УНИВЕРЗИТЕТ У БЕОГРАДУ
МАТЕМАТИЧКИ ФАКУЛТЕТ

Стеван Драговић

ОБРАСЦИ У МИКРОСЕРВИСНОЈ АРХИТЕКТУРИ И ЊИХОВА ИМПЛЕМЕНТАЦИЈА У РАДНИМ ОКВИРИМА .NET И NODE.JS

мастер рад

Београд, 2026.

Ментор:

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

Чланови комисије:

проф. др. Милена Вујошевић Јаничић, редовни професор
Универзитет у Београду, Математички факултет

проф. др. Александар Картељ, ванредни професор
Универзитет у Београду, Математички факултет

Датум одбране: _____

Породици на неизмерној подрши

Наслов мастер рада: Обрасци у микросервисној архитектури и њихова имплементација у радним оквирима *.NET* и *Node.js*

Резиме: Овај рад представља приказ пројектних образаца коришћених у микросервисној архитектури, са имплементацијом у радним оквирима *.NET* и *Node.js*. Рад обухвата теоријски увод где су уведени појмови софтверске архитектуре и пројектних образаца, као и упознавање са радним оквирима *.NET* и *Node.js*. Затим су обрађени обрасци у микросервисној архитектури: Прекидач, *API* композиција, *Transactional Outbox*. Уз сваки од образаца је дато објашњење које проблеме решавају код микросервисне архитектуре. Приказане су и имплементације образаца у *.NET* и *Node.js*, где су се поредиле архитектуре и комплексности имплементација. На крају, кроз образац *API* композиција урађена је анализа перформанси радних оквира.

Кључне речи: архитектура, обрасци, перформансе, микросервиси, програмски језици, Прекидач, *API* композиција, *Transactional Outbox*

Садржај

Глава 1

Увод

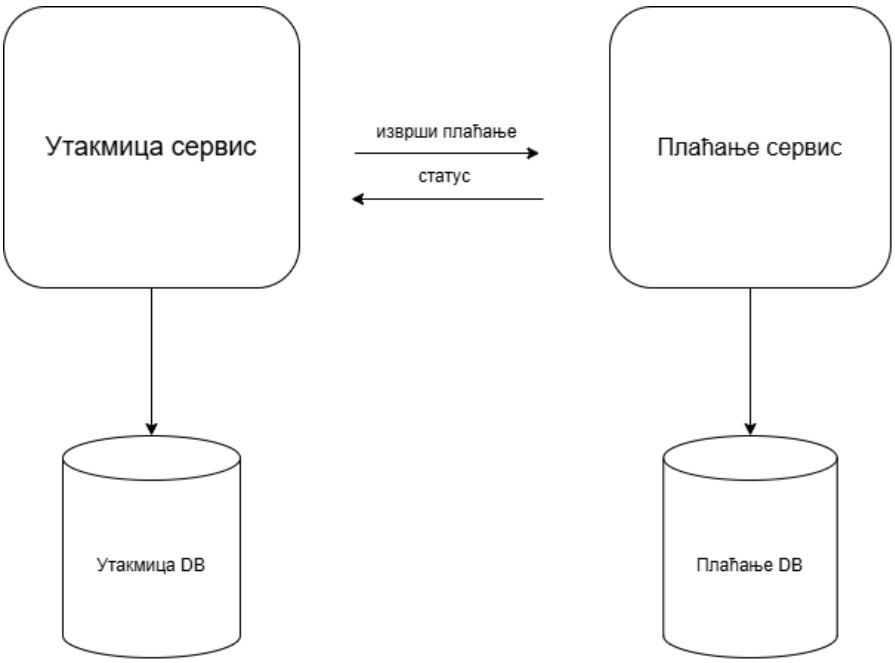
У овом раду биће приказане области у развоју софтвера које омогућују изградњу поузданих и ефикасних система, у виду микросервисне архитектуре и њених образаца. Микросервисна архитектура је једна од стандардних архитектура приликом развоја комплекснијих система, где уместо великог система постоји подела система на више засебних апликација, које раде независно једна од друге. Због комплексности коју носи микросервисна архитектура, пројектовање и имплементација таквих система представља изазов. Као одговор на те изазове, биће представљени обрасци микросервисне архитектуре, као и аналогија са обрасцима који се користе код монолитних апликација.

Микросервисна архитектура омогућава разноврсност алата за имплементацију микросервиса, па се за различите микросервисе могу користити различити алати. У овом раду ће обрасци бити представљени помоћу радних оквира *.NET* и *Node.js*. *Node.js* и *.NET* су савремени радни оквири који се користе за развој апликација на страни сервера и чест су избор алата за имплементацију микросервиса. Због разлике у парадигми којој припадају се користе у различитим случајевима, где радни оквир *.NET* почива на програмском језику *C#*, који спада у групу објектно оријентисаних програмских језика. Радни оквир *Node.js* је заснован на програмском језику *JavaScript*, који спада у групу скрипт језика. Кроз тестирање перформанси система биће извршена упоредна анализа два радна оквира.

1.1 Приказ система

У овом раду биће приказан развој система на микросервисној архитектури, који служи за приказ утакмица, као и могућност куповине улазница за утакмице. Систем се састоји из два микросервиса, микросервиса за утакмице и микросервиса за плаћање. Приликом куповине улазнице микросервис за утакмице комуницира са микросервисом за плаћање, који извршава трансакцију и шаље одговор назад о успешности трансакције.

Овако дизајниран систем је прикладан за приказивање најчешће коришћених образаца код микросервиса, због изазова који се јављају. Неки од изазова укључују конзистентност и интегритет двеју база података које чувају вредности са истом семантиком, као и перформансе система за утакмице за које постоје велика интересовања.



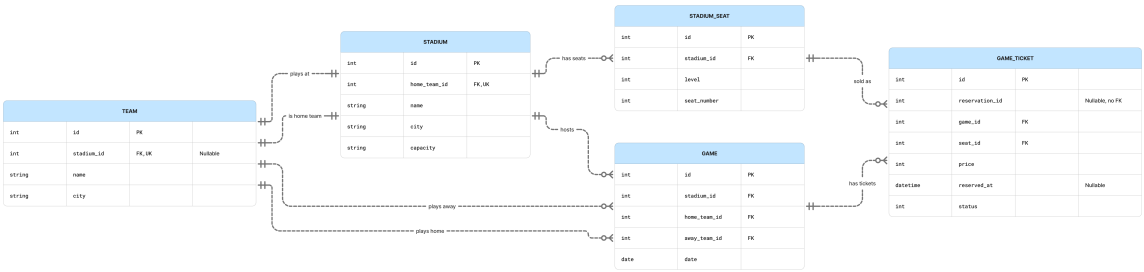
Слика 1.1: Архитектура система

Структура оба микросервиса ће бити приказана кроз њихове приступне тачке, као и кроз базе података са којима комуницирају. Табела 1.1 описује приступне тачке дефинисане код микросервиса за утакмице.

Табела 1.1: Приступне тачке микросервиса за утакмице

Метод	**Приступна тачка**	**Опис**
GET	`/get-all-games`	Приказује листу тренутних утакмица
GET	`/games/get-game-by-id/\{id\}`	Приказује детаље специфичне утакмице
GET	`/games/get-game-tickets/\{gameId\}`	Доступне карте
GET	`/games/get-seat-info/game/\{gameId\}/seat/\{seatId\}`	Приказује информацију о седишту
GET	`/games/get-ticket-info/\{gameTicketId\}`	Приказује детаље о карти
POST	`/game/ticketpay`	Извршавање плаћања карте

Што се тиче базе података, модел ентитета и односа је приказан на Слици 1.2.



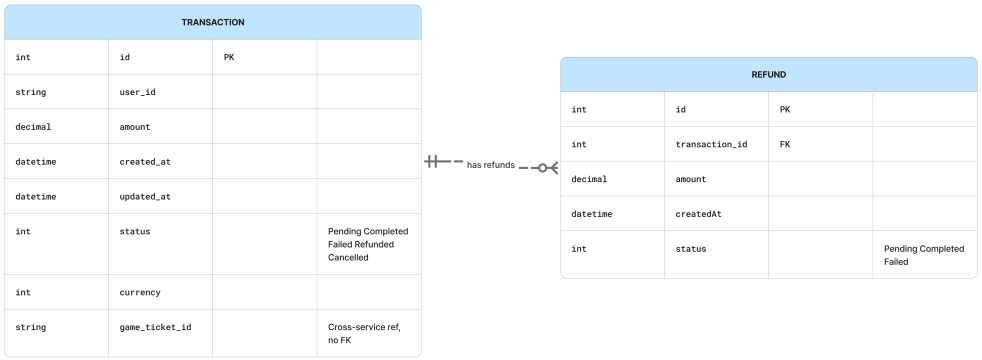
Слика 1.2: Модел ентитета и односа за утакмице

Код микросервиса за плаћање приступне тачке су описане у Табели 1.2

Табела 1.2: Приступне тачке микросервиса за плаћање

Метод	**Приступна тачка**	**Опис**
POST	`/payment/ticketpay`	Извршавање плаћања
GET	`/payment/transaction-info/gameTicket/\{gameTicketId\}/transaction/\{transactionId\}`	Информације о трансакцији

Модел ентитета и односа код микросервиса за плаћање је приказан на Слици 1.3.



Слика 1.3: Модел ентитета и односа за плаћање

Глава 2

Технологије и алати

За имплементацију микросервиса биће коришћени радни оквири *.NET* и *Node.js*. Такође, за услуге база података биће коришћена база *PostgreSQL* [15] а за редове порука *RabbitMQ* [16].

PostgreSQL представља релациону базу података која подржава трансакциони модел (*ACID*), што ће бити коришћено у овом раду ради постизања конзистентности података. Поред примитивних типова података, подржава и рад са објектно-релационим типовима, што је сврстава међу најпопуларније базе података. *RabbitMQ* представља брокер за поруке, где пошиљалац преко редова порука шаље поруку примаоцу који је обрађује. Користи се као средњи слој у комуникацији између микросервиса, где за разлику од *HTTP* позива нема директне зависности. Овим се смањује директна комуникација између микросервиса и омогућује боља отпорност на грешке.

За радни оквир *.NET* је потребно инсталирати следеће пакете

- *.NET* 9 верзија - радни оквир за градњу и покретање апликација [13]
- *Entity Framework Core* - библиотека за рад са базама података [4]
- *MassTransit* - библиотека за рад са редовима порука [11]
- *Polly* - библиотека која садржи алате за прављење безбедних система [14]

- *YARP* - библиотека за прављење обрнутог проксија [19]

Пројекат у *.NET* се иницијализује преко команде ***`dotnet new`*** при чему се онда наводи тип пројекта, назив пројекта и библиотека. Како пројекат за систем има 4 слоја, презентациони слој ће бити ***`WebAPI`*** пројекат, док остали ће бити ***`ClassLibrary`*** пројекти. На пример ако треба да се иницијализује презентациони слој за утакмице, користи се наредба

```
dotnet new webapi -n Game.API -f net9.0
```

а приликом иницијализације ***`ClassLibrary`*** пројекта уместо ***`webapi`*** користи се ***`classlib`***.

За инсталирање пакета користи се команда ***`dotnet add package`***, и за *PostgreSQL* пакети се инсталирају на следећи начин.

```
dotnet add package Npgsql.EntityFrameworkCore.PostgreSQL
dotnet add package Microsoft.EntityFrameworkCore
dotnet add package Microsoft.EntityFrameworkCore.Design
```

док се *MassTransit* пакети инсталирају преко наредбе

```
dotnet add package MassTransit
dotnet add package MassTransit.RabbitMQ
dotnet add package MassTransit.EntityFrameworkCore
```

У овом сценарију база података *PostgreSQL* и ред за поруке *RabbitMQ* су инсталирани преко *Docker*-а [3].

- За покретање *PostgreSQL* користи се наредба

```
docker compose up -d postgres
```

- За покретање *RabbitMQ* користи се наредба

```
docker compose up -d rabbitmq
```

- Приступна тачка за *RabbitMQ* се налази на порту 5672 и може се приступити преко адресе *localhost:5672*

Након иницијализације пројекта, списак библиотека које се користе за тај пројекат се налазе у датотеци *.csproj* од тог пројекта.

Такође, ако је потребно да пројекат користи ресурсе другог пројекта, може се извршити референца на пројекат преко наредбе у терминалу која почиње са *dotnet add*.

```
dotnet add Game.API/Game.API.csproj reference
Game.Application/Game.Application.csproj
```

У овој наредби наводи се да *Game.API* пројекат зависи од *Game.Application* пројекта.

За коришћење *Node.js* пројекта, потребно је инсталирати *Node.js* са адресе *https://nodejs.org/*. Затим се иницијализује *Node.js* пројекат преко наредбе

```
npm init -y
```

У пројекту се користи *Express.js* [5] библиотека, као и програмски језик *TypeScript* [18], који представља природно проширење *JavaScript* језика. Да би се *Express.js* иницијализовао са *TypeScript*-ом, користе се следеће наредбе

```
npm install express
npm install -D typescript ts-node @types/node @types/express
```

Уз то, користиће се такође *PostgreSQL* база података, и у складу са тим библиотека ***TypeORM*** [17], која служи за објектно релационо мапирање, на сличан начин као и библиотека ***Hibernate*** у програмском језику ***Java***. Оно што је потребно инсталирати за подршку за базу података су следећи пакети

- *pg* - *PostgreSQL* драјвер
- *typeorm* - библиотека за објектно релационо мапирање
- *reflect-metadata* - подршка за *TypeORM* декораторе

и ти пакети се инсталирају уз помоћ наредби

```
npm install pg
npm install typeorm
npm install reflect-metadata
```

За *RabbitMQ* се користи пакет ***amqplib*** [1], који се инсталира уз помоћ наредбе

```
npm install amqplib
```

а за обрнути прокси се користи библиотека ***http-proxy-middleware*** [9]

```
npm install http-proxy-middleware
```

За коришћење *PostgreSQL* и *RabbitMQ* користи се такође docker на начин коришћен у *.NET* пројекту.

Глава 3

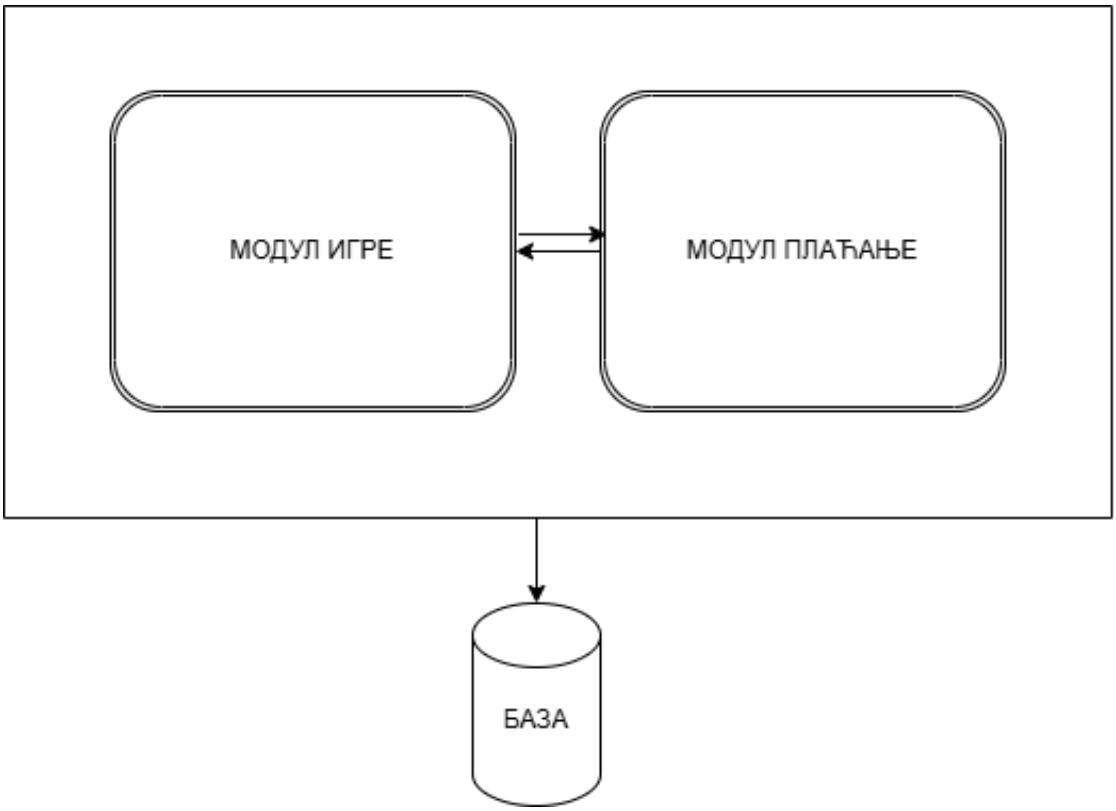
Софтверска архитектура

У развоју софтвера, софтверска архитектура представља структуру софтверског система посматрану на апстрактном нивоу [22]. Садржи главне компоненте система, као и везе између компоненти, и тиме се дефинише развојни план система. Одговарајући избор архитектуре, који се доноси на самом почетку развојног циклуса, остварује бројне бенефите, као што су одрживост, перформансе, скалабилност, и безбедност система. Лоше одлуке које се донесу на почетку развојног циклуса имају за последицу да неке од поменутих метрика неће бити остварене.

