# 5. Opis resenja problema

Kroz ovo poglavlje ce biti objasnjeni razlozi za primeni mikroservisne arhitekture i saga obrasca za distribuiranu transakciju izmedju ucesnika, tj. mikroservisa.

## 5.1 Osnove mikroservisne arhitekture

Sta je zapravo mikroservisna arhitektura? Po samom nazivu mozda ne mozemo odmah svatiti jer prenaglasava velicinu, postoje brojne definicije ove arhitekture. Neki uzimaju ime previse bukvalno i tvrde da bi servis trebao biti jako mali, drugi tvrde da bi razvoj servisa trebao trajati do dve nedelje. Adrian Kokroft (eng. Adrian Cockcroft) definise mikroservisnu arhitekturu kao servisno orijentisanu arhitekturu labavo povezanih elemenata koji imaju ograniceni konteks.

Mikroservis je jedna zasebna aplikacija, sa jedinstvenom ulogom, koja je deo sireg sistema.

Glavna karakteristika mikroservisne arhitekture je labava sprega (eng. Loose coupling) i komunikacija preko APIja. Jedan od nacina da se postigne labava sprega je primenom principa *baza podataka po servisu*, gde svaki miktroservis ima sopstveno skladiste podataka.

### 5.1.1 Kocka skaliranja

Kocka skalitanja (eng. The scale cude) definise tri odvojena nacina za skaliranje aplikacije, skaliranje po X osi, po Y osi i po Z osi, slika 5.1

Shape, rectangle

Description automatically generated  
Slika 5.1 Kocka skalitanja i ose po kojima se nezavisno moze skalirati

Skaliranje po X osi predstavlja dodavanje vise instanci, odnosno kreiranje veceg broja virtualnih cvorova, kao sto je prikazano na slici 5.2.

Diagram

Description automatically generated  
Slika 5.2 Skaliranje po X osi

Sklairanje po Z osi predstavalja prelazak sa jedne particije na vise, slika 5.3.

Diagram

Description automatically generated  
Slika 5.3 Skaliranje po Z osi

Skaliranje po Y osi je funcionalna dekompozicija koja omogucava prelazak sa monolitne na mikroservisnu arhitekture, slika 5.4.

Diagram

Description automatically generated  
Slika 5.4 Skaliranje po Y osi

### 5.1.2 Prednosti

Prilikom izgradnje velike aplikacije, prednosti primene mikroservisne arhitekture su sledece:

* *Visoke performance i brzina operacija* – svaki servis ima svoju ulogu i svoje operacije i zbog toga je izvrsavanje operacija mnogo brza,
* *Visoka skalabilnost* – svaki servis se moze skalirati nezavisno,
* *Mogucnost rada na razlicitim mrezama* – vise timova mogu raditi u isto vreme na razlicitim delovima mreze,
* *Lokalno upravljenje svake aktivnosti* – svaki tim upravlja svojim servisom,
* *Visoka fleksibilnost* – brza implementacija i brza izmena u buducnosti
* *Slobodan izbor tehnologije* – svaki tim moze izabrati tehnologiju po zelji za izradu mikroservisa,
* *Sigurnije greske* – greska u jednom servisu ne prouzrokuje pad celokupnog sistema.

Nabrojane su samo neke od mnogobrojnih prednosti implementacije mikroservisne arhitekture.

### 5.1.3 Mane

Prethodno su nabrojane prednosti ali to ne znaci da ne postoje mane, neke od mana primene mikroservisne arhitekture su sledece:

* *Arhitektura je mnogo kompleksnija,*
* *Sigurnost* – primena sigurnosti je mnogo kompleksnija jer se podaci salju i razmenjuju preko mreze usmesto lokalne memorije,
* *Testiranje* – kompleksnije je testiranje od monolita jer se mora pokrenuti servis kao i svaki drugi servis koji se oslanja na taj.

Prilikom razmatranja prednosti i mana mikroservisne arhitekture uvidja se princip Pitera Parkera ‘’*With great power comes great responsibility*’’.

### 5.1.4 Obrasci mikroservisnog arhitekturalnog jezika

Na slici 5.5 je prikazana kolekcija obrazaca mikroservisnog arhitekturalnog jezika za resavanje odredjeni problema u mikroservisnoj arhitekturi. Jezik prvo pomaze u odluci izmedju monolitnog i mikroservisnog pristupa dizajnijranja arhitekture, sa njihovim prednostima i manama. Zatim, ako se izabere mikroservisna arhitektura, jezik vam pomaze u daljem efikasnom razvoju sistema.

Jezik obrazaca se sastoji od nekoliko grupa obrazaca. S leve strane slike 5.5 je *aplikativna arhitekturalna* grupa obrazaca, monolitna i mikroservisna arhitektura. Ostatak obrazaca, desna strana slike 5.5, su resenja problema koja dolaze uz odabir mikroservisne arhitekture.

