



УНИВЕРЗИТЕТ У НОВОМ САДУ
ФАКУЛТЕТ ТЕХНИЧКИХ НАУКА У
НОВОМ САДУ



Денеш Сентмартони

Имплементација обсервабилности микросервисне архитектуре онлајн продавнице

ЗАВРШНИ РАД
- Основне академске студије -

Нови Сад, 2023



УНИВЕРЗИТЕТ У НОВОМ САДУ • ФАКУЛТЕТ ТЕХНИЧКИХ НАУКА
21000 НОВИ САД, Трг Доситеја Обрадовића 6

КЉУЧНА ДОКУМЕНТАЦИЈСКА ИНФОРМАЦИЈА


Редни број, РБР:	
Идентификациони број, ИБР:	
Тип документације, ТД:	Монографска публикација
Тип записа, ТЗ:	Текстуални штампани документ/ЦД
Врста рада, ВР:	Завршни-bachelor рад
Аутор, АУ:	Денеш Сентмартони
Ментор, МН:	др Себастијан Стоја, доцент
Наслов рада, НР:	Имплементација обсервабилности микросервисне архитектуре онлајн продавнице
Језик публикације, ЈП:	Српски (ћирилица)/Srpski (latinica)
Језик извода, ЈИ:	Српски/Енглески
Земља публикавања, ЗП:	Србија
Уже географско подручје, УГП:	Војводина
Година, ГО:	2023
Издавач, ИЗ:	Ауторски репринт
Место и адреса, МА:	Факултет техничких наука (ФТН), Д. Обрадовића 6, 21000 Нови Сад
Физички опис рада, ФО: (поглавља/страна/ цитата/табела/слика/графика/прилога)	8/29/9/0/37/0/0
Научна област, НО:	Електротехничко и рачунарско инжењерство
Научна дисциплина, НД:	Примењено софтверско инжењерство
Предметна одредница/Кључне речи, ПО:	Микросервис, обсервабилност
УДК	
Чува се, ЧУ:	Библиотека ФТН, Д. Обрадовића 6, 21000 Нови Сад
Важна напомена, ВН:	
Извод, ИЗ:	У овом раду су се имплементирани шаблони обсервабилности попут Health check API, Log aggregation и exception tracking у микросервисној архитектури онлајн продавнице како би се омогућила имплементација микросервиса са потпуном видљивошћу у реалном времену.
Датум прихватања теме, ДП:	
Датум одбране, ДО:	
Чланови комисије, КО:	Председник: др Срђан Вукмировић, редовни професор
	Члан: др Бојан Јелачић, доцент
	Члан, ментор: др Себастијан Стоја, доцент
	Потпис ментора

Образац **Q2.НА.04-05** - Издање 1



KEY WORDS DOCUMENTATION

Accession number, ANO :	
Identification number, INO :	
Document type, DT :	Monographic publication
Type of record, TR :	Textual material, printed /CD
Contents code, CC :	Bachelor thesis
Author, AU :	Deneš Sentmartoni
Mentor, MN :	Sebastijan Stoja, PhD, assistant professor
Title, TI :	Implementation of observability of microservice architecture online market
Language of text, LT :	Serbian (cyrillic script)/ Serbian (latin script)
Language of abstract, LA :	Serbian/English
Country of publication, CP :	Serbia
Locality of publication, LP :	Vojvodina
Publication year, PY :	2023
Publisher, PB :	Author reprint
Publication place, PP :	Faculty of Technical Sciences, D. Obradovića 6, 2100 Novi Sad
Physical description, PD : (chapters/pages/ref./tables/pictures/graphs/appendixes)	8/29/9/0/37/0/0
Scientific field, SF :	Electrical and computer engineering
Scientific discipline, SD :	Power Software Engineering
Subject/Key words, S/KW :	Microservice, observability
UC	
Holding data, HD :	Library of the Faculty of Technical Sciences, D. Obradovića 6, 2100 Novi Sad
Note, N :	
Abstract, AB :	In this article, observability patterns such as Health check API, Log aggregation and exception tracking are implemented in the microservice architecture online market to enable the implementation of microservices with full real-time visibility.
Accepted by the Scientific Board on, ASB :	
Defended on, DE :	
Defended Board, DB :	President: PhD Srđan Vukmirović, full professor
	Member: PhD Bojan Jelačić, Assistant professor
	Member, Mentor: PhD Sebastijan Stoja, Assistant professor
	Mentor's sign

	УНИВЕРЗИТЕТ У НОВОМ САДУ • ФАКУЛТЕТ ТЕХНИЧКИХ НАУКА 21000 НОВИ САД, Трг Доситеја Обрадовића 6	Број:
	ЗАДАТАК ЗА ДИПЛОМСКИ (BACHELOR) РАД	Датум:

(Податке уноси предметни наставник - ментор)

СТУДИЈСКИ ПРОГРАМ:	Примењено софтверско инжењерство
РУКОВОДИЛАЦ СТУДИЈСКОГ ПРОГРАМА:	др Александар Селаков, ванредни професор

Студент:	Денеш Сентмартони	Број индекса:	ПР107/2019
Област:	Електротехника и рачунарство		
Ментор:	др Себастијан Стоја, доцент		
НА ОСНОВУ ПОДНЕТЕ ПРИЈАВЕ, ПРИЛОЖЕНЕ ДОКУМЕНТАЦИЈЕ И ОДРЕДБИ СТАТУТА ФАКУЛТЕТА ИЗДАЈЕ СЕ ЗАДАТАК ЗА ДИПЛОМСКИ (BACHELOR) РАД, СА СЛЕДЕЋИМ ЕЛЕМЕНТИМА: <ul style="list-style-type: none"> - проблем – тема рада; - начин решавања проблема и начин практичне провере резултата рада, ако је таква провера неопходна; 			

НАСЛОВ ДИПЛОМСКОГ (BACHELOR) РАДА:

Имплементација обсервабилности микросервисне архитектуре онлајн продавнице
--

ТЕКСТ ЗАДАТКА:

<p>Извршити преглед литературе, идентификовати све могуће поделе постојећих сервиса на микросервисне користећи методе доступне у литератури. Имплементирати поделу онлајн продавнице. Употребити шаблоне попут Health check API, Log aggregation, distributed tracing, application metrics, exception tracking и audit logging како би се имплементирали микросервиси са потпуном видљивошћу у реалном времену. Имплементирати страницу за преглед или адаптер за постојећа мониторинг решења.</p>
--

Руководилац студијског програма:	Ментор рада:

Примерак за: <input type="checkbox"/> - Студента; <input type="checkbox"/> - Ментора
--

Sadržaj

<i>Sadržaj slika</i>	<i>2</i>
<i>Skraćenice</i>	<i>3</i>
<i>1. Uvod.....</i>	<i>4</i>
<i>1.1 . Monolitna aplikacija u odnosu na mikroservise</i>	<i>4</i>
<i>2. Korišćene tehnologije</i>	<i>7</i>
<i>3. Tranzicija monolitne arhitekture na mikroservisnu arhitekturu</i>	<i>9</i>
<i>3.1 . Učesnici u sistemu</i>	<i>9</i>
<i>3.2 . Model Podataka</i>	<i>9</i>
<i>3.3 . Implementacij mikroservisne arhitekture</i>	<i>11</i>
<i>4. Šabloni observabilnosti.....</i>	<i>21</i>
<i>5. Testiranje.....</i>	<i>26</i>
<i>6. Zaključak.....</i>	<i>27</i>
<i>7. Literatura.....</i>	<i>28</i>
<i>8. Biografija.....</i>	<i>29</i>

Sadržaj slika

- Slika 1.1.1. - Monolitna arhitektura
- Slika 1.1.2. - Mikroservisna arhitektura
- Slika 3.1. - Sistem monolitne aplikacije
- Slika 3.2.1. - Klasa *User*
- Slika 3.2.2. - Klasa *Article*
- Slika 3.2.3. - Klasa *Order*
- Slika 3.2.4. - Pomoćne klase
- Slika 3.3.1. - Posmatrana monolitna aplikacija
- Slika 3.3.2. - Ciljna mikroservisna arhitektura
- Slika 3.3.3. - Konfiguracija za bazu podataka *UserService*-a
- Slika 3.3.4. - Konfiguraciju za tabelu *Users*
- Slika 3.3.5. - Konfiguracija za tabelu *Articles*
- Slika 3.3.6. - Konfiguracija za tabelu *Orders*
- Slika 3.3.7. - Klasa *WcfUser*
- Slika 3.3.8. - Klasa *MappingProfile*
- Slika 3.3.9. - Interfejs *IwcfUserService*
- Slika 3.3.10. - Interfejs *IwcfOrderService*
- Slika 3.3.11. - Interfejs *IwcfArticleService*
- Slika 3.3.12. - Metoda *AddUser*
- Slika 3.3.13. - Interfejs *IwcfInternalUserService*
- Slika 3.3.14. - Implementacija interfejs *IwcfInternalUserService*
- Slika 3.3.15. - Kod *UserService*-a za WCF server ka veb API-ju
- Slika 3.3.16. - Reference ka servisima
- Slika 3.3.17. - Kod *UserService*-a za WCF server ka ostalim servisima
- Slika 4.1. - Bacanje izuzetka i logovanje
- Slika 4.2. - Hvatanje izuzetaka kod veb API-ja
- Slika 4.3. - Interfejs *IuserServiceHealthCheck*
- Slika 4.4. - Interfejs *IorderServiceHealthCheck*
- Slika 4.5. - Interfejs *IarticleServiceHealthCheck*
- Slika 4.6. - Implementacija metode *HealthCheck*
- Slika 4.7. - Krajnja tačka za *health check API*
- Slika 4.8. - Metoda za *health check*
- Slika 4.9. - Informacije kroz *health check API*
- Slika 4.10. - Metoda *LogAggregation*
- Slika 4.11. - Ispis svih izuzetaka
- Slika 5.1. - Prikaz logovanih izuzetaka
- Slika 5.2. - Prikaz stanja servisa