Приликом пројектовања софтвера тежи се испуњењу неких од битних карактеристика као што су флексибилност и проширивост. Флексибилност архитектуре се везује за поновну употребу одређених компоненти у систему, што даје погодност да се постојећи систем прилагоди променама које се дешавају. Проширивост архитектуре је својство додавања нових компоненти на тренутни систем без мењања постојећих компоненти. Имајући ове карактеристике у виду, током времена су се развиле разне архитектуре које настоје да их обезбеде, при чему свака нуди одређене предности и мане.

Једна од најраспрострањенијих архитектура је монолитна архитектура [24]. Постоји више начина за дефинисање монолитне архитектуре, и најједноставнији начин за дефинисање монолита се приказује кроз јединицу испоручивања, где се сва функционалност система дефинише кроз један процес. Ради боље организације система, монолит се може приказати кроз разне варијације.

Модуларни монолит је варијација монолита у којој се један процес распоређује у разне модуле, остварујући висок ниво кохезије у систему [23]. Једноставност овакве архитектуре, за разлику од дистрибуираних система, доноси погодности као што су лакше праћење и одржавање система, тестирање и поновно коришћење кôда. Постоје одређени изазови код оваквог система, и један од њих је дељена база података за све модуле. База података нема раздвојеност каква постоји на нивоу кôда, што доводи до проблема уколико постоји потреба раставити монолит и прећи на микросервисну архитектуру.



Слика 3.1: Модуларна монолит архитектура

За решавање поменутих проблема користи се сервисно-оријентисана архитектура. Сервисно оријентисана архитектура је приступ дизајну у коме више сервиса сарађује на обезбеђивању одређеног крајњег скупа функционалности, где сервис овде представља засебан процес оперативног система [24]. Комуникација међу сервисима се врши позивима преко мреже, а не путем позива метода унутар граница процеса. Овакав приступ архитектури има за циљ уна-

прећење вишекратне употребе софтвера, где две или више апликација могу да користе исте сервисе.

Еволуцију сервисно-оријентисане архитектуре представља архитектура заснована на микросервисима. Иако имају заједнички концепт дељења апликације, микросервиси се разликују од сервисно-оријентисане архитектуре у погледу величине система, где су микросервиси ситније гранулирани системи [24]. Поред величине система, разлика се огледа у начину и у протоколу комуникације: код микросервиса постоји више облика комуникације, за разлику од сервисно-оријентисане архитектуре где се типично користи стандардизован протокол.

Особина микросервиса која омогућује многе погодности јесте слаба повезаност. Та особина обезбеђује промену једног микросервиса без потребе да се мења други микросервис, а то је могуће само ако постоје добри и стабилни уговори међу микросервисима. Са таквом структуром где је нагласак на граници и уговорима међу микросервисима, омогућено је да се употребљавају различите технологије за различите микросервисе, што даје погодност развојним тимовима да користе технологије у којима имају више искуства.

Глава 4

Обрасци пројектовања софтвера и примене на микросервисну архитектуру

У развоју софтвера важна је употреба одговарајућих техника које омогућују добар квалитет програмског кôда. Квалитет програмског кôда се, поред осталог, огледа и у погледу перформанси, скалабилности и одрживости кôда. Како свака апликација тежи да постигне поменуте метрике, долази се до тога да се многе технике користе на исти или сличан начин у различитим апликацијама, што доводи до понављања кôда. Поновљени део кôда се издваја у засебан део кроз функције или модуле. Ако су два дела кôда довољно слична у имплементационом погледу, при чему је разлика само у ентитетима, онда се може говорити о решењу које је применљиво на неку класу проблема, које се сматра обрасцем за пројектовање [22].

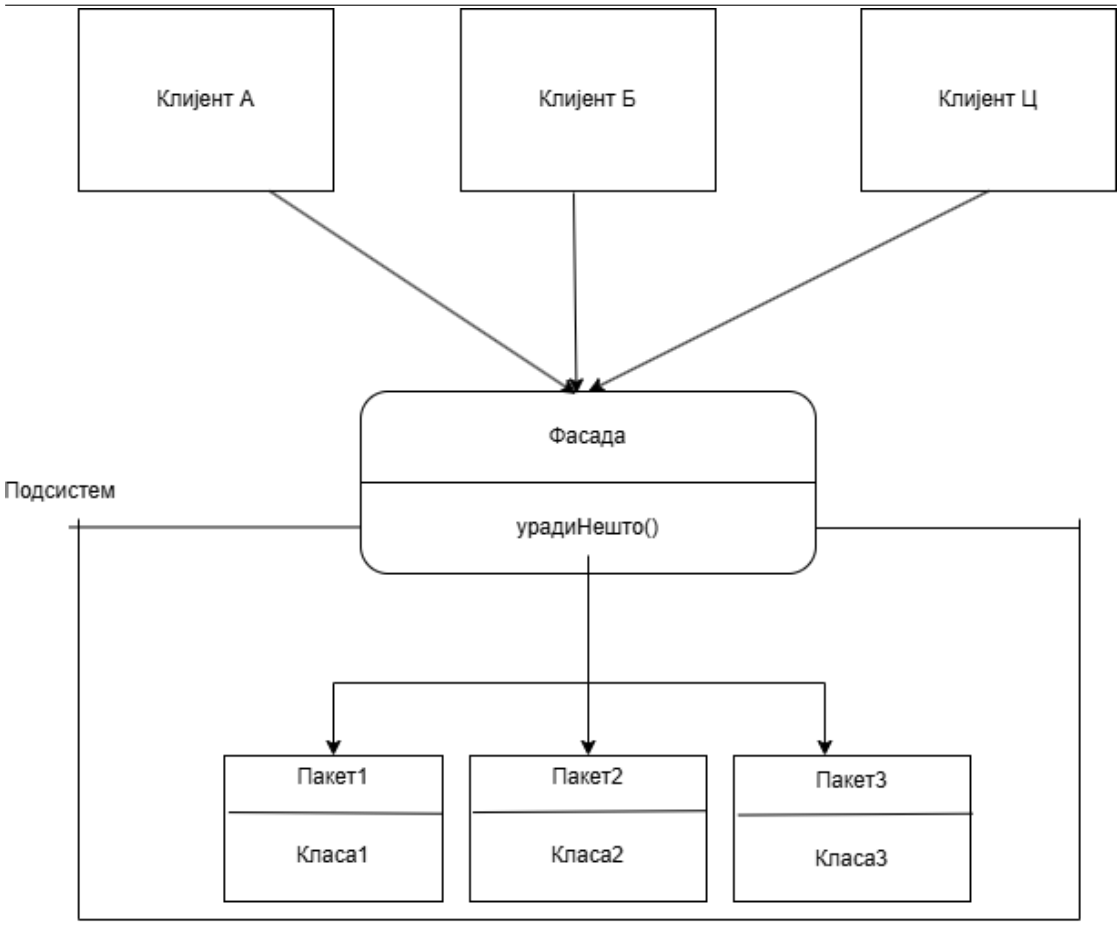
Обрасци за пројектовање софтвера се најчешће спомињу код монолитних апликација, где се примењују над компонентама у оквиру једне јединице. Најкоришћенији и најпознатији обрасци су такозвани *GOF (Gang of Four)* обрасци, који представљају скуп решења које се користе у дизајну и развоју софтвера, и та решења се деле у три категорије: градивни обрасци, структурни обрасци и обрасци понашања [20]. Градивни обрасци представљају решења за прављење нових објеката, где се на апстрактан начин приказује прављење

нових објеката као и повезаности направљених објеката. Структурни обрасци се баве организацијом објеката и класа у нове целине, при чему се тежи да те целине буду ефикасне и флексибилне. Обрасци понашања се представљају кроз алгоритме који представљају везу између објеката, где сваки објекат има своју улогу. Иако то можда није очигледно на почетку, многи од најзначајнијих образаца за пројектовање код монолитних апликација имају своје примене и у развоју микросервиса. Разлог за таквим нечим је то што дистрибуиране апликације је могуће посматрати као монолитне апликације, само на већој скали.

Код микросервисних апликација постоје још додатни проблеми везани за саму природу комуникације између микросервиса. Комуникација се најчешће одвија преко *HTTP, gRPC* захтева и редова порука, уместо функција као код монолитних апликација, и ти проблеми се огледају у виду кашњења позива. Решавање таквих проблема укључују другачије технике, као што је на пример кеширање, али на саму структуру микросервиса и логику која се користи, добар део образаца који се користе код монолитних апликација је могуће применити.

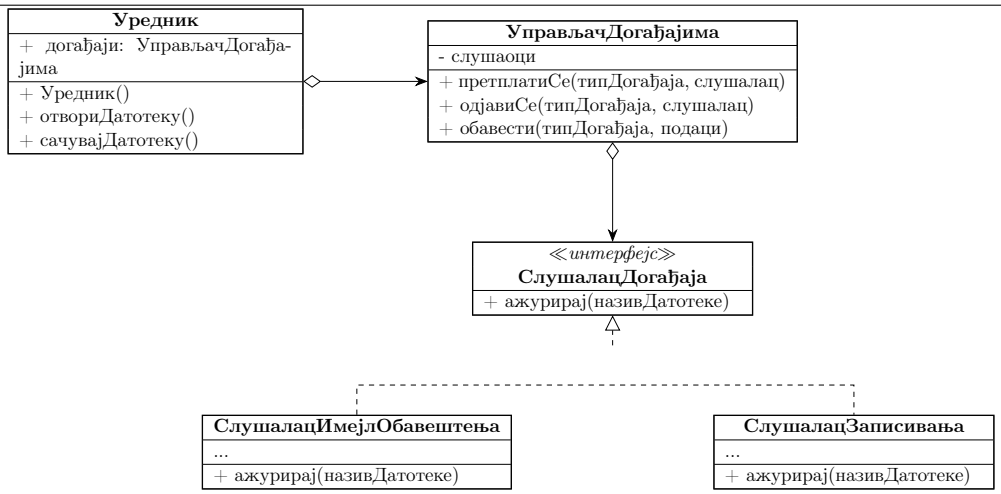
Један од образаца који има примене у монолитној и микросервисној архитектури је структурни образац Фасада (енгл. *Facade*). Код монолитних апликација користи се у поједностављењу комплексности неке библиотеке, где уместо коришћења више класа директно се креира класа фасада која управља поменутим класама из библиотеке. Разлог за овим лежи у чињеници да комплекснији подсистеми, који могу да се мењају, не треба да буду организовани у један кôд, чиме би се отежало одржавање система где би се уместо промене подсистема мењао цео кôд. Фасада се код микросервисних апликација користи ради поједностављења захтева између микросервиса. Уместо вишеструких захтева ка осталим микросервисима да би се извршио неки задатак, креира се једна фасада која ради наведено. Идеја која је настала у дистрибуираним системима а иза које стоји образац фасада јесте *API* капија (енгл. API Gateway).

Код микросервиса једна од најкоришћенијих архитектура је архитектура вођена догађајима, која омогућава лабаву спрегнутост између микросервиса који међусобно комуницирају. Образац који концептуално стоји иза ове архи-



Слика 4.1: Образац фасада

тектуре јесте образац Посматрач (енг. *Observer*), и код монолитних апликација се сврстава међу обрасце понашања. Идеја код оваквог обрасца јесте увођење механизма претплате уз помоћ којег се обавештавају објекти о догађајима које посматрају. Идеја је да постоје две улоге: улога субјекта издавача и улога посматрача. Субјекат обавештава посматрача о догађају који се десио, док посматрач прати стање догађаја, уз помоћ механизма претплате који се издваја у посебну класу. Пример је приказан на слици 4.2.



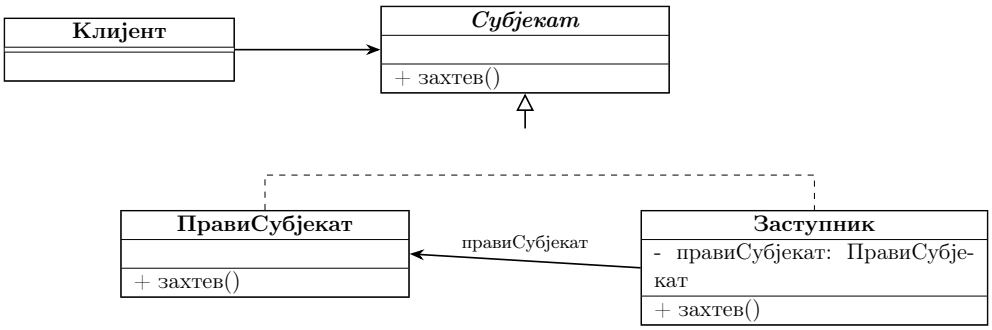
Слика 4.2: УМЛ дијаграм класа за образац Посматрач.

Слика приказује пример класе *Уредник*, која поред конструктора и метода које су везане за понашање класе, *отвориДатотеку()* и *сачувајДатотеку()*, као и поље и догађаји типа *УправљачДогађајима*. *УправљачДогађајима* класа се понаша као субјекат, која уз помоћ метода *претплатиСе()*, *одјавиСе()* чува информације о посматрачима догађаја из класе Уредник, и обавештава их преко методе обавести, након чега посматрачи зову метод ажурирај, где извршавају неку радњу након обавештења о догађају од стране субјекта издавача.

У микросервисној архитектури приликом коришћења обрасца Посматрач, уместо функцијских позива користе се редови порука, где издавач шаље поруку, а посматрач добија поруку. *Pub/Sub* образац, који је направљен према обрасцу посматрач, овде највише служи да не мора за сваког посматрача да постоји засебан ред порука већ постоји један ред порука за све посматраче.

Још један образац, који спада у категорију структурних образаца, и има примену у микросервисној архитектури јесте образац *Proxy*. Циљ иза овог обрасца је да се ограничи коришћење објекта који троши велику количину ресурса, без да се иницијализује сваки пут када се користи. Образац сугерише да се направи нова *Proxy* класа са истим интерфејсом као и класа за чији објекат постоји потреба да се уведе контролисано коришћење, и бенефит таквог приступа јесте да преко *Proxy* класе се извршавају методе без мењања

оригиналне класе.



Слика 4.3: УМЛ дијаграм класа за образац Proxy.

Код микросервисних апликација, споменут је образац *API Gateway*, који поред тога што примењује образац Фасада у ситуацијама где за један клијентски позив зове више микросервиса за извршавање логике иза које се крије интерна комплексност више микросервиса, такође имплементира образац Proxy, где клијент не зна за постојање микросервиса те само комуницира са *Gateway* односно *Gateway* контролише приступ микросервисима.

