Primjena i implementacija skalabilne mikroservisne arhitekture pomoću programskog okvira Spring Cloud

Sanković, Alen

SVEUČILIŠTE U ZAGREBU

FAKULTET ORGANIZACIJE I INFORMATIKE

V A R A Ž D I N

Alen Sanković

PRIMJENA I IMPLEMENTACIJA SKALABILNE MIKROSERVISNE ARHITEKTURE POMOĆU PROGRAMSKOG OKVIRA SPRING CLOUD

DIPLOMSKI RAD

Varaždin, 2022.

SVEUČILIŠTE U ZAGREBU

FAKULTET ORGANIZACIJE I INFORMATIKE

V A R A Ž D I N

Alen Sanković

Matični broj: 45991/17-R

Studij: Informacijsko i programsko inženjerstvo

PRIMJENA I IMPLEMENTACIJA SKALABILNE MIKROSERVISNE ARHITEKTURE POMOĆU PROGRAMSKOG OKVIRA SPRING CLOUD

DIPLOMSKI RAD

Mentor/Mentorica:

Prof. dr. sc. Kermek Dragutin

Varaždin, rujan 2022.

Alen Sanković

Izjava o izvornosti

Izjavljujem da je moj završni/diplomski rad izvorni rezultat mojeg rada te da se u izradi istoga nisam koristio drugim izvorima osim onima koji su u njemu navedeni. Za izradu rada su korištene etički prikladne i prihvatljive metode i tehnike rada.

Autor/Autorica potvrdio/potvrdila prihvaćanjem odredbi u sustavu FOI-radovi

Sažetak

Ovaj diplomski rad teorijski obrađuje mikroservisnu arhitekturu, uz dodirne točke s ostalim rasprostranjenijim oblicima monolitne i distribuirane arhitekture. Glavni fokus rada stavljen je na opće karakteristike mikroservisne arhitekture, kao trenutno jednog od popularnijih odabira arhitekture sustava. Uz implementaciju mikroservisnog sustava kroz praktični dio, koja upotpunjuje i potkrepljuje informacije iznesene u teorijskom dijelu, obrađene su značajke kao što su balansiranje opterećenja, otkrivanje servisa, API pristupnik, skalabilnosti i ostale najbolje prakse mikroservisne arhitekture. Praktični dio implementira web aplikaciju za upravljanje letovima, koristeći i detaljno obrađujući osnovne koncepte i mogućnosti Spring, točnije Spring Cloud, razvojnog okvira. Objedinjujući teorijski i praktični dio u smislenu cjelinu, istaknute su prednosti i nedostaci mikroservisne arhitekture kao trenda pri razvoju distribuiranih aplikacija.

Ključne riječi: mikroservisna arhitektura; modularnost; Web servisi; Java; Spring programski okvir; Spring Cloud

Sadržaj

1. Uvod

Trendovi, kao u svim aspektima života, postoje i u IT industriji, a ubrzan razvoj iste rezultira njihovom češćom izmjenom. Od starijih monolitnih aplikacija (eng. legacy) koje u današnje vrijeme izazivaju strah i trepet kod mladih programera, razvoj softvera krenuo je prema distribuiranom i modularnom pristupu, neovisno o razini aplikacije i sustava o kojoj se govori. Kao takav, rođen je novi pojam i ideja o mikroservisnom dizajnu arhitekture.

Ovaj rad bavi se upravo arhitekturom sustava, gdje naglasak stavlja na mikroservisnu arhitekturu, kao relativno svjež i novi pojam u svijetu razvoja arhitekture softvera. Prvi dio rada, koji predstavlja teorijski dio, kreće od općih koncepata arhitekture softvera i pristupa (metodologije) razvoja softvera, kako bi se uočilo da postoji velik broj faktora koji uvjetuju odabir arhitekture, a time i stvaranje trendova. Prije nego što se prijeđe na srž ovog rada, ukratko su obrađene neke od poznatijih i češće primjenjivih stilova arhitekture, uz nešto veći naglasak na Servisno-orijentiranu arhitekturu, zbog sličnosti s glavnom temom ovog rada, tj. mikroservisnom arhitekturom.

Nakon stilova arhitekture vrijednih spomena, slijedi teorijska obrada mikroservisne arhitekture. Od ideje i pojave mikroservisne arhitekture pa sve do njezinih značajki, prednosti i zamki, ovaj rad nastoji približiti mikroservisnu arhitekturu kroz njezina svojstva i prepoznatljivost, radije nego kroz čistu definiciju iste. Uz mikroservisnu arhitekturu također dolaze brojni uzorci dizajna koji se, na različitim razinama i u različitoj mjeri, paralelno primjenjuju te su isti pobliže obrađeni u nastavku.

Kao prijelaz s teorijskog na praktični dio rada, prvo su obrađene tehnologije čije je razumijevanje ključno za shvaćanje implementiranog sustava. Žetoni, OAuth standard i sustavi poruka neki su od primjera korištenih tehnologija. Središte praktičnog dijela jest programski okvir Spring, čije su osnovne karakteristike detaljno obrađene prije prelaska na praktični dio.

Sam praktični dio implementiran je kroz nekoliko mikroservisa i predstavlja web aplikaciju koja služi za upravljanje letovima uz brojne dodatne funkcionalnosti. Implementacija je odrađena u programskom jeziku Java, koristeći već spomenuti Spring programski okvir i ostale tehnologije. Fokus pri obradi implementacije stavljen je na Spring Cloud koji kroz praktične primjere prikazuje poveznicu između implementacije i obrađenog teorijskog sadržaja.