Skraćenice

WCF - *Windows Communication Foundations*

API - *Application Programming Interface*

TCP - *Transmission Control Protocol*

SQL - *Structured Query Language*

IIS - *Internet Information Services*

REST - *Representational State Transfer*

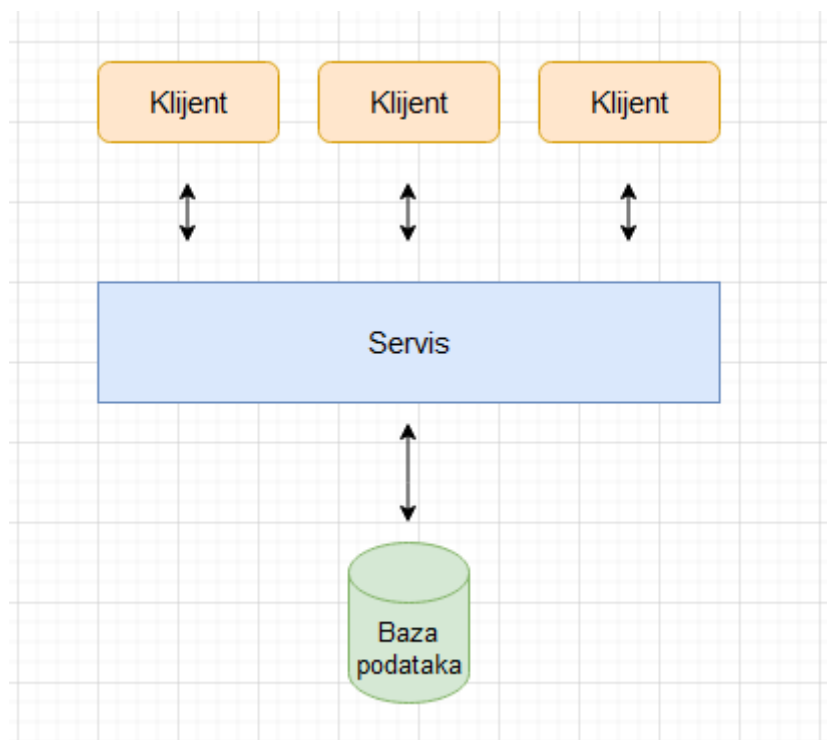
1. UVOD

Mikroservisi su jedna nova softverska arhitektura, čiji zadatak je da se obezbedi bolja komunikacija između različitih platformi, veća jednostavnost i sistem koji je lakši za korisnika. Mikroservisi se obično smatraju tehnikom razvoja softvera koja organizuje aplikaciju kao grupu slabo povezanih servisa. Mikroservisi su arhitektonski i organizacioni pristup razvoju softvera gde se softver sastoji od malih nezavisnih servisa koji komuniciraju preko dobro definisanih API-ja. Mikroservisne arhitekture čine aplikacije lakšim za skaliranje i bržim za razvoj, omogućavajući inovacije i ubrzavajući vreme do puštanja novih funkcija na tržište.

Karakteristike samih mikroservisa su da svaka komponenta servisa u arhitekturi mikroservisa se može razviti, primeniti, upravljati i skalirati bez uticaja na funkcionisanje drugih usluga. Servisi ne moraju da dele svoj kod ili implementaciju sa drugim servisima. Svaka komunikacija između pojedinačnih komponenti se odvija preko dobro definisanih API-ja. Kod inicijalne podele servisa treba gledati da se domen aplikacije podeli na manje nezavisne domene. Svaki servis ima jednu ulogu. Svaki servis treba da ima sopstvenu bazu podataka u kojoj će čuvati sve svoje podatke. Ako softverski inženjeri prošire postojeći mikroservis na način da ima više funkcionalnosti za rešavanje različitih problema a ne samo jednu onda u tom slučaju postojeći servis treba da se razbije na manje servise[1].

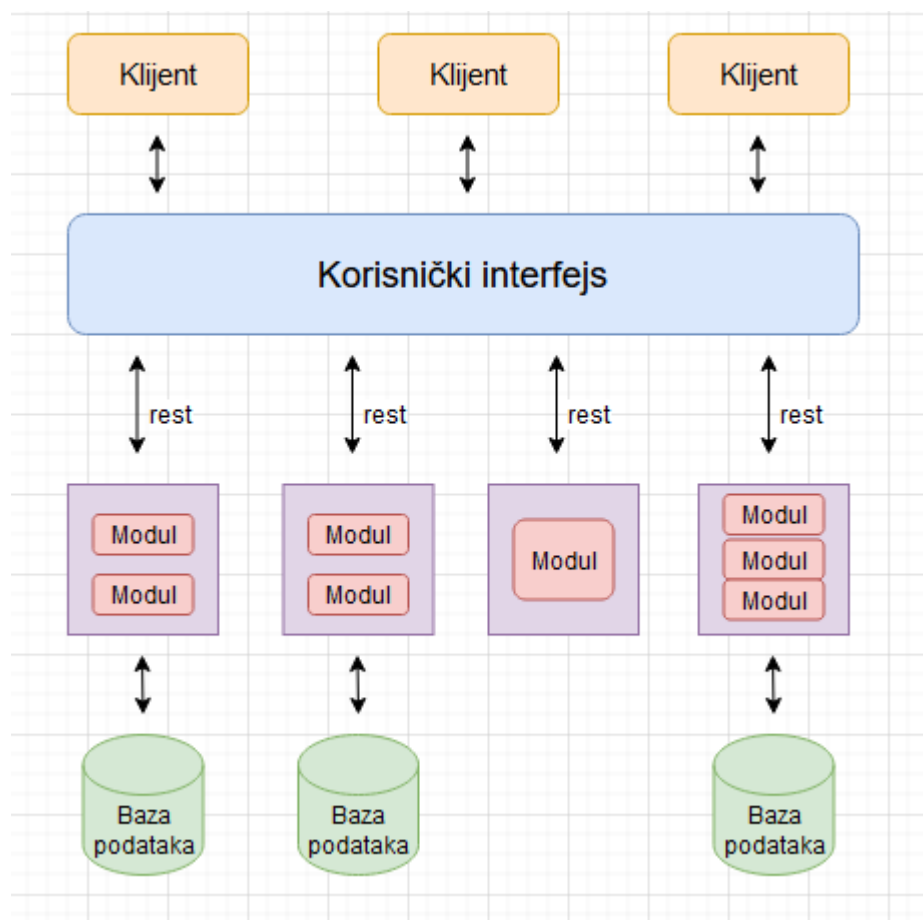
1.1. Monolitna aplikacija u odnosu na mikroservise

U softverskom inženjerstvu monolitna aplikacija je jedinstvena softverska aplikacija koja je samostalna i nezavisna od drugih aplikacija. Monolitna u ovom kontekstu znači sastavljeno sve zajedno i svi međusobno zavise jedan od drugog. Monolitne aplikacije funkcionišu kao jedna celina sa jednom bazom podataka. Sa monolitnom arhitekturom svi procesi su čvrsto povezani i rade kao jedan servis. To znači da ako jedan proces aplikacije doživi nagli porast potražnje, čitava arhitektura mora biti skalirana. Dodavanje ili poboljšanje karakteristika monolitne aplikacije postaje složenija kako baza koda raste. Ova složenost ograničava eksperimentisanje i otežava implementaciju novih ideja. Ako se jedna programska komponenta mora ažurirati drugi elementi takođe mogu zahtevati ponovo pisanje, a cela aplikacija mora ponovo da se kompajlira i testira. Monolitne arhitekture povećavaju rizik za dostupnost aplikacije jer mnogi zavisni i čvrsto povezani procesi povećavaju uticaj neuspeha jednog procesa. Ako dođe do greške u aplikaciji, onda verovatnoća da će cela aplikacija prestati sa radom su veoma velike. Popravljanje grešaka i traženje uzroka grešaka u monolitnoj aplikaciji je veoma teško i ograničeno zbog čvrsto povezanih procesa[1]. Observabilnost je isto otežana zbog same arhitekture. Na sledećoj slici (Slika 1.1.1) je prikazana arhitektura jedne tipične monolitne aplikacije.



Slika 1.1.1. Monolitna arhitektura

Koristeći mikroservisnu arhitekturu, aplikacija je izgrađena kao skup nezavisnih komponenti koje pokreću svaki proces aplikacije kao servis. Ovi servisi komuniciraju preko dobro definisanog interfejsa koristeći lagane API-je. Servisi se grade na način da svaki servis ima jednu ulogu pa zato oni neće direktno zavisiti jednih od drugih. Pošto se pokreću nezavisno svaki servis se može ažurirati, primeniti i skalirati kako bi zadovoljila potražnju za određenim funkcijama aplikacije. Ako dodje do greške u jednom servisu ili ako se iz nekog drugog razloga servis padne, sama aplikacija neće pasti nego će nastaviti da radi, kao i ostali servisi će nastaviti da rade [1]. Mikroservisi rade u realnom vremenu pa je veoma važno da se stalno prate performance, da se zabeleže događaji, da se prate stanja servisa kao i zabeleženje grešaka i izuzetaka. Observabilnost u mikroservisima se postiže metodama kao što su *health check API*, *log aggregation*, *audit logging*, *distributed tracing*, *exception tracking* i *application metrics*. Na sledećoj slici (Slika 1.1.2) će biti prikazana tipična mikroservisna arhitektura.



Slika 1.1.2. Mikroservisna arhitektura

2. KORIŠĆENE TEHNOLOGIJE

Microsoft Visual Studio je alat za programere koji se koristi da se razvije ceo razvojni ciklus na jednom mestu. To je integrisano razvojno okruženje koji se koristi za pisanje, uređivanje koda, otklanjanje grešaka i razvijanje aplikacije. Podržava 36 različitih programskih jezika. Programski jezici koji nisu ugrađeni kao što su *Python* i *Ruby*, oni su dostupni putem programskih usluga koji se instaliraju odvojeno. *Microsoft* pruža i besplatnu verziju *Visual Studio* alata pod nazivom *Community Edition* koja je dostupna bez ikakvih troškova. Ostale dve verzije su *Professional Edition* i *Enterprise Edition*, koje pružaju bolje performanse i više *feature-a*, ali se moraju plaćati.

