Moderna mikroservisna arhitektura i ekosustav mikroservisa

Đurić, Dario

SVEUČILIŠTE JOSIPA JURJA STROSSMAYERA U OSIJEKU
FAKULTET ELEKTROTEHNIKE, RAČUNARSTVA I INFORMACIJSKIH TEHNOLOGIJA

Sveučilišni studij

MODERNA MIKROSERVISNA ARHITEKTURA I EKOSUSTAV MIKROSERVISA

Diplomski rad

Dario Đurić

Osijek, 2024.

SADRŽAJ:

1. UVOD

Posljednjih godina razvoj softvera je doživio značajan pomak u smislu arhitekture aplikacija. Arhitektura aplikacija predstavlja strukturu i organizaciju softvera kako bi se postigli određeni ciljevi poput skalabilnosti, održivosti, fleksibilnosti i efikasnosti. To je temeljni koncept pri razvoju softvera koji utječe na način na koji su komponente sustava konfigurirane i kako međusobno komuniciraju.

Tradicionalne monolitne arhitekture predstavljaju pristup razvoju aplikacije kao jedne velike i masivne jedinice. Karakterizira ih jedna centralna baza podataka i usko povezane (engl. *tightly coupled*) i složene strukture koje često otežavaju daljnju implementaciju zbog međuovisnosti koda. Kao odgovor na izazove koji proizlaze iz razvoja i održavanja velikih i kompleksnih aplikacija, monolitne arhitekture sve češće počinju mijenjati modularne i skalabilne arhitekture. Ovim diplomskim radom prikazat će se pristup izradi aplikacije koristeći arhitekturu mikroservisa.

Mikroservisna arhitektura je arhitekturalni obrazac dizajna (engl. *architectural design pattern*) softvera gdje su aplikacije podijeljene na male, labavo povezane (engl. *loosely coupled*) servise koji se mogu razvijati, skalirati i pokretati neovisno jedan o drugom. Svaki servis fokusiran je na određenu poslovnu sposobnost i radi neovisno jedan o drugom. Međusobno komuniciraju putem aplikacijskih programskih sučelja (engl. *Application Programming Interface,* API) koristeći lagane (engl. *lightweight*) protokole kao što je hipertekstualni transfer protokol (engl. *Hypertext Transfer Protocol,* HTTP) ili sustave za razmjenu poruka, primjerice RabbitMQ. Prednosti ove arhitekture su višestruke, primjerice rastavljanjem aplikacija na manje usluge, povećava se mogućnost održavanja i implementacije novih funkcionalnosti. Mikroservisna arhitektura također uvodi svoj niz izazova. Složenost raspodijeljenog računalnog sustava, komunikacija među servisima i dosljednost podataka među servisima jedni su od ključnih izazova kod razvijanja sustava mikroservisnom arhitekturom.

Na primjeru aplikacije za rezerviranje smještaja u ovom radu bit će prikazane glavne značajke arhitekture mikroservisa. Ova aplikacija će omogućiti domaćinu postavljanje slobodnih datuma za rezerviranje smještaja te gostima izradu rezervacije smještaja. Koristeći IntelliJ razvojno okruženje te programski okvir (engl. *framework*) Spring programskog jezika Java implementirat će se funkcionalnosti ove aplikacije. Vrednovanje aplikacije napravit će se koristeći HTTP klijent dostupan unutar IntelliJ razvojnog okruženja.

2. PREGLED PODRUČJA RADA

Mikroservisna arhitektura programske podrške jedna je od novijih arhitektura. Ona predstavlja nadogradnju na servisno orijentiranu arhitekturu i pripada raspodijeljenim arhitekturalnim stilovima jer podrazumijeva sudjelovanje više procesa sudjeluje u radu aplikacije. Svaki proces pokreće jednu uslugu, dok svaka usluga ima svoju poslovnu domenu. Ukoliko jedna operacija zahtijeva podatke i funkcionalnosti drugih domena (servisa), potrebno je implementirati njihovu međusobnu komunikaciju.

Jedan od značajnijih primjera upotrebe mikroservisne arhitekture jest Amazon, poznati internetski maloprodajni div, čija web stranica je prema [1] 2000. godine izvedena kao jedna monolitna aplikacija. Amazonov monolit činile su uske veze između višeslojnih usluga. Svaki put kada su razvojni programeri htjeli nadograditi ili skalirati sustav morali su pažljivo odspojiti te uske veze. Bio je to mukotrpan i skup proces. Aplikacija je idalje vrlo dobro radila sve do 2001. godine. Povećanjem broja korisnika došlo je i do potrebe za većim brojem razvojnih programera. Kako su timovi rasli količina koda se povećavala. Velik broj nadogradnji imao je negativan utjecaj na razvoj i rad takvog softvera te je došlo vrijeme za promjene u strukturi same aplikacije. Suočen s potrebom potpunog refaktoriranja svoje aplikacije, Amazon je rastavio svoju monolitnu aplikaciju u male, neovisno pokrenute aplikacije specifične za jednu uslugu. Razvojni programeri su planski izdvojili dijelove koda koji su postali zasebni servisi, primjerice servis za računanje poreza. Amazonova implementacija servisno orijentirane arhitekture bila je početak današnjih mikroservisa, među koje spada još jedan Amazonov proizvod – Amazon Web Services (AWS) platforma koja pruža korištenje računalnih resursa, primjerice pohrana podataka, putem Interneta.

Drugi je primjer jedna od najkorištenijih *streaming*[1] platformi danas – Netflix. U početnim mjesecima rada tvrtka je rukovala fizičkim DVD-ovima, a *streaming* je bio samo san. Monolitni dizajn aplikacije odgovarao je trenutnim zahtjevima korisnika. 2008. godine, prema [1], doživjeli su prvo ozbiljnije oštećenje baze podataka te tri dana nisu mogli dostavljati DVD-ove. Tada se okreću visoko pouzdanim i distribuiranim sustavima te odabiru AWS kao pružatelja usluga. Proces postupnog refaktoriranja monolitne arhitekture počinje 2009. godine. Ključni korak bio je migracija platforme za kodiranje

[1] streaming – online dijeljenje sadržaja, najčešće video

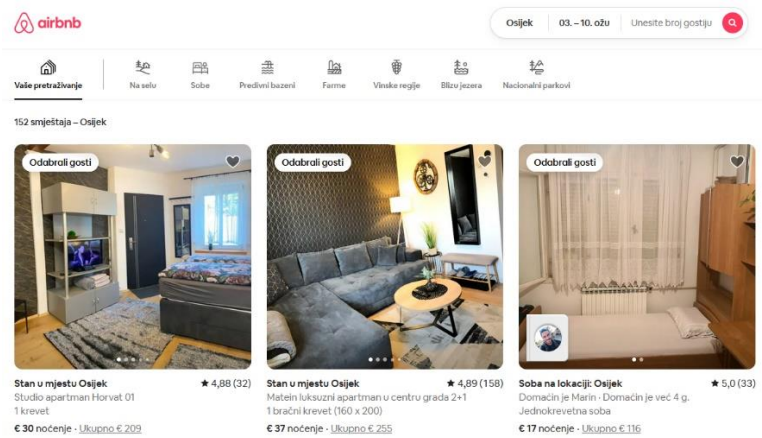
filmova koja nije bila vidljiva korisnicima kako bi se pokretala na AWS serverima kao neovisni mikroservis. Danas se arhitektura Netflix sustava sastoji od preko 700 labavo povezanih mikroservisa.

Još jedan primjer aplikacija s arhitekturom mikroservisa je Uber. Inicijalno razvijena kao monolitna aplikacija, tvrtka je prebrzo rasla da bi lako održavala originalnu arhitekturu aplikacije. Putnici i vozači su se, prema [1], povezivali s Uberovim monolitom putem API-ja. Postojala su tri adaptera s ugrađenim API-jem za funkcije poput naplate, plaćanja i tekstualnih poruka. Porastom popularnosti aplikacije, ovakva arhitektura postala je nepraktična i teška za daljnji razvoj. Razvojni programeri su izgradili pojedinačne mikroservise za funkcije poput upravljanja putnicima i upravljanja vožnjama. Timovi razvojnih programera usavršavali su se u rješavanju specifičnih problema. Zahvaljujući mikroservisima, ažuriranje jedne usluge nije utjecalo na ostale usluge te je znatno povećana otpornost na greške.

2.1. Pregled postojećih aplikacija za rezerviranje smještaja zasnovanih na mikroservisnoj arhitekturi

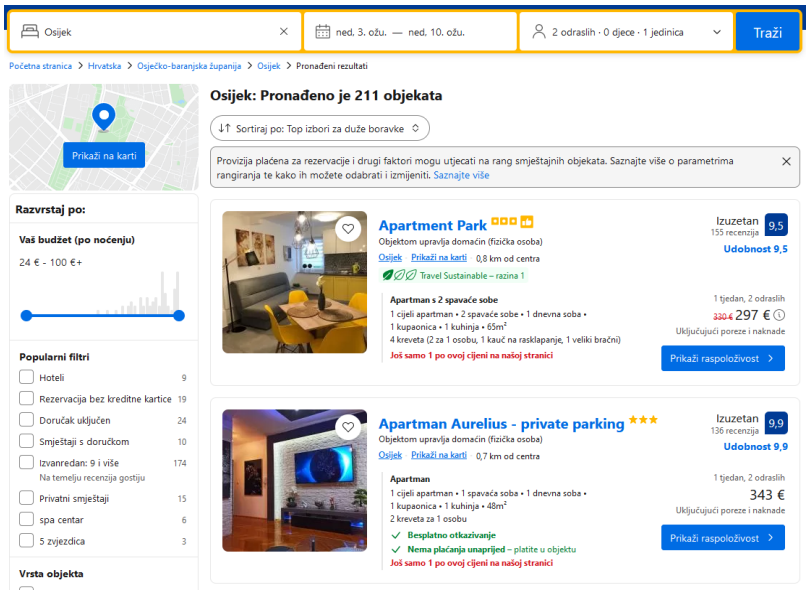
Aplikacije za rezerviranje smještaja revolucionirale su način na koji ljudi putuju te nude brojne prednosti i za putnika i za domaćina. Putnici mogu koristiti ove aplikacije za traženje smještaja prema različitim kriterijima, od kojih su najznačajniji lokacija, cijena te razne pogodnosti. Također mogu pročitati recenzije drugih putnika kako bi stekli bolji dojam o smještaju prije nego što rezerviraju smještaj. Putnicima se nudi širi izbor smještaja u odnosu na tradicionalne putničke agencije. Osim hotela, korisnici ovakvih aplikacija mogu rezervirati sobe, apartmane te čak i privatne kuće. Domaćini mogu izraditi popise za svoje nekretnine i postaviti vlastite cijene i dostupnost. Aplikacije za rezerviranje smještaja također olakšavaju domaćinima upravljanje rezervacijama i komunikaciju s gostima. Aplikacije za rezerviranje smještaja imaju potencijal riješiti još više problema u budućnosti, primjerice koristeći tehnologije virtualne stvarnosti omogućiti korisniku virtualni obilazak objekta. Neke od najpopularnijih aplikacija za rezerviranje smještaja danas su Airbnb i Booking.com. Dvije globalne platforme istaknule su se i stvorile mrežu smještaja koja osigurava korisnicima posjetu lokacijama od globalnih metropola do udaljenih mjesta za bijeg na osamljene lokacije. U nizu sličnih aplikacija izdvajaju ih jednostavnost procesa objavljivanja i rezerviranja. Korisnici u njima mogu javno recenzirati smještaj, čiji komentar može dublje informirati nove potencijalne korisnike. Airbnb i Booking.com ponajviše se razlikuju u tipu nekretnina koje se objavljuju, gdje se Booking.com više bazira na tradicionalnom načinu smještaja - hotelima, dok Airbnb sadrži više personaliziranih

smještaja, kao što su privatne kuće, stanovi i vikendice. Na slici 2.1. prema [2] prikazani su rezultati pretrage “Osijek” web stranice Airbnb.com, gdje se mogu vidjeti tri ponude.



Slika 2.1. Rezultati pretrage “Osijek” na web stranici Aribnb.com

S tehničke strane, Airbnb isprva je razvijan kao monolitna aplikacija koja je funkcionirala bez poteškoća. Kroz godine daljnjeg razvoja te porastom broja razvojnih programera koji rade na širenju aplikacije tvrtka se odlučila prijeći na arhitekturu mikroservisa. Na slici 2.2. prema [3] prikazani su rezultati pretrage “Osijek” na web stranici Booking.com gdje vidimo drukčiju ponudu smještaja.



Slika 2.2. Rezultati pretrage “Osijek ” na web stranici Booking.com

3. ZAHTJEVI NA PROGRAMSKO RJEŠENJE

Zahtjevi na programsko rješenje predstavljaju opise funkcionalnosti tog sustava. Prije početka razvoja softvera tim razvojnih programera analizira zahtjeve zatražene od klijenta. Analiza zahtjeva je proces razumijevanja i dokumentiranja kako aplikacija treba raditi te kako će ga tim razvojnih programera implementirati. To je kritičan dio procesa razvoja softvera, jer osigurava da se aplikacija ili proizvod razvija kako bi zadovoljio potrebe klijenta i njihovih budućih korisnika. Zahtjevi se obično dijele na dvije glavne vrste: funkcionalne i nefunkcionalne zahtjeve.

Aplikaciju za rezerviranje smještaja koristit će dva tipa korisnika, domaćin i gost te se podrazumijeva kako će se korisnici moći prijaviti u aplikaciju pomoću svojih korisničkih računa. Unutar aplikacije određene funkcionalnosti moći će koristiti oba tipa korisnika, primjerice dohvaćanje svih smještaja, dok će samo korisnik tipa domaćin moći postaviti ponudu smještaja i dostupne dane za određeni smještaj. Aplikacija će vraćati cjelobrojni rezultat koji se naziva HTTP statusni kod. Početni broj HTTP statusnog koda označava jednu od pet standardnih klasa odgovora.