2. Arhitektura softvera

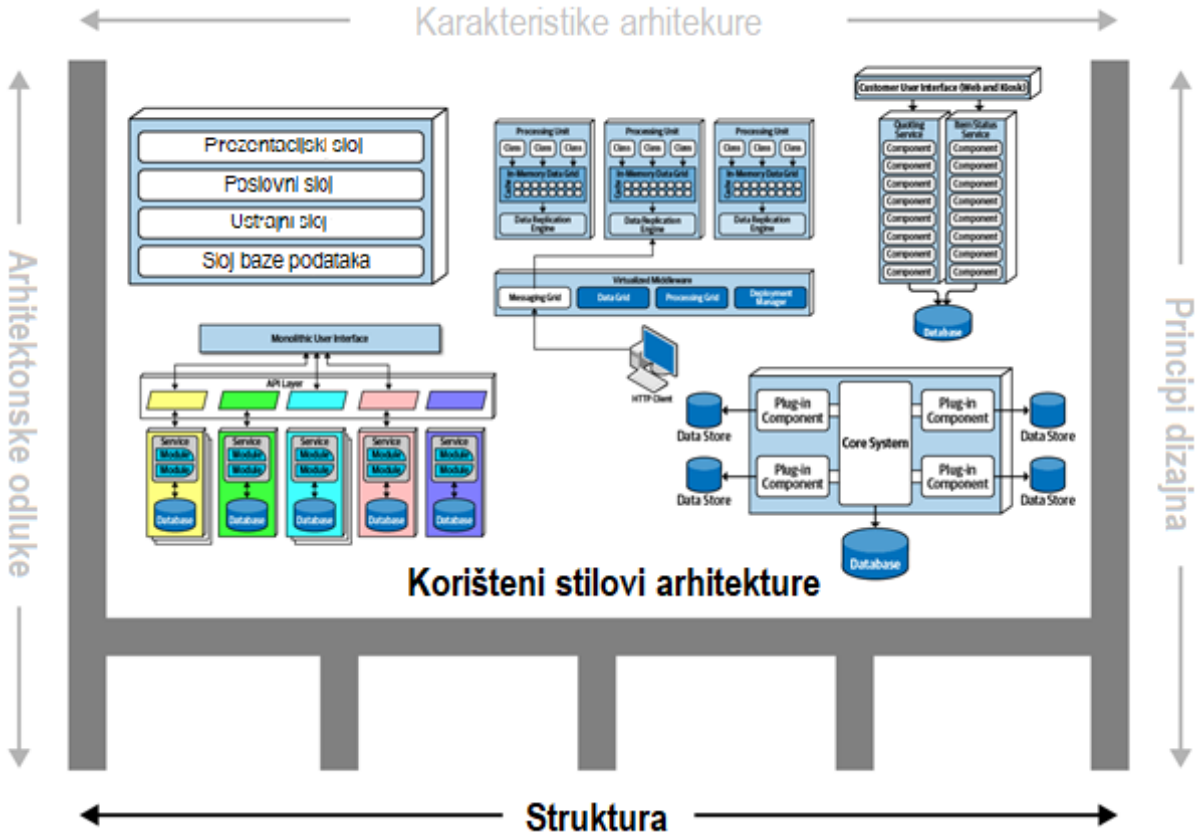
Kada se govori o izgradnji aplikacija, bile one namijenjene za Web, stolna računala, mobilne uređaje ili nešto drugo, vrlo je bitno definirati arhitekturu na kojoj će se one temeljiti. Sama arhitektura postaje sve važnija čim se radi o većem softverskom proizvodu. Osim što se odabirom prave arhitekture za određeni poslovni slučaj osigurava da se izvršavanje softvera usredotoči na ključne elemente (određeni aspekti performanse, dostupnost i sl.), također se olakšava njegovo održavanje i skalabilnost.

Richards M. i Ford N. (2020) navode da se na arhitekturu sustava može gledati kao na njegov predložak ili čak putokaz za razvoj softverskog proizvoda. Iz ove usporedbe postaje još jasnije zašto se arhitektura definira na početku, konkretno u fazi dizajna programskog proizvoda, ako se primjerice govori o vodopadnom modelu.

Na sljedećoj slici vidljive su najčešće karakteristike arhitekture koje se uzimaju u obzir. One su vrlo važne za uspješnost sustava, no čine samo dio (jednu dimenziju) koji se koristi za definiranje arhitekture sustava. Također, ni jedna od karakteristika navedenih na slici nije direktno vezana uz neku funkcionalnost sustava, već bi se moglo reći da su sveprisutne i (pozitivno) utječu na sve aspekte softvera.

Slika 1 Karakteristike arhitekture (Izvor: Richards M. i Ford N., 2020)

Sljedeći aspekt (ili dimenzija), na koju se najčešće misli kada se govori o arhitekturi, je stil arhitekture. Richards M. i Ford N. (2020) ovo još nazivaju strukturom sustava i navode da je samostalna struktura čest odgovor na pitanje o arhitekturi sustava. No, ona samostalno nije dovoljna za potpun opis arhitekture sustava. Strukture o kojima se ovdje govori su npr. mikroservisna, slojevita, microkernel i sl. Ova je podjela vizualno prikazana na sljedećoj slici, a budući da je mikroservisna struktura ono na što je ovaj rad usredotočen, taj će se dio detaljnije obraditi u kasnijem dijelu rada.



Slika 2 Stilovi arhitekture (Izvor:Richards M i Ford N., 2020)

Dvije dimenzije koje će još biti spomenute su arhitektonske odluke i principi dizajna. Arhitektonske odluke služe za definiranje pravila o konstruiranju sustava. Primjer jedne arhitektonske odluke bio bi: u kojem sloju softvera slojevite strukture dodati pristup bazi podataka. Ovisno o slučaju o kojem se radi, pristup može biti potreban u poslovnom sloju, servisnom sloju ili čak u oba sloja odjednom. S druge strane, sigurna arhitektonska odluka je da će se u prezentacijskom sloju zabraniti pristup sloju baze podataka. Dakle, arhitektonskim odlukama se definira ono što jest, a što nije dozvoljeno (ograničenja) prilikom razvoja softvera (Richards M. i Ford N., 2020).