.NET je *open-source* platforma za pravljenje desktop, web i mobilnih aplikacija koje mogu da rade na bilo kojem operativnom sistemu. *.NET* sistem uključuje alate, biblioteke i jezike koje podržavaju moderan, skalabilan i razvoj softvera visokih performansi, preveđe kod *.NET* programskog jezika u uputstva koja računarski uređaj može da obradi. Aktivna zajednica programera održava i podržava i *.NET* platformu.

.NET Framework je okvir za razvoj softvera za pravljenje i pokretanje aplikacija na *Windows-u*. *.NET Framework* je prva verzija i on je deo *.NET* platforme.

.NET Core je *open-source* okvir i koristiti se za razvoj različitih aplikacija. *Microsoft* je objavio *.NET Core* 2014. godine, i on je multi-platformski naslednik *.NET Framework-a*.

ASP .NET je veb okvir otvorenog koda, kreiran od strane *Microsoft-a*, za pravljenje modernih veb aplikacija i usluga sa *.NET-om*. *ASP .NET* proširuje *.NET* platformu alatima i bibliotekama posebno za pravljenje veb aplikacija.

ASP .NET Web API Core je okvir za pravljenje HTTP servisa kojem se može pristupiti sa bilo kog klijenta uključujući veb pregledače i mobilne uređaje. To je idealna platforma za pravljenje *RESTful* aplikacija na *.NET* okviru.

C# je nastao 2000. godine i on moderan objektno orijentisan jezik koji je deo *.NET* razvojnog okruženja koji omogućava programerima da naprave mnoge vrste sigurnih i robusnih aplikacija koje rade u *.NET-u*. *C#* obuhvata statičko, imperativno, deklarativno, funkcionalno, generičko programiranje i programske discipline orijentisane na komponente.

Representational State Transfer ili skraćeno *Rest* je arhitektonski stil za obezbeđivanje standarda između računarskih sistema na vebu, što olakšava sistemima međusodnu komunikaciju. Sistemi usaglašeni sa *Rest-om*, koji se često nazivaju *RESTfull* sistemi, odlikuju se time što su bez stanja i razdvajaju brige klijenata i servera. Implementacija klijenata i implementacija servera se mogu obavljati nezavisno, a da jedni za druge ne znaju. To znači da se kod na strani klijenata može promeniti u bilo kom trenutku bez uticaja na rad servera, a kod na strani servera može da se promeni bez uticaja na rad klijenata [8].

Hyper Text Transfer Protocol ili skraćeno HTTP je osnova veba i koristi se za učitavanje veb stranica pomoću hipertekstualnih veza. To je protokol sloja aplikacije dizajniran za prenos informacija između umreženih uređaja i radi na vrhu drugih slojeva steka mrežnih protokola. Tipičan tok preko ovog protokola uključuje klijentsku mašinu koja šalje zahtev serveru, koji zatim šalje poruku odgovora.

Entity Framework je ORM(objektno orijentisan mapper) okvir otvorenog koda za *.NET* aplikacije koje podržava *Microsoft*. Omogućava programerima da rade sa podacima koristeći objekte klase specifičnih za domen bez fokusiranja na osnovne tabele i kolone baze podataka u kojima se ovi podaci čuvaju. Sa *Entity Framework-om* programeri mogu da rade na višem nivou apstrakcije kada se rukuju sa podacima i mogu da kreiraju i održavaju aplikacije orijentisane na podatke sa manje koda u poređenju sa tradicionalnim aplikacijama [7].

Windows Communication Foundation ili skraćeno WCF je okvir za izgradnju aplikacija orijentisanih na usluge. Koristeći WCF mogu se slati podaci kao asinhrona poruka sa jedne krajnje tačke usluge na drugu. Krajnja tačka usluge može biti deo stalno dostupne usluge koju hostuje IIS

ili može biti usluga hostovana u aplikaciji. Krajnja tačka može biti klijent usluge koji zahteva podatke od krajnje tačke usluge. Poruke mogu biti jednostavne poput jednog znaka ili reči poslate kao *Extensible Markup Language* (XML) ili složene kao tok binarnih podataka [6].

Transmission Control Protocol ili skraćeno TCP je komunikacioni standard koji omogućava aplikacijskim programima i računarskim uređajima da razmenjuju poruke preko mreže. Dizajniran je da šalje pakete preko interneta i obezbeđuje uspešnu isporuku podataka i poruka preko mreža. TCP organizuje podatke tako da se mogu preneti između servera i klijenta. Garantuje integritet podataka koji se prenose preko mreže. Pre nego što se prenesu podaci, uspostavlja se veza između izvora i njegovog odredišta, za koju obezbeđuje da će ostati živa dok komunikacija ne počne. Zatim razbija velike količine podataka u manje pakete, istovremeno osiguravajući da je integritet podataka očuvan tokom celog procesa.

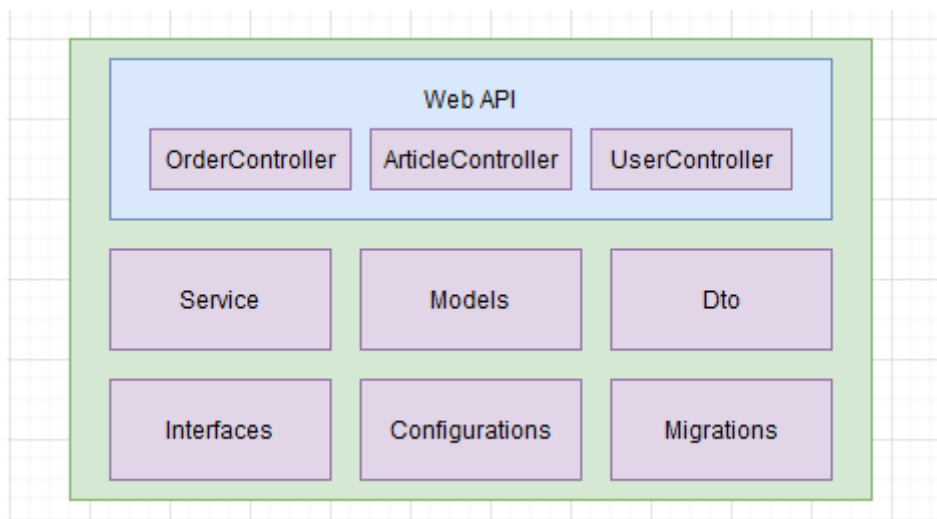
Serilog je biblioteka za evidenciju koja se lako podešava za *.NET* sa jasnim API-jem. Na dugačkoj listi *Serilog*-ovih funkcija mogu se pronaći, podrška za strukturiranog evidentiranja, što omogućava da se evidencije tretiraju kao skupovi podataka a ne kao tekst.

AutoMapper je biblioteka za *.NET* koja se koristi kad god postoji mnogo svojstava podataka za objekte i moramo ih mapirati između objekta izvorne klase i objekta odredišne klase.

Microsoft SQL Server 2019 Express je besplatno izdanje sql servera koje je bogato funkcijama koje su idealne za učenje, razvoj i napajanje desktop, veb i malih serverskih aplikacija. To je sistem za upravljanje relacionim bazama podataka koji je besplatan za preuzimanje, distribuciju i korišćenje. Sastoji se od baze podataka koja je posebno namenjena za ugrađene aplikacije i aplikacije manjeg obima.

3. TRANZICIJA MONOLITNE ARHITEKTURE NA MIKROSERVISNU ARHITEKTURU

Monolitna aplikacija koja će se transformirati u mikroservisnu aplikaciju je jedna veb aplikacija onlajn prodavnice, koja sadrži veb API, sa tri kontrolera. To su kontroleri *UserController*, *ArticleController* i *OrderController*. Sadrži servis koji je ugrađen u nju, koja rukuje sa modelima podataka, koje koriste istu bazu podataka. Dodatno sadrži komponente: *Models*, *Dto*, *Interfaces*, *Configurations* i *Migrations*. Na sledećoj slici (Slika 3.1.) je prikazan sistem monolitne aplikacije.



Slika 3.1. Sistem monolitne aplikacije

3.1. Učesnici u sistemu

Početna monolitna aplikacija je jedna veb aplikacija koja je namenjena da bude korišćena od strane korisnika koji po ulozi mogu biti kupci ili prodavci takodje postoji uloga administrator ali korisnici ne mogu da biraju tu ulogu. Administrator može da odobri ili da odbije registraciju kupca, i može da verifikuje prodavce. Prodavac može da postavlja artikle. Artikal predstavlja jedan fizički objekat koji se može kupiti. Kupac može da postavlja porudzbine gde bira koje artikle želi da kupi.

3.2. Model Podataka

U aplikaciji postoje tri entiteta. Klase *User*, *Article* i *Order* opisuju te entitete. Na slici (Slika 3.2.1.) je prikazan kod koji predstavlja klasu *User* koja opisuje korisnike.

```

public class User
{
    public int UserId { get; set; }

    public string Username { get; set; }

    public string Email { get; set; }

    public string FirstName { get; set; }

    public string LastName { get; set; }

    public string DateOfBirth { get; set; }

    public string Address { get; set; }

    public string UserType { get; set; }

    public byte[] ImageFile { get; set; }

    public string Password { get; set; }

    public Status Approved { get; set; }

    public Status Verified { get; set; }
}

```

Slika 3.2.1. Klasa *User*

Svaki korisnik u sistemu ima jedinstven ceo broj koji označava id, korisničko ime, ime, prezime, datum rođenja, adresu, ulogu korisnika, sliku, lozinku i polja koja označavaju da li je korisnik odobren i verifikovan. Na slici (Slika 3.2.2.) je prikazan kod koji predstavlja klasu *Article* koja opisuje artikal.

```

public class Article
{
    public int ArticleId { get; set; }

    public string Name { get; set; }

    public double Price { get; set; }

    public int Quantity { get; set; }

    public string Description { get; set; }

    public byte[] ImageFile { get; set; }

    public int UserSellerId { get; set; }
}