Obrasci mikroservisne arhitekture su podeljeni u tri sloja:

* *Obrasci infrastrukture* – resavaju probleme koji se odnose na infrasturkturne probleme van razvojnog okruzenja,
* *Obrasci aplikativne infrastructure* – resavaju probleme koje se odnose na infrastrukturne probleme u razvojnom okruzenju,
* *Aplikativni obrasci* – resavaju probleme suocene od stane programera.

Graphical user interface, table

Description automatically generated  
Slika 5.5 Obrasci mikroservisnog arhitekturalnog jezika

## 5.2 Upravljanje transakcijama u mikroservisima

Transakciju su bitan deo sve aplikacije, bez njih ne bi bilo moguce odrzati konsiztentnost podataka.

ACID transakcije znatno pojednostavljuju posao programera pruzajuci iluziju da svaka transakcija ima ekskluzivni pristup podacima. U mikroservisnoj arhitekturi, transakcije unutra jednog servisa i dalje mogu biti ACID, ali dolazi do problema prilikom transakcija izmedju razlicitih servisa. Resenje za takav problem je primena *sage,* redosled lokalnih transakcija vodjenim porukama, za odrzavanje konzistentnosti podataka. Izazov sa sagom je taj da su one ACD (*Atomicity, Consistency, Durability*), nedostatak se nalazi u isolaciji, stoga kao resenje aplikacija mora da ima implementirane *provitmere*, koje govore sta se desava nakon odredjenog naslog prolema, odnosno kako se resava problem.

## 5.3 Saga obrazac

Kao sto je prethodno pomenuto saga je mehanizam za odrzavanje konzistentnosti podataka u mikroservisnoj arhitekturi bez potrebre za upotrebom distribuiranih transakcija. Saga je niz lokalnih transakcija, gde se lokalno upravlja podacima upotrebom ACID transakcijama. Sistemska operacija pokrece prvi korak sage, zavrsetkom lokalne obradom tih podataka pokrece se sledece lokalna obrada, itd.

Pretpostavimo da postoje *(n + 1)th*transakcija i toliko mogucih ispada sistema, prethodnih *n* transakcija se moraju opozvati. Svaki od koraka Ti ima svoju kompanzacionu transakciju Ci koja ponistava akciju od Ti. Da bi se ponistilo svih *n* koraka saga mora da izvrsi svaki Ci korak u obrnutom redosledu, kao sto je prikazno na slici 5.6.

A picture containing graphical user interface

Description automatically generated  
Slika 5.6 Niz koraka saga usled greske

Postoje dva nacina za koordinaciju sagom: *koreografija (eng. Choreography)*, gde ucesnici razmenjuju dogadjaje bez centralizove tacke kontrole i *orkestracija (eng. Orchestration)*, gde centralizovani kontroler govori ucesnicima sage obrasca koju operaciju da izvrse.

### 5.3.1 Koreograf

Jedan od nacina za implementacju sage je pomocu koreografa. Prilikom primene koreografa, ne postoji centralni coordinator koji govori ucesnicima sage sta da rade, usmesto toga saga ucesnici se prijavljuju na dogadjeje jedni kod drugih.

Prednosti saga koreografa su:

* *Jednostavnost* – servisi objavljuju dogadjaje kad kreiraju, izmene ili obrisu objekte,
* *Slaba sprega* – ucesnici se prijavljuju na dogadjaje i nemaju direktno saznanje o drugim ucesnicima.

Mane saga koreografa su:

* *Mnogo teze za razumevanje* – za razliku od orkestratora, ne postoji mesto gde je sve definisano,
* *Kruzna zavisnost izmedju servisa* – ucesnici sage se prijavljuju na dogadjaje sto moze da stvori kruznu zavisnost,
* *Rizik cvrte sprege* – svaki saga ucesnik treba da se prijavi na sve dogadjeje koje uticu na njih.

Koreografija je dobar pristup za jednostavnije sage, ali sa mogucim manama uvek je sigurnija opcija orkestrator, koja ce biti detaljnije opisana u nastavku.

### 5.3.2 Orkestrator

Kao sto je prethodno spomenuto, orkerstrator je mnogo bolje resenje od koreografa. Definise se orkestraotr klasa cije je jedina odgovornost da kaze saga ucesnicima sta da rade. Sada orkestrator komunicira sa ucesnicima upotrebom *command/async reply-style* interakciju. Da bi se izvrsio korak, salje se komanda sa porukom ucesniku govoreci mu koju operaciju da izvrsi, nakon sto saga ucesnik izvrsi operaciju, vraca povratnu poruku orkestratoru. Orkestrator obradi povratnu poruku i odluci koji korak je sledeci za izvrsavanje.