Principi dizajna slični su arhitektonskim odlukama, no razlikuju se po „strogoći“. Principi dizajna smatraju se više smjernicama nego strogim pravilima kojih se timovi moraju pridržavati prilikom razvoja softvera. Oni ne pokrivaju sve moguće slučajeve i probleme koji se mogu javiti, već služe programeru da lakše odabere metodu kojom će riješiti neki problem. Primjer za ovaj aspekt bila bi preporuka korištenja asinkronih poruka unutar mikroservisne arhitekture kako bi se podigla razina performansi. Programerima će u ovom slučaju biti lakše odabrati način komunikacije komponenti (servisa), no jasno je da je realizacija ovisna o konkretnom slučaju (Richards M. i Ford N., 2020).

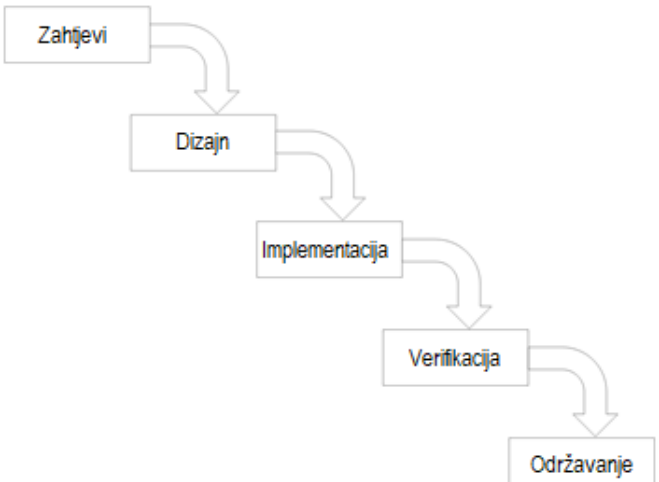
3. Metodologije razvoja softvera

U prethodnom poglavlju bilo je riječi o nekim od osnovnih karakteristika arhitekture softvera. Kako razvoj programskih rješenja i dizajn njihove arhitekture, pogotovo onih zahtjevnijih i opsežnijih, nije nešto što se odrađuje preko noći, važno je spomenuti na koji se način provodi taj postupak. Postupak koji se slijedi prilikom razvoja softverskih proizvoda, a koji se može nazvati i kuharicom, naziva se metodologija razvoja softvera. Zapravo ne postoji samo jedna metodologija, već više njih od kojih je svaka prilagođena određenom scenariju ili projektu na kojem se primjenjuje. Razlog tome je taj što svaki projekt ima drugačije zahtjeve te svaka od ovih metodologija u drugačijoj mjeri stavlja naglasak na određene dijelove. Stoga će u ovom poglavlju biti ukratko obrađene neke od najpoznatijih metodologija za razvoj softvera.

Arhitekt softvera je najčešće onaj tko odlučuje koja će se metodologija primjenjivati na projektu, no to ne znači da ju nitko drugi na projektu ne treba poznavati. Poznavanje ovih metodologija ključan je faktor uspješnosti za svakog člana projektnog tima. Svaka od ovih metodologija ima svoje prednosti i mane, a opća podjela dijeli ih na vodopadnu i agilnu metodologiju. Agilna metodologija sadrži brojne podvrste, ali je i raširenija od vodopadne (Ingeno J., 2018). Nastavak će obuhvatiti glavne karakteristike svake od ovih metodologija.

3.1. Vodopadni model

Prema Casteren W. (2017), početak ove metodologije krenuo je s člankom autora dr. Winstona Roycea, čija je tema bila upravljanje razvojem opširnih i kompleksnih softverskih rješenja. On je na temelju svojih iskustava naveo ono što je smatrao osnovama razvoja softvera. Model koji je on tada prezentirao nije identičan onome koji se primjenjuje danas, no osnovna ideja ostaje ista. Ovaj je model specifičan po tome što su koraci, tj. faze, organizirane



Slika 3 Vodopadni model (Izvor: Ingeno J, 2018)

slijedno. Osim što, kada ga se vizualno prikaže, podsjeća na vodopad, u njegovom osnovnom obliku ne postoji vraćanje na prethodnu fazu jednom kad je ona završila.

Ako se gleda sa strane arhitekture softvera, tada je jasno da su najbitnije početne faze, kao što su to faza definiranja zahtjeva i dizajna. Ovdje je posebno važno da se u tim fazama dobro definira željena arhitektura koja odgovara zahtjevima jer ne postoji vraćanje nakon što se jednom krene na fazu implementacije. Faze verifikacije i održavanja odnose se na životni ciklus softvera nakon što je cijeli implementiran, a uz to služe za potvrdu ispunjavanja svih potrebnih zahtjeva od strane softvera te za njegovo održavanje nakon isporuke (Ingeno J., 2018).

3.2. Agilne metodologije

U sljedećem će poglavlju detaljnije biti obrađene i uspoređene prednosti i mane ovih dvaju pristupa, no bitno je spomenuti da agilne metodologije ciljano nadoknađuju nedostatke tradicionalnog, tj. vodopadnog, pristupa. Razvoj agilnih metodologija došao je od profesionalnih informatičara koji ga temelje na iskustvu iz stvarnog, poslovnog okruženja. Upravo su to razlozi zbog kojih je agilni pristup u današnje vrijeme znatno popularniji i više primjenjiv u praksi. Budući da sam razvoj softvera nije tema rada, neće se ulaziti u detalje agilnih vrsta, no neke poznatije i vrijedne spomena su: Scrum, Kanban i ekstremno programiranje. Svaki od navedenih ima karakteristike koje ga dijele od drugih, no ono što je zajedničko svima jest sposobnost prilagođavanja korisničkim zahtjevima (Ingeno J., 2018).

Agilni manifesto, čiji su autori Fowler i Highsmith, temelj je agilnog pristupa. Manifesto je usredotočen na razvoj „vrijednog“ softvera koji zadovoljava potrebe korisnika. Glavna razlika u odnosu na vodopadni pristup jest taj što se faze u agilnim vrstama provode iterativno. To znači da postoje kratki vremenski ciklusi unutar kojih dio po dio softverskog rješenja prolazi svoj životni ciklus od planiranja do isporuke i verifikacije. Ovo omogućuje da se tijekom razvoja, na temelju povratnih informacija korisnika ili klijenata, određeni dijelovi prilagode te da rezultira njihovim većim zadovoljstvom s krajnjim proizvodom.