```

Slika 3.2.2. Klasa *Article*

Svaki artikal u sistemu ima jedinstveni ceo broj koji označava id, ime artikla, cenu, količinu, opis, sliku i id korisnika koji je kreirao artikal. Na slici (Slika 3.2.3.) je prikazan kod koji predstavlja klasu *Order* koja opisuje porudžbine.

```

public class Order
{
    public int OrderId { get; set; }

    public string Comment { get; set; }

    public string Address { get; set; }

    public int Price { get; set; }

    public string DateOfOrder { get; set; }

    public int UserBuyerId { get; set; }

    public string ArticleIdsString
    {
        get => string.Join(",", ArticleIds);
        set => ArticleIds = value.Split(new[] { ',' }, StringSplitOptions.RemoveEmptyEntries)
                                .Select(int.Parse)
                                .ToList();
    }

    public ICollection<int> ArticleIds { get; set; }

    public string AmountOfArticlesString
    {
        get => string.Join(",", AmountOfArticles);
        set => AmountOfArticles = value.Split(new[] { ',' }, StringSplitOptions.RemoveEmptyEntries)
                                    .Select(int.Parse)
                                    .ToList();
    }








    public ICollection<int> AmountOfArticles { get; set; }
}

```

Slika 3.2.3. Klasa *Order*

Svaka porudžbina ima jedinstven ceo broj koji označava id, komentar, adresu, cenu, datum postavke porudžbine, korisnikov id koji je postavio porudžbinu i listu artikala koju je korisnik postavio u porudžbinu. *ArticleIds* je lista id-ova artikala koji se nalaze u porudžbini, a *ArticleIdsString* je tekstualna predstava prethodne liste koja, i ovaj string će se zapravo čuvati u bazi. *AmountOfArticles* je lista celih brojeva koje predstavljaju količine poručenih artikala. *AmountOfArticlesString* je samo tekstualna predstava liste, koja će se čuvati u bazi podataka. Dodatno u sistemu se nalaze i još pomoćne *Data Transfer Object* (DTO) klase koje se koriste kod veb API-ja, u narednoj slici (Slika 3.2.4.) će se prikazati pomoćne klase.

```

▶  ArticleDto.cs
▶  OrderDetailsDto.cs
▶  OrderDto.cs
▶  UserDtoApprovedVerified.cs
▶  UserDtoLogin.cs
▶  UserDtoRegistration.cs
▶  UserDtoStatus.cs

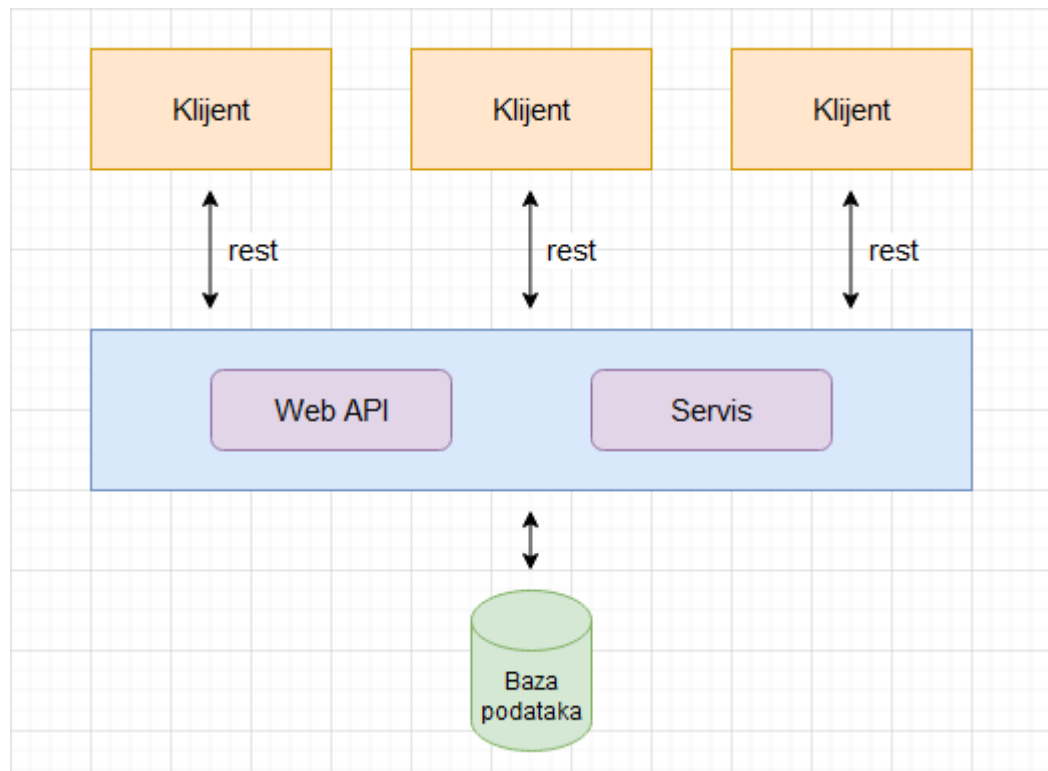
```

Slika 3.2.4. Pomoćne klase

3.3. Implementacij mikroservisne arhitekture

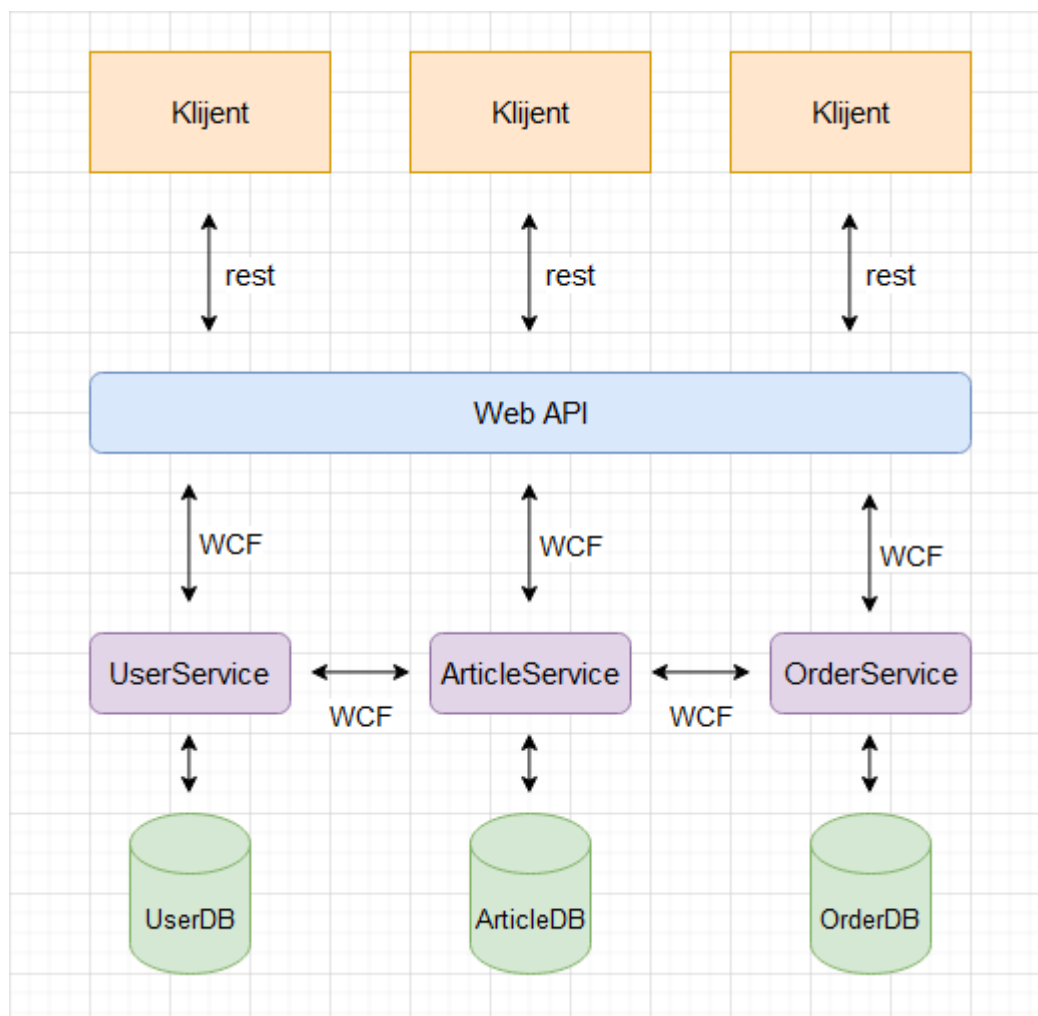
U monolitnoj aplikaciji omogućena je autentifikacija u vidu *login*-a i autorizacija preko uloga. Tri uloge koje se nalaze u sistemu su administrator,prodavac i kupac. Prilikom *login*-a izdaje

se *JavaScript Object Notation Web Token* (JWT) token koji sadrži informacije o ulozu korisnika i preko ovog tokena se vrši autorizacija. Na sledećoj slici (Slika 3.3.1.) će biti prikazana arhitektura monolitne aplikacije koja će se transformirati.



Slika 3.3.1. Posmatrana monolitna aplikacija

Mikroservisna arhitektura će korsiti ista ta tri kontrolera veb API-ja ali će se za obradu podataka napraviti tri servisa svaki sa sopstvenom bazom podataka. Servisi su *UserService* za rad sa korisnicima, *ArticleService* za rad sa artiklima i *OrderService* za rad sa porudžbinama. Servisi će preko WCF protokola sa TCP konekcijom da komuniciraju sa veb API-jem, dodatno servisi će između sebe isto da komuniciraju preko WCF protokola sa TCP konekcijom (Slika 3.3.2).



