљада порука је обрађено, такорећи одмах, тј. кроз пар секунди. Томе, наравно сведочи и претходни график, који показује брзину од 8 хиљада порука у секунди.

4.8 Број конекција над базом

Прати се стање конекција над базом података и њихов број.



Слика 4.22: Приказ стања конекција на базу података

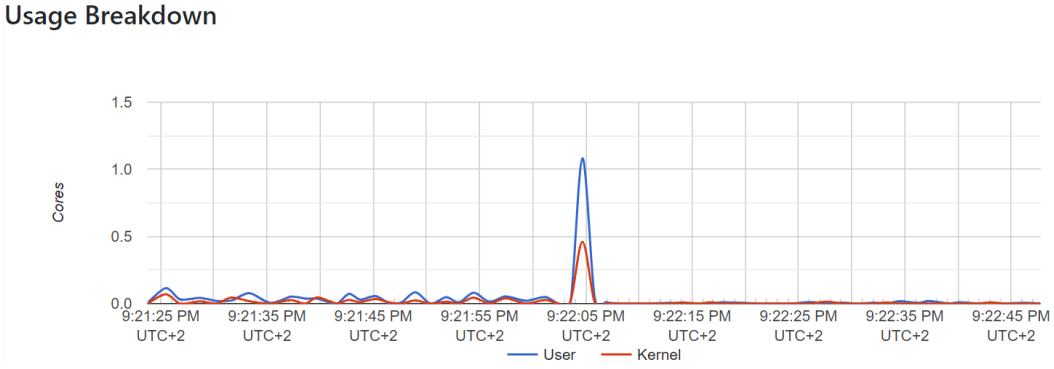
На овом графику приказан је број конекција над базом сервиса у тренуцима просечног, затим повећаног оптерећења. У табели се види просечан број заузетих конекција за овај тестни период, као и максимална вредност коју ова метрика достиже. И у тренуцима највећег оптерећења, није премашен број конекција већи од 100. А горња граница броја конекција која је дата на располагање сервису је чак 200.

	min	max	avg	current
active	0	72	1.33	0
idle	7	100	49.8	10
max	100	100	100	100

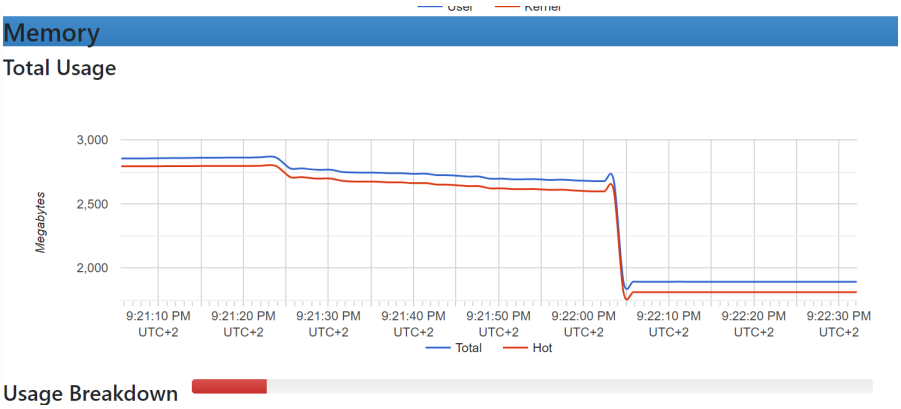
Табела 4.4: Број конекција на базу груписан по стањима

На графику се уочава да није велики број активних конекција, што може да завара и да наведе на то да не постоји повећана активност базе. Наиме активне конекције су оне којима су тренутно идентификовани покренути упити; у извесном смислу, ово је заиста колико конекција може активно да ради у исто време. Када би био активан велики број конекција континуирано и то се видело на графику, то би значило да велики број конекција чека над базом на извршење неких упита и да је узалуд заузето. Овде се не приказује тако нешто, што је добар показатељ. Неактивне или беспослене (енгл. *idle*)