Глава 5

Радни оквири *Node.js* и *.NET*

У овом поглављу биће представљени радни оквири *.NET* и *Node.js* кроз теоријски увод и објашњење начина њиховог извршавања. Биће извршена упоредба модела на којима се заснивају, као и разлике у архитектури апликација развијених у поменутим радним оквирима. Кроз поменуте разлике биће истакнуте различите парадигме на којима почивају радни оквири.

5.1 *Node.js*

Node.js је *JavaScript* платформа за развој веб апликација на страни сервера, која омогућава коришћење истих алата и парадигми за софтвер на страни сервера и на страни клијента. Базирана је на програмском језику *JavaScript* и омогућава извршавање *JavaScript* програма изван веб прегледача. Засновано је на Гугловој *V8* машини, која прима *JavaScript* програм и извршава се у веб прегледачу. *V8* машина је написана у програмском језику *C++* и карактерише је портабилност на разним оперативним системима. Иако се *JavaScript* сматра интерпретираним језиком, *V8* машина користи *JIT* компилацију како би се убрзало извршавање програма.

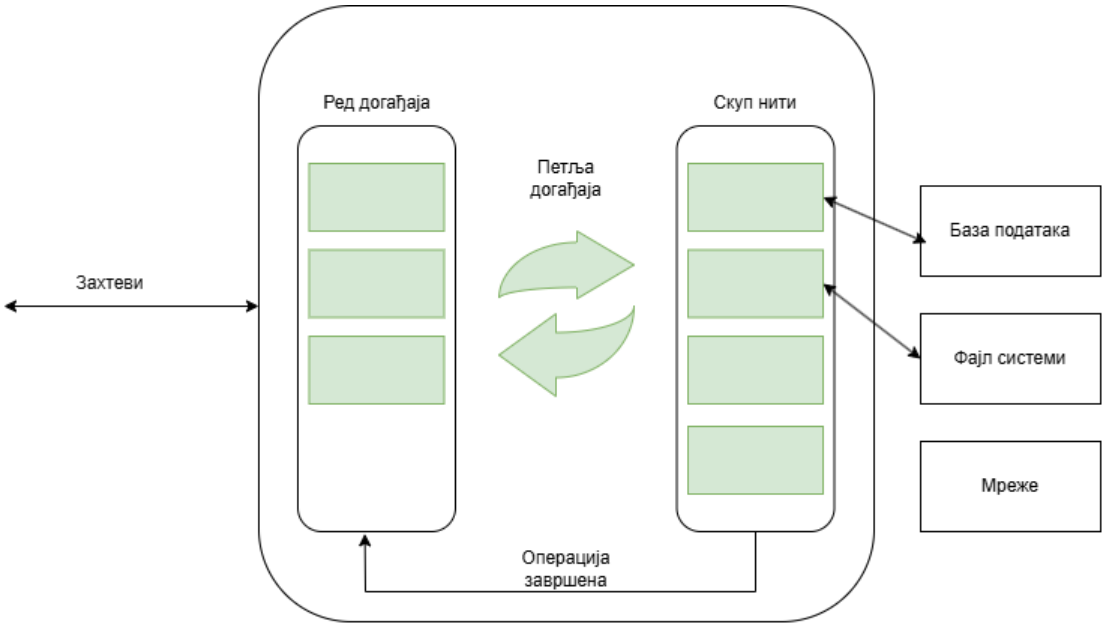
За разлику од компилираних језика који користе више нити ради скалирања апликација, *Node.js* користи једнонитни модел асинхроног програмирања вођен догађајима [21].

Такав модел се састоји од неколико делова: реда догађаја, скупа нити и

петље догађаја која обрађује потребне операције.

Принцип рада се своди на следећи начин:

1. Када сервер прими захтев од клијента, тај захтев се смешта у ред догађаја.
2. Уколико догађај не садржи блокирајућу операцију, окружење ће обрадити тај захтев и одговор послати клијенту
3. Уколико догађај садржи блокирајућу операцију, окружење додаје догађај скупу нити како би била реализована блокирајућа операција.
4. Када се изврши блокирајућа операција радна нит припрема одговор и шаље петљи за догађаје, која одговор прослеђује клијенту.



Слика 5.1: Петља догађаја

За развијање апликација на страни сервера користе се разни радни оквири који омогућавају једноставнији развој, и неки од најпопуларнијих радних оквира су *Express.js*, *Nest.js*, *Fastify*. У наставку, због једноставности и могућности дефинисања архитектуре користиће се радни оквир *Express.js*.

5.2 *.NET*

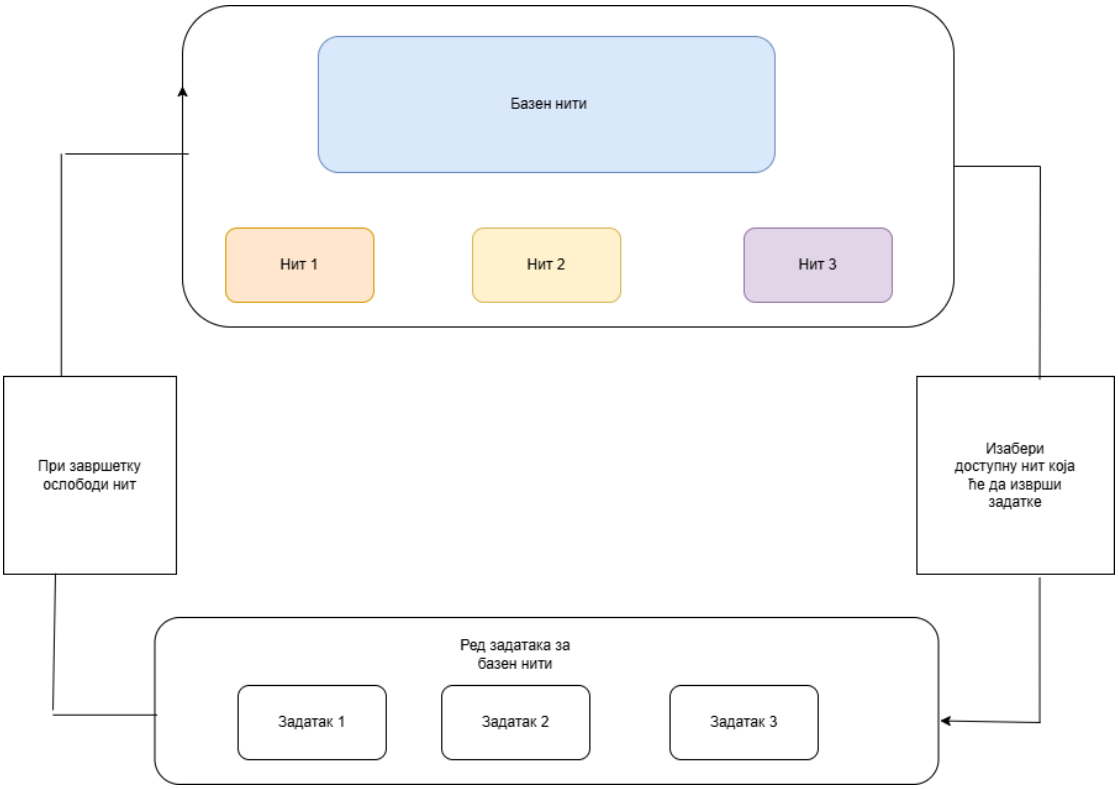
.NET је вишеплатформски радни оквир који подржава више програмских језика, уз језик *C#* који се сматра најпопуларнијим и најкоришћенијим. Садржи *Common Language Runtime (CLR)* за управљање извршењем кôда као и *Base Class Library (BCL)*, богату библиотеку класа за изградњу апликација. Омогућава аутоматско управљање меморијом кроз сакупљаче смећа (енгл. *garbage collector*), као и строгу типизираност. Слично као код *Node.js* платформе, омогућује конкурентност кроз кључне речи *async/await*, уз класу *Task* која омогућава асинхроно извршавање. Богати скуп библиотека са оптимизованим функционалностима омогућава коришћење *.NET* платформе на више оперативних система.

.NET има следеће компоненте

- Извршно окружење које извршава апликациони кôд.
- Библиотеке - омогућавају основне операције као што је нпр. парсирање *JSON* ниске.
- Компајлер - компајлира изворни кôд језика који подржава *.NET* платформа у извршни кôд.
- *SDK* и други алати - омогућавају изгрању и праћење програма.
- Радни оквири - користе погодности библиотека нижег нивоа, као и извршног окружења и дефинишу начин на који се гради апликација као и животни век.

За разлику од *Node.js*, који користи концепт петље догађаја, *.NET* користи концепт базена нити (енгл. *thread pools*) [26]. Петља догађаја је користила једну нит која је распоређивала догађаје у ред извршавања, док базен нити користи више нити ради извршавања захтевних улазно излазних операција. Пошто је креирање и брисање нити захтевна операција, *CLR* садржи кôд који управља базеном нити, који представља скуп нити доступних апликацији, и сваки *CLR* садржи по један такав базен нити. Интерно базен нити садржи ред

долазећих захтева. Приликом иницијализације, базен нити је празан, и кад треба извршити асинхрону операцију, креира се нова нит којој се додељује операција са врха реда. Након завршетка задатка, нит се не уништава, већ се враћа у базен нити где чека на наредну операцију.



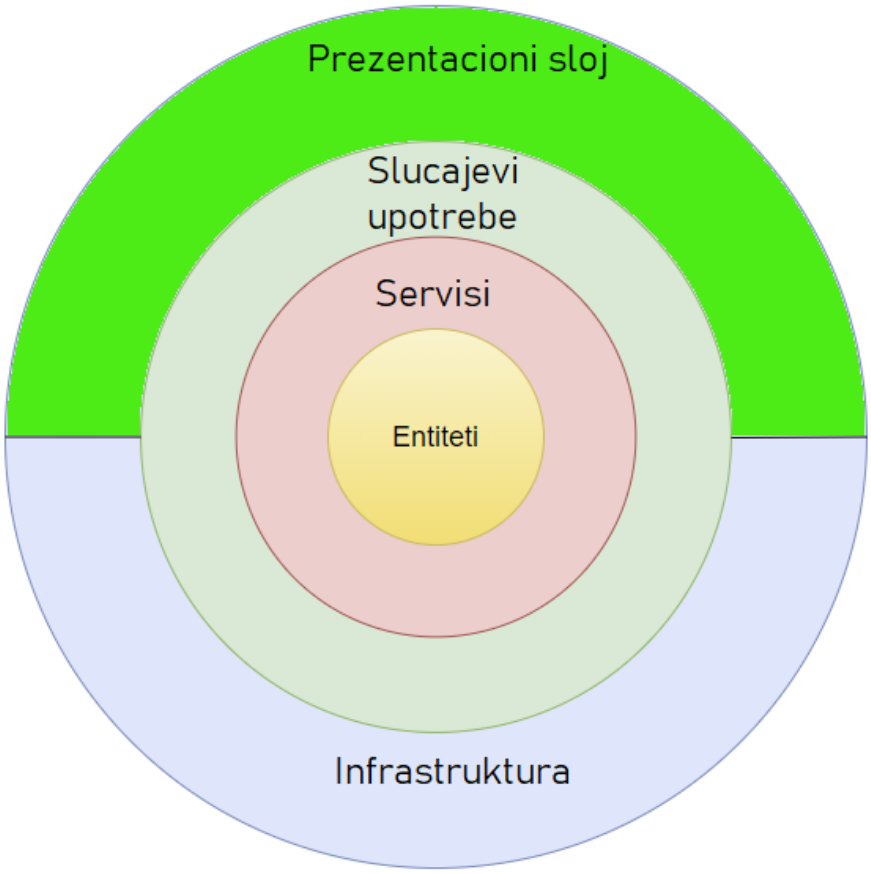
Слика 5.2: Ред базена

Иако имају сличну синтаксу језици *C#* и *TypeScript*, начин извршавања улазно излазних операција представља основну разлику између два језика, и тиме се и за различите типове апликација користе.

5.3 Поређење архитектура

У овом одељку биће представљене архитектуре које се користе у *.NET* и *Node.js* апликацијама, и разлози због чега за два радна оквира се користе две различите архитектуре. Ради демонстрације архитектура, приказаће се дијаграми који осликују архитектуре апликација, као и токови програма.

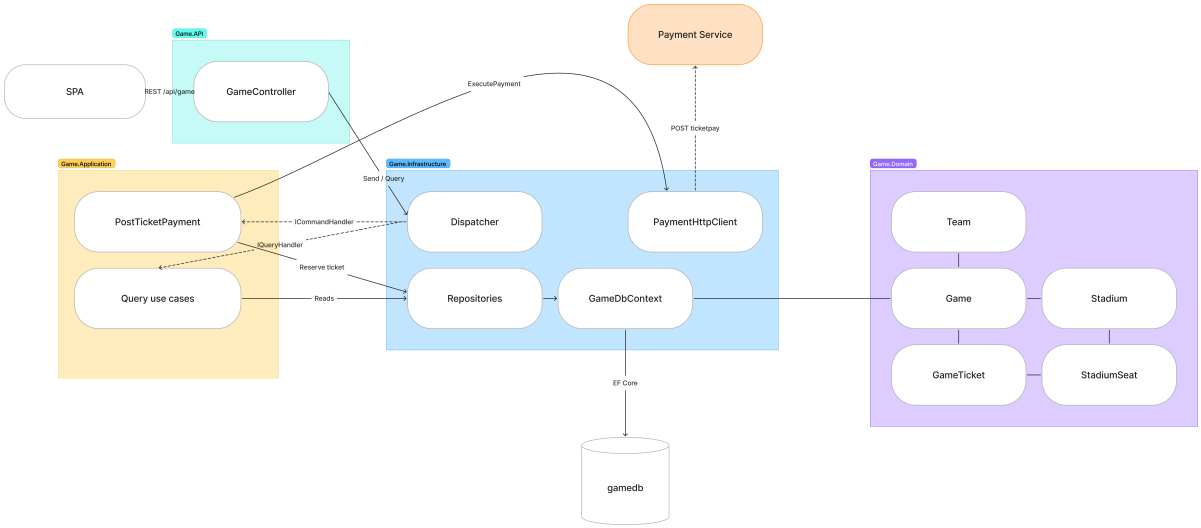
За развијање апликација у *.NET* радном оквиру користи се тип архитектуре где јасно раздвајају одговорности сваког од слоја, који се зове чиста архитектура. Чиста архитектура има следећи облик



Слика 5.3: Чиста архитектура [8]

Овде се виде следећи слојеви: презентациони слој, који представља улазну тачку система где су дефинисане приступне тачке система, апликациони слој, који садржи случајеве употребе и пословну логику система, затим слој домена који садржи све ентитете система, као и инфраструктурни део, који служи за коришћење ресурса као што је база података, third-party интерфејси и остало.