Slika 4 Agilni model (Izvor: Jayathilaka C., 2020)

3.3. Usporedba agilnog i vodopadnog pristupa

Nakon navedenih glavnih karakteristika dvaju modela, u ovom će se poglavlju oni usporediti kako bi se stvorila potpuna slika koja prikazuje razloge njihove primjene.

Na strani same aplikacijske domene, agilni pristup radi s malim timovima te za cilj ima brzo podizati vrijednost te reagirati na promjene u zahtjevima. Vodopadni pristup oslanja se na uspješno predviđanje zahtjeva te njihovu stabilnost. Radi se u većim timovima nego kod agilnog pristupa te su projekti uglavnom većeg opsega (Casteren W., 2017). Već je ovdje vidljiv problem vodopadnog pristupa u tome što korisnici „ne mogu opipati“ ono što će dobiti sve do kada to ne bude u potpunosti završeno. A nekome tko se ne bavi informatikom može biti vrlo teško predočiti sliku samo na temelju dokumentacije. Ovakav scenarij često rezultira potrebama za naknadnim promjenama koje su u vodopadnom pristupu iznimno skupe, mjerilo se u vremenskim ili novčanim jedinicama (Ingeno J., 2018).

Tehnički aspekt agilnog pristupa na strani zahtjeva prioritizira tzv. priče (eng. story), koji predstavljaju neformalne slučajeve koji su podložni promjenama. Vodopadni pristup ovaj dio smatra potpuno predvidljivim i formaliziranim. Proces razvoja je već bio spomenut te se kod agilnog pristupa radi o jednostavnom dizajnu koji se izvršava u kratkim ciklusima, a glavna prednost je jeftinije refaktoriranje i prilagođavanje. Vodopadni model ovaj dio ima potpuno suprotno te se kod dizajna ulazi u dubinu, ciklusi su dugi i naknadno refaktoriranje se uglavnom želi izbjeći zbog njegove cijene. Da bi se definirali daljnji zahtjevi, agilni pristup koristi testiranje, dok se kod vodopadnog provode testovi koji su planirani u prethodnoj fazi (Casteren W., 2017).

4. Modularnost

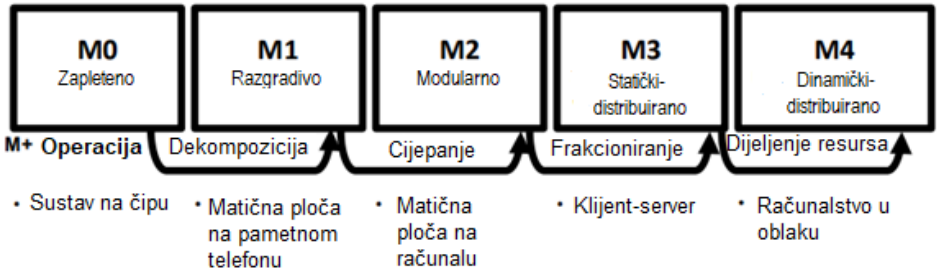
Iako sav softver nije moguće izgraditi koristeći isti stil arhitekture, danas je popularna i smatra se dobrom praksom, izgradnja sustava koji su modularni. Modul se prema Richardsu M. i Fordu N. (2020) smatra standardiziranom neovisnom jedinicom koja može biti korištena u izgradnji veće i kompleksnije cjeline. Ovo omogućuje da se svaki modul u modularnom sustavu tretira kao dio slagalice koji u bilo kojem trenu može biti zamijenjen drugim dijelom (modulom) koji je kompatibilan s postojećim sustavom.

Modul je čest i generički naziv za ono što većina platformi smatra komponentama. One su uglavnom učahurene, što znači da njihov unutrašnji sadržaj nije poznat ostatku sustava, već s njime komuniciraju kroz dobro definirana sučelja. Sučelja osiguravaju da komponenta zadovoljava potrebe sustava, dok se konkretan način implementacije prepušta njima samima. Prema Garlanu D. (2008), stilovi arhitekture koji koriste modularnost najviše su karakteristični za objektno orijentirane programske jezike. Definiranje i podjela dijelova na komponente nije lagan zadatak, no u ovom radu neće se ulaziti u detalje tog postupka. Ipak, vrijedno je spomenuti koje razine modularnosti postoje, kao i opći proces kojim se one mogu postići.

4.1. Metode definiranja distribuirane arhitekture

Prema M. Moslehu i suradnicima (2018), postoje dvije općenite metode za odabir idealne arhitekture. Prva metoda se temelji na skupini kvalitativnih sistemskih intuicija kojima se odabiru moguće alternative dizajna. Druga metoda uzima u obzir fiksan skup alternativa i pokušava odrediti koja od njih je optimalna. Često je potrebno izvršiti više iteracija ovih metoda da bi se donijela odluka. Autori također navode jedinstveni postupak koji obuhvaća ove dvije metode i pridonosi uspješnosti odabiru alternativa. Taj je postupak prikazan na sljedećoj slici.