- Informativni odgovori – HTTP statusni kod: 100 – 199
- Uspješni odgovor – HTTP statusni kod: 200 – 299
- Poruka preusmjeravanja – HTTP statusni kod: 300 – 399
- Odgovor za klijentske greške – HTTP statusni kod: 400 – 499
- Odgovor za poslužiteljske greške – HTTP statusni kod: 500 – 599

3.1. Funkcionalni zahtjevi

Funkcionalni zahtjevi opisuju specifične funkcionalnosti sustava kako bi zadovoljio potrebe korisnika. Primjerice, za sustav koji će omogućiti iznajmljivanje smještaja, funkcionalni zahtjev bi mogao biti: “Sustav mora omogućiti korisnicima provjeru je li smještaj dostupan unutar zadanih datuma”. Funkcionalni zahtjevi na sustav, prema [4], bit će raspisani u formatu *Dano-Kada-Tada* (engl. *Given-When-Then)*. Zadani format može se predstaviti kao kriterij prihvatljivosti sustava. Funkcionalni zahtjevi za izradu aplikacije za rezerviranje smještaja su detaljno raspisani u tablici funkcionalnih zahtjeva:

FZ-1: Izrada ponude smještaja, uspješni slučaj	**FZ-2: Izrada ponude smještaja, neuspješan slučaj**
DANA je mogućnost izrade ponude smještaja i ako je korisnik prijavljen kao 'domaćin' KADA korisnik unese atribute za izradu ponude smještaja: • adresa (tekst, minimalno 10 znakova) • broj ljudi (cijeli broj, veći od 0) • cijena (decimalni broj, veći od 0.0) i pozove endpoint[2] za izradu ponude smještaja TADA aplikacija mora validirati podatke i ako su točni, unijeti podatke u bazu i kao rezultat vratiti HTTP statusni kod 201.	DANA je mogućnost izrade ponude smještaja KADA korisnik ne unese sve potrebne atribute ili ih ne dostavi točne TADA aplikacija mora validirati podatke, uočiti greške i vratiti odgovor sa popisom atributa koji su krivo navedeni ili nisu dostavljeni i kao rezultat vratiti HTTP statusni kod 400.
FZ-3: Dohvaćanje ponude smještaja	**FZ-4: Dohvaćanje svih ponuđenih smještaja**
DANA je mogućnost dohvaćanja smještaja KADA korisnik pozove endpoint za dohvaćanje smještaja uz atribut: • uuid[3] (tekst, 32 znaka) TADA aplikacija mora validirati podatke i vratiti odgovarajući smještaj iz baze podataka i vratiti HTTP statusni kod 200.	DANA je mogućnost dohvaćanja smještaja KADA korisnik pozove endpoint za dohvaćanje smještaja TADA aplikacija mora vratiti sve smještaje iz baze podataka i vratiti HTTP statusni kod 200.
FZ-5: Uređivanje smještaja, uspješan slučaj	**FZ-6: Brisanje smještaja, uspješan slučaj**
DANA je mogućnost uređivanja smještaja KADA korisnik pozove endpoint za uređivanje smještaja uz atribut: • uuid (tekst, 32 znaka) TADA aplikacija mora validirati podatke i ako su točni, unijeti podatke u bazu i kao rezultat vratiti HTTP statusni kod 201.	DANA je mogućnost brisanja smještaja KADA korisnik pozove endpoint za brisanje smještaja uz atribut: • uuid (tekst, 32 znaka) TADA aplikacija mora validirati podatke i ako su točni, obrisati odgovarajući smještaj iz baze podataka i kao rezultat vratiti HTTP statusni kod 200.
FZ-7: Uređivanje smještaja, neuspješan slučaj	**FZ-8: Brisanje smještaja, neuspješan slučaj**
DANA je mogućnost uređivanja smještaja KADA korisnik ne unese potreban atribut ili ga ne dostavi točno TADA aplikacija mora validirati podatke i vratiti odgovor sa popisom atributa koji su krivo navedeni ili nisu dostavljeni i kao rezultat vratiti HTTP statusni kod 400.	DANA je mogućnost brisanja smještaja KADA korisnik ne unese potreban atribut ili ga ne dostavi točno TADA aplikacija mora validirati podatke, uočiti greške i vratiti odgovor sa popisom atributa koji su krivo navedeni ili nisu dostavljeni. i kao rezultat vratiti HTTP statusni kod 400.
FZ-9: Dohvaćanje dostupnosti smještaja, uspješan slučaj	**FZ-10: Dohvaćanje dostupnosti smještaja, neuspješan slučaj**
DANA je mogućnost dohvaćanja dostupnosti smještaja KADA korisnik pozove endpoint za dohvaćanje dostupnosti smještaja uz atribute: • uuid (tekst, 32 znaka) • startDate (datum u formatu yyyy-mm-dd) • endDate (datum u formatu yyyy-mm-dd) TADA aplikacija mora vratiti podatke o smještaju uz podatak je li smještaj dostupan unutar zadanog vremenskog perioda i kao rezultat vratiti HTTP statusni kod 200.	DANA je mogućnost dohvaćanja dostupnosti smještaja KADA korisnik ne unese sve potrebne atribute ili ih ne dostavi točne TADA aplikacija mora validirati podatke, uočiti greške i vratiti odgovor sa popisom atributa koji su krivo navedeni ili nisu dostavljeni i kao rezultat vratiti HTTP statusni kod 400.

[2] endpoint – završna pristupna točka, (engl. Uniform Resource Locator – URL)

[3] uuid – jedinstveni identifikator unutar aplikacije

FZ-11: Izrada dostupnosti smještaja, uspješan slučaj	**FZ-12: Izrada dostupnosti smještaja, neuspješan slučaj**
DANA je mogućnost dodavanja dostupnosti smještaja KADA korisnik pozove endpoint za dohvaćanje dostupnosti smještaja uz atribute: • uuid (tekst, 32 znaka) • startDate (datum u formatu yyyy-mm-dd) • endDate (datum u formatu yyyy-mm-dd) TADA aplikacija sprema podatke o dostupnosti smještaja i kao rezultat vraća HTTP statusni kod 200.	DANA je mogućnost dodavanja dostupnosti smještaja KADA korisnik ne unese sve potrebne atribute ili ih ne dostavi točne TADA aplikacija mora validirati podatke, uočiti greške i vratiti odgovor sa popisom atributa koji su krivo navedeni ili nisu dostavljeni i kao rezultat vratiti HTTP statusni kod 400.
FZ-13: Izrada plaćanja, uspješan slučaj	**FZ-14: Izrada plaćanja, neuspješan slučaj**
DANA je mogućnost izrade plaćanja KADA korisnik pozove endpoint za izradu plaćanja uz atribut: • balance (cijeli broj, veći od 0) TADA aplikacija sprema plaćanje u bazu podataka i vraća HTTP statusni kod 200.	DANA je mogućnost izrade plaćanja KADA korisnik ne unese sve potrebne atribute ili ih ne dostavi točne TADA aplikacija mora validirati podatke, uočiti greške i vratiti odgovor sa popisom atributa koji su krivo navedeni ili nisu dostavljeni i vraća HTTP statusni kod 400.
FZ-15: Dohvaćanje plaćanja, uspješan slučaj	**FZ-16: Dohvaćanje svih plaćanja**
DANA je mogućnost dohvaćanja plaćanja KADA korisnik pozove endpoint za dohvaćanje plaćanja uz atribut: • uuid (tekst, 32 znaka) TADA aplikacija vraća odgovarajuće plaćanje iz baze podataka i vraća HTTP statusni kod 200.	DANA je mogućnost dohvaćanja plaćanja KADA korisnik pozove endpoint za dohvaćanje plaćanja i ne unese atribut TADA aplikacija vraća sva plaćanja iz baze podataka i vraća HTTP statusni kod 200.
FZ-17: Brisanje plaćanja, uspješan slučaj	**FZ-18: Brisanje plaćanja, neuspješan slučaj**
DANA je mogućnost brisanja plaćanja KADA korisnik pozove endpoint za brisanje plaćanja uz atribut: • uuid (tekst, 32 znaka) TADA aplikacija briše odgovarajuće plaćanje iz baze podataka i vraća HTTP statusni kod 200.	DANA je mogućnost brisanja plaćanja KADA korisnik ne unese sve potrebne atribute ili ih ne dostavi točne TADA aplikacija mora validirati podatke, uočiti greške i vratiti odgovor sa popisom atributa koji su krivo navedeni ili nisu dostavljeni i vraća HTTP statusni kod 400.
FZ-19: Izrada rezervacije, uspješan slučaj	**FZ-20: Izrada rezervacije, neuspješan slučaj**
DANA je mogućnost izrade rezervacije I ako je korisnik prijavljen kao 'gost' KADA korisnik pozove endpoint za izradu rezervacije uz atribut: • placeUuid (tekst, 32 znaka) • startDate (datum u formatu yyyy-mm-dd) • endDate (datum u formatu yyyy-mm-dd) i ako je smještaj dostupan za zadane datume TADA aplikacija poziva servis za naplatu kako bi provjerila stanje računa i sprema zadanu rezervaciju u bazu podataka i kao rezultat vraća HTTP statusni kod 200.	DANA je mogućnost izrade rezervacije KADA korisnik pozove endpoint za izradu rezervacije i korisnik ne unese sve potrebne atribute ili ih ne dostavi točne TADA aplikacija mora validirati podatke, uočiti greške i vratiti odgovor sa popisom atributa koji su krivo navedeni ili nisu dostavljeni i kao rezultat vratiti HTTP statusni kod 400.
FZ-21: Dohvaćanje rezervacije, uspješan slučaj	**FZ-22: Dohvaćanje svih rezervacija**
DANA je mogućnost dohvaćanja rezervacije KADA korisnik pozove endpoint za dohvaćanje rezervacije uz atribut: • uuid (tekst, 32 znaka) TADA aplikacija vraća rezervaciju iz baze podataka i kao rezultat vraća HTTP statusni kod 200.	DANA je mogućnost dohvaćanja rezervacije KADA korisnik pozove endpoint za dohvaćanje rezervacije TADA aplikacija vraća sve rezervacije iz baze podataka i vraća HTTP statusni kod 200.

FZ-23: Brisanje rezervacije, uspješan slučaj	FZ-24: Brisanje rezervacije, neuspješan slučaj
DANA je mogućnost brisanja rezervacije KADA korisnik pozove endpoint za brisanje rezervacije uz atribut: • uuid (tekst, 32 znaka) TADA aplikacija briše odgovarajuću rezervaciju iz baze podataka i vraća HTTP statusni kod 200.	DANA je mogućnost brisanja rezervacije KADA korisnik pozove endpoint za brisanje rezervacije i ne unese sve potrebne atribute ili ih ne dostavi točne TADA aplikacija mora validirati podatke, uočiti greške i vratiti odgovor sa popisom atributa koji su krivo navedeni ili nisu dostavljeni i kao rezultat vratiti HTTP statusni kod 400.

3.2. Nefunkcionalni zahtjevi

Nefunkcionalni zahtjevi opisuju kvalitete sustava, kao što su njegove performanse, sigurnost, pouzdanost i uporabljivost. Opisuju karakteristike koje softver mora imati te kako će softver raditi. Mogu se podijeliti u nekoliko skupina, odnosno svaki nefunkcionalni zahtjev odgovara na pitanje:

- Performanse – Koliko brzo će aplikacija vratiti rezultat?
- Skalabilnost – Koliko će se performanse aplikacije promijeniti u slučaju većeg opterećenja?
- Pouzdanost – Koliko su česte kritične greške aplikacije?
- Održivost – Koliko dugo traje prosječno vrijeme nedostupnosti aplikacije?
- Sigurnost – Koliko su zaštićene informacije koje aplikacija sprema?

Jedan općeniti nefunkcionalni zahtjev za aplikacije koje će omogućiti iznajmljivanje smještaja su: "Sustav mora spriječiti neovlašten pristup podacima kriptiranjem i/ili postupkom autorizacije". Osim navedenog nefunkcionalni zahtjevi za izradu ove aplikacije su sljedeći:

- Aplikacija bi trebala implementirati OAuth 2.0[4] za autorizaciju i autentifikaciju – Zato što OAuth 2.0 pruža snažan programski okvir za sigurnu autorizaciju i autentifikaciju. Time se osigurava pristup samo registriranim korisnicima. Korisnici se mogu prijaviti u aplikaciju i samo ovlašteni korisnici mogu pristupiti osjetljivim podacima.
- Aplikacija bi trebala biti horizontalno skalabilna koristeći tehnologije kao što je Docker – Zato što horizontalna skalabilnost osigurava kako će aplikacija moći podnijeti povećano opterećenje dodavanjem više instanci mikroservisa na više poslužitelja ili koristeći više

[4] OAuth 2.0 – industrijski standardni protokol za autorizaciju

Docker spremnika (engl. *container*). Time se osigurava optimalna izvedba i pravilna raspodjela resursa.

- Aplikacija bi trebala optimizirati upite na bazu podataka i osigurati odgovor u roku 500 milisekundi – Zato što će se time znatno poboljšati iskustvo korisnika (engl. *user experience*). Time se povećava vjerojatnost kako će korisnik uspješno rezervirati smještaj.
- Aplikacija bi trebala biti sposobna odgovoriti na više istovremenih zahtjeva – Zato što se u vrijeme praznika i turističke sezone očekuje veći broj istovremeno aktivnih korisnika. Skalabilnost osigurava kako će aplikacija zadržati konstantan odziv i dostupnost čak i kada dolazi do povećanja prometa na aplikaciji.
- Aplikacija bi trebala biti lako održiva i sposobna za nadogradnju – Zato što će se aplikacija dodatno razvijati s porastom broja korisnika. Druga verzija sustava sadržavati će grafičko sučelje, dok je planirano dodavanje novog servisa odgovornog za slanje računa putem emaila.

4. OPIS KORIŠTENIH TEHNOLOGIJA

Analizom navedenih zahtjeva donesena je odluka kako će se softver razvijati pomoću arhitekture mikroservisa zato što se pravilnom implementacijom arhitekture mikroservisa osigurava skalabilnost, nastavak razvoja aplikacije i skaliranje, raznolikost tehnologija, povećava se otpornost na greške te poboljšava autonomija razvojnog tima. Programski jezik Java odabran je zbog lakog pokretanja na svim operacijskim sustavima te programski okvir Spring Boot. Kako se radi o desktop aplikaciji, odabran je Maven alat za automatiziranu izradu aplikacija. Aplikacija u izvornoj verziji neće sadržavati grafičko sučelje nego će korisnici s aplikacijom komunicirati putem API-ja koje omogućuje prijenos reprezentativnih stanja (engl. *Representational State Transfer,* REST). Umjesto servisa za upravljanje korisnicima, aplikacija za rezerviranje smještaja će koristiti autorizacijski poslužitelj Keycloak. Keycloak će autorizirati korisnike te generirati *JSON Web Token* (JWT) koji će korisniku omogućiti pristup sustavu. Za pravilan rad sustava potrebne su sinkrona i asinkrona komunikacija između servisa. Za sinkronu komunikaciju koristiti će se REST API, a za asinkronu će se implementirati napredni protokol za smještanje poruka u red čekanja (engl. *Advanced Message Queuing Protocol,* AMQP) izveden pomoću RabbitMQ poslužitelja. Detaljnije objašnjenje svih korištenih tehnologija slijedi u nastavku poglavlja.