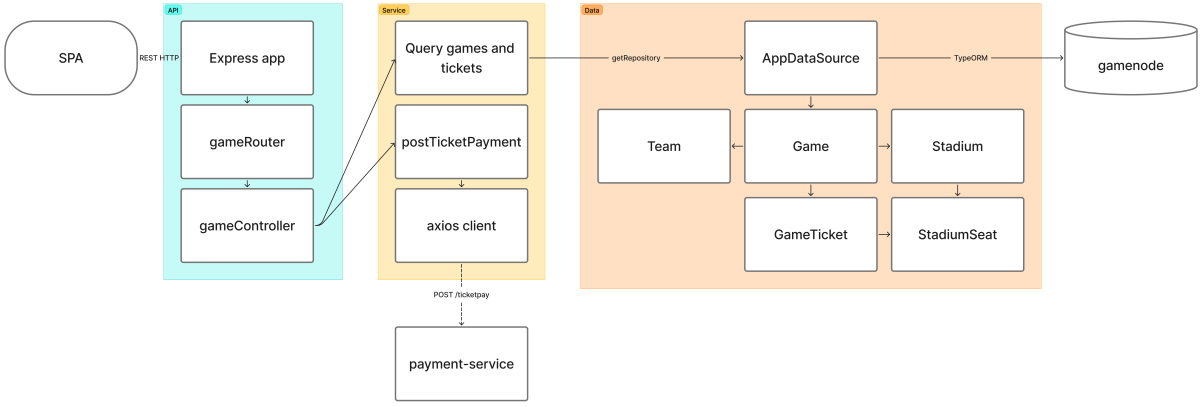
У центру чисте архитектуре налази се слој домена и слој домена не зависи ни од једног другог слоја. Ово има смисла јер у модерном развоју софтвера фокус је над ентитетима, јер они представљају део система који се ретко мења, због слабе повезаности међу самим ентитетима. Затим постоји зависност апликационог слоја од слоја домена, као и зависност презентационог слоја од апликационог слоја. Инфраструктурни слој зависи од апликационог слоја, и ту се види разлика у односу на тролојну архитектуру. Овако организоване зависности омогућују независност пословне логике од радних оквира, и тиме се смањују ограничења система. Такође апликациони слој може да се тестира без коришћења базе података или било којих других екстерних елемената што чини тестирање лакшим. Архитектура апликације приказана је на слици 5.4.



Слика 5.4: Чиста архитектура (.NET)

За разлику од *.NET* система, *Node.js* систем користи тролојну архитектуру: презентациони слој, слој сервиса и слој ентитета. Презентациони слој зависи од слоја сервиса, док слој сервиса зависи од слоја ентитета. Архитектура апликације приказана је на слици 5.5.

Одмах на почетку могу се приметити разлике у архитектури два радна оквира, и то најпре види у броју слојева, као и у зависности између слојева.



Слика 5.5: Трослојна архитектура (Node.js)

Таква организација између слојева омогућава код *.NET* архитектуре мања ограничења система, при чему пословна логика не зависи од радних оквира.

Архитектура *.NET* апликације има додатни апликациони слој, који садржи интерфејс ка бази података, као и дефинисање обрасца преко којег се комуницира са базом података, у овом случају *CQRS* [2]. *CQRS* шаблон служи за раздвајање упита и команди, ради бољег скалирања апликације, уместо једног модела који служи и за читање и писање. Главне методе су Send и Query, где Query се користи код GET захтева, док Send се користи код преосталих *HTTP* захтева. *CQRS* се користи у комбинацији са *Mediator* обрасцем [12], где презентациони слој шаље тип команде или упита апликационом слоју. Руковалац за тај тип у апликационом слоју прима захтев и обрађује га. За разлику од *Node.js* апликације, презентациони слој код *.NET* апликације не зна за постојање руковалаца захтева, те шаље само тип команде или упита.

Начин на који је имплементирана комуникација ка бази података, што се у *.NET* систему налази у инфраструктурном слоју а у *Node.js* систему се налази у слоју сервиса, представља једну од кључних разлика. Слој сервиса у *Node.js* систему се позива директно из презентационог слоја, док у *.NET* систему инфраструктурни слој се не позива ни из једног слоја, већ у позадини имплементира методе дефинисане у апликационом слоју. Да би се слојеви могли повезати и програм могао радити користи се образац уметање зависности

(енгл. *Dependency injection*) где се за конкретне интерфејсе из апликационог слоја дефинишу класе које имплементирају методе из интерфејса, преко контејнера. Контејнери поред давања типова интерфејсима, дефинишу и животни век везивања. На тај начин се обезбеђује лабава спрегнутост између слојева. Такође, у инфраструктурном слоју, преко истог обрасца, дефинише се и диспечер који служи за имплементирање *Mediator* обрасца. Како слој домена не зависи ни од једног другог слоја, у инфраструктурном слоју се дефинише конекција ка бази података као и креирање табела на основу класа из слоја домена.

Поставља се питање због чега на сличан начин није дефинисана архитектура у *Node.js* систему, и разлог лежи у парадигми језика *JavaScript*. За разлику од програмског језика *C#* који је објектно оријентисан и коме су класе носиоци стања, у језику *JavaScript* носиоци стања су функције. Преко функцијских затворења (енгл. *Closure*) у *JavaScript*-у се везују инстанце, јер инстанце живе унутар функције па се везују за понашање. Због тога је дефинисање контејнера у *JavaScript*-у непотребно, за разлику од програмског језика *C#* где инјекција мора да иде кроз конструкторе па је потребно имати контејнер.

Глава 6

Обрасци у микросервисној архитектури

У овој секцији приказаће се како операције које укључују рад са више микросервиса могу да се извршавају ефикасније и то ће бити приказано кроз обрасце за пројектовање у домену микросервисне архитектуре. Случај употребе на којем ће бити извршена демонстрација неких образаца је пример са приступном тачком `/ticketpay`. На том примеру ће се приказати најпре стандардни облик комуникације преко *HTTP* метода између два обрасца, затим кроз обрасце Прекидач и *Transactional Outbox* ће бити приказана унапређења како би систем био поузданији. Преко обрасца *API* композиција, због његове улоге, ће се тестирати перформансе *.NET* и *Node.js* система. Изворни кôдови апликација се налазе на *GitHub*-у, у јавно доступним репозиторијумима [6] [7].

Обрасци који ће се приказати који представљају решење поменутих проблема добрим делом почивају на обрасцима код монолитних апликација. Пре него што се употребе обрасци приказаће се основни облик комуникације између микросервиса код приступне тачке `/ticketpay`.

6.1 Случај употребе плаћања карте - пример комуникације *HTTP* позивима

Код случаја употребе плаћања карте микросервис за утакмице шаље *HTTP* захтев за уплатом микросервису за плаћање, и ако прође уплата микросервис за утакмице ажурира информацију о купљеној карти. Како се у том процесу комуникација одвија путем *HTTP* захтева, постоји неколико ситуација које могу да спрече успешно извршавање процеса.

Једна од ситуација се може десити када откаже један од микросервиса где би се процес извршио половично, што резултира у неконзистентним базама података за микросервисе. Ту се такође јавља проблем константног слања захтева микросервису који није у функцији, што може да изазове рањивост самог микросервиса.

6.1.1 Имплементација у радном оквиру *Node.js*

Главна логика се дешава у датотеци *gameService.ts* унутар сервис слоја. Ту је имплементирана пословна логика где је остварена комуникација са базом података, као и комуникација са микросервисом за плаћање, уз помоћ библиотеке *axios*. Приликом касније имплементације образаца, тај део ће бити највише подложан променама.

```
1  const postTicketPayment = async (ticketSeatPayment: TicketSeatPayment):
   ↪  Promise<PaymentResult | null> => {
2      const gameTicketRepository =
       ↪  AppDataSource.getRepository(GameTicket);
3
4      const gameTicket = await gameTicketRepository.findOne({
5          where: {
6              id: ticketSeatPayment.ticketId
7          },
8      });
9
10     if (!gameTicket) {
```

```
11          return null;
12      }
13
14      const reservationId = ticketSeatPayment.reservationId ?? Date.now();
15
16      const reserved = await tryReserveTicket(gameTicket.id,
        ↪  reservationId);
17
18      if (!reserved) {
19          throw new Error("Karta nije dostupna za kupovinu");
20      }
21
22      const paymentRequest: PaymentRequest = {
23          GameTicketId: String(gameTicket.id),
24          UserId: ticketSeatPayment.userId,
25          Amount: gameTicket.price,
26          Currency: ticketSeatPayment.currency,
27          ReservationId: String(reservationId),
28      };
29
30      let result: PaymentResult;
31
32      try {
33          const client = axios.create({
34              baseURL: PAYMENT_SERVICE_URL,
35          });
36
37          const response = await client.post<PaymentResult>("/ticketpay",
            ↪  paymentRequest);
38          result = response.data;
39      } catch (error) {
40          await releaseReservedTicket(gameTicket.id);
41          throw error;
42      }
43
```

```
44      if (!result || result.status !== PaymentStatus.Completed) {
45          await releaseReservedTicket(gameTicket.id);
46          throw new Error("Placanje nije uspelo");
47      }
48
49      const confirmed = await confirmTicket(gameTicket.id);
50
51      if (!confirmed) {
52          const latestTicketState = await gameTicketRepository.findOne({
53              where: {
54                  id: gameTicket.id,
55              },
56          });
57
58          if (latestTicketState?.status === TicketStatus.SOLD) {
59              return result;
60          }
61
62          throw new Error("Placanje je uspelo, ali potvrda karte nije
            ↪  uspela");
63      }
64
65      return result;
66  }
```

Овде је приказан процес где се прво добија информација о карти, где се проверава да ли тај објекат постоји, и након тога покушај резервације карте. Да би се избегла трка са ресурсима, у којој би више корисника покушало да купи карту истовремено, најпре се врши резервација карте, и после тога позив ка микросервису за плаћање где ако је плаћање успешно статус карте се мења у продат, а ако је неуспешно статус карте се мења у доступан. Након што захтев стигне до сервиса за плаћање, ту се дешава процес уплате и ту се користи логика из датотеке *paymentService.ts*.

```
1  const executePayment = async (ticketSeatPayload: TicketSeatPayload) :
   ↪  Promise<PaymentResult | null> => {
2      const transactionRepository =
       ↪  AppDataSource.getRepository(Transaction);
3      try {
4          const status = ticketSeatPayload.Amount <= 0 ?
           ↪  PaymentStatus.Failed : PaymentStatus.Completed;
5
6          const transaction = transactionRepository.create({
7              game_ticket_id: ticketSeatPayload.GameTicketId,
8              user_id: ticketSeatPayload.UserId,
9              amount: ticketSeatPayload.Amount,
10             currency: ticketSeatPayload.Currency,
11             status
12         } as Partial<Transaction>);
13
14         const saved = await transactionRepository.save(transaction);
15
16         return {
17             transactionId: saved.id,
18             gameTicketId: saved.game_ticket_id,
19             userId: saved.user_id,
20             amount: saved.amount,
21             currency: saved.currency,
22             status: saved.status,
23             createdAt: saved.created_at,
24             message: "Uspesno ste izvrsili placanje"
25         }
26     } catch (error) {
27         return null;
28     }
29 }
```

6.1.2 Имплементација у радном оквиру *.NET*

У *.NET* радном окружењу пословна логика се дефинише у апликационом слоју, унутар датотеке PostTicketPaymentCommandHandler.cs

Како *`PostTicketPaymentCommand`* команди у *CQRS* обрасцу одговара један руковалац *`PostTicketPaymentCommandHandler`*, где се у методи *`Handle`* обрађује захтев из методе *`Send`*, приказаће се у тој методи процес резервације карте и плаћања.

```
1  public async Task<PaymentResultDto> Handle(PostTicketPaymentCommand
   ↪  command)
2  {
3      var dto = command.TicketSeatPaymentDTO;
4      if (dto.ReservationId == Guid.Empty) dto.ReservationId =
       ↪  Guid.NewGuid();
5
6      var reserved = await
       ↪  _gameRepository.TryReserveTicketAsync(dto.GameTicketId,
       ↪  dto.ReservationId);
7
8      if (!reserved)
9      {
10         return new PaymentResultDto
11         {
12             GameTicketId = dto.GameTicketId,
13             Status = PaymentStatus.Failed,
14             Message = "Karta nije dostupna"
15         };
16     }
17
18     try
19     {
20         var result = await _paymentClient.ExecutePaymentAsync(dto);
21
22         if (result.Status == PaymentStatus.Completed)
23             await _gameRepository.ConfirmTicketAsync(dto.GameTicketId);
```

```
24          else
25              await _gameRepository.ReleaseTicketAsync(dto.GameTicketId);
26          return result;
27      }
28      catch (Exception ex)
29      {
30          await _gameRepository.ReleaseTicketAsync(dto.GameTicketId);
31          return new PaymentResultDto
32          {
33              GameTicketId = dto.GameTicketId,
34              Status = PaymentStatus.Failed,
35              Message = $"Placanje neuspesno: {ex.Message}"
36          };
37      }
38  }
```

Овде се користи атрибут *_gameRepository* инстанце *IGameRepository* као и *_paymentClient* инстанце *IPaymentClient* који представља интерфејс са методом *ExecutePaymentAsync*. У датотеци *InfrastructureServiceRegistration* се налази регистрација адресе микросервиса за плаћање.

```
1  var paymentServiceUrl = configuration["Services:PaymentServiceUrl"] ??
   ↪  "http://localhost:5001/api/payment/";
2
3  services.AddHttpClient<IPaymentClient, PaymentHttpClient>(client =>
4  {
5      client.BaseAddress = new Uri(paymentServiceUrl);
6  });
```

Овде је описана регистрација класе *PaymentHttpClient* као *HTTPClient*, чиме се омогућава да све операције унутар класе *PaymentHttpClient* користе адресу дефинисану из конфигуационе датотеке *appsettings.json*.

```
1  "Services": {
2    "PaymentServiceUrl": "http://localhost:5074/api/payment/"
```

```
}
```

Метод *ExecutePaymentAsync* из *PaymentHttpClient* садржи објекат *HttpClient* класе да би било могуће реализовати *HTTP* захтеве.

Метод садржи асинхрони захтев ка *ticketpay* приступној тачки ка микросервису за плаћање, и тај део у наредним поглављима, приликом приказа различитих образаца, ће бити подлежан променама.

```
public async Task<PaymentResultDto>
↪ ExecutePaymentAsync(TicketSeatPaymentDTO paymentDto)
{
    var response = await _httpClient.PostAsJsonAsync("ticketpay",
    ↪ paymentDto);
    response.EnsureSuccessStatusCode();

    var result = await
    ↪ response.Content.ReadFromJsonAsync<PaymentResultDto>();
    if (result == null)
    {
        throw new InvalidOperationException("Payment servis vratio
        ↪ prazan odgovor");
    }

    return result;
}
```

Уплата се извршава у микросервису за плаћање у датотеци *TransactionService.cs* унутар инфраструктурног пројекта.

```
public async Task<PaymentResultDto> ExecutePayment(TicketSeatPaymentDTO
↪ ticketSeatPaymentDTO)
{
    var status = ticketSeatPaymentDTO.Amount <= 0 ? PaymentStatus.Failed
    ↪ : PaymentStatus.Completed;

```

```
5       var transaction = new Transaction
6       {
7           Id = Guid.NewGuid(),
8           GameTicketId = ticketSeatPaymentDTO.GameTicketId,
9           UserId = ticketSeatPaymentDTO.UserId,
10          Amount = ticketSeatPaymentDTO.Amount,
11          Currency = ticketSeatPaymentDTO.Currency,
12          Status = status,
13          CreatedAt = DateTime.UtcNow,
14          UpdatedAt = DateTime.UtcNow
15      };
16
17      _paymentDbcontext.Transactions.Add(transaction);
18      await _paymentDbcontext.SaveChangesAsync();
19
20      return new PaymentResultDto
21      {
22          TransactionId = transaction.Id,
23          GameTicketId = transaction.GameTicketId,
24          UserId = transaction.UserId,
25          Amount = transaction.Amount,
26          Currency = transaction.Currency,
27          Status = transaction.Status,
28          CreatedAt = transaction.CreatedAt,
29          Message = status == PaymentStatus.Completed
30  ? "Placanje izvrseno uspesno"
31  : "Placanje nije izvrseno"
32      };
33  }
```