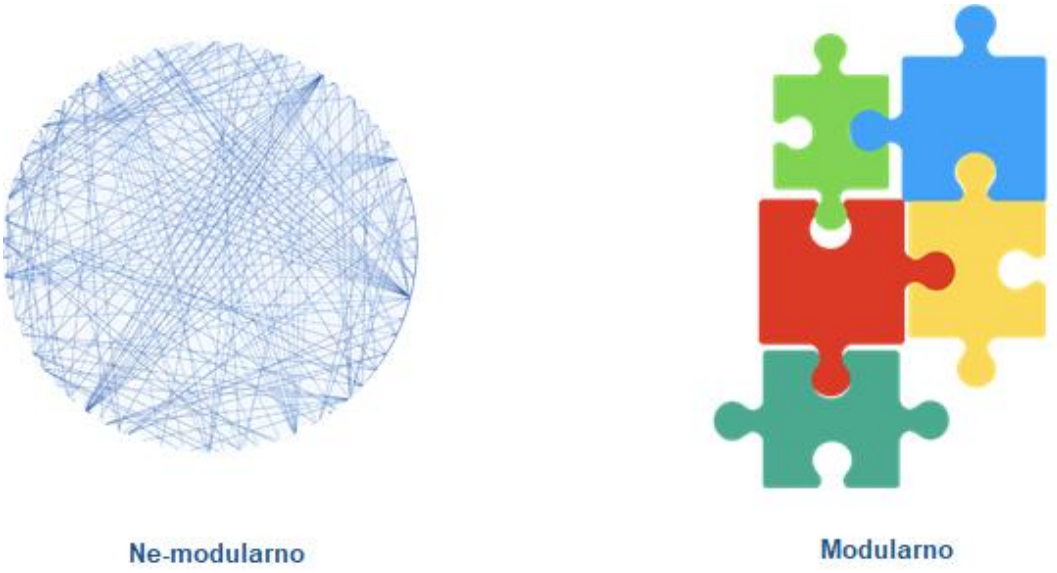
Slika prikazuje pet razina modularnosti koje su označene slovom M i indeksima redom brojevima od 0 do 4. Nulta (najniža, početna) razina predstavlja sustav koji je čvrsto povezan do te mjere da ne postoji način da se razlikuju fizički niti funkcijski dijelovi. Kao primjer je naveden sustav na čipu koji sadrži elektroničke sustave integrirane na jednom mjestu. Vidljivo je da svaka sljedeća razina podiže količinu modularnosti do te mjere da su komponente sustava u potpunosti distribuirane kao što je to riječ kod sustava „u oblaku“.



Slika 5 Razine modularnosti sustava (Izvor: M. Mosleh i suradnici (2018)

4.2. Distribuirana i monolitna arhitektura

Korištenje modularnog pristupa rezultira već spomenutom distribuiranom arhitekturom, dok se s druge strane klasični pristup arhitekturi naziva monolitnom arhitekturom. Kod monolitne arhitekture cijeli sustav spada u jednu teško promjenjivu i čvrsto povezanu cjelinu. Vizualna usporedba je vidljiva na sljedećoj slici. Monolitan pristup možda se ne čini primamljivim izborom pored modularnog pristupa, no aplikacije koje koriste monolitnu arhitekturu vrlo su česte.



Slika 6 Modularna i monolitna arhitektura (Izvor: Khalfallah H. B., 2020)

Vidljivo je da su u ne-modularnim, tj. monolitnim sustavima svi pozivi isprepleteni tako da čine mrežu. Kaže se da je programski kôd čvrsto povezan i to je ono što se uči kao loša praksa. Prema tome, svi dijelovi aplikacije, kao što su sučelje, poslovna logika, autorizacija, pristup bazi podataka i sl., nalaze se na jednom mjestu (Ingeno J., 2018). Ovo dosta komplicira dodavanje i izmjenu postojećih dijelova sustava s obzirom na to da se u tom slučaju treba

napraviti velik broj prilagodbi kako bi sustav i dalje željeno funkcionirao. Na strani modularne ilustracije vidljivo je da je potrebno osigurati da je komponenta kompatibilna samo na određenom dijelu koji je, kao što je već spomenuto, najčešće definiran sučeljem.

4.2.1. Zablude vezane uz distribuiranu arhitekturu

U ovom će se dijelu rada spomenuti neke od zabluda vezanih uz distribuirane sustave te objasniti zašto one ne vrijede.

Glavne prednosti distribuirane arhitekture nad monolitnom su poboljšane performanse, skalabilnost i dostupnost, ali često se zanemaruju ključne negativne strane koje mogu biti velika prepreka pri razvoju i održavanju distribuiranih sustava. Budući da se komunikacija komponenti u distribuiranim arhitekturama često odvija preko mreže, ona se uvelike oslanja na njezinu stabilnost. Prva zabluda do koje dolazi jest da se na mrežu uvijek moguće osloniti. Ne postoji način da se u potpunosti garantira da u jednom trenu neće doći do pada mreže što rezultira problemima u sustavu. Primjerice, jedna komponenta zbog poteškoća u mreži ne dobije odgovor na zahtjev koji je poslala i ne može nastaviti s radom (Richards M. i Ford N., 2020).

Sljedeća zabluda jest da ne postoji latencija prilikom poziva neke metode ili funkcije. Kada se radi o monolitnoj arhitekturi, latencija se mjeri u nanosekundama (ili mikrosekundama) i zanemariva je. Ako ipak govorimo o distribuiranoj arhitekturi, tada latencija (vrijeme potrebno za stizanje odgovora) postiže puno veće vrijednosti što usporava rad sustava i može biti kritično.

Sigurnost je aspekt koji treba imati na umu uvijek kada se radi s mrežom. Distribuirana struktura s raznim komponentama na različitim mjestima može biti zahtjevna za realizaciju sigurne razmjene informacija. Zabluda o kojoj se ovdje radi jest da je mreža uvijek sigurna. Stalnim korištenjem privatnih virtualnih mreža, vatrozida i sl., često se zaboravlja na to da se svaka komponenta (krajnja točka u mreži) treba osigurati, što može rezultirati i usporavanjem rada cijelog sustava. No, jasno je da kod monolitnih sustava ovaj problem nije toliko ozbiljan.

Još neke zablude vrijedne spomena su: smatranje da se topologija mreže nikada ne mijenja te da neće biti potrebno raditi naknadne izmjene u komunikaciji komponenti, da je samo jedan administrator dovoljan da nadzire rad (u monolitnoj arhitekturi to jest slučaj, no ne i u distribuiranoj), zanemarivanje cijene prijenosa podataka preko mreže itd. Prepreka koja se ovdje još spominje je praćenje rada aplikacije koja se znatno komplicira kada se dodaje (i povećava) broj komponenti. Sada se neće ulaziti u detalje, ali će ovaj aspekt biti posebno obrađen u kasnijem dijelu rada budući da je se javlja i vrlo je bitan u mikroservisnoj arhitekturi (Richards M. i Ford N., 2020).