4.1. Spring

Spring [5] je jedan od mnogih programskih okvira temeljen na Java programskom jeziku. Programski okviri pojednostavljuju razvoj aplikacija pružajući gotove biblioteke i alate za automatizaciju. Ovaj programski okvir slijedi mnoga načela programiranja među kojima se izdvajaju ubrizgavanje ovisnosti (engl. *Dependency Injection,* DI) i inverzija kontrole (engl. *Inversion of Control,* IoC). Ubrizgavanje ovisnosti u Spring programskom okviru koristi se za smanjenje ovisnosti između različitih komponenata unutar aplikacije. Preuzima odgovornost održavanja životnog ciklusa objekta. Najčešće se DI postiže koristeći IoC koncept, za što postoje tri načina:

- ubrizgavanjem konstruktora (engl. *constructor injection*)
- ubrizgavanjem metode za postavljanje (engl. *setter injection*)
- ubrizgavanjem polja (engl. *field injection*)

U kontekstu Spring programskog okvira kontrola se pomoću koncepta inverzija kontrole prepušta Spring spremniku pomoću dva sučelja: (engl. *interface*) *BeanFactory* i *ApplicationContext*. Time se osigurava kako se komponente aplikacije neće same instancirati čime se promiče održivost sustava. Ključni koncept unutar programskog okvira Spring je *bean*[5]. Može se predstaviti kao objekt kojim upravlja Springov IoC spremnik. Ima temeljnu ulogu u upravljanju i povezivanju komponenti unutar Spring aplikacije. *Bean* se definira korištenjem određenih oznaka (engl. *annotations*), konfiguracijskih datoteka pisanih u XML-u (engl. *eXtensible Markup Language*) ili konfiguracijskih klasa Java programskog jezika. Kada se klasa ili metoda deklarira kao *bean*, Spring će voditi brigu o njegovoj konfiguraciji, ovisnostima i životnom ciklusu. Oznake, primjerice *@Component*, *@Service*, *@Repository* i *@Controller* označavaju klase kao *bean* i omogućuju njihovo automatsko otkrivanje od strane IoC spremnika. Oznaka *@Bean* može se koristiti na metodama koje moraju vratiti kao rezultat inicijaliziranu implementaciju neke klase. Najčešće su to metode konfiguracijskih klasa i metode klasa potrebnih za ispravan rad cjelokupnog sustava, primjerice klase za komunikaciju između servisa.

Slika 4.1. Spring logo

4.1.1. Spring Boot

Spring Boot [6] predstavlja proširenje Spring programskog okvira koja olakšava stvaranje samostojećih (engl. *stand-alone*) aplikacija. Pruža kolekciju početnih ovisnosti (engl. *starter dependencies*), koje pojednostavljuju uključivanje uobičajenih biblioteka i okvira. Ne zahtijeva posebnu XML konfiguraciju, nego na temelju *classpatha*[6] i drugih definiranih uvjeta osigurava automatsku konfiguraciju. Spring Boot nudi ugrađene servere kao što su Tomcat i Jetty te se

[5] bean – U doslovnom prijevodu grah, no kontekst je potekao od zrna kave koje može biti spremljeno u staklenku (engl. *jar*) – koja predstavlja ekstenziju zapakirane Java datoteke
[6] classpath – lokacija na kojoj Java traži klase i resurse potrebne za izvođenje programa

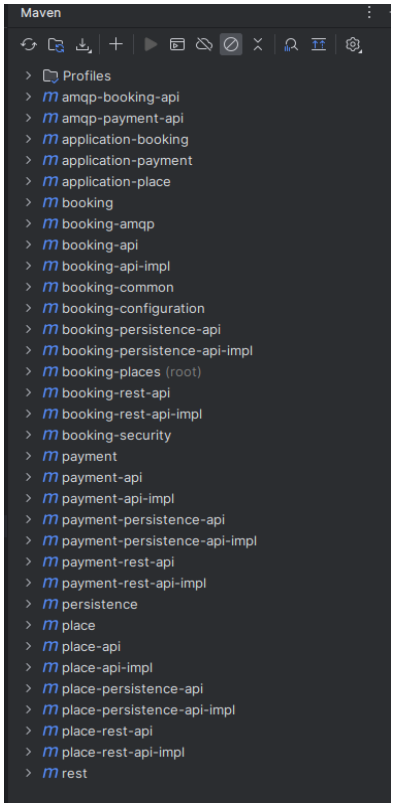
prvenstveno koristi za brzu izradu REST API-ja. Početne ovisnosti pomažu u smanjenju broja ovisnosti koje se moraju ručno dodati kako bi se omogućila određena funkcionalnost. Primjerice, dodavanjem *Spring Boot Starter Web* ovisnosti, omogućuje se implementacija REST upravljača (engl. *controller*) Neke od početnih ovisnosti koje koristi Spring Boot su:

- *Spring Boot Starter* – pruža plugin i ovisnosti za upravljanje Maven aplikacijama
- *Spring Boot Starter Web* – početna ovisnost za web, REST i MVC[7] aplikacije, koristi Tomcat
- *Spring Boot Starter Security* – početna ovisnost za sigurnost
- *Spring Boot Starter Logging* – početna ovisnost za bilježenje

4.2. Maven

Maven [7] je alat za automatiziranu izradu te alat za upravljanje java paketima. Prvenstveno se koristi u projektima pisanima u Java programskom jeziku, međutim može se koristiti i u drugim programskim jezicima. Ključne značajke i koncepti Mavena su projektni objektni model (POM), koji je XML datoteka pod nazivom *pom.xml*. POM definira konfiguraciju projekta, postavke izgradnje, ovisnosti, dodatke i druge informacije povezane s projektom. Maven omogućuje navođenje Maven ovisnosti koje projekt zahtjeva koje se mogu postaviti na udaljeni repozitorij. Maven zapakira projekt u Maven projekt te implementiranjem aplikacije koristeći više modula unutar jednog projekta omogućuje se dodavanje ovisnosti jednog modula o drugom. Unutar projekta može dodati i vanjske ovisnosti, uključujući njihove verzije, a Maven će automatski preuzeti i upravljati tim ovisnostima iz udaljenih repozitorija. Maven slijedi unaprijed definirani životni ciklus izrade programa s fazama kao što su prevođenje programskog koda iz čovjeku čitljivog teksta u tekst čitljiv računalu (engl. *compile*), testiranje, pakiranje[8] instalacija i postavljanje (engl. *deploy*). Na slici 4.2. prikazan je Maven *taskbar*[9] unutar razvojnog okruženja IntelliJ, na kojem su prikazani svi moduli unutar aplikacije za rezerviranje smještaja.

[7] MVC – *Model-View-Controller* koncept implementacije aplikacije kao 3 logičke komponente
[8] maven package – specifični Maven zadatak (engl *goal*) za prevođenje koda u distributivni format, primjerice .jar
[9] taskbar – traka koja pruža brži pristup i upravljanje zadacima



Slika 4.2. Maven *taskbar* unutar IntelliJ IDEA razvojnog okruženja

4.3. Docker

Docker [8] je platforma otvorenog koda (engl. *open-source*) koja razvojnim programerima omogućuje automatizaciju implementacije i upravljanja aplikacijama unutar Dcoker spremnika. Spremnici su labavo povezana, izolirana programska okruženja koja pakiraju aplikacije i njihove ovisnosti, omogućujući im dosljedan rad u različitim računalnim okruženjima. Time se, primjerice, pruža mogućnost implementiranja koda u više različitih verzija programskog jezika unutar više Docker spremnika, bez da oni utječu jedan na drugog. Određena verzija programskog jezika bit će instalirana samo unutar spremnika gdje se nalazi aplikacija koja zahtijeva tu verziju. Docker slikovna datoteka (engl. *docker image*) je samostalan paket samo za čitanje koji sadrži sve potrebne komponente za pokretanje aplikacije. Slike se mogu verzionirati, dijeliti putem Interneta i ponovno koristiti, što olakšava distribuciju i implementaciju aplikacija. Implementiranjem *docker-compose.yml* konfiguracijske datoteke znatno se pojednostavljuje proces pokretanja više povezanih spremnika. Na slici 4.3. prikazana je datoteka *docker-compose.yml* unutar koje se nalaze tri servisa: Postgres, RabbitMQ te Keycloak. Pokretanjem navedene datoteke pokreću se sva tri servisa. Nije nužno

pokrenuti sve servise, odnosno mogu se odabrati samo oni servisi koji su trenutno potrebni. Unutar razvojnog okruženja IntelliJ postoji blok *Services* pomoću kojeg se može ručno pokrenuti pojedinačne resurse neovisno o ostalim komponentama.

```yaml
version: '3.3'

services:
  postgres:
    image: "postgres"
    environment:
      POSTGRES_USER: root
      POSTGRES_PASSWORD: root
    ports:
      - "5432:5432"

  rabbitmq:
    image: "rabbitmq:3-management-alpine"
    ports:
      - "5672:5672"
      - "15672:15672"

  key-cloak:
    image: quay.io/keycloak/keycloak:latest
    environment:
      KEYCLOAK_ADMIN: admin
      KEYCLOAK_ADMIN_PASSWORD: ${KEYCLOAK_ADMIN_PASSWORD}
      DB_VENDOR: h2
    command:
      - start-dev
    ports:
      - "9000:8080"
```

Slika 4.3. Primjer konfiguracijske datoteke *docker-compose.yml*

4.4. PostgreSQL

Postgres, odnosno PostgreSQL [9] je *open-source* sustav za upravljanje relacijskim bazama podataka. Omogućuje pohranu podataka u strukturirane tablice s unaprijed definiranim shemama. Podržava SQL jezik za postavljanje upita, manipuliranje i upravljanje podacima. Omogućuje integritet transakcije[10], dopuštajući grupiranje više operacija u jednu transakciju i vraćanje unatrag (engl. *rollback*) u slučaju kvarova te osigurava *ACID* svojstva. Transakcija predstavlja logičku jedinicu rada koja se sastoji od

[10] integritet transakcije – očuvanje ispravnosti podataka tijekom izvođenja transakcija

jednog ili više upita baze podataka. *ACID* predstavlja četiri svojstva koja pridonose očuvanju pouzdanosti podataka tijekom izvođenja transakcija, a ta svojstva su:

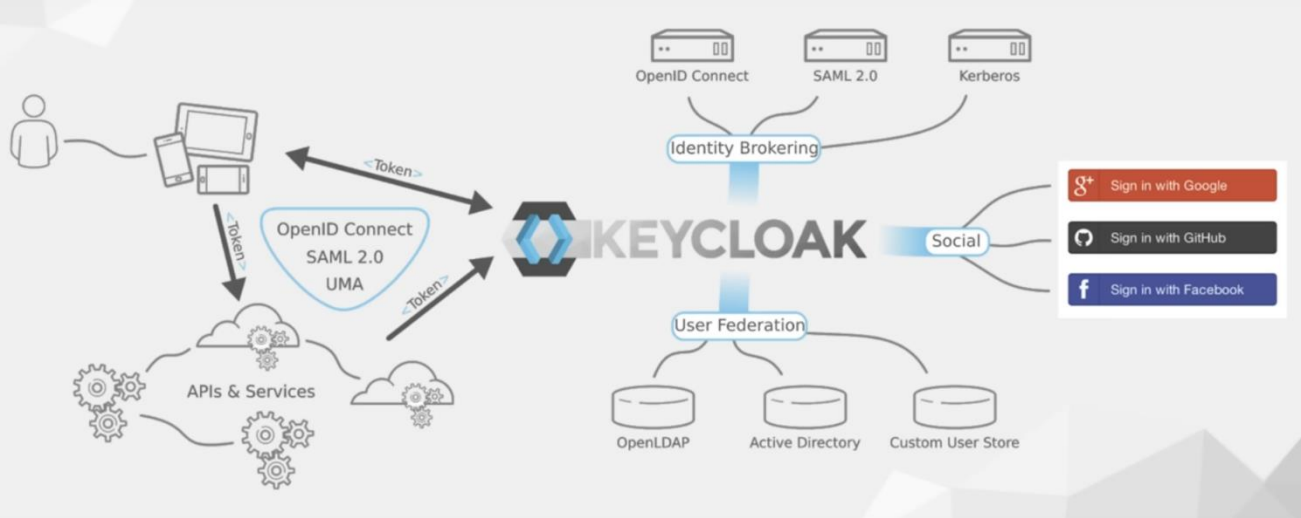
- atomarnost (engl. *atomicity*) – ako radnja s bazom podataka u sklopu transakcije završi pogreškom, tada cijela transakcija prestaje te baza ostaje nepromijenjena
- dosljednost (engl. *consistency*) – osigurava ispravno unošenje podataka u bazu podataka
- izolacija (engl. *isolation*) – omogućuje istovremenu radnju s bazom podataka tako da izgleda kao da su radnje obavljene jedna iza druge
- trajnost (engl. *durability*) – sve završene transakcije ostat će zapisane i nepromijenjene u slučaju kvara sustava

Omogućuje različite tehnike optimizacije performansi, kao što su indeksiranje, optimizacija upita i paralelno izvođenje te je dizajniran za rukovanje radnim opterećenjima velikog volumena. PostgreSQL ima aktivnu zajednicu razvojnih programera koja pridonosi njegovom razvoju, održavanju i poboljšanju. Ima bogat ekosustav proširenja, alata i biblioteka koje poboljšavaju njegovu funkcionalnost i integriraju se s drugim tehnologijama.

4.5. Keycloak

Keycloak [10] je *open-source* autorizacijski poslužitelj koji nudi jednostavno upravljanje korisnicima (engl. *user management*). Može se koristiti kao samostalni poslužitelj ili integrirati u postojeće sustave. U kontekstu arhitekture mikroservisa, Keycloak igra ključnu ulogu u pružanju usluga autentifikacije i autorizacije. Omogućuje definiranje i upravljanje korisničkim identitetima, ulogama i dopuštenjima na središnjem mjestu, čime eliminira potrebu za pojedinačnim servisima za upravljanje korisnicima, čineći sustav lakšim za održavanje i skalabilnijim. Keycloak izdaje pristupne JWT tokene pomoću industrijski standardiziranih protokola. Omogućuje definiranje uloga i dopuštenja za različite tipove korisnika. Mikroservisi mogu potvrditi pristupne tokene za autentifikaciju i autorizaciju dolaznih zahtjeva. Ovaj mehanizam provjere autentičnosti temeljen na tokenu osigurava sigurnu komunikaciju bez stanja između mikroservisa. Na slici 4.4. prema [11] opisana je arhitektura autorizacijskog poslužitelja Keycloak, gdje se mogu uočiti glavni elementi servisa: sigurna komunikacija koristeći pristupne tokene, posredovanje identitetima te mogućnost povezivanja korisničkog računa s računima servisa Google, GitHub i Facebook. Uključuje različite sigurnosne

značajke, primjerice podršku za istek tokena, multi-faktorsku autentifikaciju (MFA) i siguran prijenos tokena. Također pomaže u rješavanju zahtjeva usklađenosti, kao što je Opća uredba o zaštiti podataka (engl. *General Data Protection Regulation,* GDPR), pružajući značajke za upravljanje pristankom korisnika i zaštitu podataka.