6.2 Образац Прекидач

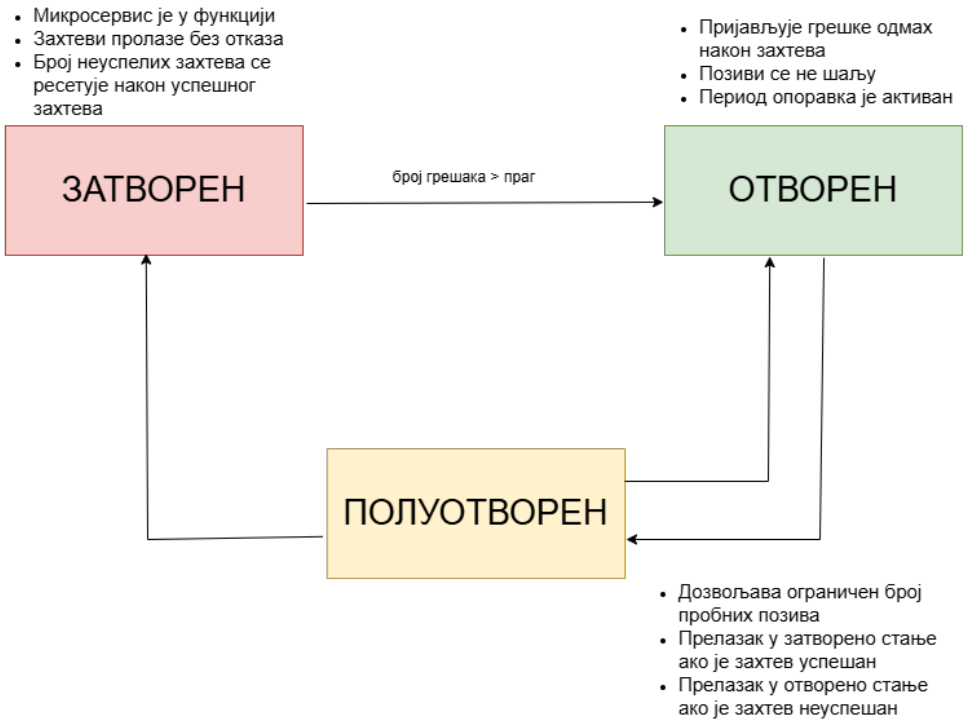
Процес који је описан у вези комуникације микросервиса за утакмице и микросервиса за плаћање је извршив само под условом да сваки од микро-

сервиса ради без отказивања [24]. До отказивања може доћи из разлога као што су софтверске грешке, или хардверске грешке где долази до отказивања дискова. Да би микросервиси наставили да раде чак иако дође до отказа у једном од микросервиса, морају да знају шта је довело до отказа. Најчешћи разлози су грешке чији кôд почиње бројем 5, а може бити и превелики тајмаут. Тајмаути представљају временско ограничење на које се чека приликом одговора од стране сервера и осетљив су део за руковање грешкама. Ако се тајмаут постави на превелику вредност, може доћи до загушења сервера, а ако се постави на премалу вредност, онда може доћи до ситуације да сервер не стигне на време да пошаље одговор. Једна од ситуација која може да изазове рањивост сервера је мноштво поновних позива, где се успорава рад сервера и време чекања на одговор.

Решење за овакав проблем је механизам који би спречио коришћење ресурса за комуникацију са сервером који је у отказу, где би корисник могао да добије бржи одговор о стању система што би дало време серверу који је у отказу да се опорави, и такав механизам нуди образац Прекидач.

Образац Прекидач нуди начин да кроз његова стања опише доступност сервера, а то су стања затворен, отворен и полуотворен. Када је прекидач у затвореном стању тада је могуће успоставити конекцију са сервером, и захтеви се одвијају без грешака сервера. Да не би дошло до успоравања система и трошења ресурса, у случају да се деси неколико неуспешних захтева прекидач прелази у отворено стање, што спречава слање захтева ка микросервису који није у функцији.

Да се не би дошло у ситуацију да се након опоравка микросервиса чека дуго на поновни захтев или да се „пусте” поново сви захтеви након одређеног броја секунди, уводи се полуотворено стање код прекидача. У полуотвореном стању прекидач пушта само један захтев да провери доступност микросервиса, и за случај да је захтев успешан, прекидач се затвара, а за случај да је захтев неуспешан прекидач се поново отвара.



Слика 6.1: Образац Прекидач

6.2.1 Имплементација у радном оквиру *Node.js*

У радном оквиру *Node.js*, због слабе подршке екстерних библиотека, биће представљена имплементација кроз класу `CircuitBreaker` унутар директоријума `utils`.

Параметри класе представљају интерфејс `CircuitBreakerOptions`

```
1 interface CircuitBreakerOptions {
2   failureThreshold: number;
3   recoveryTimeout: number;
4   successThreshold: number;
5   callTimeout?: number;
6 }
```

- `failureThreshold` - максималан број узастопно неуспелих позива након којих се прекида комуникација са микросервисом и прекидач прелази у

стање отворен.

- *recoveryTimeout* - временски период током којег прекидач остаје у отвореном стању пре него што пређе у полуотворено стање да провери да ли је микросервис доступан.
- *successThreshold* - број успешно узастопних пробних позива потребних да би се комуникација са микросервисом поново успоставила.
- *callTimeout* - максимално време чекања одговора појединачног позива.

Стање се дефинише кроз тип, где се дефинишу сва могућа стања у коме може прекидач да се нађе.

```
type State = 'CLOSED' | 'OPEN' | 'HALF_OPEN';
```

Променљива *state* поменутог типа се дефинише унутар класе, док информацију о броју успешних и неуспешних захтева се чува у променљивим *failures* і *successes*. Унутар конструктора класе као параметар се прослеђује објекат интерфејса *CircuitBreakerOptions* из разлога што вредности тих параметара приликом иницијализације треба да подесе, док остале вредности треба да буду сакривене.

Метода *call()* представља улазну тачку за извршавање свих позива према микросервису, при чему се пре самог извршавања проверава стање прекидача.

```
async call<T>(action: () => Promise<T>): Promise<T> {
    if (this.state === 'OPEN') {
      if (this.lastFailureTime !== null && Date.now() -
      ↪  this.lastFailureTime >= this.recoveryTimeout) {
        this.transitionTo('HALF_OPEN');
      } else {
        throw new Error('Prekidac je otvoren');
      }
    }

    if (this.state === 'HALF_OPEN') {
```

```
11      if (this.halfOpenInFlight) {
12        throw new Error('Prekidac je poluotvoren, zahtev je u toku');
13      }
14      this.halfOpenInFlight = true;
15    }
16
17    try {
18      const result = await this.runAction(action);
19      this.onSuccess();
20      return result;
21    } catch (error) {
22      this.onFailure();
23      throw error;
24    }
25  }
```

У случају да је стање *OPEN*, проверава се да ли је истакао период опоравка, и ако није програм избацује изузетак, иначе прелази у полуотворено стање, где дозвољава извршавање пробног захтева. Да би се спречило извршавање више пробних захтева, користи се променљива ***halfOpenInFlight***, и док је један пробни захтев у току, сви остали захтеви су одбијени. Само извршавање позива се догађа у методи ***runAction()*** која извршава прослеђену асинхрону функцију.

Уколико је дефинисан тајмер кроз ***callTimeout*** параметар, користи се ***Promise.race()*** ради паралелног покретања позива прослеђене асинхроне функције и тајмера.

```
1  private async runAction<T>(action: () => Promise<T>): Promise<T> {
2      if (this.callTimeout === null) {
3        return action();
4      }
5
6      let timer: ReturnType<typeof setTimeout> | undefined;
7      const timeoutPromise = new Promise<never>((_, reject) => {
8        timer = setTimeout(
```

```
9         () => reject(new Error('Circuit breaker call timed out')),
10        this.callTimeout as number
11      );
12    });
13
14    try {
15      return await Promise.race([action(), timeoutPromise]);
16    } finally {
17      if (timer) {
18        clearTimeout(timer);
19      }
20    }
21  }
```

Уколико истекне максимално време чекања позив се прекида и третира као неуспешан. Након извршавања тајмер се уклања како не би остао активан у меморији.

У случају успешно позива извршава се метода *onSuccess()*, где ако је прекидач у затвореном стању број неуспешних позива се поставља на нулу, а уколико је у полуотвореном стању повећава се успешан број пробних позива, и ако је тај број једнак *successThreshold*, прекидач прелази у затворено стање.

```
1  private onSuccess() {
2      if (this.state === 'HALF_OPEN') {
3        this.halfOpenInFlight = false;
4        this.successes++;
5        if (this.successes >= this.successThreshold) {
6          this.transitionTo('CLOSED');
7        }
8        return;
9      }
10
11     this.failures = 0;
12   }
```

У случају неуспешног позива извршава се метода *onFailure()*. Када је прекидач у затвореном стању, повећава се број узастопних неуспеха, и ако тај број достигне вредност параметра *failureThreshold*, позива се метод *open()* који мења стање прекидача у отворено и бележи време последњег неуспешног захтева. Ако дође до грешке у полуотвореном стању, аутоматски се прекидач пребацује у отворено стање.

```
private onFailure() {
    if (this.state === 'HALF_OPEN') {
      this.halfOpenInFlight = false;
      this.open();
      return;
    }

    this.failures++;
    if (this.failures >= this.failureThreshold) {
      this.open();
    }
  }
```

Метода *transitionTo* служи за прелазак прекидача у стање дефинисано параметром типа *State*.

```
private transitionTo(newState: State) {
    if (this.state === newState) {
      return;
    }

    this.state = newState;
    this.failures = 0;
    this.successes = 0;

    if (newState !== 'HALF_OPEN') {
      this.halfOpenInFlight = false;
    }
```

```
13    }
```

Класа се примењује у датотеци *gameService.ts*, где се дефинише инстанца класе *CircuitBreaker*

```
const paymentCircuitBreaker = new CircuitBreaker({
    failureThreshold: 5,
    recoveryTimeout: 30000,
    successThreshold: 1,
    callTimeout: 5000,
});
```

И микросервис за плаћање позива се у методи *tryTicketPay*.

```
const tryTicketPay = async (paymentRequest: PaymentRequest):
↪  Promise<PaymentResult> => {
    const client = axios.create({
            baseURL: PAYMENT_SERVICE_URL,
            timeout: PAYMENT_TIMEOUT_MS,
        });

    const response = await paymentCircuitBreaker.call(() =>
        client.post<PaymentResult>("/ticketpay", paymentRequest)
    );

    return response.data;
}
```

Унутар методе *call* постављен је *POST* HTTP захтев ка микросервису за плаћање.

6.2.2 Имплементација у радном оквиру *.NET*

Кад је у питању имплементација у радном оквиру *.NET*, приказаће се решење које представља стандард кад је у питању радни оквир *.NET*, а то је

коришћење библиотеке *Polly*. За имплементацију самог обрасца потребно је инсталирати библиотеке *Polly* и *Polly.Extensions.Http*.

```
static IAsyncPolicy<HttpResponseMessage> GetCircuitBreakerPolicy()
{
    return HttpPolicyExtensions
        .HandleTransientHttpError()
        .CircuitBreakerAsync(2, TimeSpan.FromSeconds(30));
}
```

Овде су дефинисана правила која се тичу прекидача где *.HandleTransientHttpError()* реагује на привремене грешке, а *CircuitBreakerAsync(2, 30s)* након два узастопна неуспеха прелази у отворено стање на 30 секунди.

```
static IAsyncPolicy<HttpResponseMessage> GetRetryPolicy()
{
    return HttpPolicyExtensions
        .HandleTransientHttpError()
        .WaitAndRetryAsync(3, retryAttempt =>
        ↪  TimeSpan.FromSeconds(Math.Pow(2,
        ↪  retryAttempt)));
}
```

У овом делу је дефинисана *Retry* техника која се примењује када микросервис ради, али има привремене грешке. Обе технике се повезују са *HttpClient*-ом на следећи начин

```
services.AddHttpClient<IPaymentClient, PaymentHttpClient>(...)
    .SetHandlerLifetime(TimeSpan.FromMinutes(5))
    .AddPolicyHandler(retryPolicy)
    .AddPolicyHandler(circuitBreakerPolicy);
```

Retry техника се региструје прва, док *CircuitBreaker* техника друга, зато што свако понављање пролази кроз прекидач, па више понављања може да брже отвори прекидач.

6.3 *API* композиција

Код монолитних апликација, код којих је присутна једна база података, операцију добијања податка који садржи податке из више табела је могуће урадити спајањем по примарном кључу и на тај начин добити податак из више табела.

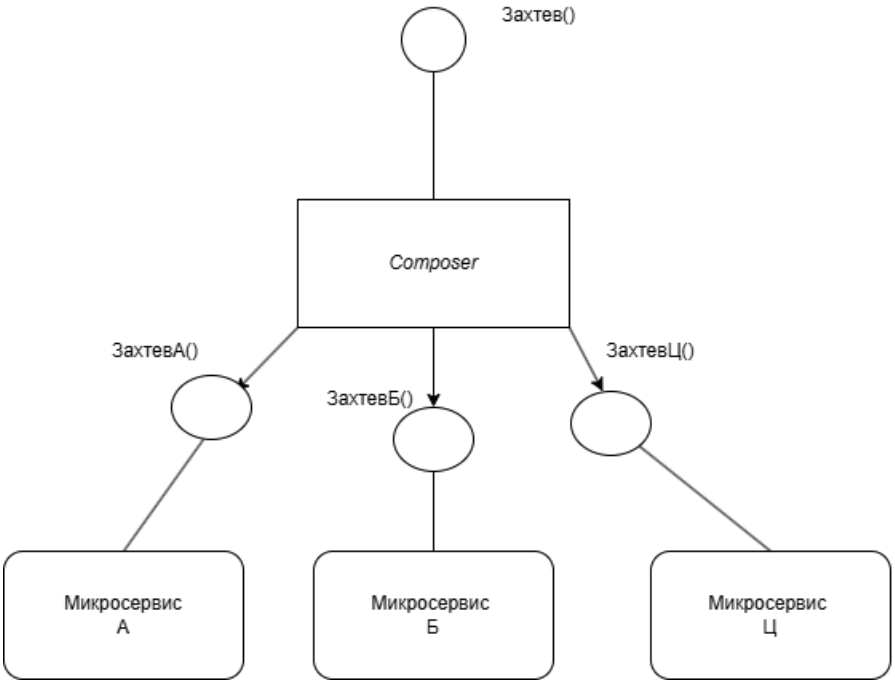
Пошто код микросервисне архитектуре за сваки микросервис постоји посебна база података, спајање табела би било могуће када би се користила техника дистрибуиране базе података, али такав приступ би повећао спрегу између микросервиса. Ако један микросервис може да мења податке које чита други микросервис, то може довести до грешака у другом микросервису. Због тога ће се користити образац *API* композиција, где постоји централизовани микросервис одакле се читају подаци и касније се спајају [25].

У обрасцу *API* композиција, поред микросервиса, за учесника се дефинише и компонента која има улогу *Composer*-а. Такава компонента може бити клијентска веб апликација која паралелно чита податке из више микросервиса. Такав начин не представља добру праксу, због тога што трпе перформансе клијентске апликације, па се уместо ње користи посебан микросервис који има улогу *Composer*-а, који чита податке из осталих микросервиса. Тај микросервис има улогу *API* капије, која се понаша као обрнути прокси (енгл. *Reverse Proxy*).

Обрнути прокси је мрежни микросервис који се налази између клијента и осталих микросервиса. Када клијент пошаље *HTTP* захтев, обрнути прокси га обрађује и транспарентно прослеђује одговарајућем серверу. Представља заштиту идентитета и сигурности микросервиса са којима је повезан.

Обрнути прокси има неколико улога:

- *Улога безбедности* - скривање IP адресе и структуре микросервиса од спољног света.
- *Балансирање оптерећења* - да не би један микросервис био преоптерећен захтевима, захтев се дистрибуира на више сервера.
- *Кеширање* - чува копије често тражених садржаја ради бржег одговора.



Слика 6.2: API композиција

Као пример обрасца *API* капија користиће се приступна тачка *get-transaction-info*, која ће нам дати информацију о трансакцији проширену са информацијом о купљеној карти.