5. Uzorci arhitekture

Primjeri stilova arhitekture, kao i podjela na distribuiranu i monolitnu vrstu, spomenuti su na početku rada. Ono što se u početku nazivalo stilovima zapravo su uzorci dizajna na arhitekturalnoj razini. Uzorci dizajna poznata su i bitna komponenta za svakog programskog inženjera, a isto vrijedi za one na razini arhitekture softvera. Ovi uzorci opisuju opće karakteristike arhitekture i usmjeravaju dizajnere sustava, kao i same razvojne inženjere, na to kako dizajnirati komponente sustava. Osim toga, uzorci su od velike koristi kada je potrebno odlučiti na koji će način komponente sustava komunicirati (Richards M., 2015).

U ovom će poglavlju biti napravljena detaljna podjela poznatih uzoraka arhitekture prema njihovoj vrsti. Također će se dublje ući u svaki stil arhitekture da bi se dodatno pojasnili te kako bi se uočila potreba za istima. Postoji velik broj poznatih uzoraka koji su definirani, primjerice u seriji knjiga „Pattern-Oriented Software Architecture“ (POSA), no ovdje će biti obrađeni samo neki.

Ova podjela napravljena je prema podjeli autora Richards M. i Ford N. (2020). U monolitne stilove svrstavaju se:

- Slojevita arhitektura
- Cjevovodna arhitektura
- Mikrokernel arhitektura

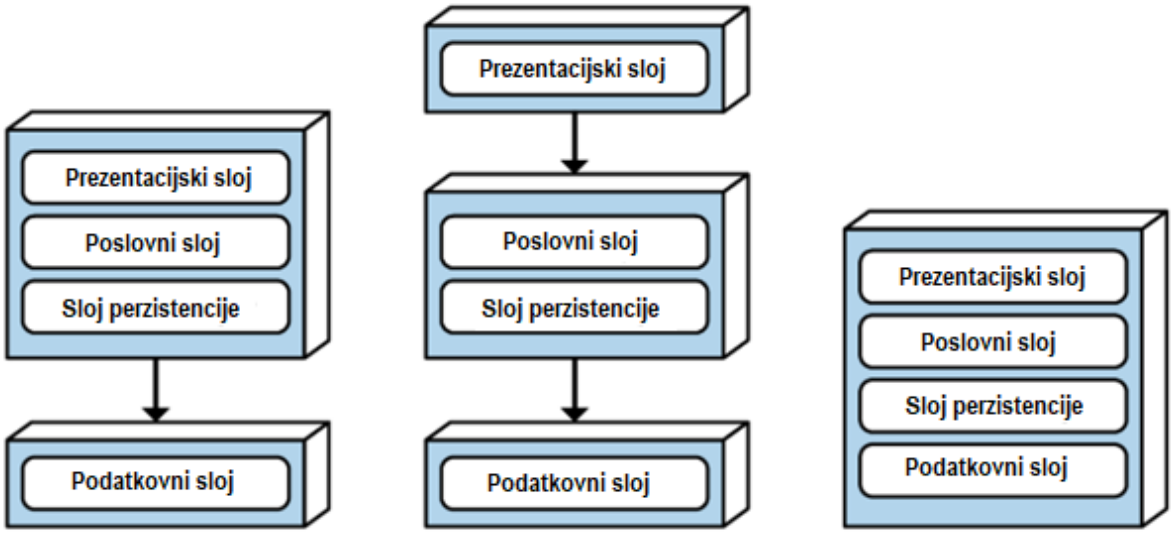
Pod distribuiranu arhitekturu svrstan je ostatak, a to su:

- Arhitektura temeljena na događajima
- Arhitektura temeljena na prostoru
- Mikroservisna arhitektura

5.1. Slojevita arhitektura

Korištenje slojevite arhitekture pri dizajniranju softvera jedan je od najčešćih pristupa ovom problemu. Glavna ideja je podijeliti aplikacijsku strukturu na dijelove, koji predstavljaju horizontalne slojeve (razine) aplikacije, tako da je svaki sloj „iznad“ onog niže razine. Prema tome, postaje jasno da svaki sloj ovisi o slojevima koji su „ispod„ njega, no neovisan je o svim slojevima na višim razinama (Ingeno J., 2018). Vrijedno je spomenuti da se, prema Richardsu M. i Fordu N. (2020), slojeviti stil arhitekture može svrstati u anti-uzorke kao što su „uzorak arhitekture prema implikaciji“ i „uzorak slučajne arhitekture“. Razlog tome je to što, ako tim koji radi na programskom rješenju krene u razvoj bez točne ideje o kojoj se arhitekturi radi, vrlo je velika šansa da će to na kraju biti slojevita arhitektura.

Svaki sloj na određenoj razini općenito ima neku zadaću unutar aplikacije. To, primjerice, može biti prezentacija sadržaja, pristup podacima ili samo obrada poslovne logike. U teoriji, sam uzorak ne definira koje su to točno razine zato što taj dio varira od aplikacije do aplikacije, ovisno o njezinim zahtjevima. Ipak, postoje neki najčešći slojevi koji su de facto postali standard. Spomenuti slojevi su prezentacijski sloj, poslovni sloj, sloj perzistencije i sloj (baze) podataka. U nekim slučajevima može doći do toga da se neki do ovih slojeva implementiraju kao jedan - ako ne postoji posebna potreba za izdvajanjem. Isto tako, ako je to potrebno, na ove spomenute je moguće dodati još slojeva (Richards M., 2015). Na sljedećoj slici vidljive su različite varijacije kako slojevi mogu biti raspoređeni.