Slika 3.3.2. Ciljna mikroservisna arhitektura

U aplikaciji za bazu podataka upotrebljen je *Entity Framework* koji će na osnovu klasa kreirati bazu podataka. Sami podaci će se čuvati u *Microsoft SQL Server 2019 Express*. Svaki servis ima konfiguraciju za povezivanje na bazu i konfiguraciju za kreiranje tabele na osnovu klasa. Na slici (Slika 3.3.3.) je prikazan kod koji predstavlja konfiguraciju za bazu podataka *UserService*-a koja se nalazi u *App.config* fajlu. U *connectionString* promenljivi je naveden naziv baze podataka.

```
<entityFramework>
  <providers>
    <provider invariantName="System.Data.SqlClient"
      type="System.Data.Entity.SqlServer.SqlProviderServices, EntityFramework.SqlServer" />
  </providers>
</entityFramework>
<connectionStrings>
  <add name="UserCS" connectionString="data source=.\sqlcxpress;initial catalog=UserDB;integrated security=True;"
    providerName="System.Data.SqlClient" />
</connectionStrings>
```

Slika 3.3.3. Konfiguracija za bazu podataka *UserService*-a

Za *UserService* naziv baze podataka će biti *UserDB*, za *ArticleService* će biti *ArticleDB* a za *OrderService* će biti *OrderDB*. Na slici (Slika 3.3.4.) je prikazan kod koji predstavlja konfiguraciju za tabelu *Users* u bazi sa nazivom *UserDB* koju koristi *UserService*.

```

class UserDBContext : DbContext
{
    public UserDBContext() : base("name = UserCS")
    {
    }

    public DbSet<User> Users { get; set; }

    protected override void OnModelCreating(DbModelBuilder modelBuilder)
    {
        modelBuilder.Entity<User>().HasKey(x => x.UserId);

        modelBuilder.Entity<User>().Property(x => x.Username).HasMaxLength(22);
        modelBuilder.Entity<User>().HasIndex(x => x.Username).IsUnique();

        modelBuilder.Entity<User>().Property(x => x.Email).HasMaxLength(70);
        modelBuilder.Entity<User>().HasIndex(x => x.Email).IsUnique();

        modelBuilder.Entity<User>().Property(x => x.FirstName).HasMaxLength(20);

        modelBuilder.Entity<User>().Property(x => x.LastName).HasMaxLength(20);

        modelBuilder.Entity<User>().Property(x => x.DateOfBirth).HasMaxLength(20);

        modelBuilder.Entity<User>().Property(x => x.Address).HasMaxLength(70);

        modelBuilder.Entity<User>().Property(x => x.UserType).HasMaxLength(10);

        modelBuilder.Entity<User>().Property(x => x.Password).HasMaxLength(70);

        modelBuilder.Entity<User>().Property(x => x.Approved).HasColumnAnnotation("DefaultValue", Status.UNDEFINED);
        modelBuilder.Entity<User>().Property(x => x.Verified).HasColumnAnnotation("DefaultValue", Status.UNDEFINED);
    }
}

```

Slika 3.3.4. Konfiguraciju za tabelu *Users*

Klasa *UserDbContext* nasleđuje klasu *DbContext* preko koje kreira tabelu *Users* na osnovu klase *User*. Primarnu ključ tabele *Users* će biti jedinstven identifikator klase *User* a to je ceo broj koji se generiše pri dodavanja novog korisnika u bazu podataka. Polje email mora da jedinstveno i ima dodatno ograničenje da može da bude maksimalno 22 karaktera dugačko. Ostala polja isto imaju ograničenje za maksimalnu dužinu, dok podešena je i uobičajena vrednost za polja *Approved* i *Verified*. Na narednoj slici (Slika 3.3.5.) je prikazan kod koji predstavlja konfiguraciju za tabelu *Articles* u bazi sa nazivom *ArticleDB* koju koristi *ArticleService*.

```

public class ArticleDBContext : DbContext
{
    public ArticleDBContext() : base("name = ArticleCS")
    {
    }

    public DbSet<Article> Articles { get; set; }

    protected override void OnModelCreating(DbModelBuilder modelBuilder)
    {
        modelBuilder.Entity<Article>().HasKey(x => x.ArticleId);

        modelBuilder.Entity<Article>().Property(x => x.Name).HasMaxLength(30);

        modelBuilder.Entity<Article>().Property(x => x.Description).HasMaxLength(70);
    }
}

```

Slika 3.3.5. Konfiguracija za tabelu *Articles*

Ova konfiguracija se nalazi u *ArticleDBContext*-u koja nasleđuje klasu *DbContext*. Primarni ključ tabele će biti *ArticleId* koji jedinstveno identifikuje svaki artikal, to je ceo broj koji se generiše

pri dodavanju novog artikala u bazu. Ostala dva polja imaju ograničenje za dužinu. Na slici (Slika 3.3.6.) je prikazan kod koji predstavlja konfiguraciju za tabelu *Orders* koja se nalazi u bazi podataka sa nazivom *OrderDB*.

```
public class OrderDBContext : DbContext
{
    public OrderDBContext() : base("name = OrderCS")
    {
    }

    public DbSet<Order> Orders { get; set; }

    protected override void OnModelCreating(DbModelBuilder modelBuilder)
    {
        modelBuilder.Entity<Order>().HasKey(x => x.OrderId);

        modelBuilder.Entity<Order>().Property(x => x.Comment).HasMaxLength(70);

        modelBuilder.Entity<Order>().Property(x => x.Address).HasMaxLength(70);

        modelBuilder.Entity<Order>()
            .Ignore(u => u.ArticleIds);

        modelBuilder.Entity<Order>()
            .Ignore(u => u.AmountOfArticles);
    }
}
```

Slika 3.3.6. Konfiguracija za tabelu *Orders*

Ova konfiguracija se nalazi u *OrderDBContext* klasi koja nasleđuje klasu *DbContext*. Primarni ključ ove tabele je jedinstveni identifikator porudžbine, a to je polje *OrderId*. Ovo polje je ceo broj koji se generiše pri dodavanju porudžbine u bazu podataka. Dve liste koje se nalaze u klasi se ignoriše zato što će se čuvati kao liste u bazi podataka.

Komunikacija između veb API-ja i servisa je preko WCF portokola. Da bi se mogli razmenjivati podaci neophodno je da se napravne posebne klase koje su serijalizovane. Klasa *WcfUser* je primer klase *User* koja je serijalizovana da bi moglo da se prenosi kroz WCF. Na slici (Slika 3.3.7.) je prikazan kod koji predstavlja klasu *WcfUser*.

```
[DataContract]
public class WcfUser
{
    [DataMember]
    public int UserId { get; set; }
    [DataMember]
    public string Username { get; set; }
    [DataMember]
    public string Email { get; set; }
    [DataMember]
    public string FirstName { get; set; }
    [DataMember]
    public string LastName { get; set; }
    [DataMember]
    public string DateOfBirth { get; set; }
    [DataMember]
    public string Address { get; set; }
    [DataMember]
    public string UserType { get; set; }
    [DataMember]
    public byte[] ImageFile { get; set; }
    [DataMember]
    public string Password { get; set; }
    [DataMember]
    public Status Approved { get; set; }
    [DataMember]
    public Status Verified { get; set; }
}
```

Slika 3.3.7. Klasa *WcfUser*

Klasa *WcfUser* sadrži sva polja kao i klasa *User* ali se koristi atribut *DataMember* za polja i atribut *DataContract* za klasu kako bi ona mogla da se razmenjuje izmedju servisa i veb API-ja. Klase *Article* i *Order* su na isti način serijalizovane, a njih predstavljaju klase *WcfArticle* i *WcfOrder*. U *ClassLibrary* projektu se nalaze još i *Data Transfer Class*(DTO) klase koje sadrže samo neophodne podatke koju potrebne za kontroler, i te klase sadrže attribute za serijalizaciju. Da bi se sve ove razlilite klase mogle da se razmenjuju korisiti se i pomoćna klasa *MappingProfile* preko se mogu mapirati klase jednu na drugu. Na sledećoj slici (Slika 3.3.8.) je prikazan kod koji predstavlja klasu *MappingProfile*.

```
public class MappingProfile : Profile
{
    public MappingProfile()
    {
        CreateMap<WcfUser, UserDtoRegistration>();
        CreateMap<WcfUser, UserDtoLogin>();
        CreateMap<WcfUser, UserDtoApprovedVerified>();
        CreateMap<WcfUser, User>();

        CreateMap<UserDtoRegistration, WcfUser>();
        CreateMap<UserDtoLogin, WcfUser>();
        CreateMap<UserDtoApprovedVerified, WcfUser>();
        CreateMap<User, WcfUser>();
    }
}
```

Slika 3.3.8. Klasa *MappingProfile*

Klasa *MappingProfile* se koristi u *UserService*, i ona nasleđuje klasu *Profile*, i preko metode *CreateMap* mapira klase jedne na druge. Svaki servis ima interfejs za obradu podataka i rad sa bazom podataka, interfejs za komunikaciju izmedju samih servisa i interfejs koji opisuje *health check api* server koji ce se objasniti u daljem delu rada. Na slici (Slika 3.3.9.) je prikazan kod koji predstavlja interfejs *IWcfUserService*.

```
[ServiceContract]
public interface IWcfUserService
{
    [OperationContract]
    UserDtoRegistration AddUser(UserDtoRegistration newUser);
    [OperationContract]
    string LoginUser(UserDtoLogin userDto);
    [OperationContract]
    UserDtoRegistration GetUserById(int id);
    [OperationContract]
    UserDtoApprovedVerified GetUserByEmail(string email);
    [OperationContract]
    List<UserDtoApprovedVerified> GetUsers();
    [OperationContract]
    UserDtoRegistration UpdateUser(int id, UserDtoRegistration userDtoRegistration);
    [OperationContract]
    UserDtoStatus UpdateUserStatus(UserDtoStatus userDtoStatus);
    [OperationContract]
    bool UploadImageMS(MemoryStream ms, string email);
}
```

Slika 3.3.9. Interfejs *IWcfUserService*

Interfejs *IWcfUserService* izlaže metode sa radom sa korisnicima kao i sa komunikacijom sa bazom podataka. Sve ove metode se pozivaju u *UserController*-u. Na slici (Slika 3.3.10.) je prikazan kod koji predstavlja interfejs *IWcfOrderService*.

```
[ServiceContract]
public interface IWcfOrderService
{
    [OperationContract]
    OrderDto AddOrder(string email, OrderDto orderDto);
    [OperationContract]
    List<OrderDto> GetAllOrders();
    [OperationContract]
    List<OrderDto> GetAllCurrentOrdersForSeller(string email);
    [OperationContract]
    List<OrderDto> GetAllPastOrdersForSeller(string email);
    [OperationContract]
    Tuple<List<OrderDto>, List<OrderDto>> GetCurrentAndPastOrdersForBuyer(string email);
    [OperationContract]
    void CanceledOrder(int id);
    [OperationContract]
    OrderDetailsDto GetOrderDetails(int id, string email);
}
```