6.3.1 Имплементација у радном оквиру *Node.js*

Кад је у питању имплементација у радном оквиру *Node.js*, креираће се нови микросервис који има улогу *API* капије, и биће иницијализован као *Express* пројекат на начин на који је рађен са претходним микросервисима. Библиотека која ће се користити за обрнути прокси јесте *http-proxy-middleware*.

У датотеци *app.ts* дефинишу се Proxy и Composer.

```
app.use(
    "/game",
    createProxyMiddleware({
        target:"http://localhost:3000",
        changeOrigin: true,
```

```
6           pathRewrite: {
7               "^/game": "/"
8           }
9       })
10  );
11
12  app.use(
13      "/payment",
14      createProxyMiddleware({
15          target:"http://localhost:3001",
16          changeOrigin: true,
17          pathRewrite: {
18              "^/payment": "/"
19          }
20      })
21  );
```

Овде је дефинисан *Proxy* за наш образац. Може се дефинисати на начим да су додате приступне тачке за сервер, у нашем примеру game и payment, које захтев прослеђују одговарајућем микросервису. Ако се узме за пример *GET http://localhost:3002/game/get-all-games*, овде се дешава следеће:

1. Средњи слој види приступну тачку */game*
2. *pathRewrite* брише */game* и putanja постаје */get-all-games*
3. Захтев иде на адресу *http://localhost:3000/get-all-games*, што је захтев ка микросервису за утакмице.
4. Одговор од микросервиса се врати клијенту онакав какав јесте.

Слично се дешава и када клијент пошаље захтев на приступној тачки */payment*, где у том случају захтев бива прослеђен микросервису за плаћање.

Преко приступне тачке */get-transaction-info*, дефинише се Composer који преко *URL* параметара које је добио шаље захтев ка микросервису за

утакмице и микросервису за плаћање, где добија одговарајуће информације и од једног и од другог микросервиса.

```
app.get("/get-transaction-info/:gameTicketId/:transactionId", async (req: Request, res:
↪  Response, next: NextFunction) => {
    const gameClient = axios.create({baseURL: "http://localhost:3000"});
    const paymentClient = axios.create({baseURL: "http://localhost:3001"});

    const { gameTicketId, transactionId } = req.params;

    try{
        const [gameTicketInfo, transactionInfo] = await
        ↪  Promise.all([gameClient.get<any>(`/get-ticket-info/${gameTicketId}`),
        ↪  paymentClient.get<any>

        (`/transaction-info/gameTicket/${gameTicketId}/transaction/${transactionId}`)]);

        return res.status(200).json({
            gameTicketInfo: gameTicketInfo.data,
            transactionInfo: transactionInfo.data
        });
    } catch (err) {
        return res.status(500).json({message:"Interna greska"});
    }
});
```

Овде капија позива паралелно захтеве ка оба микросервиса да би добила информацију коју спаја у један објекат.

6.3.2 Имплементација у радном оквиру *.NET*

Што се тиче имплементације у *.NET* радном оквиру, креираће се посебан *WebAPI* пројекат који ће имати улогу *API* капије. Да би се користио обрнути прокси користиће се библиотека *YARP* и она се користи у датотеци *appSettings.json*. Она се чита у датотеци *Program.cs* приликом покретања апликације, и то у линији.

```
builder.Services.AddReverseProxy()
    .LoadFromConfig(builder.Configuration.GetSection("ReverseProxy"));
```

У датотеци *appSettings.json* се дефинише конфигурација

```
"ReverseProxy": {
  "Routes": {
    "games-route": {
      "ClusterId": "game-cluster",
      "Match": {
        "Path": "/games-api/{**catch-all}"
      },
      "Transforms": [
        { "PathPattern": "/api/Game/{**catch-all}" }
      ]
    },
    "payments-route": {
      "ClusterId": "payment-cluster",
      "Match": {
        "Path": "/payments-api/{**catch-all}"
      },
      "Transforms": [
        { "PathPattern": "/api/Payment/{**catch-all}" }
      ]
    }
  },
  "Clusters": {
    "game-cluster": {
      "Destinations": {
        "game-api": {
          "Address": "http://localhost:5177/"
        }
      }
    },
    "payment-cluster": {
      "Destinations": {
        "payment-api": {
          "Address": "http://localhost:5074/"
        }
```

```
35        }
36      }
37    }
38  }
```

Конфигурација је састављена од делова *Routes* и *Clusters*. Део ***Routes*** дефинише правила у зависности који *URL* стигне, и у складу са тим games-route и payments-route представљају идентификаторе рута. ***ClusterId*** представља идентификатор кластера у виду микросервиса који обрађује ту руту, ***Match.Path*** представља јавну адресу за капију где **catch-all = је све оно што следи после те адресе, а то су приступне тачке захтева. ***Transforms*** и ***PathPattern*** преписују путању на адресу одговарајућег микросервиса пре слања захтева.

Део ***Clusters*** представља параметар који описује где се налазе микросервиси преко којих шаље захтев *API* капија, и ***game-cluster*** и ***payment-cluster*** су посебне групе дестинација. Део ***Destinations*** представља листу адреса микросервиса, и ако се користи балансирање оптерећења, што иначе нуди библиотека *YARP*, може постојати више адреса, и те дестинације су ***game-api*** и ***payment-api***. ***Address*** је права база *URL* адреса микросервиса, и та адреса се користи и за конфигурацију ***HTTPClient***-а.

```
1  var gameApiBase = builder
2  .Configuration["ReverseProxy:Clusters:game-cluster:Destinations:game-api:Address"]
3      ?? throw new InvalidOperationException("game-cluster destination not configured");
4  var paymentApiBase = builder
5  .Configuration["ReverseProxy:Clusters:payment-cluster:Destinations:payment-api:Address"]
6      ?? throw new InvalidOperationException("payment-cluster destination not
       ↪  configured");
7
8  builder.Services.AddHttpClient("game", c => c.BaseAddress = new Uri(gameApiBase));
9  builder.Services.AddHttpClient("payment", c => c.BaseAddress = new
   ↪  Uri(paymentApiBase));
```

Composer, слично као и у радном оквиру *Node.js*, дефинише се у приступној датотеци ***Program.cs***, пратећи сличан начин за позивање микросервиса као и у *Node.js* радном оквиру.

```
app
.MapGet("/GetGameTransaction/game/{gameTicketId:guid}/transaction/{transactionId:guid}",
↪  async (
    Guid gameTicketId,
    Guid transactionId,
    [FromServices] IHttpClientFactory httpClientFactory,
    CancellationToken ct) =>
{
    var gameClient = httpClientFactory.CreateClient("game");
    var paymentClient = httpClientFactory.CreateClient("payment");

    var gameTicketTask=  gameClient.GetFromJsonAsync<object>(
        $"api/Game/get-ticket-info/{gameTicketId}", ct);
    var transactionTask =  paymentClient.GetFromJsonAsync<object>(

        ↪  $"api/Payment/get-transaction-info/game/{gameTicketId}/transaction/{transactionId}",
        ↪  ct);

    await Task.WhenAll(gameTicketTask, transactionTask);

    return Results.Ok(new
    {
        gameTicketId,
        transactionId,
        gameTicketInfo = gameTicketTask.Result,
        transactionInfo = transactionTask.Result
    });
});
```

6.4 Образац *Transactional Outbox*

Кад су у питању обрасци комуникације између два микросервиса, приказан је образац Прекидач који служи да спречи сталан покушај захтева ка микросервису који није у функцији. То је један од проблема када је у питању комуникација између два микросервиса.

Један од проблема може бити, због одржавања конзистентности између две базе података, да је потребно реализовати механизам дистрибуиране трансакције. У примеру комуникације микросервиса за утакмице и микросервиса за плаћање може се десити да кад микросервис за плаћање уради свој део посла, након операције *COMMIT* да мрежа постане недоступна, што резултира у то-

ме да микросервис за утакмице не зна какав се исход десио. Да би се спречио такав сценарио, потребно је увести механизам да се догађаји који морају бити извршени не изгубе, и решење за то нуди образац *Transactional Outbox* [25].

Поред обрасца *Transactional Outbox*, могуће је имплементирати и образац дистрибуиране трансакције. Циљ дистрибуиране трансакције јесте да изврши операције као што би биле извршене у монолитним системима, и начин да се то постигне јесте преко двофазног комита. Као што му име каже, двофазни комит се састоји од две фазе: фазе припреме и фазе извршавања, и обема фазама управља централни координатор. Фаза припреме подразумева да координатор пита сваку од база да ли су спремне да изврше потребне операције, и пре него што се изврши *COMMIT* у другој фази, свака од база извршава себи одговарајуће операције које су потребне. Оног момента кад су обе базе способне даураде *COMMIT* операцију, оне обавештавају координатора да могу и да ће урадити *COMMIT* операцију у другој фази, и ако било која од база нема могућност да уради обећање, координатор наређује базама даураде *ROLLBACK* операцију и све базе се враћају на стање пре почетка трансакције.

Постоји мноштво мана код оваквог централизованог приступа, и тиче се пре свега мрежног протокола. За имплементацију оваквог механизма протоколи који користе базе података са координатором морају бити исти, а многе нерелационе базе података као што су *MongoDB* и *Cassandra*, не подржавају систем дистрибуираних трансакција, као и модерни брокери за поруке *RabbitMQ* и *Apache Kafka*, о којима ће бити речи касније. Оно што је главни проблем са дистрибуираним трансакцијама јесте да представљају синхрони метод за извршавање, и ако нека база није спремна да изврши *COMMIT* операцију, она блокира координатора и цео процес је у чекању.

Једно од решења овог проблема нуди образац *Transactional Outbox*, који суштински уместо синхроне комуникације нуди решење у виду асинхроне комуникације, чиме се отклања блокирање микросервиса и одговарајућих база података везаних за микросервисе. Идеја која стоји иза овог обрасца јесте да информације о догађају који се десио никада неће бити изгубљене, па самим тим што приликом комуникације сваки сервис сигурно добија информацију о догађају који се десио, конзистентност база података одговарајућих микросервиса је стално присутна.

За имплементацију *Transactional Outbox* обрасца су додатно потребне две ствари, *Outbox* табела за сваку од база података где се чувају догађаји који су извршени, као и позадински обављач послова.

Процес којим се извршава овај образац иде тако што микросервис након што обради догађај не шаље директно захтев ка другом микросервису, већ у једној трансакцији упише одговарајућу информацију у табелу где је и до сада писао, и у *Outbox* табели пише информацију о догађају, како се не би изгубила ниједна информација. Позадински обављач послова затим чита из табеле информацију о догађају и обавештава други микросервис о том догађају. Сама комуникација може да се одвија преко REST архитектуре, при чему се мора водити рачуна о томе да се рукује непредвиђеним околностима као и код директне комуникације.

Оно што је стандард јесте да комуникација се дешава асинхроно преко редова порука, и то омогућује слабу повезаност међу микросервисима и тиме се испуњава једно од начела у развоју софтвера. Комуникација иде тако да клијент шаље захтев ка микросервису у виду слања поруке, и ако се очекује да микросервис одговори на захтев, то ће урадити тако што ће проследити поруку назад. Ниједан од микросервиса не зна одакле је послат захтев у виду поруке, порука је једино што стиже. Архитектура која омогућава овакав начин комуникације се зове архитектура вођена догађајем.

Комуникацију преко редова порука омогућава брокер за поруке који представља централну компоненту код архитектуре вођене догађајима. По улози и начину на који се испоручују поруке постоји неколико типова брокера за поруке. Сваки брокер садржи *header* који представља кључ-вредност тип података којим се описују подаци који су послати. Поред *header* садржи и *body* део, главни део поруке, који може бити или у текстуалном или у бинарном формату.

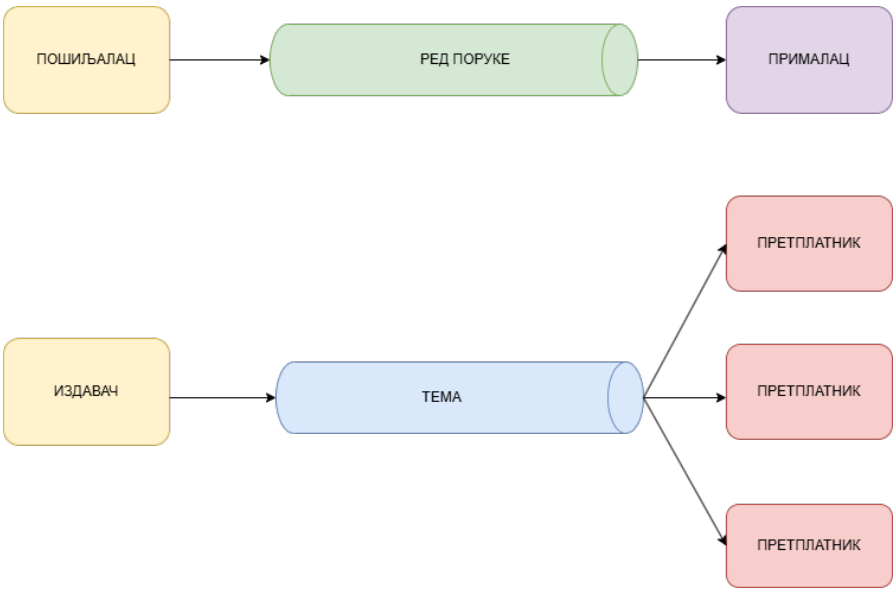
По типу, порука може бити:

- Документ - генеричка порука која садржи само податке и прималац поруке одређује како ће да је интерпретира.
- Команда - порука која назначује операцију која мора да се изврши као и параметре операције.

- Догађај - порука која говори да се кôд пошиљаоца десила промена која је описана у поруци.

Кад је у питању начин комуникације преко порука, канали за поруке могу бити:

- *Point to Point* - порука иде од пошиљаоца до тачно једног примаоца, где прималац преузима поруку, обрађује је и брише је из реда. Пример *RabbitMQ*.
- *Publish/Subscribe* - порука иде од пошиљаоца до више прималаца, који могу бити преплаћени на одређену тему. Сви примаоци који су преплаћени на тему добијају исту поруку. Пример *Apache Kafka*.



Слика 6.3: Point to Point *и* Publisher/Subscriber *редови порука*

За образац *Transactional Outbox* користи се *Point to Point* начин комуникације. Оба микросервиса имају улогу и пошиљаоца и примаоца порука. У примеру комуникације микросервиса за утакмице са микросервисом за плаћање, образац се примењује на следећи начин:

1. Микросервис за утакмице, уколико је карта доступна, резервише карту и уписује информације о догађају у *Outbox* табелу у једној трансакцији.

2. Позадински обављач послова код микросервиса за утакмице, чита поруке које су прослеђене у растућем редоследу и сваку од њих прослеђује реду поруке, и те поруке обрађује микросервис за плаћање .

3. Прималац поруке, микросервис за плаћање, прима догађај од стране микросервиса за утакмице, и у једној трансакцији извршава плаћање и уписује извршену операцију у *Outbox* табелу у једној трансакцији.

4. Позадински обављач послова код микросервиса за плаћање, чита поруке које су прослеђене у растућем редоследу и сваку од њих прослеђује реду поруке, и те поруке обрађује микросервис за утакмице.

5. Микросервис за утакмице, након прочитане поруке, резервисану карту потврђује за случај да је микросервис за утакмице успешно извршио догађај. Иначе ако је догађај неуспешно извршен, микросервис за утакмице укида резервацију за карту и карта постаје слободна за резервацију.