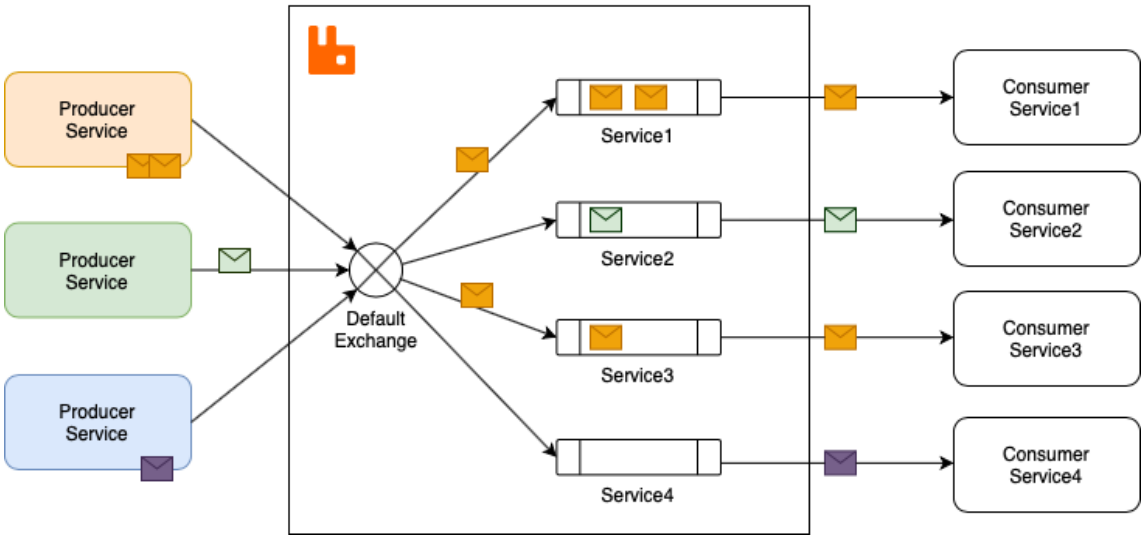


Slika 4.4. Pregled Keycloaka

4.6. RabbitMQ

RabbitMQ [12] je posrednik (engl. *broker*) koji služi za razmjenu poruka između aplikacija ili komponenti u raspodijeljenom računalnom sustavu. Implementira napredni protokol za slanje poruka u red čekanja odnosno AMQP. Omogućuje asinkronu komunikaciju između servisa dopuštajući proizvođačima (engl. *producer*) slanje poruka u redove čekanja (engl. *queue*), a potrošačima (engl. *consumer*) dohvaćanje poruke iz tih redova čekanja. Asinkrona komunikacija osigurava normalan rad sustava u slučaju da neka poruka nije odmah pročitana. Redovi čekanja djeluju kao međuspremnici koji drže poruke sve dok se ne potroše (pročitaju). RabbitMQ koristi razmjenu (engl. *exchange*) za usmjeravanje poruka od proizvođača do reda čekanja. Poruke se pomoću ključa za usmjeravanje (engl. *routing key*) povezuju (engl. *binding*) s određenim redom čekanja čime se osigurava pravilna komunikacija. RabbitMQ podržava obrazac slanja poruka objavi-pretplati (engl. *publish-subscribe*), gdje se jedna poruka emitira većem broju korisnika. Ovaj obrazac omogućuje slanje poruka u više redova ili potrošača istovremeno. Koristi mehanizam potvrde poruke kako bi osigurao pouzdanu isporuku poruke. Potrošači šalju potvrdu brokeru nakon obrade poruke, potvrđujući da je poruka uspješno obrađena. Ako korisnik ne uspije potvrditi poruku, RabbitMQ je može ponovno staviti u red

čekanja za ponovnu dostavu. Na slici 4.5. prema [13] prikazana je jednostavna izmjena poruka, gdje se može vidjeti asinkrona komunikacija s primjerima praznog reda te redova u kojima poruke čekaju potrošača.



Slika 4.5. Skica izmjene poruka pomoću RabbitMQ

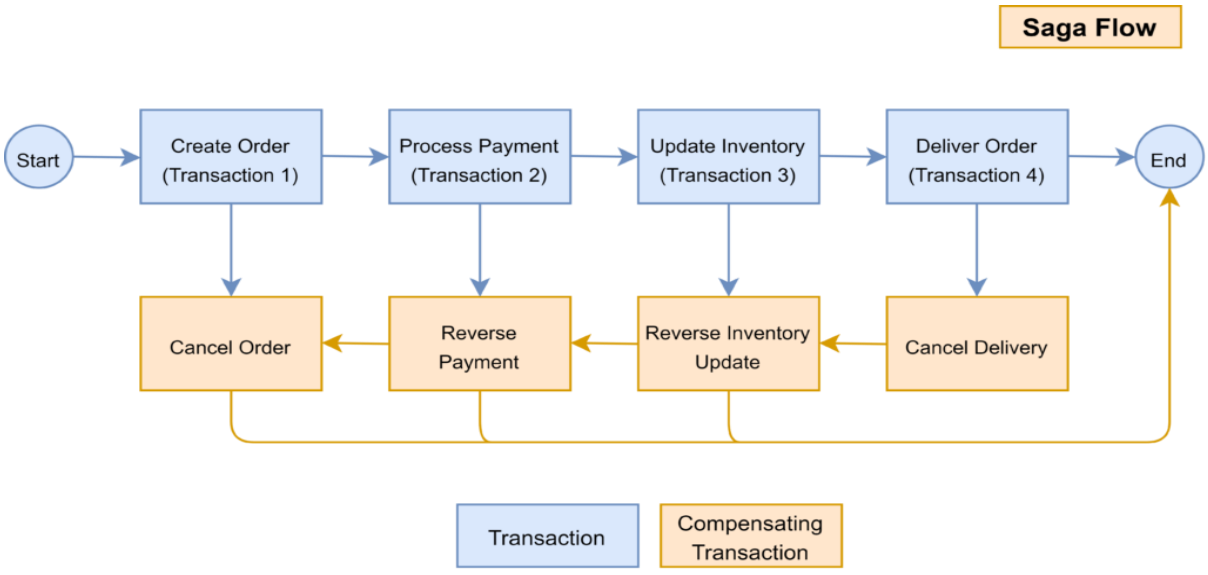
4.7. REST

REST [14] je skup principa i stil arhitekture za razvijanje umreženih aplikacija. Pruža jednostavan i fleksibilan pristup izgradnji raspodijeljenih sustava koje mogu lako koristiti različiti klijenti, uključujući web preglednike, mobilne aplikacije i druge poslužitelje. REST je bez stanja, što znači da svaki zahtjev od klijenta prema poslužitelju sadrži sve potrebne informacije za obradu tog zahtjeva. Poslužitelj ne održava nikakvo stanje klijenta između zahtjeva. Sve tretira kao resurs koji je obično predstavljen u obliku XML ili JSON (engl. *JavaScript Object Notation*) datoteke. Nudi skup standardnih metoda za interakciju s resursima. Ove metode su općenito poznate kao CRUD operacije:

- POST – stvaranje (engl. *create*)
- GET – čitanje (engl. *read*)
- PUT/PATCH – ažuriranje (engl. *update*)
- DELETE – brisanje (engl. *delete*).

4.8. Saga

U kontekstu mikroservisa Saga [15] je obrazac dizajna (engl. *design pattern*) koji se koristi za održavanje dosljednosti podataka i rukovanje dugotrajnim transakcijama preko više mikroservisa. Različiti servisi često imaju vlastite lokalne baze podataka i rade neovisno, no jedna operacija može uključivati više servisa i zahtijevati suradnju radi održavanja cjelokupnosti podataka. Saga predstavlja slijed lokalnih transakcija, gdje svaka odgovara operaciji unutar pojedinog mikroservisa. To su uglavnom operacije baze podataka ili servisni pozivi unutar mikroservisa. Orkestriranje (engl. *orchestration*) prestavlja proces koordinacije u kojoj središnji kontroler odnosno orkestrator govori sudionicima sage koje lokalne transakcije trebaju izvršiti. Orkestrator sage prati izvršenje lokalnih transakcija te odlučuje koje će korake pokrenuti sljedeće na temelju ishoda prethodnih koraka. Ako se dogodi kvar u bilo kojem koraku Sage, orkestrator Sage pokreće kompenzacijske radnje kako bi poništio ili kompenzirao promjene napravljene prethodnim koracima. Ove kompenzacijske radnje osmišljene su kako bi vratile sustav u dosljedno stanje. Saga se nastavlja sve dok se svi koraci uspješno ne izvrše ili se ne može primijeniti kompenzacijska radnja. Na slici 4.6. prema [16] prikazan je tijek Saga obrasca u scenariju online narudžbe s plaćanjem i dostavom.



Slika 4.6. Prikaz toka Saga obrasca

5. ARHITEKTURA PROGRAMSKOG RJEŠENJA

Arhitektura programskog rješenja predstavlja komponente aplikacije, način njihovog međudjelovanja te načela i smjernice koje upravljaju dizajnom i razvojem sustava. Prilikom planiranja razvijanja nekog softvera tim razvojnih programera mora donijeti odluku o njenoj strukturi, odnosno arhitekturi. Ne može se eksplicitno reći koja je arhitektura bolja, zato što sve arhitekture imaju svoje prednosti i nedostatke. Osim ranije spomenutih mikroservisne i monolitne arhitekture neki od najpoznatijih arhitekturalnih stilova su:

- slojevita arhitektura (engl. *layered architecture)*
- arhitektura model-pogled-kontroler (engl. *model-view-controller architecture* – MVC)
- arhitektura klijent-poslužitelj (engl. *client-server architecture*)
- objektno orijentirana arhitektura (engl. *object oriented architecture*)
- arhitektura pokrenuta događajima (engl. *event driven architecture*)
- servisno orijentirana arhitektura (engl. *service oriented architecture* – SOA)
- cjevovodna arhitektura (engl. *pipe and filter architecture*)
- mikrojezgrena arhitektura (engl. *microkernel architecture*)

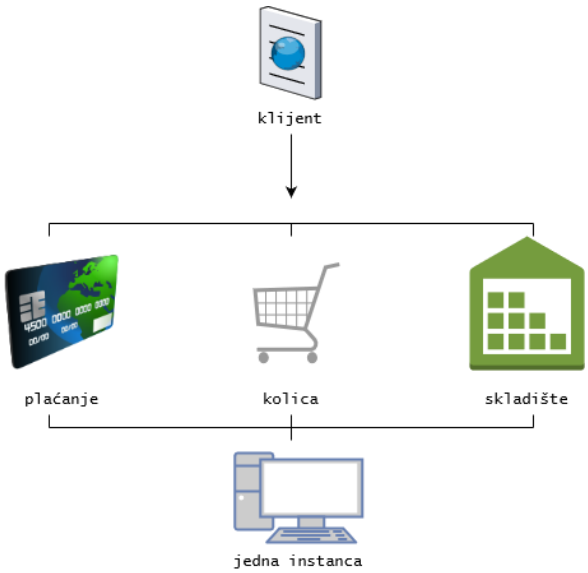
U nastavku poglavlja detaljnije će se objasniti monolitna arhitektura i mikroservisna arhitektura. Ove dvije arhitekture često su korištene prilikom implementacije sotvera. Mnoge tvrtke često imaju projekte u prvotno implementirane koristeći monolitnu arhitekturu, a s njegovim rastom dođu do potrebe za migraciju na arhitekturu mikroservisa. Mnoge prednosti mikroservisne arhitekture su mane monolitne i obrnuto.

5.1. Monolitna arhitektura

Monolitna arhitektura predstavlja tradicionalni model softvera kao jedne velike cjeline. Ukoliko se razvija mala i jednostavna aplikacija monolitna arhitektura može biti najbolji izbor. Monolitna arhitektura pruža mnoge prednosti zato što su aplikacije lake za razvijanje te ih karakterizira centralizirana baza podataka. Sve funkcionalnosti integrirane su unutar istog procesa izvršavanja. Takve aplikacije su neovisne o drugim aplikacijama. Svi dijelovi aplikacije dijele logiku i

funkcionalnosti te su integrirani unutar iste baze koda. Komponente unutar jedne takve cjeline su usko povezane. Pokretanje je uveliko olakšano zato što se pokreće samo jedna datoteka koja predstavlja cijeli sustav. Testiranje i pronalazak grešaka također su olakšani zato što postoji jedinstvena baza koda. Iako pruža mnoge prednosti, monolitna arhitektura ima svojih nedostataka. Jedna greška može zaustaviti rad cijele aplikacije. Implementacija novih funkcionalnosti može biti iznimno kompliciranа. Rastavljanje usko povezanih funkcionalnosti često postaje mukotrpan posao za razvojne programere.

Primjer aplikacije izrađene koristeći monolitnu arhitekturu može biti jednostavna aplikacija online trgovine. Uz potrebno grafičko sučelje, takva aplikacija treba imati mogućnost narudžbe i plaćanja. Na slici 4.1. prikazana je struktura monolitne aplikacije koja ima tri funkcionalnosti unutar jedne cjeline: kupovinu, skladište i plaćanje. Na taj način više različitih poslovnih sposobnosti ovise jedna o drugoj čime se povećava količina potrebnog rada prilikom promjena i implementiranja novih funkcionalnosti. To je samo početak izazova koji proizlaze iz daljnjeg razvoja monolitne aplikacije. Skaliranje monolitne aplikacije može biti izazov jer je potrebno skalirati cijelu aplikaciju, čak i ako je samo jedan dio pod povećanim opterećenjem. Razvoj novih značajki i samo održavanje sustava također često postaju kompleksni zbog usko povezanih komponenata. Dodavanje novih mogućnosti, primjerice dostave kupljenih artikala ili slanja računa putem e-maila, u aplikaciju sa slike 5.1. bio bi naporan posao koji bi zahtijevao refaktoriranje velikih dijelova koda.