Slika 7 Varijacije rasporeda slojevite arhitekture (Izvor: Richards M. i Ford N., 2020)

5.1.1. Zatvoreni i otvoreni slojevi

Bitan koncept u slojevitoj arhitekturi je koncept zatvorenih i otvorenih slojeva. Iz naziva je jasno da jedan sloj može biti zatvoren ili otvoren, no pitanje je što to zapravo znači? Budući da su slojevi naslagani jedan na drugog, zamislimo ih kao stog. Zahtjev koji će doći u aplikaciju, krenut će od najvišeg sloja na stogu (to je uglavnom prezentacijski) i propagirat će se niz stog.

Ako se radi o zatvorenom sloju, tada znači da svi zahtjevi koji prolaze stog, moraju proći kroz svaki sloj tim redom koji je zadan. Primjerice, čest redoslijed slojeva je (od vrha prema dnu) prezentacijski, poslovni i podatkovni. U slučaju da prezentacijski sloj nema potrebe za pozivanjem poslovnog sloja, on ipak mora proslijediti sve zahtjeve poslovnom sloju da bi ih on mogao proslijediti dalje do podatkovnog sloja (Ingeno J., 2018).

Prema Richards M. i Ford N. (2020), „preskakanje“ slojeva je 2000-tih godina bilo nazvano uzorkom brzog čitača (eng. fast-lane reader pattern). Ako su slojevi koje nije moguće preskočiti

bili nazvani zatvorenim, tada je jasno da se u ovom slučaju radi o otvorenim slojevima. Javlja se pitanje koji pristup ovom stilu je bolje koristiti – otvorene ili zatvorene slojeve?

Posljednji koncept koji se javlja, da bi se odgovorilo na postavljeno pitanje, jest koncept slojeva izolacije (eng. layers of isolation). Richards M. (2015) navodi da ovaj koncept govori o tome da su slojevi u slojevitoj arhitekturi međusobno izolirani i ne bi trebali utjecati na rad ostalih slojeva u arhitekturi. Uvjet za realizaciju ovog koncepta jest da svi slojevi budu zatvoreni. Stoga, ako se želi postići izoliranost slojeva, tako da međusobno ne utječu jedan na drugoga i čak do neke mjere omogućuju njihovu zamjenu sa slojem iste funkcije, potrebno je koristiti pristup temeljen na zatvorenim slojevima. Primjer zamjene slojeva bi bio slučaj kada stari prezentacijski sloj temeljen na JavaServer Faces (JSF) zamijenimo prezentacijskim slojem koji se temelji na React.js programskom okviru.

5.1.2. Prednosti i nedostaci slojevite arhitekture

Primjenom zatvorenih slojeva smanjuje se kompleksnost sustava i postiže podjela odgovornosti (eng. Separation of Concerns). Međuovisnost slojeva znatno je smanjena što omogućuje već spomenutu zamjenu različitih implementacija određenog sloja. Zbog manje međuovisnosti, postaje lakše testirati svaki sloj budući da je dovoljno usredotočiti se samo na sloj koji se trenutno testira. Iz istog je razloga veća razina ponovno iskoristivog programskog kôda (ako, primjerice, dvije aplikacije koriste isti sloj pristupa podacima).

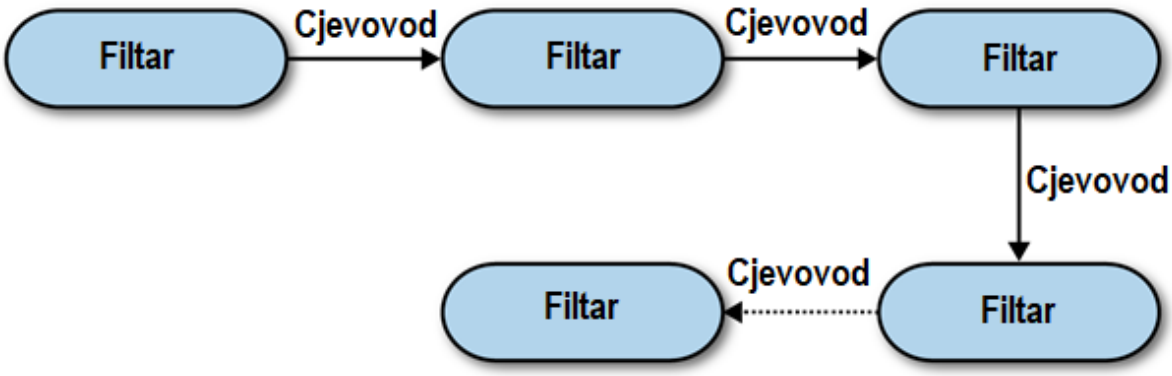
Iako se slojevi mogu dizajnirati kao neovisne cjeline, to često nije slučaj i zbog toga promjene u aplikaciji zahtijevaju promjene i u drugim slojevima. Potrebno je dobro definirati koji dio programskog kôda spada u koji sloj, što nekada nije lak zadatak. Propagiranje zahtjeva kroz svaki sloj može negativno utjecati na performanse aplikacije. Prema svemu navedenom, slojevita arhitektura jest vrlo česta opcija na koju se programski inženjeri odlučuju, no ako se ulazi dublje u analizu performansi i cijene ove arhitekture, potrebno je dobro razmisliti o odabiru pristupa implementaciji (Ingeno J., 2018).

5.2. Cjevovodna arhitektura

Cjevovodna arhitektura temelji se na konceptima filtera koji su povezani cjevovodima. Filteri predstavljaju jedinice koje su međusobno jednosmjerno povezane pomoću cjevovoda, a svaki je filter samostalna jedinica koja uglavnom ne pamti stanja i ima jedan zadatak unutar aplikacije. Nizanjem filtera, tj. zadataka, realizira se funkcionalnost aplikacije. Složeniji zadaci dijele se na manje cjeline tako da svaki dio može biti odrađen od strane jednog filtera.