6.4.1 Имплементација у радном оквиру *Node.js*

Кад је у питању имплементација у радном оквиру *Node.js* прво што ће се урадити јесте да ће се додати `OutboxMessage` ентитет унутар `src/data/entity`.

```
1  import { Column, Entity, Index, PrimaryGeneratedColumn } from "typeorm";
2
3  @Entity()
4  export class OutboxMessage {
5      @PrimaryGeneratedColumn()
6      id!: number;
7
8      @Index({ unique: true })
9      @Column({ type: "uuid" })
10     message_id!: string;
11
12     @Column({ type: "varchar" })
13     type!: string;
14
```

```
    @Column({ type: "varchar" })
    routing_key!: string;

    @Column({ type: "text" })
    payload!: string;

    @Column({ type: "timestamp" })
    created_at!: Date;

    @Column({ type: "timestamp", nullable: true })
    processed_at?: Date | null;

    @Column({ type: "int", default: 0 })
    retry_count!: number;

    @Column({ type: "text", nullable: true })
    error?: string | null;
}
```

У датотеци *gameService.ts* унутар методе *postTicketPayment* је имплементиран први корак код обрасца, где у једној трансакцији се врши резервација карте, уколико је то могуће, и упис догађаја у *Outbox* табелу.

```
const postTicketPayment = async (ticketSeatPayment: TicketSeatPayment):
↪  Promise<PaymentResult | null> => {
    const queryRunner = AppDataSource.createQueryRunner();
    const gameTicketRepository =
    ↪  AppDataSource.getRepository(GameTicket);
    const outboxRepository = AppDataSource.getRepository(OutboxMessage);

    await queryRunner.connect();
    await queryRunner.startTransaction();

    try {
        const transactionalGameTicketRepository =
        ↪  queryRunner.manager.withRepository(gameTicketRepository);
```

```
11          const transactionalOutboxRepository =
            ↪  queryRunner.manager.withRepository(outboxRepository);
12
13          const updated = await transactionalGameTicketRepository.update(
14              {id: ticketSeatPayment.ticketId, status:
                ↪  TicketStatus.AVAILABLE},
15              {status: TicketStatus.PENDING, reserved_at: new
                ↪  Date(Date.now()), reservation_id:
                ↪  ticketSeatPayment.reservationId}
16          );
17
18          if(updated.affected === 0) {
19              await queryRunner.rollbackTransaction();
20
21              return {
22                  gameTicketId: ticketSeatPayment.ticketId,
23                  status: PaymentStatus.Failed,
24                  message: "Karta nije dostupna"
25              };
26          }
27
28          const paymentRequestedMessage: PaymentRequestedMessage = {
29              reservationId: ticketSeatPayment.reservationId ?? null,
30              gameTicketId: ticketSeatPayment.ticketId,
31              userId: ticketSeatPayment.userId,
32              amount: ticketSeatPayment.amount,
33              currency: ticketSeatPayment.currency,
34          };
35
36          const outboxMessage = new OutboxMessage();
37          outboxMessage.message_id = randomUUID();
38          outboxMessage.type = MessageTypes.PaymentRequested;
39          outboxMessage.routing_key = RoutingKeys.PaymentRequested;
40          outboxMessage.payload =
            ↪  \emph{JSON}.stringify(paymentRequestedMessage);
```

```
41          outboxMessage.created_at = new Date();
42
43          await transactionalOutboxRepository.save(outboxMessage);
44          await queryRunner.commitTransaction();
45
46          return {
47              gameTicketId: ticketSeatPayment.ticketId,
48              status: PaymentStatus.Pending,
49              message: "Rezervacija uspesna, cekanje na placanje",
50              userId: ticketSeatPayment.userId,
51              currency: ticketSeatPayment.currency,
52              amount: ticketSeatPayment.amount
53          }
54      } catch (error){
55          await queryRunner.rollbackTransaction();
56
57          return {
58              gameTicketId: ticketSeatPayment.ticketId,
59              status: PaymentStatus.Failed,
60              message: "Karta nije dostupna"
61          }
62      }
63  }
```

Након завршетка трансакције, догађај још није стигао у ред за чекање, па је потребно имплементирати процесор за слање у реда за поруке, а самим тим и ред за поруке.

Да би се имплементирао механизам реда за поруке, дефинише се заједнички директоријум shared, и то се дефинише као npm пројекат, да би обе апликације могле да имају референцу на исти пројекат. Разлог за тиме је то да се оба микросервиса морају усагласити око истих назива за делове који су им битни.

Унутар *src/shared*, је приказана топологија брокера за поруке у датотеци *topology.ts*.

```
export const EXCHANGE = "outbox.exchange";
export const EXCHANGE_TYPE = "direct";

export const DEAD_LETTER_EXCHANGE = "outbox.dlx";
export const DEAD_LETTER_QUEUE = "outbox.dlq";
export const DEAD_LETTER_ROUTING_KEY = "outbox.dead-letter";

export const RoutingKeys = {
    PaymentRequested: "payment.requested",
    PaymentCompleted: "payment.completed",
} as const;

export const Queues = {
    PaymentRequested: "payment-requested",
    PaymentCompleted: "payment-completed",
} as const;

export const MessageTypes = {
    PaymentRequested: "PaymentRequested",
    PaymentCompleted: "PaymentCompleted",
} as const;

export type RoutingKey = (typeof RoutingKeys)[keyof typeof RoutingKeys];
export type QueueName = (typeof Queues)[keyof typeof Queues];
export type MessageType = (typeof MessageTypes)[keyof typeof
↪  MessageTypes];
```

Овде постоји неколико типова константи, и први тип константи се везује за начин комуникације као и место где ће шаљу поруке, и то су константе *EXCHANGE* и *EXCHANGE_TYPE*, где се врши директна комуникација која иде на *outbox.exchange*.

Други тип константи је везан за поруке које нису доспеле до примаоца.

- *DEAD_LETTER_EXCHANGE* - размена за поруке које нису доспеле
- *DEAD_LETTER_QUEUE* - ред порука у које се смештају неуспешно извршене поруке.
- *DEAD_LETTER_ROUTING_KEY* - веза између размене и реда.

Трећи тип је везан за адресе порука, и то се дефинише унутар *RoutingKeys*.

- *PaymentRequested* - константа која има вредност „payment.requested" и пошиљалац је микросервис за утакмице, док је прималац микросервис за плаћање.
- *PaymentCompleted* - константа која има вредност „payment-completed" и пошиљалац је микросервис за плаћање, док је прималац микросервис за утакмице.

Четврти тип константи је везан за називе редова где примаоци примају поруку, и за наш пример нама ће требати два реда порука, дефинисаних у *Queues* променљивој: *PaymentRequested*, *PaymentCompleted*.

- *PaymentRequested* - ред порука који слуша микросервис за плаћање.
- *PaymentCompleted* - ред порука који слуша микросервис за утакмице.

Пети тип константи је везан за типове догађаја под променљивом *MessageTypes*, који такође имају вредност PaymentRequested и *PaymentCompleted*. Иако *RoutingKeys* и *MessageTypes* имају слична имена константи, немају исту намену. Код *MessageTypes* чува се семантика догађаја, тј. оно што иде у базу података у виду payload-а и касније у ред за поруке, док RoutingKeys чувају адресе рутирања.

У датотеци *connection.ts*, методи *assertTopology* се дешава физичко креирање структуре *RabbitMQ*.

```
1 const assertTopology = async (ch: Chan, consumer: ConsumerBinding) => {
2     await ch.assertExchange(EXCHANGE, EXCHANGE_TYPE, { durable: true });
3
```

```
    await ch.assertExchange(DEAD_LETTER_EXCHANGE, "direct", { durable:
    ↪ true });
    await ch.assertQueue(DEAD_LETTER_QUEUE, { durable: true });
    await ch.bindQueue(DEAD_LETTER_QUEUE, DEAD_LETTER_EXCHANGE,
    ↪ DEAD_LETTER_ROUTING_KEY);

    await ch.assertQueue(consumer.queue, {
        durable: true,
        deadLetterExchange: DEAD_LETTER_EXCHANGE,
        deadLetterRoutingKey: DEAD_LETTER_ROUTING_KEY,
    });
    await ch.bindQueue(consumer.queue, EXCHANGE, consumer.routingKey);

    await ch.prefetch(10);
};
```

Конекција ка *RabbitMQ* је приказана у методи connectToRabbitMQ

```
export const connectRabbitMq = async (
    consumer: ConsumerBinding,
    ready?: (channel: Chan) => Promise<void> | void,
    retries = 10,
): Promise<Chan> => {
    binding = consumer;
    onReady = ready;
    closing = false;

    for (let attempt = 1; attempt <= retries; attempt++) {
        try {
            await establish();
            return channel!;
        } catch (err) {
            const message = err instanceof Error ? err.message :
            ↪ String(err);
            console.warn(`RabbitMQ connect attempt ${attempt}/${retries}
            ↪ failed: ${message}`);
```

```
17              if (attempt === retries) throw err;
18              await new Promise((resolve) => setTimeout(resolve,
                ↪  RECONNECT_DELAY_MS));
19          }
20      }
21
22      throw new Error("Unable to connect to \emph{RabbitMQ}");
23  };
```

и у app.ts дефинише се конекција приликом покретања програма

```
1   await connectRabbitMq(
2               { queue: Queues.PaymentCompleted, routingKey:
                ↪  RoutingKeys.PaymentCompleted },
3               (channel) => registerPaymentCompletedConsumer(channel),
4           );
```

Ово је пример конекције на *RabbitMQ* у микросервису за утакмице, где се дефинише ред и адреса реда где се, након што микросервис за плаћање изврши радњу, очекује порука о извршеној радњи, и у функцији повратног позива дефинише се шта се позива кад прималац прими поруку а то је *`(channel) => registerPaymentCompletedConsumer(channel)`*. Функција се приликом конекције чува у променљивој *`onReady`*. Затим се retries број пута креира канал за поруке у методи establish.

```
1   const establish = async () => {
2       connection = await amqp.connect(RABBITMQ_URL);
3       channel = await connection.createChannel();
4
5       await assertTopology(channel, binding!);
6
7       connection.on("error", (err) => console.error("RabbitMQ greska pri
        ↪  konekciji:", err.message));
8       connection.on("close", () => {
9           if (closing) return;
```

```
10          console.warn(`RabbitMQ konekcija zatvorena, ponovno pokretanje
            ↪  za ${RECONNECT_DELAY_MS}ms`);
11          channel = null;
12          connection = null;
13          setTimeout(() => {
14              establish().catch((err) => console.error("RabbitMQ
                ↪  rekonekcije neuspesna:", err.message));
15          }, RECONNECT_DELAY_MS);
16      });
17
18      if (onReady) await onReady(channel);
19
20      console.log("RabbitMQ konektovan i topologija podesena");
21  };
```

Креира се канал у *`channel = await connection.createChannel()`* и за тај канал позивање функције повратног позива, која се чува у *`onReady`*, асинхроно *`await onReady(channel)`*.

Функција повратног позива која се користи приликом конекције на *RabbitMQ* је *`registerPaymentCompletedConsumer`*, која за параметар прима канал који је дефинисан приликом конекције, и користећи методу *`channel.consume`*, која садржи ред на ком се чека порука као и функцију повратног позива која садржи поруку, порука се парсира у објекат који одговара параметру метода *`PaymentCompletedMessage`* за које, у зависности од успешности плаћања, се потврђује резервација карте, или се отказује резервација.

```
1   import { Channel, ConsumeMessage } from "amqplib";
2   import { PaymentCompletedMessage, Queues } from "@app/contracts";
3   import { gameService } from "../../services/gameService";
4
5   export const registerPaymentCompletedConsumer = (channel: Channel) => {
6       channel.consume(Queues.PaymentCompleted, async (msg: ConsumeMessage
        ↪  | null) => {
7           if (!msg) return;
8
```

```
9           try {
10              const event = \emph{JSON}.parse(msg.content.toString()) as
                ↪  PaymentCompletedMessage;
11
12              if (event.success) {
13                  await gameService.confirmTicket(event.gameTicketId);
14              } else {
15                  await gameService.releaseTicket(event.gameTicketId);
16              }
17
18              channel.ack(msg);
19          } catch (err) {
20              console.error("Failed to process payment-completed
                ↪  message:", err);
21              channel.nack(msg, false, false);
22          }
23      });
24  };
25
```

Након дефинисања конекције ка брокеру за поруке, може се дефинисати и позадински обављач послова у датотеци *outboxProcessor.ts*.

```
1   import { DataSource } from "typeorm";
2   import { OutboxMessage } from "../data/entity/OutboxMessage";
3   import { publishMessage } from "./publisher/publishMessage";
4
5   const BATCH_SIZE = 10;
6
7   export const processOutbox = async (dataSource: DataSource):
    ↪  Promise<void> => {
8       await dataSource.transaction(async (manager) => {
9           const messages = await manager
10              .getRepository(OutboxMessage)
11              .createQueryBuilder("message")
```

```
            .setLock("pessimistic_write")
            .setOnLocked("skip_locked")
            .where("message.processed_at IS NULL")
            .orderBy("message.created_at", "ASC")
            .take(BATCH_SIZE)
            .getMany();

        for (const message of messages) {
            try {
                await publishMessage(message.routing_key,
                ↪  message.payload, {
                    messageId: message.message_id,
                    type: message.type,
                });

                message.processed_at = new Date();
                message.error = null;
                await manager.save(message);
            } catch (err) {
                message.retry_count += 1;
                message.error = err instanceof Error ? err.message :
                ↪  String(err);
                await manager.save(message);
                console.error(`Failed to publish outbox message
                ↪  ${message.message_id}:`, message.error);
            }
        }
    });
};

export const startOutboxProcessor = (dataSource: DataSource, intervalMs
↪  = 2000): NodeJS.Timeout => {
    let running = false;

    return setInterval(async () => {
```

```
43          if (running) return;
44          running = true;
45          try {
46              await processOutbox(dataSource);
47          } catch (err) {
48              console.error("Outbox processor run failed:", err);
49          } finally {
50              running = false;
51          }
52      }, intervalMs);
53  };
```

Метода ***startOutboxProcessor*** се дефинише у *app.ts* датотеци, и унутар те методе, на сваке две секунде позива се метода ***processOutbox***. Метода ***processOutbox*** узима поруке које нису обрађене(***processed_at*** има вредност ***null***) и позивају за сваку поруку ***publishMessage*** методу из ***publishMessage.ts*** датотеке.

```
1   import { EXCHANGE } from "@app/contracts";
2   import { getChannel } from "../connection";
3
4   export interface PublishOptions {
5       messageId?: string;
6       type?: string;
7   }
8
9   export const publishMessage = async (
10      routingKey: string,
11      payload: string,
12      options: PublishOptions = {},
13  ): Promise<void> => {
14      const channel = getChannel();
15      const ok = channel.publish(EXCHANGE, routingKey,
        ↪  Buffer.from(payload), {
16          persistent: true,
```

```
17          contentType: "application/json",
18          messageId: options.messageId,
19          type: options.type,
20      });
21      if (!ok) {
22          await new Promise<void>((resolve) => channel.once("drain", () =>
            ↪  resolve()));
23      }
24  };
```

Кад је у питању коришћење редова за поруке код микросервиса за плаћање, механизам је исти као и код микросервиса за утакмице, само што за Consumer-а поруке позива се логика за плаћање, а порука се шаље након успешног обављеног плаћања, на следећи начин

```
1   const executePayment = async (request: PaymentRequestedMessage):
    ↪  Promise<PaymentResult | null> => {
2       const queryRunner = AppDataSource.createQueryRunner();
3
4       await queryRunner.connect();
5       await queryRunner.startTransaction();
6
7       try {
8           const status = request.amount <= 0 ? PaymentStatus.Failed :
            ↪  PaymentStatus.Completed;
9
10          const transaction = queryRunner.manager.create(Transaction, {
11              game_ticket_id: String(request.gameTicketId),
12              user_id: request.userId,
13              amount: request.amount,
14              currency: request.currency,
15              status,
16          } as Partial<Transaction>);
17
```

```
18          const savedTransaction = await
            ↪  queryRunner.manager.save(transaction);
19
20          const paymentCompletedMessage: PaymentCompletedMessage = {
21              reservationId: request.reservationId,
22              gameTicketId: request.gameTicketId,
23              transactionId: savedTransaction.id,
24              success: status === PaymentStatus.Completed,
25              message: status === PaymentStatus.Completed
26                  ? "Placanje izvrseno uspesno"
27                  : "Placanje izvrseno neuspesno",
28          };
29
30          const outboxMessage = new OutboxMessage();
31          outboxMessage.message_id = randomUUID();
32          outboxMessage.type = MessageTypes.PaymentCompleted;
33          outboxMessage.routing_key = RoutingKeys.PaymentCompleted;
34          outboxMessage.payload =
            ↪  \emph{JSON}.stringify(paymentCompletedMessage);
35          outboxMessage.created_at = new Date();
36
37          await queryRunner.manager.save(outboxMessage);
38
39          await queryRunner.commitTransaction();
40
41          return {
42              transactionId: savedTransaction.id,
43              gameTicketId: savedTransaction.game_ticket_id,
44              userId: savedTransaction.user_id,
45              amount: savedTransaction.amount,
46              currency: savedTransaction.currency,
47              status: savedTransaction.status,
48              createdAt: savedTransaction.created_at,
49              message: "Uspesno ste izvrsili placanje",
50          };
```

```
51      } catch (error) {
52          await queryRunner.rollbackTransaction();
53          return null;
54      } finally {
55          await queryRunner.release();
56      }
57  }
```

Након извршене уплате уписује се у *Outbox* табелу информација о радњи.