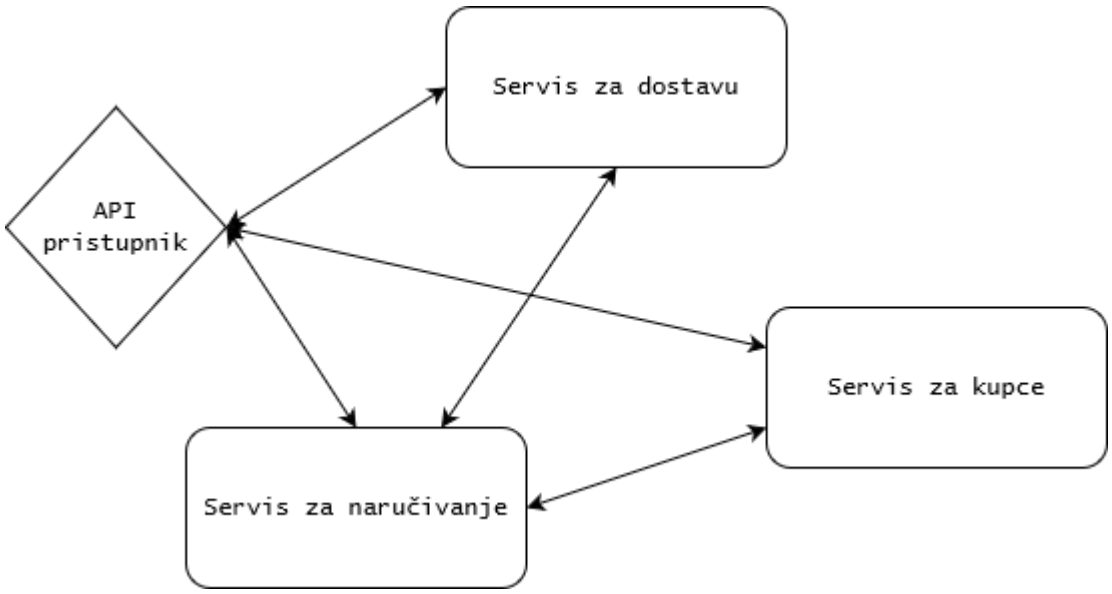


Slika 5.1. Struktura aplikacije implementirane koristeći monolitnu arhitekturu

Međutim, ako se razvija složena aplikacija koja ima visoke zahtjeve za skalabilnost i otpornost, monolitna struktura neće biti odgovarajuće rješenje. Mikroservisna arhitektura ističe se kao idealan način za ispunjavanje takvih zahtjeva.

5.2. Mikroservisna arhitektura

Mikroservisna arhitektura [15] predstavlja pristup izgradnji softvera kao skupa malih, neovisnih i labavo povezanih servisa. Servisi se mogu samostalno pokretati, a međusobno koordinirano surađuju kako bi isporučili cjelokupnu funkcionalnost sustava. Svaki servis unutar sustava fokusiran je na određenu poslovnu sposobnost i radi kao zaseban proces ili spremnik. Mikroservisi međusobno komuniciraju putem definiranih API-ja, obično preko protokola kao što su HTTP/REST ili pomoću sustava za razmjenu poruka. Komunikacija se dijeli na sinkronu i asinkronu te se na razini aplikacije dogovaraju posebni portovi za komunikaciju. Port predstavlja broj kojim je svaki servis definiran unutar mreže. Mikroservisi imaju decentralizirano upravljanje podacima, svaki servis upravlja svojom bazom podataka. Nužno je osigurati konzistentnost podataka prilikom komunikacije između servisa. Jedan od načina jest uvođenje transakcija. Mikroservisna arhitektura pruža veliku fleksibilnost pri izboru tehnologija koje treba koristiti ovisno o potrebama tog servisa. Podržava horizontalnu skalabilnost, dopuštajući zasebnim servisima skaliranje neovisno jedan o drugom. Mikroservisi se često pokreću pomoću prakse kontinuirane integracije i kontinuirane isporuke (engl. *continuous integration/continuous delivery*, CI/CD). Često su korišteni alati za automatizaciju kao Docker i platforme za orkestraciju spremnika kao što je Kubernetes. Labava povezanost povećava toleranciju na pogreške, što znači da kvar na jednom servisu neće ugasiti cijeli sustav. Na slici 5.2. prikazan je sustav rađen u arhitekturi mikroservisa, sličan sustavu sa slike 5.1.



Slika 5.2. Sustav za naručivanje rađen koristeći arhitekturu mikroservisa

Bitno je spomenuti kako mikroservisi nude brojne benefite kada se koriste u pravom kontekstu. Ne preporuča se koristiti arhitekturu mikroservisa za bilo koje programsko rješenje, zato što bespotrebno može uvesti mnoge izazove, primjerice osiguranje konzistentnosti podataka i održavanje pravilne komunikacije između servisa. Već spomenuti primjer sa slike 5.1. gdje je prikazana aplikacija online trgovine odlično bi odgovarala prednostima arhitekture mikroservisa. Svako željeno proširenje poput uvođenja dostave, dodavanje novih načina plaćanja ili implementiranje sustava za povrat robe riješilo bi se dodavanjem novog servisa. Financijski i zdravstveni sustavi, aplikacije za rezerviranje smještaja, *streaming* servisi samo su neki primjeri sustava kojima bi mikroservisna arhitektura bila pogodna.

5.3. Migracija aplikacije s monolita na mikroservis

Kada aplikacija postane prevelika, jedan od načina za daljnji razvoj jest migracija na mikroservisnu arhitekturu. Kako bi se proces migriranja uspješno proveo potrebno je detaljno analizirati module i njihove ovisnosti. Prilikom razdvajanja modula, preporuča se inkrementalno razdvajanje počevši od najmanjeg modula. Time će se postupno povećavati broj malih modula koji će na kraju činiti mikroservisnu aplikaciju. Posebnu pažnju treba predati komunikaciji, jer su monolitni moduli statički povezani i koriste funkcionalnosti referenciranjem na drugi modul. Implementacija komunikacijskih kanala unutar mikroservisa obavit će se preko raznih protokola i slanjem poruka. Potrebno je implementirati mnoštvo ispravno povezanih komunikacijskih kanala. Migriranjem monolitnog

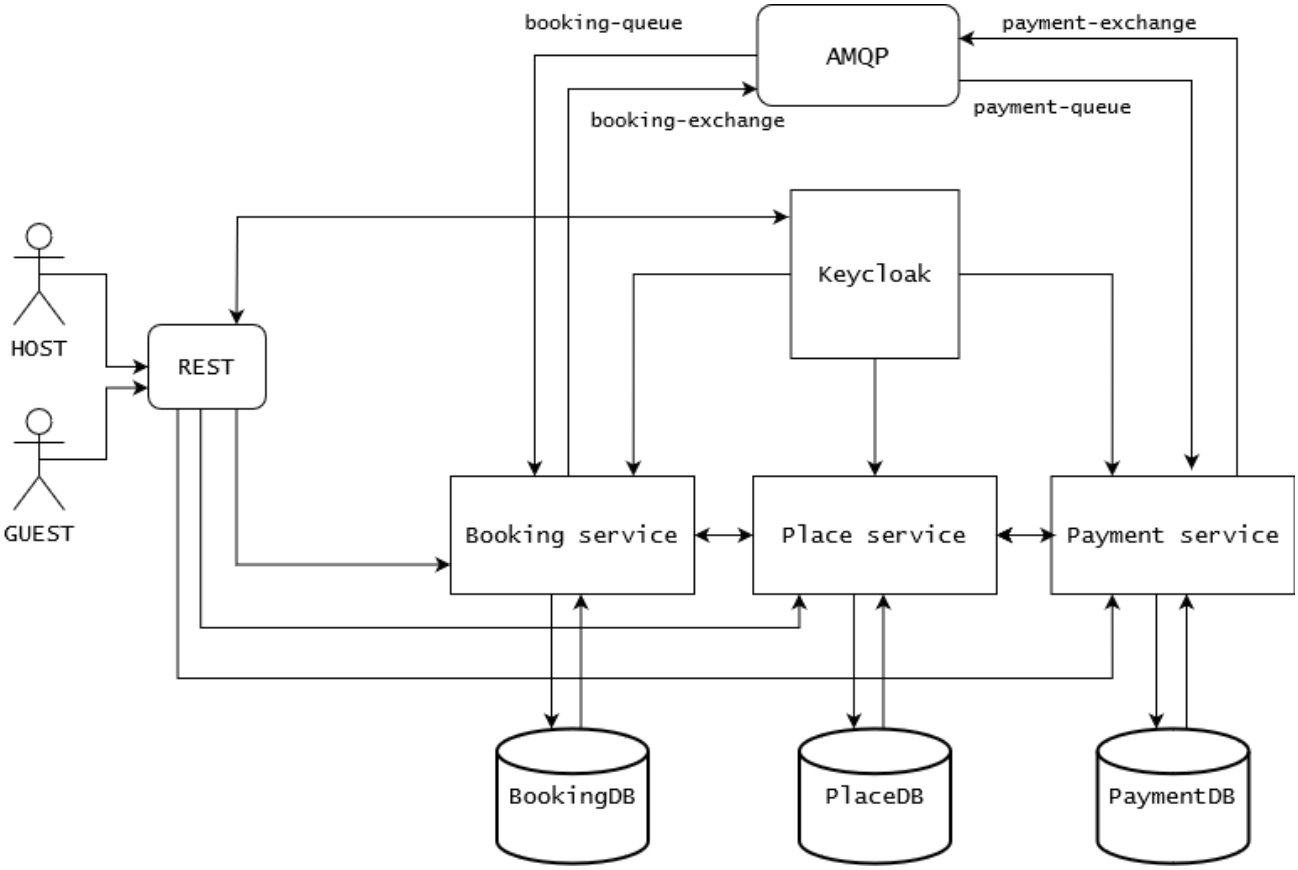
sustava migriraju i baze podataka. Potrebno je odlučiti hoće li se koristiti dijeljena baza podataka ili jedna baza po usluzi. Preferirani način je jedna baza po usluzi, ali ovom implementacijom kompliciraju se transakcije. Gubi se mogućnost referenciranja stranog ključa (engl. *foreign key*) kada se svi atributi ne nalaze u istoj bazi podataka. Konzistentnost baze podataka osigurat će se tako da će se neuspjele operacije spremati u spremnik te će se pokušati izvesti nakon nekog vremenskog perioda. Sustav neko vrijeme neće u biti usklađen, ali će se nakon vremenskog perioda izvršiti potrebne operacije te će se sustav uskladiti.

Uz već spomenuti primjer Airbnb [17], poznat je primjer Netflixa [17] koji je 2009. uočio sve teži razvoj svoje brzo rastuće *streaming* platforme. Monolitna arhitektura nije mogla zadovoljiti povećanu potražnju za uslugama *streaminga*. Tijekom procesa migracije Netflix je morao održavati u funkciji i svoje *cloud-based*[11] poslužitelje i svoje poslužitelje u vlastitim prostorima kako bi osigurao neometan rad. Postali su tehnološki lider te jedna od prvih velikih tvrtki koje su uspješno migrirale s monolitne na arhitekturu mikroservisa zasnovanu na računalnom oblaku. Danas gotovo sve velike tvrtke, primjerice Google, Amazon i IBM koriste arhitekturu zasnovanu na računalnom oblaku ili su u procesu migracije na mikroservis.

[11] cloud-based – zasnovan na računalnom oblaku, predstavlja spremanje podataka na poslužitelju

6. PROGRAMSKO RJEŠENJE

U ovom poglavlju detaljnije će se opisati sama izrada aplikacije za rezerviranje smještaja. Opisat će se pojedinačni dijelovi aplikacije izrađeni po dobivenim zahtjevima već spomenutim tehnologijama. Struktura aplikacije prikazana je na slici 6.1, gdje se može vidjeti kako postoje dva tipa korisnika: domaćin (engl. *host*) i gost (engl. *guest*).



Slika 6.1. Struktura aplikacije *Booking Places*

Svaki korisnik se samostalno povezuje u sustav koristeći Keycloak, koji će osigurati sigurnu prijavu za svaki tip korisnika. Generira se JWT koji sadrži podatke kao što su zaglavlje (engl. *header*), potpis (engl. *signature*), koristan teret (engl. *payload*) te generalne podatke korisnika: korisničko ime, email adresa, ime, prezime i uloge (engl. *roles*) kojima pripada. Unutar sustava postavljene su dvije uloge domaćin i gost. Dekodiranjem JWT-a prikazani su navedeni podaci te se na slici 6.2. vidi dekodirani

koristan teret korisnika s ulogom domaćin. Keycloak šalje dva tokena: pristupni (engl. *access*) token i token za osvježavanje (engl. *refresh token*), čije trajanje se može definirati unutar administratorske konzole Keycloaka. Provjera ispravnosti i dešifriranje tokena obavlja se u pozadinskom sustavu unutar klase *KeycloakFilter*. Ta klasa uzima token iz autorizacijskog zaglavlja, dekodira ga iz Base64 formata te provjerava njegov potpis. Provjeravanjem potpisa osigurava se pristup aplikaciji i njenim metodama samo ovlaštenim korisnicima.


```
{
  "exp": 1707232853,
  "iat": 1707214853,
  "jti": "a8a3e59d-8c77-47c7-b8eb-2d551239840f",
  "iss": "http://localhost:9000/realms/booking-places",
  "aud": "account",
  "sub": "a7b8b6c6-91ed-418e-8ef3-ee3fe2b3e2e9",
  "typ": "Bearer",
  "azp": "microservices",
  "session_state": "80e620ae-58da-4310-9852-
bbd87b3d1ee9",
  "acr": "1",
  "allowed-origins": [
    "/*"
  ],
  "realm_access": {
    "roles": [
      "default-roles-booking-places",
      "offline_access",
      "HOST",
      "uma_authorization"
    ]
  },
  "resource_access": {
    "account": {
      "roles": [
        "manage-account",
        "manage-account-links",
        "view-profile"
      ]
    }
  },
  "scope": "email profile",
  "sid": "80e620ae-58da-4310-9852-bbd87b3d1ee9",
  "email_verified": true,
  "name": "Dario Đurić",
  "preferred_username": "dario.djuric",
  "given_name": "Dario",
  "family_name": "Đurić",
  "email": "dario.djuric@dice.hr"
}
```


Slika 6.2. Dekodirani *Payload* JSON Web Token korisnika 'domaćin'

6.1. Konfiguracija aplikacije za rezerviranje smještaja

Osnovna konfiguracijska datoteka unutar svakog pojedinačnog Maven modula je datoteka *pom.xml*. Također, postoji i *pom.xml* datoteka unutar korijenskog (engl. *root*) modula koja sadrži dodatne konfiguracijske informacije kao što su podaci o organizaciji, razvojnim programerima, podaci o svojstvima, primjerice verzija Jave i Spring Boot-a te podaci o ovisnostima (engl. *dependency*). Maven ovisnost je datoteka ekstenzije ".jar" koju koristi Java aplikacija. Datoteke s ekstenzijom ".jar", punim imenom Java arhiv (engl. *Java ARchive*), su zapakirane datoteke koje sadrže podatke o

Java klasama i resursima. Svaki put kada se u projekt doda nova Maven ovisnost pokrenut će se preuzimanje ".jar" datoteke s centralnog repozitorija te ga spremiti u lokalni Maven repozitorij koji se nalazi u *${USER_HOME}/.m2* mapi.