Prednosti saga orkestratora:

* *Jednostravnije zavisnosti* – ne postoji mogucnost kruzne zavisnosti kao kod koordinatora, saga aktivira ucesnika a ucesnici ne aktiviraju sagu,
* *Slaba sprega* – svaki servis implementira API koji je aktiviran od stane orkestratora,
* *Poboljsava razdvajanje briga (eng*. *Separation of concerns) i pojednostavljuju poslovnu logiku* – logika sage koordinatora je smestena unutra saga orkestratora.

Iako je orkestrator bolji od koordinator, to ne znaci da nema svoje mane, najveca mana orkestratora je *rizik od centralizacije prevelike poslovne logike orkestratora*.

## 5.4 Slucaj koriscenja

Kao sto je spomenuto u poglavlju 4.2, treba se teziti sto vecem rasterecenju servisa, odnosno kreiranje sa jednom ulogom. U tom poglavlju su bili prikazani slucajevi koriscenja monolitnog sistema i uoceno je da MAANPP20 ima veliko opterecenje i stoga je taj monolitni servis “razbijen” na vise manjih mikroservisa gde svaki ima svoju ulogu. Na slici 5.7 je prikazan dijagram slucaja koriscenja registrovanog korsnika u mikroservisnom sistemu. Kao sto je napomenuto, u poglavljaju 4.6, dodati su novi servisi i svaki servis kao i uslugu koju pruza je prikazano razlicitom bojom radi lakse uocljivosti.

Diagram

Description automatically generated  
Slika 5.7 Slucaj koriscenja registrovanog korisnika u mikroservisnom sistemu

## 5.5 Model podataka

U sledecim listinzima ce biti predstavljeni modeli koji ucestvuju u toku podataka. Moguci tokovi podataka bice predstavljeni u narednom poglavlju 5.6.

Napomena, zbog izrade diplomskog su kreirani elemntarni modeli za svaki pojedinacni mikroservis, koji sluze za demonstraciju sage.

Svaki od sledecih modela *SagaFlightReservation, Automobile, Hotel, Payment* su modeli od kojih su kreirane tabele u svakoj od baza mikroservisa u kojima se cuvaju rezervacije.

public class SagaFlightReservation

{

[Key]

public Guid Guid { get; set; }

public Guid UserId { get; set; }

public string Grad { get; set; }

public double Cena { get; set; }

}

Listing 5.1 Klasa *SagaFlightReservation*

public class Automobile

{

[Key]

public int id { get; set; }

public string Grad { get; set; }

public int? Avaible { get; set; }

public DateTime? Updated { get; set; }

}

Listing 5.2 Klasa *Automobile*

public class Hotel

{

[Key]

public int id { get; set; }

public string Grad { get; set; }

public int? Avaible { get; set; }

public DateTime? Updated { get; set; }

}

Listing 5.3 Klasa *Hotel*

public class Payment

{

[Key]

public int id { get; set; }

public Guid UserId { get; set; }

public double? Avaible { get; set; }

public DateTime? Updated { get; set; }

}

Listing 5.4 Klasa *Payment*

U listingu 5.5 je prikazana klasa *FlightStateData* koja predstavlja jednu instancu neke sage rezervaciije koja moze biti uspesna ili neuspesna, u zavisnost od toka podataka. Da bi mogla stanja sage da se cuvaju u bazi, mora se kreirati tabela od pomenute klase iz listinga 5.5 kao sto je prikazano na listingu 5.6.

public class FlightStateData : SagaStateMachineInstance

{

public Guid CorrelationId { get; set; }

public string CurrentState { get; set; }

public DateTime? CreationDateTime { get; set; }

public DateTime? CancelDateTime { get; set; }

public string UserId { get; set; }

public Guid FlightId { get; set; }

public int CarId { get; set; }

public int HotelId { get; set; }

public int PaymentId { get; set; }

public double price { get; set; }

public string grad { get; set; }

}

Listing 5.5 Klasa *FlightStateData*

USE [MAANPP20-SAGA]  
GO  
SET ANSI\_NULLS ON  
GO  
SET QUOTED\_IDENTIFIER ON  
GO  
CREATE TABLE [dbo].[FlightStateData](

[CorrelationId] [uniqueidentifier] NOT NULL,

[CurrentState] [nvarchar](max) NULL,

[CreationDateTime] [datetime] NULL,

[CancelDateTime] [datetime] NULL,

[UserId] [nvarchar](max) NULL,

[FlightId] [uniqueidentifier] NULL,

[CarId] [int] NULL,

[HotelId] [int] NULL,

[PaymentId] [int] NULL,

[price] [float] NULL,

[grad] [nvarchar](max) NULL,

CONSTRAINT [PK\_FlightStateData] PRIMARY KEY CLUSTERED (

[CorrelationId] ASC

)WITH (PAD\_INDEX = OFF, STATISTICS\_NORECOMPUTE = OFF, IGNORE\_DUP\_KEY = OFF, ALLOW\_ROW\_LOCKS = ON, ALLOW\_PAGE\_LOCKS = ON) ON [PRIMARY]  
) ON [PRIMARY] TEXTIMAGE\_ON [PRIMARY]  
GO