Ово је био пример имплементације у *Node.js* радном оквиру, где је било потребно на почетку дефинисати топологију брокера, као и позадински обављач послова који је служио за слање порука из *Outbox* табеле.

6.4.2 Имплементација у радном оквиру *.NET*

Што се тиче имплементације у радном оквиру *.NET*, за цео процес користиће се библиотека *MassTransit*, поред тога што примарно нуди алате за рад са дистрибуираним системима, укључујући подршку за *RabbitMQ* брокера за поруке, нуди подршку за *Transactional Outbox* образац, где не постоји потреба да се користи екстерна библиотека да би се имплементирао позадински обављач задатака, као ни експлицитно да се дефинише *OutboxMessage* табела.

Конфигурација за редове порука и за остале потребне ствари које ће за *Outbox* образац у погледу табела и спољног обављача послова дати *MassTransit*, се дефинише у *InfrastructureServiceRegistration.cs*.

```
1  services.AddMassTransit(x =>
2      {
3      x.AddConsumer<PaymentCompletedConsumer>();
4
5      x.AddEntityFrameworkOutbox<GameDbContext>(o =>
6      {
7          o.QueryDelay = TimeSpan.FromSeconds(1);
8          o.UsePostgres();
```

```
9           o.UseBusOutbox();
10      });
11
12      x.AddConfigureEndpointsCallback((context, name, cfg) =>
13      {
14          cfg.UseEntityFrameworkOutbox<GameDbContext>(context);
15      });
16
17      x.UsingRabbitMq((context, cfg) =>
18      {
19          cfg.Host("localhost", "/", h =>
20          {
21              h.Username("guest");
22              h.Password("guest");
23          });
24          cfg.ConfigureEndpoints(context);
25      });
26  });
```

У линији *x.AddConsumer<PaymentCompletedConsumer>()* дефинисана је логика по којој микросервис за утакмице прима поруку. У тој класи чији је параметар интерфејс *IGameRepository*, који подсећања ради чини скуп метода за рад са базом података, дефинисана је метода Consume. У методи *Consume* је описан процес који ради микросервис за утакмице, и након што прими поруку, ако је процес плаћања успешан, потврђује се резервација карте, иначе се отказује.

```
1   public async Task Consume(ConsumeContext<PaymentCompletedMessage>
    ↪  context)
2   {
3       var message = context.Message;
4
5       if (message.Success)
6       {
```

```
7           var confirmed = await
            ↪ _gameRepository.ConfirmTicketAsync(message.GameTicketId);
8       }
9       else
10      {
11          var released = await
            ↪ _gameRepository.ReleaseTicketAsync(message.GameTicketId);
12      }
13  }
```

Резервацију карте је приказана у датотеци *GameService.cs*, где на сличан начин као и у *Node.js* постоји само ажурирање карте у резервисану, али нема експлицитног уписа догађаја у *Outbox* табелу.

```
1   public async Task<bool> ReserveTicketAndPublishAsync(
2       Guid ticketId,
3       Guid reservationId,
4       PaymentRequestedMessage message)
5   {
6       await using var transaction = await
        ↪ _gameDbContext.Database.BeginTransactionAsync();
7       try
8       {
9           var updated = await _gameDbContext.GameTickets
10              .Where(gt => gt.Id == ticketId && gt.Status ==
                ↪ TicketStatus.Available)
11              .ExecuteUpdateAsync(setters => setters
12                  .SetProperty(gt => gt.Status, TicketStatus.Reserved)
13                  .SetProperty(gt => gt.ReservedAt, DateTime.UtcNow)
14                  .SetProperty(gt => gt.ReservationId, reservationId)
15              );
16
17          if (updated == 0)
18          {
19              await transaction.RollbackAsync();
```

```
20              return false;
21          }
22
23          await _publishEndpoint.Publish(message);
24          await _gameDbContext.SaveChangesAsync();
25          await transaction.CommitAsync();
26          return true;
27      }
28      catch
29      {
30          await transaction.RollbackAsync();
31          throw;
32      }
33  }
```

Као додатак у односу на претходне имплементације овог примера је *_publishEndpoint* параметар, интерфејса *IPublishEndpoint*, преко којег уз помоћ методе *Publish* шаље поруку у брокер. Овде након позива те методе, порука не иде директно у брокер, већ *UseBusOutbox()* пресреће *Publish* и уписује ред у *MassTransit* табеле *OutboxMessage / OutboxState* на *SaveChangesAsync* и тек после *CommitAsync* уписује поруку у брокер.

Кад је у питању коришћење редова за поруке код микросервиса за плаћање, механизам је исти као и код микросервиса за утакмице, само што за Consumer-а поруке се зове логика за плаћање, а порука се шаље након успешног обављеног плаћања. У микросервису за плаћање, радња се мења тако што након уплате преко *SaveChangesAsync* уписује се у *Outbox* табелу радњу, која је директно доступна из *MassTransit*-а.

```
1  public async Task ExecutePaymentWithOutboxAsync(PaymentRequestedMessage
   ↪  message)
2  {
3      var status = message.Amount <= 0 ? PaymentStatus.Failed :
       ↪  PaymentStatus.Completed;
4
5      var transaction = new Transaction
```

```
6       {
7           Id = Guid.NewGuid(),
8           GameTicketId = message.GameTicketId,
9           UserId = message.UserId,
10          Amount = message.Amount,
11          Currency = (PaymentCurrency)message.Currency,
12          Status = status,
13          CreatedAt = DateTime.UtcNow,
14          UpdatedAt = DateTime.UtcNow
15      };
16
17      _paymentDbcontext.Transactions.Add(transaction);
18
19      var paymentCompleted = new PaymentCompletedMessage
20      {
21          ReservationId = message.ReservationId,
22          GameTicketId = message.GameTicketId,
23          TransactionId = transaction.Id,
24          Success = status == PaymentStatus.Completed,
25          Message = status == PaymentStatus.Completed
26              ? "Payment executed successfully"
27              : "Payment failed (invalid amount)"
28      };
29
30      await _publishEndpoint.Publish(paymentCompleted);
31      await _paymentDbcontext.SaveChangesAsync();
32  }
```

Закључак је да *.NET* нуди бољу подршку за овакав образац.

Глава 7

Анализа перформанси

Анализа перформанси је урађена на обрасцу *API* композиција, с обзиром на то да обрасци Прекидач и *Transactional Outbox* имају за циљ да прикажу поузданост система. Тестирање перформанси је извршено конкурентним захтевима, коришћењем библиотеке ***k6*** [10].

Библиотека може да се инсталира на више оперативних система, и то преко оперативног система Windows преко наредбе

```
winget install k6 -source winget
```

и преко оперативног система Linux наредбом

```
sudo gpg -k
```

За потребе тестирања креираће се датотека ***script.js*** у програмском језику *JavaScript* који користи библиотеку k6.

```
1 import http from 'k6/http';
2
3 export default function () {
4   http.get('http://localhost:3002/get-transaction-info/2/1');
5 }
```

Под ***http.get*** се ставља *URL* адреса захтева која се шаље *API* капији за информацију о уплати и карти, и овде је постављена *Node.js* адреса микросервиса који ради ту акцију. На сличан начин приликом тестирања *.NET* апликације ставља се *URL* адреса захтева којег обрађује *.NET* микросервис.

Тестирање се врши наредбом

```
k6 run -vus 100 -duration 1m script.js
```

Ова команда тестира оптерећење апликације тако што 100 виртуелних корисника активно шаље захтев на ту адресу 60 секунди. Из оваквог тестирања добијају се метрике као што је број захтева и време извршавања.

Резултати, респективно најпре за *.NET* апликацију па затим и за *Node.js* апликацију су приказани на следећим сликама.

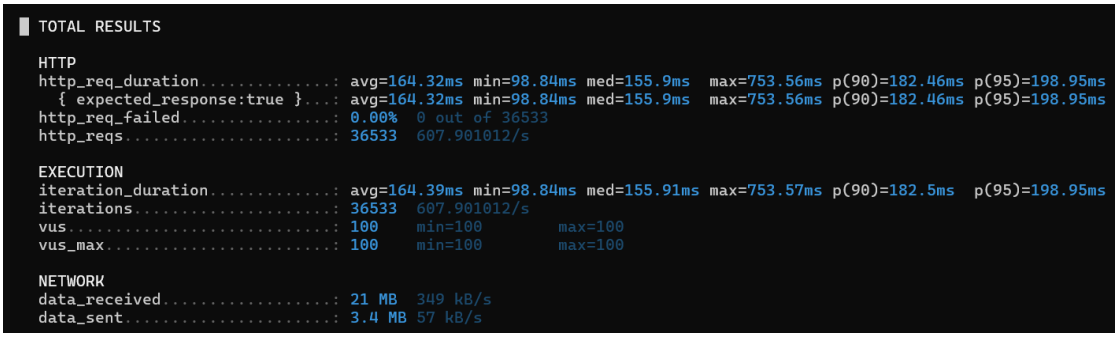
```
TOTAL RESULTS

HTTP
http_req_duration..............: avg=97.85ms min=5.94ms med=92.53ms max=1.58s p(90)=115.39ms p(95)=134.14ms
  { expected_response:true }...: avg=97.76ms min=5.94ms med=92.52ms max=1.57s p(90)=115.39ms p(95)=134ms
http_req_failed................: 0.00% 4 out of 61275
http_reqs......................: 61275 1020.702313/s

EXECUTION
iteration_duration.............: avg=97.93ms min=5.94ms med=92.58ms max=1.59s p(90)=115.44ms p(95)=134.19ms
iterations.....................: 61275 1020.702313/s
vus............................: 100   min=100        max=100
vus_max........................: 100   min=100        max=100

NETWORK
data_received..................: 36 MB 603 kB/s
data_sent......................: 11 MB 183 kB/s
```

Слика 7.1: Анализа перформанси .NET



Слика 7.2: Анализа перформанси Node.js

На сликама се види да су перформансе *.NET* система значајно боље, где је *.NET* обрадио 61,275 захтева, док је у истом временском интервалу *Node.js* обрадио 36,533 захтева.

Просек *.NET* система кад је у питању време извршавања је 97.85ms, а медијана (med) 92.53ms. Код *Node.js* система просек је 164.32ms а медијана (med) 155.9ms, док је *.NET* скоро дупло бржи у просеку. Занимљиво је приметити да је и минимално време код *.NET* система скоро 20 пута брже него код *Node.js* система. Изузетак је максимално време захтева (max), где бољи резултат показује *Node.js* систем.

Анализа перформанси говори у прилог да *.NET*, поред свог модела више нити који може да обради велики број захтева, нуди и широк скуп библиотека које пружају високе перформансе као што су *YARP* и EFCore. Како овај тест није мерио само перформансе радних оквира *.NET* и *Node.js* већ и њихових библиотека, резултати су ограничени на конкретне имплементације и избор радних оквира и библиотека.

Глава 8

Закључак

.NET и *Node.js* су радни оквири који нуде програмерима широк спектар библиотека и радних оквира за развој модерних веб апликација. *.NET* се традиционално користио за прављење сложених (енг. enterprise) апликација, и његов вишенитни модел је одговарао том задатку. *Node.js* као новији радни оквир, је неколико година уназад јако напредовао у популарности, што због брзине, што због једноставности коришћења, као и због могућности да се *JavaScript* може користити на страни сервера.

Примењујући обрасце за микросервисну архитектуру, закључак је да *.NET* нуди бољу подршку у виду већег броја алата који пружају високе перформансе, али је зато и значајно комплекснији од *Node.js* система. Приликом избора неких од двеју радних оквира треба узети у обзир комплексност апликације као и искуство развојног тима. *.NET*, иако има боље перформансе и већи екосистем, није најприкладнији за коришћење код изградње мањих система или ако је развојни тим мање искусан.

Библиографија

[1] Amqplib документација. https://amqp-node.github.io/amqplib/.

[2] CQRS образац. https://learn.microsoft.com/en-us/azure/architecture/patterns/cqrs.

[3] Docker dokumentacija. https://docs.docker.com/manuals/.

[4] Entity Framework Core документација. https://learn.microsoft.com/en-us/ef/core/.

[5] Express.js документација. https://expressjs.com/.

[6] GitHub репозиторијум .NET апликације. https://github.com/ste1drag/Microservice-patterns-dotnet.

[7] GitHub репозиторијум Node.js апликације. https://github.com/ste1drag/Microservice-patterns-nodejs.

[8] GitHub репозиторијум пројекта са курса „Информациони системи”. https://github.com/SpotRusherZ/evidencija-zaposlenih.

[9] http-proxy-middleware документација. https://www.jsdocs.io/package/http-proxy-middleware.

[10] K6 документација. https://grafana.com/docs/k6/latest/.

[11] MassTransit документација. https://masstransit.massient.com/.

[12] Mediator образац. https://mediatr.io/.

[13] .NET документација. https://learn.microsoft.com/en-us/dotnet/.

[14] Polly документација. https://www.pollydocs.org/.

[15] PostgreSQL документација. https://www.postgresql.org/docs/.

[16] RabbitMQ документација. https://www.rabbitmq.com/docs.

[17] TypeORM документација. https://typeorm.io/.

[18] TypeScript документација. https://www.typescriptlang.org/docs/.

[19] Yarp документација. https://learn.microsoft.com/en-us/aspnet/core/fundamentals/servers/yarp/getting-started?view=aspnetcore-10.0.

[20] Elisabeth Robson Eric Freeman. *Um caruje: Projektni obrasci*. Mikro knjiga, 2021.

[21] Vladimir Filipović. Materijali sa predavanja „Uvod u veb i internet tehnologije". https://matfuvit.github.io/info/predavanja/tekstovi/JavaScript-Programiranje-Pomocu-Node.html.

[22] Saša Malkov. *Razvoj softvera*. Matematički fakultet, 2024.

[23] Sam Newman. *Monolith to Microservices*. O'REILLY, 2019.

[24] Sam Newman. *Izgradnja mikroservisa*. CET, 2022.

[25] Chris Richardson. *Microservices Patterns: With examples in Java*. Manning, 2019.

[26] Jeffrey Richter. *CLR via C#*. Microsoft Press, 2012.

Биографија аутора

Стеван Драговић је рођен 9. јануара 1997. године у Београду, где је завршио основну и средњу школу. Математички факултет је уписао 2016. године а завршио је у фебруару 2023. године, на модулу Рачунарство и информатика. Исте године уписује мастер студије на модулу Математика и рачунарство. Професионално искуство је стекао радећи у школи као наставник математике и у индустрији, где је стекао искуства у програмским језицима *C#* и *TypeScript*.