конекције су оне које су отворене за базу података (већина оквира то ради и одржава конекције у животу, нешто слично као скупови нити), које се не користе тренутно, али свака потреба за новом конекцијом узима једну од ових конекција и претвара је у активну. На графику се уочава значајан пораст ових конекција, не и активних. Разлог томе је јер се све одвија превише брзо, а упити над базом не трају дуго, односно сама активност сваке од конекција је јако кратка. Постоји само један моменат на графику где су конекције активне и то између 50 и 100 њих, али толико кратка и брза да не може бити забележена као метрика кроз дужи временски период. Такође, број неактивних конекција је порастао на 100 у тренуцима великог оптерећења (не би биле креиране да нису биле потребне) што показује да постоји активност.



Слика 4.23: Стање CPU-а контејнера базе



Слика 4.24: Стање RAM меморије контејнера базе

Глава 5

Закључак

У оквиру рада представљена је примена и рад AMQP-а і *Docker*-а. Прва као технологија за комуникацију међу микросервисима, друга као део одржавања и покретања апликација као контејнера у развојном и продукционом окружењу.

AMQP је представљен кроз *RabbitMQ*, где су истакнуте предности које са собом носи у односу на друге сличне софтверске компоненте/брокере. А превасходно предности које AMQP омогућује у осносу на стандардну, синхрону комуникацију. Сва 3 микросервиса *EvEnter* апликације комуницирају искључиво асинхроно коришћењем поменутог протокола. Овим путем је показано како овај начин комуникације у различитим реалним ситуацијама функционише у пракси, да у брзини не заостаје за синхроном REST комуникацијом, а да у односу на REST доноси бенефите. Оваква комуникација је омогућила да се поновно слање врши без проблема, да се рационалније и боље користе постојећи ресурси апликације у виду броја нити, односно броја акција које могу бити обављене, са смањеним временом чекања.

Када говоримо о скалабилности, на чему је и био главни акценат читавог рада, без *Docker*-а би све ишло много теже. Показано је колико *Docker* олакшава функционисање и рад програмеру приликом развоја и рада са микросервисима. А наравно, оно што је главно, како се на лак и једноставан начин могу у само неколико минута повећати капацитети апликације и тако увећати број корисника које апликација може да опслужи. Те на овај начин олакшава рад како програмерима тако и *DevOps* инжењерима. Такође игра главну улогу и у скалирању самог брокера, где би, услед великог оптерећења, у неколико минута могла бити покренута једна или више инстанци

самог *RabbitMQ* брокера. У том случају би се једна растеретила, а проток информација и рад апликације био бољи.

Након извршеног тестирања оптерећења апликације кроз разне случајеве употребе, са великим бројем конкурентних корисника изложене су детаљне статистике и резултати тестова. Ови подаци и графици су показали како у различитим сегментима процесор, меморија, број конекција над базом, одзив, итд, реагују и раде, односно како се апликација понаша у произвољно креираном продукционом окружењу. Наравно, показано је и да није могуће само произвољно повећавање капацитета апликације или броја контејнера, односно вертикално и хоризонтално скалирање, без претходног мониторинга, праћења стања ресурса апликације и знања о понашању одређених делова кода. Већ је неопходно скалирање вршити са разумевањем и свешћу о томе који су ресурси угрожени, на који начин и због чега, те је контрола рада апликације пре „слепог" скалирања неопходна. Такође, јако је битно напоменути, да о томе како ће се неки микросервис скалирати треба водити рачуна приликом самог развоја. И посебно обратити пажњу на паметно коришћење ресурса апликације, на њихово ослобађање и дељење и на то да се из нехата, у развоју, не створи уско грло. Уколико се у оквиру апликације због саме логике створи уско грло, може се доћи у ситуацију да ни додавање нових контејнера, који би потенцијално требало да увећају број конкурентних корисника који могу да користе апликацију, у томе не помаже и да онда имамо систем који није скалабилан.