Slika 3.3.10. Interfejs *IWcfOrderService*

Interfejs *IWcfOrderService* izlaže metode sa radom sa porudžbinama i sa komunikacijom sa bazom podataka. Sve ove metode se pozivaju u *OrderController*-u. Na slici (Slika 3.3.11.) je prikazan kod koji predstavlja interfejs *IWcfArticleService*.

```
[ServiceContract]
public interface IWcfArticleService
{
    [OperationContract]
    ArticleDto AddArticle(string email, ArticleDto articleDto);
    [OperationContract]
    List<ArticleDto> GetArticles();
    [OperationContract]
    List<ArticleDto> GetArticlesByEmailForSeller(string email);
    [OperationContract]
    void DeleteArticleById(int id);
    [OperationContract]
    ArticleDto UpdateArticle(ArticleDto articleDto);
    [OperationContract]
    List<ArticleDto> GetArticlesForCart(List<int> articleIds);
    [OperationContract]
    bool UploadImage(MemoryStream ms, int id);
}
```

Slika 3.3.11. Interfejs *IWcfArticleService*

Interfes *IWcfArticleService* izlaže metode sa radom sa porudžbinama i sa komunikacijom sa bazom podataka. Sve ove metode se pozivaju u *ArticleController*-u. U *Providers* folderu *UserService*-a se nalaze klase koje implementiraju interface-ove. Klasa *UserServicProvider* implementira interface *IWcfUserService*. Na narednoj slici (Slika 3.3.12.) je prikazan kod koji prestavlja implementaciju metode *AddUser*.

```

public UserDtoRegistration AddUser(UserDtoRegistration newUser)
{
    var eventLogger = new LoggerConfiguration()
        .WriteTo.File(eventLoggerPath, outputTemplate:
            "{Timestamp:yyyy-MM-dd HH:mm:ss.fff zzz} [{Level:u3}] {Message:lj}{NewLine}")
        .CreateLogger();
    var exceptionLogger = new LoggerConfiguration()
        .WriteTo.File(exceptionLoggerPath, outputTemplate:
            "{Timestamp:yyyy-MM-dd HH:mm:ss.fff zzz} [{Level:u3}] {Message:lj}{NewLine}{Exception}")
        .CreateLogger();

    if (_dbContext.Users.Any(x => x.Email == newUser.Email))
    {
        exceptionLogger.Error("Email already in use!");
        Log.CloseAndFlush();
        throw new FaultException("Email already in use!");
    }
    if (_dbContext.Users.Any(x => x.Username == newUser.Username))
    {
        exceptionLogger.Error("Username already in use!");
        Log.CloseAndFlush();
        throw new FaultException("Username already in use!");
    }

    WcfUser user = _mapper.Map<WcfUser>(newUser);
    user.Password = GetHashValueInString(newUser.Password);
    User u = _mapper.Map<User>(user);

    _dbContext.Users.Add(u);
    _dbContext.SaveChanges();

    eventLogger.Information("Successfully added user to the database!", DateTime.Now);
    Log.CloseAndFlush();

    return _mapper.Map<UserDtoRegistration>(user);
}

```

Slika 3.3.12. Metoda *AddUser*

Metoda *AddUser* se nalazi u *UserServiceProvider* klasi. Ovu metodu zove veb API. Metoda vraća korisnika na osnovu *mail*-a koji je prosleđen. Na slici (Slika 3.3.13.) je prikazan kod koji predstavlja interfejs *IWcfInternalUserService*.

```

[ServiceContract]
public interface IWcfInternalUserService
{
    [OperationContract]
    WcfUser GetUserByEmail(string email);
}

```

Slika 3.3.13. Interfejs *IWcfInternalUserService*

Interfejs *IWcfInternalUserService* izlaže metodu koja se koristi kod komunikacije između servisa. Na slici (Slika 3.3.14.) je prikazan kod koji predstavlja implementiranu metodu *GetUserByEmail* koja se nalazi u klasi *InternalUserServiceProvider*.


```

public class InternalUserServiceProvider : IWcfInternalUserService
{
    private readonly UserDBContext _dbContext;
    private readonly IMapper _mapper;
    private readonly string binFolderPath;
    private readonly string exceptionLoggerPath;
    private readonly string eventLoggerPath;

    public InternalUserServiceProvider()
    {
        MapperConfiguration config = new MapperConfiguration(cfg => {
            cfg.AddProfile<MappingProfile>();
        });
        _mapper = _mapper = config.CreateMapper();
        _dbContext = new UserDBContext();
        binFolderPath = Path.Combine(AppDomain.CurrentDomain.BaseDirectory, "bin");
        exceptionLoggerPath = Path.Combine(binFolderPath, "UserServiceExceptionLog.txt");
        eventLoggerPath = Path.Combine(binFolderPath, "UserServiceEventLog.txt");
    }

    public WcfUser GetUserByEmail(string email)
    {
        var eventLogger = new LoggerConfiguration()
            .WriteTo.File(eventLoggerPath, outputTemplate:
                "{Timestamp:yyyy-MM-dd HH:mm:ss.fff zzz} [{Level:u3}] {Message:l1}{NewLine}")
            .CreateLogger();
        var exceptionLogger = new LoggerConfiguration()
            .WriteTo.File(exceptionLoggerPath, outputTemplate:
                "{Timestamp:yyyy-MM-dd HH:mm:ss.fff zzz} [{Level:u3}] {Message:l1}{NewLine}{Exception}")
            .CreateLogger();

        try
        {
            User user = _dbContext.Users.FirstOrDefault(u => u.Email == email);
            eventLogger.Information("Successfully retrieved user by email!", DateTime.Now);
            Log.CloseAndFlush();
            return _mapper.Map<WcfUser>(user);
        }
        catch (Exception e)
        {
            exceptionLogger.Error(e.Message);
            Log.CloseAndFlush();
            throw e;
        }
    }
}

```

Slika 3.3.14. Implementacija interfejsa *IWcfInternalUserService*

Metoda *GetUserByEmail* se koristi za komunikaciju izmedju servisa. Kod sva tri servisa u njihovim *Program* klasama su napravljeni WCF servisi i izložili krajnje tačke za povezivanje. Na slici (Slika 3.3.15.) je prikazan kod koji predstavlja deo koda koji služi za pokretanje WCF servera u *UserService-u*.

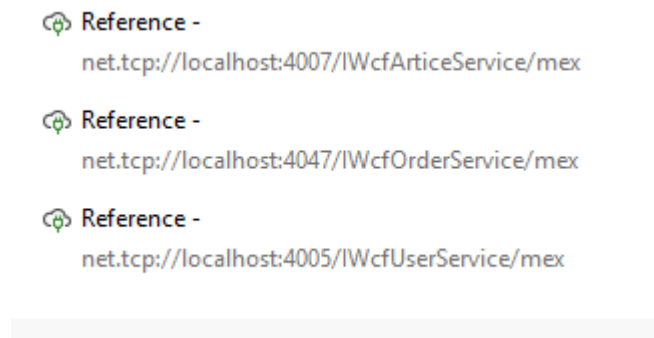
```

ServiceHost wcfUserService = new ServiceHost(typeof(UserServiceProvider));
wcfUserService.AddServiceEndpoint(typeof(IWcfUserService), new NetTcpBinding(),
    new Uri("net.tcp://localhost:4005/IWcfUserService"));
ServiceMetadataBehavior smb = new ServiceMetadataBehavior();
wcfUserService.Description.Behaviors.Add(smb);
Binding mexbinding = MetadataExchangeBindings.CreateMexTcpBinding();
wcfUserService.AddServiceEndpoint(typeof(IMetadataExchange), mexbinding,
    "net.tcp://localhost:4005/IWcfUserService/mex");
wcfUserService.Open();

```

Slika 3.3.15. Kod *UserService-a* za WCF server ka veb API-ju

Tu se izlaže krajnja tačka ka kojoj će *UserController* imati konekciju i koja će pozivati metode. Da bi se mogli razmenjivati podaci između servisa i veb API-ja neophodno je da se krajnja tačka nadogradi mex produžetkom. Mexbinding omogućuje *metadata* exchange. Na slici (Slika 3.3.16.) su prikazane reference ka servisima.



Slika 3.3.16. Reference ka servisima

Preko ovih referenci će se omogućiti da se povežu na WCF server i da razmenjuju podatke. Na slici (Slika 3.3.17.) je prikazan kod koji otvara WCF konekciju i izlaže krajnju tačku preko koje će ostali servisi moći da komuniciraju.

```
ServiceHost wcfInternalUserService = new ServiceHost(typeof(InternalUserServiceProvider));  
wcfInternalUserService.AddServiceEndpoint(typeof(IWcfInternalUserService),  
    new NetTcpBinding(), new Uri("net.tcp://localhost:4000/IWcfInternalUserService"));  
wcfInternalUserService.Open();
```

Slika 3.3.17. Kod *UserService*-a za WCF server ka ostalim servisima

4. ŠABLONI OBSERVABILNOSTI

Mikroservisi imaju ubedljive prednosti u svetu softverkog inženjerstva. Međutim sa zasebnim servisima raspoređenim na različitim *host*-ovima, praćenje desetina ili čak stotina mikroservisa može biti izazovno. Sa većim obimom i složenošću dolazi i veća potreba za uočljivošću. Postoji mnogo potencijalnih tačaka neuspeha i stalnih ažuriranja u arhitekturi mikroservisa koje se ne mogu rešiti tradicionalnim rešenjima za praćenje. Mnogi nepoznati dinamički faktori u distribuiranom okruženju čine neophodnim da se dizajnom ugradi vidljivost u sistem. Mehanizmi za posmatranje obezbeđuju vidljivost distribuiranog sistema kako bi pomogli programerima da razumeu performanse svoje aplikacije. Observabilnost nudi neophodnu kontrolu za brzo identifikovanje i rešavanje problema [2]. Šabloni obesrvabilnosti koju su implementirane u rešenju mikroservisa su *exception tracing*, *health check API* i *log aggregation*. Ove metode koje su implementirane će se u daljem tekstu objasniti.