Jedan od razloga za odabir Spring Boot razvojnog okvira je korištenje automatske konfiguracije Spring Boot aplikacija. Automatska konfiguracija je značajka koja omogućuje automatsko konfiguriranje Spring *beana* na temelju vanjskih ovisnosti. Proces autokonfiguracije unutar modula Spring Boot obavlja tako što pretražuje klase jednog modula, dok za učitavanje *beana* iz vanjskih modula čita jednu od definiranih datoteka:

- *spring.factories* – za Spring Boot verzije manje od 3
- *org.springframework.boot.autoconfigure.AutoConfiguration.imports* – za Spring Boot verzije veće od 3

Te datoteke sadrže klase u *classpathu* koje Spring Boot učitava prilikom pokretanja aplikacije i na temelju njih inicijalizira *bean*.

Unutar aplikacije za rezerviranje smještaja implementirana je klasa *ConfigurationModuleAutoConfiguration* klasa kao globalna konfiguracijska klasa. Sadrži oznaku *@EnableConfigurationProperties* pomoću koje se osigurava registracija klasa čija se vanjska svojstva žele učitati. Na slici 6.3. nalazi se *ConfigurationModuleAutoConfiguration* klasa, u koju su dodane svojstvene (engl. *properties*) klase svakog servisa pojedinačno. Osim svojstvenih klasa, tu su dodane metode za asinkronu komunikaciju i generiranje jedinstvene identifikacijske oznake te klase za vrijeme i preslikavanje poruka. Svaki servis pravi svoju razmjenu (engl. *exchange*) koja se koristi za asinkronu komunikaciju.

```java
@Configuration
@EnableConfigurationProperties({
        ApplicationProperties.class,
        BookingProperties.class,
        DatasourceProperties.class,
        PlaceProperties.class,
        PaymentProperties.class
})
public class ConfigurationModuleAutoConfiguration {


    @Bean
    public TopicExchange bookingExchange() {
        return ExchangeBuilder.topicExchange(BOOKING_EXCHANGE).durable( isDurable: true).build();
    }

    @Bean
    public TopicExchange paymentExchange() {
        return ExchangeBuilder.topicExchange(PAYMENT_EXCHANGE).durable( isDurable: true).build();
    }

    @Bean
    public JavaTimeProvider utcJavaTimeProvider() {
        return new UtcJavaTimeProvider();
    }

    @Bean
    public Validator amqpValidator() { return new OptionalValidatorFactoryBean(); }

    @Bean
    public MessageConverter jsonMessageConverter(ObjectMapper objectMapper) {
        return new Jackson2JsonMessageConverter(objectMapper);
    }

    @Bean
    public IdGenerator idGenerator() { return new JdkIdGenerator(); }
}
```

Slika 6.3. Konfiguracijska klasa *ConfigurationModuleAutoConfiguration*

Osim navedenih mogućnosti za konfiguraciju projekta koriste se i datoteke s ekstenzijom ".yaml". Unutar globalne datoteke *application-config.yaml* postavljena su dodatna svojstva vanjskih servisa kao što su RabbitMQ i Feign, dok svaki zasebni servis ima vlastitu datoteku *application.yaml* u kojoj je definiran port te informacije za povezivanje servisa s bazom podataka. Servisi sadrže dodatne konfiguracijske klase naziva *PersistenceAutoconfiguration* s oznakama *@Configuration* i *@EnableJpaRepositories*. Te klase konfiguriraju ispravan rad sa shemama unutar baze podataka. Eksplicitno su naznačena tri svojstva:

- lokacija entiteta unutar paketa projekta
- lokacija Flyway migracije
- metode odgovorne za rad s bazom podataka

6.2. Flyway

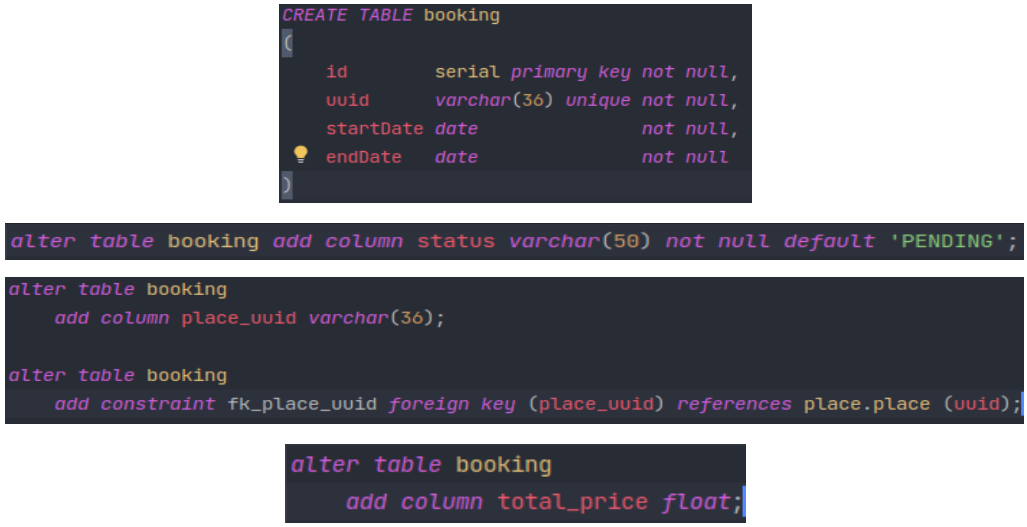
Tijekom razvoja aplikacije sheme unutar baze podataka često zahtijevaju promjene koje uključuju dodavanje, izmjenu ili brisanje stupaca. Svaka promjena u shemi baze podataka mora se sačuvati u obliku datoteke koja se naziva migracija. Svaka migracija ima svoju jedinstvenu verziju koja omogućuje praćenje redoslijeda primjene promjena. Verzioniranje tablica omogućuje praćenje i upravljanje promjenama u strukturi tablica tijekom vremena. Za implementaciju verzioniranja koristio se Flyway koji pomoću ".sql" skripti osigurava da aplikacija sama sebi pripremi bazu. Već izvršene migracije se više ne mogu mijenjati zato što Flyway sprema razne metapodatke[12] u istu bazu u tablicu *flyway_schema_history*. Ključni metapodaci su:

- verzija – prva verzija je 1.0.0.0
- ime skripte – razvojni programer mora pratiti dogovoreni standard imenovanja (engl. *naming convention*) primjerice prva se migracija zove *V1.0.0.0__{naziv}.sql*, a sve sljedeće uvećavaju verziju za 1
- kontrolni zbroj (engl. *checksum*) – numerički identifikator koji se generira na temelju sadržaja podataka, čime omogućuje otkrivanje i sprječavanje slučajnih ili namjernih promjena u ".sql" datotekama

Prilikom pokretanja svakog pojedinog servisa automatski se validiraju sve migracije unutar *classpatha*. Svaki servis ima konfiguracijsku klasu u koju sprema podatke o lokaciji Flyway migracijskih skripti. Ukoliko sustav uoči grešku, servis se neće pokrenuti.

Unutar aplikacije za rezerviranje smještaja napravljeno je sedam Flyway skripti. Broj nije velik zato što su zahtjevi bili dobro raspisani te nije bilo potrebe za velikim promjenama. Na slici 6.4. prikazane su četiri ".sql" skripte korištene u *Booking* servisu. Prva skripta kreira tablicu *booking* s četiri atributa. Druga skripta u postojeću tablicu *booking* dodaje atribut *status* kojeg automatski postavlja na vrijednost PENDING ukoliko se ne postavi drukčije. Treća skripta u postojeću tablicu *booking* dodaje atribut *place_uuid* i strani ključ *fk_place_uuid*, dok je posljednjom skriptom dodana ukupna cijena.

[12] metapodatak – podatak koji opisuje druge podatke, primjerice format, datum izrade, autor



Slika 6.4. Flyway migracije *Booking* servisa

6.3. Svojstvo strukture više modula

Struktura više modula Maven projekta snažna je značajka koja pomaže razvojni programerima olakšati upravljanje ovisnostima i proces izgradnje aplikacije. Struktura više modula predstavlja hijerarhijsku strukturu u kojoj roditeljski (engl. *parent*) modul sadrži jedan ili više modula djece (engl. *child*). Svaki modul predstavlja zaseban Java projekt sa vlastitom *pom.xml* datotekom, koja predstavlja osnovnu konfiguracijsku datoteku unutar svakog pojedinačnog modula. Osim konfiguracije projekta *pom.xml* datoteka sadrži popis korištenih ovisnosti. Moduli djece mogu nasljeđivati ovisnosti, čime je omogućeno centralizirano upravljanje ovisnostima iz roditeljskog modula. Unutar roditeljske *pom.xml* datoteke postavljaju se svojstva, primjerice verzija Java programskog jezika, koja se pomoću *dependencyManagement* oznake nasljeđuje u svim modulima djece. Tako se znatno povećava ponovno korištenje definiranih Java sučelja i promiče princip razdvajanja odgovornosti (engl. *Separation of Concerns)*.

6.4. Servisi aplikacije za rezerviranje smještaja

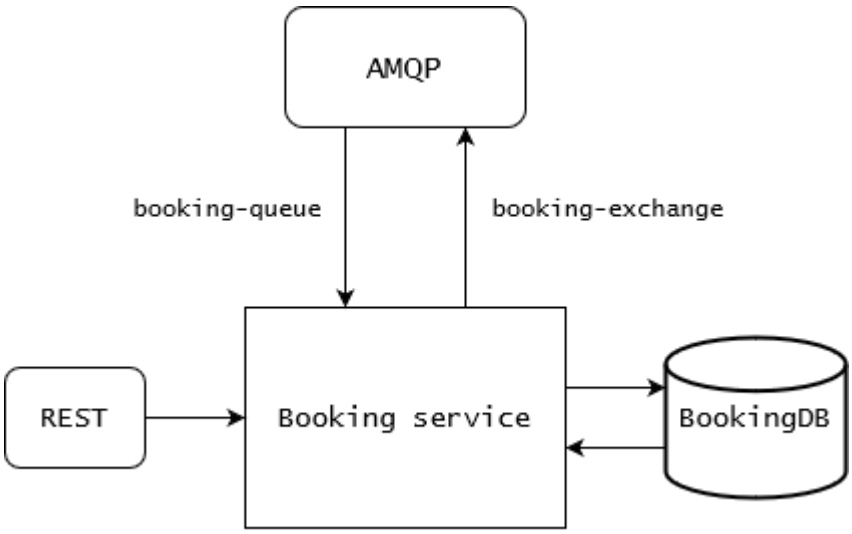
Ključni dio cijele aplikacije, odnosno poslovna logika napravljena je u tri servisa. Ti servisi su labavo povezani kako bi se u budućnosti aplikacija lako mogla proširiti sa novim funkcionalnostima. Svaki servis ima svoj pristup bazi podataka čime se osigurava da samo jedan servis može raditi operacije s njom. Korišten je obrazac baza podataka po servisu (engl. *database per service*) te se zbog

jednostavnije implementacije koristio koncept shema po servisu (engl. *schema per service*) [18]. Međusobno su neovisni u smislu da se svi zasebno mogu pokrenuti, ali to ne znači da mogu pružiti sve funkcionalnosti sami. Neke funkcionalnosti zahtijevaju samo jedan servis, dok neke ne mogu obaviti kompletnu radnju bez drugih. Primjerice, pokretanjem servisa *Place* bez pokretanja ostalih servisa, korisnik ima mogućnost dodavanja novih smještaja. Sustav je kompletan kada su sva tri servisa pokrenuta istovremeno te čine cjelovitu aplikaciju. **FZ-19: Izrada rezervacije, uspješan slučaj** ne može se obaviti ako sva tri servisa nisu pokrenuta. Servis *Place* vraća podatke je li smještaj dostupan u zadano vrijeme i cijenu noćenja, a servis *Payment* će obaviti naplatu ako korisnik ima dovoljno stanje računa o odnosu na cijenu noćenja pomnoženu s brojem noćenja.

Svi servisi pokreću se lokalno te im se pristupa preko definiranih portova na lokaciji *http://localhost/{port}*. U sljedećim potpoglavljima opisat će se pojedinačni servisi te njihova međusobna suradnja.

6.4.1. Booking servis

Booking servis je središnji servis aplikacije za rezerviranje smještaja te se nakon njegove izgradnje (engl. *build*) njemu pristupa lokalno na portu 8082. Za ovaj servis napravljena je shema baze podataka *booking* te tablica *booking* u koju se upisuje nova rezervacija. Komunicira s drugim servisima sinkrono preko definiranih REST API-ja i asinkrono za što su implementirane klase *BookingPublisherImpl* koja objavljuje poruke u *booking-exchange* i klasa *CompleteBookingConsumer* koja prima poruke iz *complete-booking-queue*. Na slici 6.5. prikazana je skica servisa *Booking*. Implementira funkcionalnosti vezane za rezerviranje te za većinu metoda ne zahtijeva odgovor ostalih servisa. Jedina metoda koja zahtijeva ostale servise je metoda *createBooking()*, koja bez ostalih servisa ne može izvršiti rezerviranje smještaja. Prilikom rezervacije, sustav prvo provjerava je li smještaj slobodan u zadanom vremenu, onda ima li željeni korisnik dovoljno stanje računa. Šalje se upit (engl. *request*) prema servisu *Place*, koji vraća odgovor (engl. *response*).



Slika 6.5. Servis *Booking*

Prilikom izvođenja metoda za rad s rezervacijama aplikacija vraća presliku entiteta *BookingEntity* koja se preslikava u *BookingDTO*. *DTO* predstavlja objekt za prijenos podataka (engl. *Data Transfer Object*) koji se koristi prilikom izvođenja metoda. Takav objekt se preslikava (engl. *mapping*) prije spremanja u bazu podataka. Time se dodatno osigurava ispravnost podataka u bazi te se nudi mogućnost skrivanja osjetljivih podataka prilikom komunikacije između servisa.

Unutar *BookingController* klase definirana su četiri endpointa koje pruža *Booking* servis:

- POST *http://localhost/8082/api/v1/booking/{placeUuid}* – koristi se za izradu nove rezervacije, *placeUuid* predstavlja jedinstveni identifikator mjesta kojeg korisnik želi rezervirati
- GET *http://localhost/8082/api/v1/booking* – koristi se za dohvaćanje svih kreiranih rezervacija
- GET *http://localhost/8082/api/v1/booking/{uuid}* – koristi se za dohvaćanje jedne rezervacije, *uuid* predstavlja jedinstveni identifikator rezervacije
- DELETE *http://localhost/8082/api/v1/booking/{uuid}* – koristi se za brisanje rezervacije, *uuid* predstavlja jedinstveni identifikator rezervacije

6.4.2. Place servis

Place servis implementira funkcionalnosti vezane za smještaj. Ovaj servis sadrži shemu baze podataka *place* s dvije tablice: *place* i *place_availability*. Tablica *place* čuva općenite informacije o smještaju, primjerice cijena po noći, dok tablica *place_availability* ima samo informacije o vremenskim periodima u kojima je taj smještaj dostupan za rezerviranje. Sinkrono komunicira s ostalim servisima putem REST API-ja, dok asinkrona komunikacija nije implementirana. Na slici 6.6. prikazana je skica servisa *Place*.