Сигурно да и апликација која је развијена има мана и места за побољшање, међутим, за потребе овог рада је и више него довољно послужила сврси и осликала понашање у реалном окружењу. Праћена и што је најважније у овом контексту, скалирана на једноставан начин. Неки наредни кораци у испитивању скалабилности апликације *EvEnter* били би издвајање базе података на одвојени сервер са већим капацитетом и могућа промена пословне логике уколико се утврди да операције над базом могу да се оптимизују. Након тога, даље хоризонтално скалирање био би циљ.

Глава 6

Литература

Литература

Anna, J. (2019). *Open source monitoring stack: Prometheus and grafana.* Retrieved 25/06/2023, from https://www.bizety.com/2019/01/25/open-source-monitoring-stack-prometheus-and-grafana/

Brazil, B. (2020). *Cpu scheduling metrics from the node exporter.* Retrieved 25/06/2023, from https://www.robustperception.io/cpu-scheduling-metrics-from-the-node-exporter/

Carr, B. (2023). *Tx - row lock contention.* Rampant. Retrieved 25/06/2023, from http://www.rampant-books.com/art_enq_TX_row_lock_contention.htm

Deshpande, A. (2020). Solving distributed transaction management problems in microservices architecture using saga. Retrieved 25/06/2023, from https://developer.ibm.com/articles/use-saga-to-solve-distributed-transaction-management-problems-in-a-microservices-architecture/

Garvie, L. (2022). *Pub-sub vs. message queues.* Retrieved 25/06/2023, from https://www.baeldung.com/pub-sub-vs-message-queues

Hazelcast. (2021). *Time series database.* Retrieved 25/06/2023, from https://hazelcast.com/glossary/time-series-database/

Kanjilal, J. (2021). *Dependency injection vs inversion of control.* Retrieved 25/06/2023, from https://www.codeguru.com/csharp/dependency-injection-vs-inversion-control/

Karim, Z. (2019). *Take the confusion out of docker, vms, and*

microservices. Retrieved 25/06/2023, from `https://developer.ibm.com/articles/breaking-down-docker-and-microservices/`

Mahmoud, Q. (2004). *Comparing messaging standards: Jms vs amqp.* Retrieved 25/06/2023, from `https://www.oracle.com/technical-resources/articles/java/intro-java-message-service.html`

Marchioni, F. (2022). *Comparing messaging standards: Jms vs amqp.* Retrieved 25/06/2023, from `http://www.mastertheboss.com/jbossas/jboss-jms/comparing-messaging-standards-jms-vs-amqp`

Marić, M. (2017). *Operativni sistemi* (2nd ed.). Univerzitet u Beogradu – Matematički fakultet.

Poulton, N. (2020). Docker deep dive. *Packt Publishing.*

RedHat. (2023). *What is docker?* Retrieved 25/06/2023, from `https://opensource.com/resources/what-docker`

Team, I. C. (2021). *What is service-oriented architecture (soa)?* Retrieved 25/06/2023, from `https://www.ibm.com/topics/soa#toc-soa-vs-mic-BjTfju28`

TechTarget. (2018). *Advanced message queuing protocol (amqp).* Author. Retrieved 25/06/2023, from `https://www.techtarget.com/whatis/definition/Advanced-Message-Queuing-Protocol-AMQP`

Биографија аутора

Урош Познан (*Београд, 20. септембар 1994.*) завршио је основну и средњу школу у Горњем Милановцу. Након завршене гимназије „Таковски устанак" (друштвено-језички смер, билингвално одељење) уписује Математички факултет 2013. године. Дипломирао је на студијском програму „Математика", модул „Рачунарство и информатика" у јулу 2019. године. Након тога уписује мастер студије на истом модулу и смеру.

Каријеру започиње у септембру 2017. године четворомесечном праксом у фирми *EzDerm*. Затим од децембра 2018. године започиње рад у ИТ индустрији као *Java* програмер у фирми *Mozzart d.o.o*, где и данас ради.