Listing 5.6 Kveri za kreiranje tabele *FlightStateData*

Najbitniji model je prikazan u listingu 5.7. Model predstavlja poruku koja se prenosi kroz tok podataka.

public interface IStartFlight

{

public string UserId { get; }

public Guid FlightId { get; }

public int CarId { get; }

public int HotelId { get; }

public int PaymentId { get; }

public double price { get; }

public string grad { get; }

}

Listing 5.7 Interfejs *IStartFlight*

Primer pracenja jednog stanja u sagi dat je u listingu 5.8. Metoda *During* ima ulogu da prati *FlightStarted* stanje i na osnovu nekog dogadjaja koji je definisan u *When* ona izvrsi odredjenu akciju *Then* i predje na drugo stanje uz pomoc ulancane metode *TransitionTo(CarStarted).* Metoda *During* moze da ima neogranicen broj uslova godjaja, ali nije preporuceno preterivati sa brojem dogadjaja.

During(FlightStarted,

When(CarStartedEvent)

.TransitionTo(CarStarted),

When(FlightCancelledEvent)

.Then(context => context.Instance.CancelDateTime = DateTime.Now)

.TransitionTo(Cancelled));

Listing 5.8 Primer pracenja sage jednog stanja

## 5.6 Tok podataka i arhitekturalni pogled na resenje

Razliciti scenariji toka podataka ucesnika sage ce biti objasnjeno i prikazano kroz sledece scenarije:

1. Uspesna rezervacija
2. Neuspesna rezervacija automobila
3. Neuspesna rezervacija hotela
4. Neuspesno placanje

Na sledeci slikama ce biti prikazani dijagrami sekvenci koji predstavljaju dijagrame interakcije jer opisuju kako i kojim redosledom grupa objekata radi zajedno.

Prvi scenario, prikazan na slici 5.6, predstavlja savrseni scenario gde nigde u sistemu nije nastala greska. Kada stigne zahtev rezervacije od klijenta na avio mikroservis, u daljem tekstu avio-ms, ta rezervacija se sacuva i probudi se saga masina, u daljem tekstu saga, koja prosledjuje zahteve ka sledecim mikroservisima za dalju obradu.

Diagram

Description automatically generated  
Slika 5.6 Uspesan tok podataka

U sledecem scenario, slika 5.7, greska nastane u automobilskom mikroservis, u daljem tekstu car-ms. Car-ms ne uspe da rezervise vozilo i nekog razloga, kao sto je npr. nedostatak vozila za taj grad. On javi sagi da nije uspeo, saga zatim javlja ucesniku avio-ms da obrise rezervaciju leta.

Diagram

Description automatically generated  
Slika 5.7 Neuspesna rezervacija vozila

Na slici 5.8 prikazan je treci scenario. U tom scenario se desila greska u hotelskom mikroservisu, u daljem tekstu hotel-ms. Hotel-ms je isto javio sagi da se desila greska, kao car-ms u prethodnom scenariu. Zatim saga javlja car-msu da obrise rezervaciju auta i javlja avio-ms da obrise rezervaciju leta.

Diagram

Description automatically generated  
Slika 5.8 Neuspesna rezervacija hotela

Poslednji scenario, prikazan na slici 5.9, se desi u mikroservisu za placanje celokupne usluge, odnosno svih rezrvacija, dalje u tekstu payment-ms. Payment-ms isto kao i hotel-ms i car-ms u prethodna dva slucaja javi sagi za neuspesno placanje i saga dalje orkestrira brisanje svih rezervacija.

Diagram

Description automatically generated  
Slika 5.9 Neuspesno placanje

U poglavlju 4.3 je bio prikazan i objasnjen pogled na arhitekturalno resenje monolitnog sistema. Za taj isti sistem, koji je preradjen u mikroservise, prikana je arhitektura na slici 5.10. Korsinici isto komuniciraju preko protokola HTTPS sa prednjom stranom, prednja strana komunicira sa samo jednim mikroservisom, odnosno avio mikroservisom. Zatim taj servis pobudi sagu, i saga na osnovu predefinisanih akcija pobudjuje, tj. odredjuje koji mikroservis sledeci izvrsava lokalne akcije.

Diagram

Description automatically generated  
Slika 5.10 Arhitekturalni pogled na mikroservisno resenje