Slika 6.6. *Place* servis

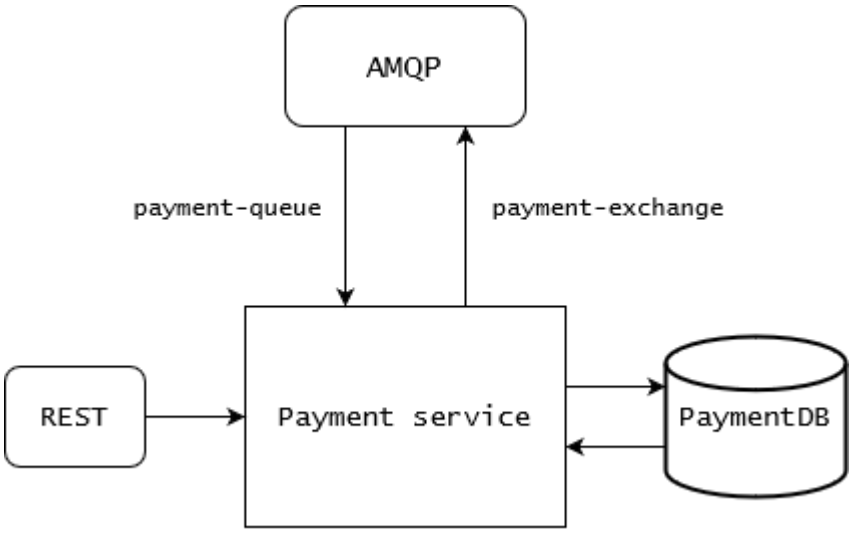
Servis se pokreće lokalno te mu se pristupa preko porta 8084. Prilikom izvođenja metoda za rad sa smještajem koriste se dva entiteta: *PlaceEntity* i *PlaceAvailabilityEntity*, odnosno njihove *DTO* preslike. Kako jedan smještaj može imati više vremenskih blokova kada je dostupan za rezerviranje, *Place* i *PlaceAvailability* su u odnosu jedan naprema više. Unutar *PlaceController* klase definirano je sedam endpointa koje pruža *Place* servis:

- POST *http://localhost/8084/api/v1/places* – koristi se za izradu ponude novog smještaja
- GET *http://localhost/8084/api/v1/places* – koristi se za dohvaćanje svih kreiranih smještaja
- GET *http://localhost/8084/api/v1/places/{uuid}* – koristi se za dohvaćanje jednog smještaja, *uuid* predstavlja jedinstveni identifikator smještaja
- DELETE *http://localhost/8084/api/v1/places/{uuid}* – koristi se za brisanje smještaja, *uuid* predstavlja jedinstveni identifikator smještaja
- PUT *http://localhost/8084/api/v1/places/{uuid}* – koristi se za uređivanje smještaja, *uuid* predstavlja jedinstveni identifikator smještaja
- GET *http://localhost/8084/api/v1/places/{uuid}/availability* – koristi se za dohvaćanje dostupnosti smještaja, *uuid* predstavlja jedinstveni identifikator smještaja, *startDate* predstavlja početni datum, *endDate* predstavlja krajnji datum

- POST *http://localhost/8084/api/v1/places/availability* – koristi se za izradu nove dostupnosti smještaja

6.4.3. Payment servis

Payment servis implementira funkcionalnosti vezane za plaćanje. Komunicira s drugim servisima sinkrono preko definiranih REST API-ja i asinkrono sa servisom *Booking* za što su implementirane klase *PaymentPublisherImpl* koja objavljuje poruke u *payment-exchange* i klasa *ProcessPaymentConsumer* koja prima poruke iz *payment-queue*. Unutar sheme baze podataka *payment* nalazi se tablica *payment* koja čuva informaciju o stanju računa pojedinog korisnika. Na slici 6.7. prikazana je skica servisa *Payment*.



Slika 6.7. *Payment* servis

Servis se pokreće lokalno te mu se pristupa preko porta 8086. Metode unutar servisa rade s *DTO* preslikom entiteta *PaymentEntity*. Unutar *PaymentController* klase definirano je pet endpointa koje pruža *Payment* servis:

- POST *http://localhost/8086/api/v1/payment* – koristi se za izradu novog plaćanja
- GET *http://localhost/8086/api/v1/payment* – koristi se za dohvaćanje svih kreiranih plaćanja
- GET *http://localhost/8086/api/v1/payment/{uuid}* – koristi se za dohvaćanje jednog plaćanja, *uuid* predstavlja jedinstveni identifikator plaćanja

- DELETE *http://localhost/8086/api/v1/payment/{uuid}* – koristi se za brisanje plaćanja, *uuid* predstavlja jedinstveni identifikator plaćanja
- PUT *http://localhost/8086/api/v1/payment/{uuid}* – koristi se za uređivanje plaćanja, *uuid* predstavlja jedinstveni identifikator plaćanja

6.5. REST klijenti

Jedan od implementiranih načina komunikacije između servisa je preko REST API-ja. Svaki servis ima set definiranih REST API-ja koji su opisani u potpoglavlju 6.4. Primjerice, prilikom rezervacije smještaja, *Booking* servis šalje upit na *Place* servis kako bi dobio informaciju o dostupnosti smještaja, zatim ako je smještaj dostupan šalje upit na *Payment* servis kako bi dobio informaciju o stanju računa korisnika. Spring Boot programski okvir pruža mogućnost implementiranja jednostavnog REST klijenta koji se zove Feign client. Feign predstavlja deklarativni klijent, prema [19], koji nudi mogućnost definiranja sučelja čime pojednostavljuje upućivanje HTTP zahtjeva. Definirano sučelje predstavlja udaljeni servis s kojim se želi komunicirati. Zbog multi modularne strukture ne mora se ponovno definirati sučelje, nego se dodaje ovisnost za definicijom u modul unutar kojeg se definira *Feign* klijent. Klijentski modul mora produžiti (engl. *extend*) sučelje od već postojeće kontroler klase i dodati oznaku *@FeignClient*. Kao parametre oznake potrebno je postaviti putanju na servis kako bi znao kojem servisu taj klijent treba pristupiti te ime *Feign* klijenta.

Za potrebe ovog sustava implementirana su dva Feign klijenta: *PaymentFeignRestApiClient* i *PlaceFeignRestApiClient*. Na slici 6.8. prikazane su njihove definicije. Place Feign klijent koristi se kod komunikacije između *Booking* i *Place* servisa za vrijeme rezervacije smještaja. Tijekom upita o raspoloživosti smještaja, odnosno metode *checkAvailability()*, Feign klijent vraća odgovor u obliku *PlaceAvailabilityResponse*. Ukoliko se vrijednost *isAvailable* vrati kao *false* doći će do greške.

```
@FeignClient(value = "placeFeignRestApiClient", url = "http://localhost:8084")
public interface PlaceFeignRestApiClient extends PlaceController {
```

```
@FeignClient(value = "paymentFeignRestApiClient", url = "http://localhost:8086")
public interface PaymentFeignRestApiClient extends PaymentController {
}
```

Slika 6.8. Sučelja Feign clienta

6.6. AMQP

Drugi implementirani način komunikacije predstavlja implementacija AMQP protokola implementirana pomoću *RabbitMQ* posrednika. Omogućuje asinkronu komunikaciju između servisa tako što dopušta slanje poruka u redove čekanja te dohvaćanje poruka iz redova čekanja. Ovaj koncept posebno je koristan u arhitekturi mikroservisa zato što su servisi neovisni jedni o drugima. U slučaju da servis nije upaljen, poruka će čekati u redu i pročitati će se tek nakon što se sustav ponovno pokrene.

Tijekom razvoja ovog sustava implementirana su dva sučelja *BookingPublisher* i *PaymentPublisher* s definiranom metodom *publish()*. Te metode odgovorne su za slanje *BookingMessage* odnosno *PaymentMessage* poruke. Osim navedenih sučelja implementirane su dvije potrošač klase s oznakom *@RabbitListener* i metodom *receive()*. *CompleteBookingConsumer* je klasa koja prima poruke iz *booking-queue* reda čekanja. Red je dodan kao atribut unutar oznake *@RabbitListener* te će se pozvati metoda *receive()* svaki put kada poruka dođe u red čekanja. Metoda *receive()* obrađuje i ispisuje pristiglu *PaymentMessage* poruku. *ProcessPaymentConsumer* klasa implementirana je na isti način. Unutar oznake *@RabbitListener* postavlja se ime reda čekanja *payment-queue* te svaki put kada poruka dođe u red čekanja, metoda *receive()* obrađuje i ispisuje *BookingMessage* poruku.

Konfiguracija *RabbitMQ* posrednika implementirana je u više klasa. Unutar globalne konfiguracijske klase *ConfigurationModuleAutoConfiguration* inicijalizirana su dva *beana* za tematsku razmjenu (engl. *topic exchange*). Korištena je tematska razmjena kako bi se omogućilo povezivanje razmjena s pravilnim usmjerivačkim ključem (engl. *routing key*). Unutar servisa *Booking* i *Payment* implementirane su *BookingAmqpConfig* i *PaymentAmqpConfig* klase koje su odgovorne za inicijalizaciju *beana* povezivanja i *beana* reda čekanja, što je prikazano na slici 6.9.

```java
@Configuration
public class BookingAmqpConfig {

    public static final String BOOKING_QUEUE = "complete-booking-queue";  2 usages

    @Bean
    public Queue completeBookingQueue() { return QueueBuilder.durable(BOOKING_QUEUE).build(); }

    @Bean
    public Binding completeBookingBinding() {
        return BindingBuilder
                .bind(completeBookingQueue())
                .to(new TopicExchange(AmqpConstants.PAYMENT_EXCHANGE))
                .with(AmqpConstants.PAYMENT_ROUTING_KEY);

    }
}
```

Slika 6.9. Klasa *BookingAmqpConfig*

7. VREDNOVANJE PROGRAMSKOG RJEŠENJA

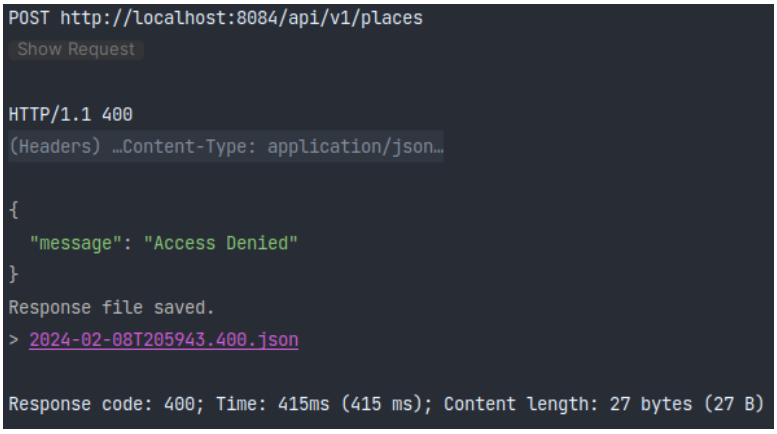
U ovom poglavlju prikazat će se rad aplikacije za rezerviranje smještaja. Pomoću već navedenih funkcionalnih zahtjeva provest će se proces verifikacije. Verificiranjem se dobiva odgovara na pitanje: "Je li sustav ispravno napravljen?". Proces verifikacije izvodit će se pozivanjem metoda iz sva tri servisa te usporedbom dobivenih i očekivanih odgovora. U radu će se prikazati uspješni slučajevi za četiri HTTP metode: POST, GET, PUT i DELETE. Zbog sličnosti u rezultatima neće se prikazati rezultati svih funkcionalnih zahtjeva koji su opisani u poglavlju 3.1, nego samo po jedan rezultat svake metode. Odgovor aplikacije prikazan je JSON porukom i povratnim HTTP statusnim kodom. Osim uspješnih slučajeva prikazat će se jedan slučaj pozivanja metode koristeći korisnika koji nema pravo na poziv te metode te dva neuspješna slučaja.

Prije pokretanja sustava potrebno je unutar Docker spremnika pokrenuti tri servisa: PostgreSQL, Keycloak te RabbitMQ. Koristeći razvojno okruženje IntelliJ navedeni Docker servisi mogu se upaliti na tri načina:

- pokretanjem datoteke *docker-compose.yml*
- iz bloka *Services* pokrenuti spremnik *booking-places*
- preko Terminala koristeći naredbu: `docker compose up`

Tijekom verifikacije sustava sva tri servisa bit će pokrenuta. Metode će se pozivati direktno iz razvojnog okruženja IntelliJ. Redom će se referencirati funkcionalni zahtjevi te će se prikazati rezultat poziva metode.

Prije vrednovanja metoda objašnjenih u potpoglavlju 6.4. vrednovat će se autorizacija. Na slici 7.1. prikazan je pokušaj poziva metode koristeći korisnika koji nema pravo na poziv te metode. Sustav će vratiti poruku "Pristup odbijen". Koristeći korisnika s ulogom gost pokušat će se pozvati metoda za izradu ponude novog smještaja, koja je dozvoljena samo korisnicima s ulogom domaćin.



Slika 7.1. Odgovor aplikacije nakon upita korisnika s pogrešnom ulogom

Pozivanjem POST metode očekuje se izrada novog zapisa u bazu podataka. Na slici 7.2. prikazan je odgovor aplikacije nakon obrađenog zahtjeva za izradu novog smještaja. Pozvana je prva metoda opisana u potpoglavlju 6.4.2. sa svim potrebnim atributima: ime, adresa, broj soba, maksimalni broj gostiju i cijena. Aplikacija automatski generira atribut *uuid* koji služi kao jedinstveni identifikator unutar aplikacije. Kao rezultat vraćena je JSON poruka s unesenim podacima te HTTP statusni kod 201 čime je ispunjen **FZ-1: Izrada ponude smještaja, uspješni slučaj**.

Slika 7.2. Odgovor aplikacije nakon upita za izradu novog smještaja

Osim navedene POST metode unutar aplikacije za rezerviranje smještaja implementirane su još tri POST metode:

- metoda za izradu nove rezervacije

- metoda za izradu novog plaćanja
- metoda za dodavanje dostupnih dana

Pozivanjem GET metode od aplikacije se očekuje dohvaćanje zapisa iz baze podataka. Za svaki servis implementirane su dvije GET metode:

- metoda koja dohvaća sve zapise iz baze podataka
- metoda koja dohvaća određeni zapis po *uuid*-u iz baze podataka

Unutar servisa *Place* implementirana je i GET metoda za dohvaćanje dostupnosti smještaja kojeg se definira koristeći *uuid*. Na slici 7.3. prikazan je odgovor aplikacije na zahtjev za dohvaćanje određene rezervacije iz baze podataka. Kao rezultat vraćena je poruka u JSON formatu koja sadrži informacije o rezervaciji. Servis *Booking* pomoću *uuid*-a dohvaća rezervaciju te vraća HTTP statusni kod 200 čime je ispunjen **FZ-21: Dohvaćanje rezervacije, uspješan slučaj**.