Može doći do grešaka dok mikroservis obrađuje zahtev. Ako je servis dobro programirana instanca servisa će izbaciti izuzetak. Ovi izuzeci mogu uključivati greške i poruke koda. Izuzeci su kritični za uočljivost [3]. Kao opšte pravilo, svi izuzeci treba da se evidentiraju, a programeri treba da budu obavешteni kada se izuzeci pojave u njihovom kodu, tako da mogu da istraže i otkriju osnovni uzrok [5]. U sva tri servisa se loguju izuzeci i svaki servis ima svoj log fajl. Ti log fajlovi se nalaze u bin folderu servisa, a nazivi tih fajlova su *UserServiceExceptionLog.txt*, *ArticleServiceExceptionLog.txt* i *OrderServiceExceptionLog.txt*. Na slici (Slika 4.1.) je prikazan kod koji predstavlja način na koji je implementirano bacanje izuzetaka i logovanje tih izuzetaka u metodi *UserService*-a.

```
if (_dbContext.Users.Any(x => x.Email == newUser.Email))
{
    exceptionLogger.Error("Email already in use!");
    Log.CloseAndFlush();
    throw new FaultException("Email already in use!");
}
if (_dbContext.Users.Any(x => x.Username == newUser.Username))
{
    exceptionLogger.Error("Username already in use!");
    Log.CloseAndFlush();
    throw new FaultException("Username already in use!");
}
```

Slika 4.1. Bacanje izuzetka i logovanje

Baca se izuzetak i taj isti se loguje. Dodatno ne baca se običan izuzetak nego se baca *FaultException* koji specifično koristan kod WCF servisa jer se ovaj izuzetak može propagirati do veb API-ja gde se onda može klijentska strana na adekvatan način obavestiti. Na slici (Slika 4.4.) je prikazan kod koji predstavlja hvatanje izuzetaka u *UserController*-u.

```

[HttpPost("registration")]
public ActionResult CreateUser([FromBody] UserDtoRegistration userDto)
{
    try
    {
        return Ok(client.AddUserAsync(userDto));
    }
    catch (FaultException e)
    {
        if (e.Message == "Email already in use!")
            return StatusCode(StatusCodes.Status409Conflict, e.Message);
        else if (e.Message == "Username already in use!")
            return StatusCode(StatusCodes.Status409Conflict, e.Message);
        else
            return StatusCode(StatusCodes.Status500InternalServerError, e.Message);
    }
}
}

```

Slika 4.2. Hvatanje izuzetaka kod veb API-ja

Na osnovu izuzetka koji bačen na serverskoj strani, taj izuzetak se propagira do veb API-ja koji onda na osnovu poruke samog izuzetka na adekvatan način obaveštava klijentsku stranu.

Veoma važna metoda obesrvabilnosti je *health check API*. *Health check API* omogućava praćenje zdravlja, drugim rečima stanje samih servisa. Ako iz nekog razloga servis ne može da izvršava zadatke onda to treba odmah da se sinalizira kroz *health check API* [4]. Svaki servis ima interface kroz koji signalizira u kom je stanju. Na slici (Slika 4.3.) je prikazan kod koji predstavlja interfejs *IUserServiceHealthCheck*.

```

[ServiceContract]
public interface IUserServiceHealthCheck
{
    [OperationContract]
    string HealthCheck();
}

```

Slika 4.3. Interfejs *IUserServiceHealthCheck*

Interfejs *IUserServiceHealthCheck* servisa *UserService* ima metodu *HealthCheck*, koja služi za proveru stanja servisa. Na slici (Slika 4.4.) je prikazan kod koji predstavlja interfejs *IOrderServiceHealthCheck*.

```

[ServiceContract]
public interface IOrderServiceHealthCheck
{
    [OperationContract]
    string HealthCheck();
}

```

Slika 4.4. Interfejs *IOrderServiceHealthCheck*

Interfejs *IOrderServiceHealthCheck* servisa *OrderService* ima metodu *HealthCheck*, koja služi za proveru stanja servisa. Na slici (Slika 4.5.) je prikazan kod koji predstavlja interfejs *IArticleServiceHealthCheck*.

```
[ServiceContract]
public interface IArticleServiceHealthCheck
{
    [OperationContract]
    string HealthCheck();
}
```

Slika 4.5. Interfejs *IArticleServiceHealthCheck*

Interfejs *IArticleServiceHealthCheck* servisa *ArticleService* ima metodu *HealthCheck*, koja služi za proveru stanja servisa. Na sledećoj slici (Slika 4.6.) je prikazan kod koji predstavlja implementiranu metodu *HealthCheck* u *UserService*-u.

```
public class UserServiceHealthCheckProvider : IUserServiceHealthCheck
{
    private readonly string binFolderPath;
    private readonly string exceptionLoggerPath;

    public UserServiceHealthCheckProvider()
    {
        binFolderPath = Path.Combine(AppDomain.CurrentDomain.BaseDirectory, "bin");
        exceptionLoggerPath = Path.Combine(binFolderPath, "UserServiceExceptionLog.txt");
    }

    public string HealthCheck()
    {
        var exceptionLogger = new LoggerConfiguration()
            .WriteTo.File(exceptionLoggerPath, outputTemplate:
                "{Timestamp:yyyy-MM-dd HH:mm:ss.fff zzz} [{Level:u3}] {Message:l1}{NewLine}{Exception}")
            .CreateLogger();

        try
        {
            using (var dbContext = new UserDBContext())
            {
                var dbConnection = dbContext.Database.Connection;
                dbConnection.Open();
                dbConnection.Close();
            }
        }
        catch (Exception e)
        {
            exceptionLogger.Error(e.Message);
            Log.CloseAndFlush();
            return "Status : UNHEALTHY, UserService cant connect to the database!";
        }

        return "Status : HEALTHY, UserService is running and healthy!";
    }
}
```

Slika 4.6. Implementacija metode *HealthCheck*

Ova metoda proverava da li može da se konektuje ka bazi podataka. Ako je došlo do neke greške u bazi podataka ili baza podataka ne reaguje na pozive iz nekih drugih razloga, onda se u tom slučaju šalje poruka da je servis u nezdravom stanju. Implementacije ove metode je u svakom

servisu ista. Svaki servis kroz WCF protokol sa TCP konekcijom izlaže krajnju tačku na kojoj se nalazi *HealthCheck* metoda. Na slici (Slika 4.7.) je prikazan kod koji predstavlja WCF server sa TCP komunikacijom koja izlaže krajnju tačku za *health check API*.

```
ServiceHost wcfUserServiceHealthCheck = new ServiceHost(typeof(UserServiceHealthCheckProvider));
wcfUserServiceHealthCheck.AddServiceEndpoint(typeof(IUserServiceHealthCheck),
    new NetTcpBinding(), new Uri("net.tcp://localhost:5003/IUserServiceHealthCheck"));
wcfUserServiceHealthCheck.Open();
```

Slika 4.7. Krajnja tačka za *health check API*

Zasebna komponenta adapter ima dve metode jedna uspostavlja komunikaciju sa kranjim tačkama sva tri servisa preko *health check API* i ispisuje zdravlje servisa a druga metoda prikazuje sve logove iz sva tri log fajla koja se nalaze u servisima. Na slici (Slika 4.8.) je prikazan kod koji predstavlja metodu *GetHealthsOfServices*.

```
private static void GetHealthsOfServices()
{
    ChannelFactory<IUserServiceHealthCheck> userServiceChannelFactory =
        new ChannelFactory<IUserServiceHealthCheck>(new NetTcpBinding(),
            new EndpointAddress("net.tcp://localhost:5003/IUserServiceHealthCheck"));
    ChannelFactory<IArticleServiceHealthCheck> articleServiceChannelFactory =
        new ChannelFactory<IArticleServiceHealthCheck>(new NetTcpBinding(),
            new EndpointAddress("net.tcp://localhost:5001/IArticleServiceHealthCheck"));
    ChannelFactory<IOrderServiceHealthCheck> orderServiceChannelFactory =
        new ChannelFactory<IOrderServiceHealthCheck>(new NetTcpBinding(),
            new EndpointAddress("net.tcp://localhost:5002/IOrderServiceHealthCheck"));
    while (true)
    {
        Console.WriteLine("Health check for the services: ");
        try
        {
            IUserServiceHealthCheck userServiceHealthCheckChannel = userServiceChannelFactory.CreateChannel();
            Console.WriteLine(userServiceHealthCheckChannel.HealthCheck());
        }
        catch (Exception e)
        {
            Console.WriteLine("Status : ERROR, UserService is unreachable!");
        }
        try
        {
            IArticleServiceHealthCheck articleServiceHealthCheckChannel = articleServiceChannelFactory.CreateChannel();
            Console.WriteLine(articleServiceHealthCheckChannel.HealthCheck());
        }
        catch (Exception e)
        {
            Console.WriteLine("Status : ERROR, ArticleService is unreachable!");
        }
        try
        {
            IOrderServiceHealthCheck orderServiceHealthCheckChannel = orderServiceChannelFactory.CreateChannel();
            Console.WriteLine(orderServiceHealthCheckChannel.HealthCheck());
        }
        catch (Exception e)
        {
            Console.WriteLine("Status : ERROR, OrderService is unreachable!");
        }
        Console.WriteLine("-----");
    }
}
```

Slika 4.8. Metoda za *health check*

Metoda *GetHealthsOfServices* se nalazi u adapteru koja na zahtev uspostavlja konekciju prema *health check API* krajnjim tačkama koje izlažu servisi. Odgovor koji dobija od servisa

ispisuje na konzolu, ako iz nekog razlog ne može da pristupi nekoj krajnjoj tački to znači da je servis pao i u tom slučaju ispisuje adekvatnu poruku. Na slici (Slika 4.9.) je prikazan terminal na kom adapter prikazuje informacije koje dobija kroz *health check API* koje opisuju stanje servisa.

```
Status : HEALTHY, UserService is running and healthy!  
Status : HEALTHY, ArticleService is running and healthy!  
Status : HEALTHY, OrderService is running and healthy!
```