Slika 7.3. Odgovor aplikacije nakon upita za dohvaćanje određene rezervacije

Pozivanjem PUT metode aplikacija dopušta uređivanje postojećeg zapisa unutar baze podataka. Implementirane su dvije PUT metode:

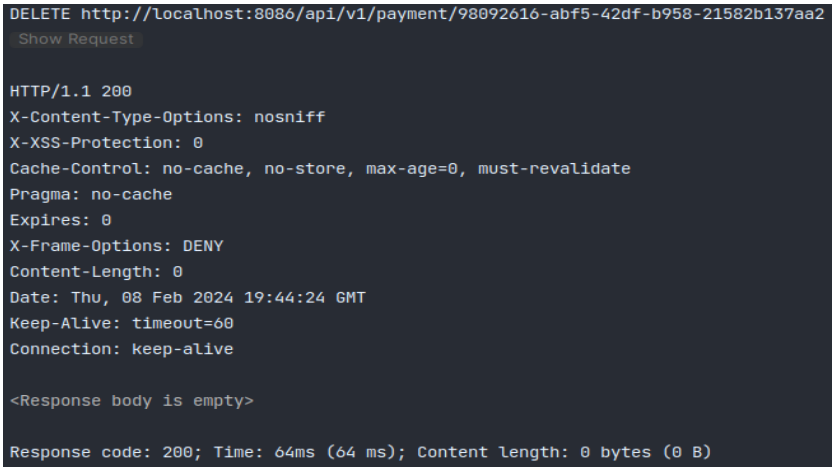
- metoda za uređivanje plaćanja
- metoda za uređivanje smještaja

U pozivu metode predaje se JSON zapis sa svim potrebnim atributima. Na slici 7.4. prikazan je odgovor aplikacije nakon upita za uređivanje smještaja. Servis *Place* uređuje postojeći zapis sa

predanim atributima, sprema ga u bazu podataka te vraća JSON poruku i HTTP statusni kod 200 čime je ispunjen **FZ-5: Uređivanje smještaja, uspješan slučaj**.

Slika 7.4. Odgovor aplikacije nakon upita za uređivanje smještaja

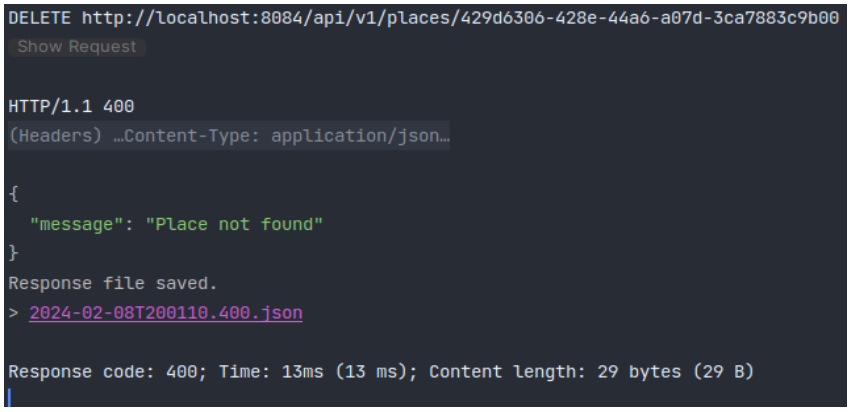
Pozivanjem DELETE metode aplikacija briše definirani zapis iz baze podataka. Svaki servis ima jednu implementiranu DELETE metodu. Na slici 7.5. prikazan je odgovor aplikacije nakon upita za brisanje plaćanja. Aplikacija briše zapis definiran pomoću *uuid*-a te vraća HTTP statusni kod 200 bez ikakve poruke. Time je ispunjen **FZ-17: Brisanje plaćanja, uspješan slučaj**.



Slika 7.5. Odgovor aplikacije nakon upita za brisanje plaćanja

Osim navedenih odgovora na uspješne zahtjeve implementirani su i negativni odgovori aplikacije. Negativni odgovori imaju HTTP statusni kod u vrijednosti između 400 i 599. Na slici 7.6. prikazan je

negativan odgovor aplikacije. Korisnik unosi krivi *uuid* smještaja te aplikacija vraća HTTP statusni kod 400 i poruku "Smještaj nije pronađen." čime je ispunjen **FZ-8: Brisanje smještaja, neuspješan slučaj**.



Slika 7.6. Odgovor aplikacije nakon neispravnog upita za brisanje smještaja

Ukoliko postojeći korisnik pokuša stvoriti novo plaćanje njegov zahtjev će biti odbijen, čime je ispunjen **FZ-14: Izrada plaćanja, neuspješan slučaj**. Sustav pomoću Keycloak autorizacijskog poslužitelja automatski dodjeljuje *uuid* za svakog korisnika. Na slici 7.7. se uz HTTP statusni kod 400 može pročitati i poruka u JSON formatu koja opisuje grešku.

Slika 7.7. Odgovor aplikacije nakon upita postojećeg korisnika za izradu plaćanja

8. ZAKLJUČAK

Razvoj programske podrške korištenjem mikroservisne arhitekture potiče stvaranje modularnih servisa koji se mogu neovisno implementirati i koristiti. Svaki servis odgovoran je za točno jednu poslovnu mogućnost. Servisi skladno komuniciraju putem REST API-ja, promičući labavu međusobnu povezanost i osiguravajući prilagodljivost u skladu s promjenjivim zahtjevima. Implementacija koristeći mikroservisnu arhitekturu potiče inovacije dopuštajući razvojnim timovima zaseban rad na odvojenim funkcionalnostima te brže postavljanje proizvoda na tržište. Time tvrtke uspijevaju učinkovitije odgovoriti na potrebe korisnika i poboljšavaju toleranciju na greške.

Aplikacija izrađena u ovom radu omogućuje domaćinu objavljivanje ponude za smještaj. Domaćin odvojeno unosi i podatke o dostupnosti smještaja te ima mogućnost pregleda trenutno rezerviranih smještaja. Korisnik gost ima mogućnost slanja upita o svim ponuđenim smještajima te njihovim pojedinačnim dostupnostima. Prije nego što se dopusti izrada nove rezervacije smještaja aplikacija za rezerviranje smještaja provjerava dostupnost i stanje računa korisnika gost. Kada su sve provjere uspješno završene korisniku se s njegovog računa naplati izračunati iznos, izrađuje se nova rezervacija te zapisuje u bazu podataka.

Aplikacija izrađena u ovom diplomskom radu sadrži implementirana je s tri servisa *Booking*, *Payment* i *Place*. Servisi su međusobno komuniciraju sinkrono i asinkrono, a interakcija s korisnikom obavlja se pomoću REST API-ja. Svaki servis ima zasebne tablice u shemi unutar jedne zajedničke baze podataka. Aplikacija za rezerviranje smještaja ima prostora za nadogradnje, a zbog izrade koristeći mikroservisnu arhitekturu te se nadogradnje mogu lako implementirati. Neki od takvih primjera nadogradnji su servisi poput onih za slanje računa putem maila. Najveća nadogradnja sustava bila bi implementacija grafičkog sučelja.

Motivaciju za izradu aplikacije koristeći mikroservisnu arhitekturu bila je učenje modernog pristupa razvoju aplikacija koji postaje sve češće korišten. Aplikacija za rezerviranje smještaja činila se kao jednostavan primjer kojim se mogu pokazati brojne prednosti mikroservisne arhitekture kao što su razdvajanje ovisnosti, korištenje različitih tehnologija i neometan razvoj različitih servisa.

LITERATURA

[1] Jeremy H, 4 Microservices Examples: Amazon, Netflix, Uber, and Etsy [online], DreamFactory, dostupno na: https://blog.dreamfactory.com/microservices-examples/ [17.02.2024.]

[2] Airbnb [online], dostupno na: https://hr.airbnb.com/s/Osijek--Hrvatska/homes?tab_id=home_tab&refinement_paths%5B%5D=%2Fhomes&flexible_trip_lengths%5B%5D=one_week&monthly_start_date=2024-03-01&monthly_length=3&monthly_end_date=2024-06-01&price_filter_input_type=0&channel=EXPLORE&query=Osijek%2C%20Hrvatska&place_id=ChIJdYByaajnXEcRlRGkIVZyjFs&date_picker_type=calendar&checkin=2024-03-03&checkout=2024-03-10&source=structured_search_input_header&search_type=autocomplete_click [12.02.2024.]

[3] Booking [online], dostupno na: https://www.booking.com/searchresults.hr.html?ss=Osijek%252C+Osje%C4%8Dko-baranjska+%C5%BEupanija%252C+Croatia [12.02.2024.]

[4] Chisel glossary, Agile & Development: What Is Acceptance Criteria and How to Write It? [online], Chisel Labs, Inc., dostupno na: https://chisellabs.com/glossary/what-is-acceptance-criteria/ [26.09.2023.]

[5] Spring [online], dostupno na: https://spring.io/ [10.09.2023.]

[6] Spring Boot [online], dostupno na: https://spring.io/projects/spring-boot [12.09.2023.]

[7] Apache Maven Project [online], dostupno na: https://maven.apache.org/ [10.09.2023.]

[8] PostgreSQL [online], dostupno na: https://www.postgresql.org/ [10.09.2023.]

[9] Docker [online], dostupno na: https://www.docker.com/ [10.09.2023.]

[10] Keycloak [online], dostupno na: https://www.keycloak.org/ [10.09.2023.]

[11] Stian Thorgersen, Keycloak Intro [online], YouTube, dostupno na: https://www.youtube.com/watch?v=duawSV69LDI [10.09.2023.]

[12] RabbitMQ [online], dostupno na: https://www.rabbitmq.com/ [10.09.2023.]

[13] Gabor Olah, An introduction to RabbitMQ – What is RabbitMQ? [online], Erlang Solutions, dostupno na: https://www.erlang-solutions.com/blog/an-introduction-to-rabbitmq-what-is-rabbitmq/ [10.09.2023.]

[14] REST API [online], dostupno na: https://restfulapi.net/ [10.09.2023.]

[15] Chris Richardson - Microservices Patterns: With examples in Java, Manning, 2019.

[16] Saga Pattern in Microservices [online], Baeldung, dostupno na: https://www.baeldung.com/cs/saga-pattern-microservices [10.09.2023.]

[17] Jessica Tai, The Human Side of Airbnb's Microservice Architecture [online], InfoQ, dostupno na: https://www.infoq.com/presentations/airbnb-culture-soa/ [27.09.2023.]

[18] Ketan Varshneya, Understanding design of microservices architecture at Netflix [online], Techahead, dostupno na: https://www.techaheadcorp.com/blog/design-of-microservices-architecture-at-netflix/ [27.09.2023.]

[19] Chris Richardson, Pattern: Database per service [online], Microservices Architecture, dostupno na: https://microservices.io/patterns/data/database-per-service.html [27.09.2023.]

[20] Spring Cloud OpenFeign [online], Spring, dostupno na: https://docs.spring.io/spring-cloud-openfeign/docs/current/reference/html/ [27.09.2023.]

SAŽETAK

Mikroservisna arhitektura je jedna od modernih i popularnih arhitektura softvera. Često je korištena jer omogućuje razvoj aplikacije kao grupe raspodijeljenih servisa koji zajedničkom suradnjom pružaju kompletnu uslugu koju nudi cijela aplikacija. Neki od primjera aplikacija koje se zasnivaju na mikroservisnoj arhitekturi su i aplikacije za rezerviranje smještaja, poput Airbnb i Booking.com. U svrhu prikaza prednosti navedene arhitekture, ali i kako bi se predstavili izazovi u izradi iste u ovom diplomskom radu izrađena je jednostavna aplikacija za rezervaciju smještaja. U tu su svrhu detaljno definirani zahtjevi na aplikaciju te njezina strukture. Aplikacija je izrađena u Java programskom jeziku upotrebom programskog okvira Spring Boot. Također, korišten je alat za upravljanje projektima Maven kako bi se dodatno pojednostavio proces izrade aplikacije. Komunikacija između servisa obavlja se sinkrono putem definiranih REST API-ja te asinkrono putem brokera RabbitMQ. Brigu o korisnicima vodi autorizacijski poslužitelj Keycloak koji osigurava pristup aplikaciji upotrebom JSON Web Tokena. Baza podataka napravljena je pomoću PostgreSQL sustava za upravljanje bazama podataka, dok je za verzioniranje i migracije odgovoran alat Flyway. Koristeći Docker spremnike pokreću se sustavi RabbitMQ, Keycloak i PostgreSQL. Aplikacija je vrednovana pozivanjem implementiranih metoda i time je pokazano kako aplikacija ispunjava funkcionalne zahtjeve.

Ključne riječi: mikroservisna arhitektura, konfiguracija sustava, međuservisna komunikacija, rezervacija smještaja, Spring Boot

ABSTRACT

Modern microservice architecture and microservice ecosystem

Microservices architecture is a modern and popular software architecture. It is often chosen because it enables the implementation of an application as a collection of distributed services, whose mutual cooperation provides the full service intended by the application. Notable examples of accommodation booking applications developed using microservices architecture include Airbnb and Booking.com. This architecture's advantages, as well as some challenges encountered during its development, are highlighted in a master's thesis, for which an accommodation booking application was developed. Detailed requirements for this application were established, covering both its structure and functional needs. The application was developed in the Java programming language, utilizing the Spring Boot framework. The Maven automation tool was selected to simplify the application building process. Communication between services is facilitated synchronously through REST API calls and asynchronously via the RabbitMQ broker. User management is handled by Keycloak, which employs JSON Web Tokens for authorization. The database is managed using the PostgreSQL Database Management System, with Flyway overseeing versioning and migrations. Docker containers are utilized to deploy RabbitMQ, Keycloak, and PostgreSQL. The application was evaluated by testing the implemented methods, confirming that all functional requirements were met.

Keywords: booking accomodation, communication between services, microservices architecture, Spring Boot, system configuration

ŽIVOTOPIS

Dario Đurić rođen je 09. kolovoza 1996. godine u Slavonskom Brodu. Osnovnu školu završava u Orašju u Bosni i Hercegovini uz koju završava i Osnovnu glazbenu školu. Upisuje Opću gimnaziju u Županji nakon čijeg završetka upisuje Elektrotehnički fakultet u Osijeku. Tijekom stručnog studija obavljao je stručnu praksu u tvrtci Atos Convergence. Završetkom upisuje studij Razlikovne obveze nakon čega upisuje sveučilišni diplomski studij Računarstva. Na prvoj godini diplomskog studija zapošljava se u DICE Digital Innovation Center. Za vrijeme studiranja bavio se i volontiranjem. Stekao je izvrsnu razinu poznavanja engleskog jezika kojeg koristi u društvene i profesionalne svrhe osim kojeg poznaje i njemački jezik.