Slika 4.9. Informacije kroz *health check API*

Šablon *log aggregation* je proces prikupljanja, standardizacije i konsolidacije podataka dnevnika iz čitave aplikacije kako bi se olakšala i pojednostavila analiza dnevnika. Bez agregacije dnevnika programeri bi morali ručno da organizuju, pripremaju i pretražuju podatke dnevnika iz brojnih izvora kako bi iz njih izvukli korisne informacije [9]. Na slici (Slika 4.10.) je prikazan kod koji predstavlja metodu *LogAggregation* koja implementira agregaciju dnevnika.

```
private static void LogAggregation() {  
    string fullPath = AppDomain.CurrentDomain.BaseDirectory;  
    string midlePath = fullPath.Substring(0, fullPath.LastIndexOf("\\", fullPath.LastIndexOf("\\", fullPath.LastIndexOf("\\") - 1) - 1));  
    string basePath = Path.GetDirectoryName(midlePath);  
    string userLogPath = Path.Combine(basePath + @"\UserService\bin\Debug\bin", "UserServiceExceptionLog.txt");  
    string articleLogPath = Path.Combine(basePath + @"\ArticleService\bin\Debug\bin", "ArticleServiceExceptionLog.txt");  
    string orderLogPath = Path.Combine(basePath + @"\OrderService\bin\Debug\bin", "OrderServiceExceptionLog.txt");  
  
    if (File.Exists(userLogPath)) {  
        Console.WriteLine("UserService exceptions: ");  
        string fileContents = File.ReadAllText(userLogPath);  
        Console.WriteLine(fileContents);  
    }  
    if (File.Exists(orderLogPath)) {  
        Console.WriteLine("OrderService exceptions: ");  
        string fileContents = File.ReadAllText(orderLogPath);  
        Console.WriteLine(fileContents);  
    }  
    if (File.Exists(articleLogPath)) {  
        Console.WriteLine("ArticleService exceptions: ");  
        string fileContents = File.ReadAllText(articleLogPath);  
        Console.WriteLine(fileContents);  
    }  
}
```

Slika 4.10. Metoda *LogAggregation*

Na početku metode prikupse se putanja za sva tri servisa. Svaki servis ima bin folder gde se nalazi log fajl. Pristupi se tim log fajlovima i isčitaju se svi podaci. Na slici (Slika 4.11.) je prikazan terminal adaptera, gde adapter ispisuje sve izuzetke koji se nalaze u svim servisima, nakon pozivanja metode *LogAggregation*.

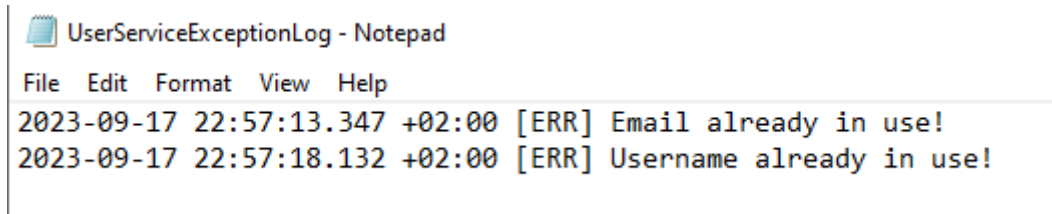
```
UserService exceptions:  
2023-09-17 22:57:13.347 +02:00 [ERR] Email already in use!  
2023-09-17 22:57:18.132 +02:00 [ERR] Username already in use!  
  
ArticleService exceptions:  
2023-09-22 00:59:59.830 +02:00 [ERR] Invalid email, no user with that email!
```

Slika 4.11. Ispis svih izuzetaka

5. TESTIRANJE

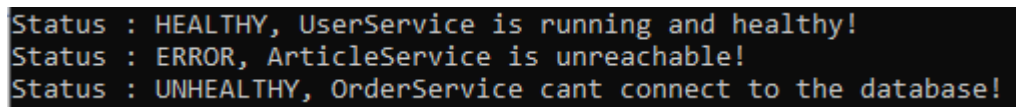
U ovom poglavlju će biti prikazani test slučajevi gde se mogu videti implementirani šabloni observabilnosti.

Ako korisnik želi da se registruje a pritom registrovanja koristio je već postojeći mejl ili korisničko ime onda će se u *UserService*-u logovati izuzetak. Taj exception se nalazi u *UserServiceExceptionLog.txt*. Na slici (Slika 5.1.) je prikazan *UserServiceExceptionLog* fajl u kom se nalaze svi evidentirani izuzeci.



Slika 5.1. Prikaz logovanih izuzetka

U sledećem primeru će biti prikazan način na koji adapter prikazuje različita zdravlja u kojem servisi mogu biti preko *health check API*-ja. *OrderService*-u će se obrisati baza podataka, *ArticleService* će pasti usled greške a *UserService* će biti potpuno zdrav bez ikakvih grešaka, u ovom slučaju adapter će ove poruke prikazati. Na slici (Slika 5.2.) je prikazan terminal adaptera gde se prikazu stanja svih servisa.



Slika 5.2. Prikaz stanja servisa

ArticleService je pao usled neke greške pa se za nju signalizira da je pao. *OrderService* ne može da pristupi bazi podataka, ali on nije prestao sa radom nego nastavlja da radi, zbog ovog razloga je u nezdravom stanju.

6. ZAKLJUČAK

Transformacija monolitnih aplikacija na mikroservisnu arhitekturu u mnogo slučajeva je dobar pristup, i sa tom tranzicijom dobijaju se svi benefiti koje pruža mikroservisna arhitektura., a to su: skalabilnost, održivost, lakšu izmenu koda, bržu reakciju na greške, lakše otklanjanje grešaka i lakšu održivost same aplikacije. Postoje slučajevi kada ta tranzicija i nije najbolje rešenje a to je kad su male aplikacije koje nemaju jako puno komponenti, i koje nemaju jako puno klijentskih zahteva za obradu, a sama aplikacija nije teška za održavanje onda u tom slučaju ne treba forsirati izmenu arhitekture. Aplikacije koje su velike, koje imaju velik broj komponenti, velik broj obrada podataka i velik broj zahteva te aplikacije definitivno treba prebaciti na mikroservisnu arhitekturu. Veoma pažljivo treba odabrati šablon koji će se koristiti i na koji način će se formirati servisi [1]. Mikoservisna arhitektura je arhitektonski stil koja je još veoma neistrsžena i nema direktne obrazce na koji način treba da se vrši tranzicija, ali sve više kompanija koristi ovu arhitekturu i u budućnosti će se još više koristiti ova arhitektura.

Monolitna veb aplikacija onlajn prodavnice koja je prikazana, objašnjena na početku je bila čvrsto povezana, teška za održavanje i ako bi došlo do greške ili izuzetka cela aplikacija bi prestala sa radom. Tranzicijom na mikroservisnu arhitekturu svi ti nedostaci su otklonjeni. Primenila se podela na nezavisne servise koji kumuniciraju sa veb API-jem. Svaki servis ima svoju bazu podataka sa kojom komunicira nezavisno od ostalih servisa. Dodatno implementirani su šabloni observabilnosti *health check api*, *exception tracing* i na ovaj način omogućena je observabilnost cele aplikaciju u realnom vremenu. Kroz ove implementirane šablone mogu se pratiti performanse servisa, *health* servisa, pojava grešaka i izuzetaka pa i reakcije na te događaje [2].

U mikroservisnoj arhitekturi koja prikazana u rešenju koristila se WCF komunikacija. WCF je veoma pogodna i efikasna komunikacija, ali je zastarela i više se ne razvija niti se koristi u modernim aplikacijama. Da bi se omogućila modernija, brža i efikasnija komunikacija zato bi se koristio *gRPC*. Okvir *gRPC* je moderan *open source* okvir visokih performansi koji može da radi u bilo kojem okruženju. Može efikasno da poveže servise između centara podataka sa podrškom koja se može priljučiti za balansiranje opterećenja, praćenje, proveru ispravnosti i autentifikaciju. Dodatno u rešenju se koristio *Entity Framework*. *Entity Framework* je veoma pogodan alat koji omogućuje brz rad sa bazom, kao i komunikacija sa bazom preko *Language Integrated Query* (LINQ) sintakse, ali on ima jednu manu a to je performansa. *Entity Framework*-of bogat skup funkcija i dodatni sloj aptrakcije mogu dovesti do sporijih performansi. Alternativno rešenje, koje pruža brže performasne je da se koriste SQL upiti. *Entity Framework* mora da prevodi funkcije u SQL upite, pa brže je da se pozivaju odmah SQL upiti bez prevođenja.

7. LITERATURA

- [1] THÖNES, Johannes. Microservices. *IEEE software*, 2015, 32.1: 116-116.
- [2] INDRASIRI, Kasun, et al. Observability. *Microservices for the Enterprise: Designing, Developing, and Deploying*, 2018, 373-408.
- [3] KENT, Karen Ann; SOUPPAYA, Murugiah. Guide to Computer Security Log Management:. 2006.
- [4] MINKKILÄ, Juuso. Health Check API and Service Discovery in Microservice Environment. 2019.
- [5] GOODENOUGH, John B. Exception handling: Issues and a proposed notation. *Communications of the ACM*, 1975, 18.12: 683-696.
- [6] MCMURTRY, Craig, et al. Windows Communication Foundation Unleashed (Wcf)(Unleashed). 2007
- [7] SINGH, Rahul Rajat, et al. *Mastering Entity Framework*. Packt Publishing, 2015.
- [8] WILDE, Erik; PAUTASSO, Cesare (ed.). *REST: from research to practice*. Springer Science & Business Media, 2011.
- [9] Stanojevic, M., Capko, D., Lendak, I., Stoja, S., & Jelacic, B. (2023). Fighting Insider Threats, with Zero-Trust in Microservice-based, Smart Grid OT Systems. *Acta Polytechnica Hungarica*, 20(6).

8. BIOGRAFIJA

Deneš Sentmartoni je rođen 28.09.2000. godine u Novom Sadu, Srbija. Osnovnu školu “Sonja Marinković” završio je 2015. godine. Srednju elektrotehničku školu “Mihajlo Pupin” završio je 2019. godine. Smer Primenjeno softversko inženjerstvo na Fakultetu tehničkih nauka upisao je 2019. godine. Ispunio je sve obaveze i položio sve ispite predviđene studijskim programom. Živi u Novom Sadu. Od stranih jezika priča engleski jezik.