Način na koji filteri međusobno komuniciraju, tj. izmjenjuju podatke, jest pomoću cjevovoda koji predstavljaju kanale komunikacije u ovoj vrsti arhitekture. Cjevovodi su uglavnom

jednosmjerni i prenose podatke od točke do točke (od filtera do filtera) te mogu prenositi različite vrste podataka. No, kako bi se podigla razina performansi, dobra praksa je prenositi manje količine podataka (Richards M. i Ford N., 2020).



Slika 8 Primjer cjevovodne arhitekture (Izvor: Richards M i Ford N., 2020)

5.2.1. Vrste filtera

U ovom će se dijelu rada ukratko opisati četiri vrste filtera prema Richardsu M. i Fordu N. (2020.), a to su:

- Proizvođač (eng. Producer)
- Pretvarač (eng. Transformer)
- Tester (eng. Tester)
- Potrošač (eng. Consumer)

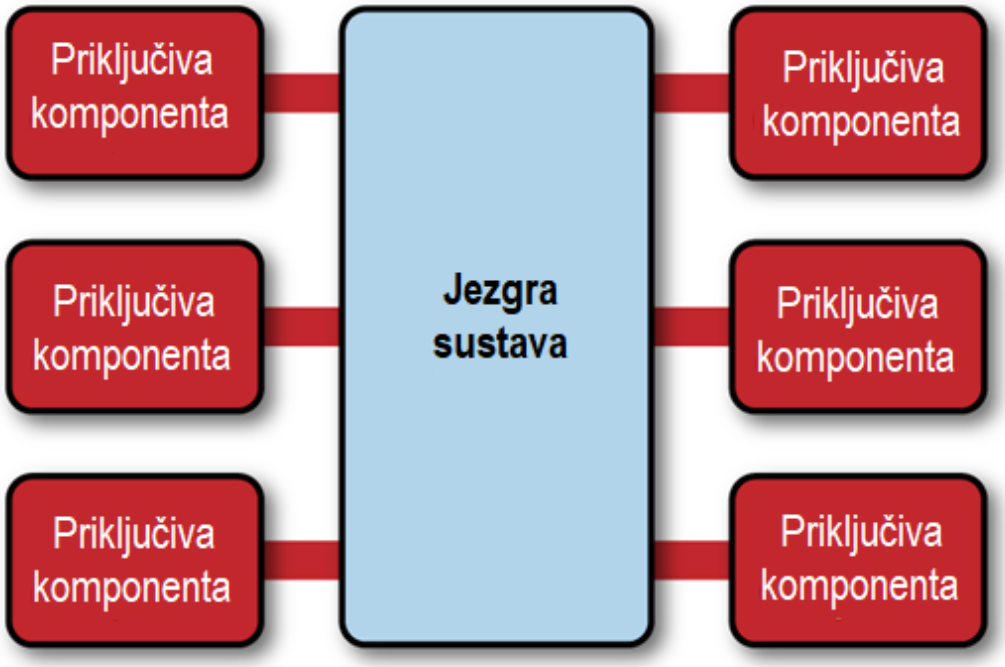
Proizvođač kao filter predstavlja početnu točku u procesu i u pravilu je samo izvor informacija za ostale filtere. Pretvarač je vrsta filtera koji, nakon što zaprimi podatke od cjevovoda, radi neku operaciju nad podacima koja ih može transformirati u drugi oblik te ih zatim prosljeđuje izlaznom cjevovodu. Ovaj filter može se usporediti s funkcijom *map* ako govorimo o programskom jeziku Java i njegovim mogućnostima strujanja podataka (eng. stream).

Tester nad podacima testira zadane kriterije te opcionalno može kreirati neki izlaz, ovisno o rezultatu testa, koji dalje šalje na izlazni cjevovod. Taj je filter ujedno sličan funkciji *reduce*. Zadnja vrsta filtera je potrošač koji, baš suprotno od proizvođača, predstavlja završnu točku u procesu tako da pohranjuje podatke u bazu podataka (ili neko drugo eksterno mjesto za pohranu) ili ih prikazuje korisniku (Richards M. i Ford N., 2020).

5.3. Mikrokernel arhitektura

Prema Richards M. (2015), mikrokernel arhitektura sastoji se od dvije glavne komponente, a to su jezgra i moduli. Jezgra predstavlja minimalnu funkcionalnost koju aplikacija zahtjeva da bi ispunjavala zadatak. Moduli prema tome proširuju osnovnu funkcionalnost tako da se nadodaju na postojeću jezgru. Korištenjem modula omogućeno je lagano proširivanje aplikacije novim funkcionalnostima, ali i njihova izolacija i fleksibilnost.

Mikrokernel je ime dobio po tome što velik broj operacijskih sustava implementira taj arhitektonski uzorak. Osnovna funkcionalnost koju predstavlja jezgra najčešće se temelji na poslovnoj logici koja je najvažnija za određeni poslovni slučaj. Na sljedećoj slici prikazana je mikrokernel arhitektura.



Slika 9 Mikrokernel arhitektura (Izvor: Richards M., 2015)

Postoje varijacije prema kojima se može implementirati jezgra. Jedan od primjera bilo bi korištenje slojevite arhitekture unutar jezgre, odnosno korištenje osnovnih slojeva kao što je npr. prezentacijski. S druge strane, jezgra može sadržavati komponente koje zajedno realiziraju potrebnu funkcionalnost. Moduli se naknadno samo „priključe“ u jezgru, najčešće jednosmjernom komunikacijom kao kod cjevovodne arhitekture te tako proširuju sustav dodatnim procesiranjem, funkcionalnostima i općenito poboljšanjima za cijeli sustav (Richards M. i Ford N., 2020).

5.4. Arhitektura temeljena na događajima

Ingeno J. (2018) događaj definira kao pojavu nečeg važnog za cijeli softver. Najčešće se tu radi o promjeni stanja koje uvjetuje daljnji tijek odvijanja rada aplikacije, a dodavanje narudžbe u košaricu Web trgovine ili izvršavanje plaćanja samo su neki od primjera događaja u Web aplikacijama.

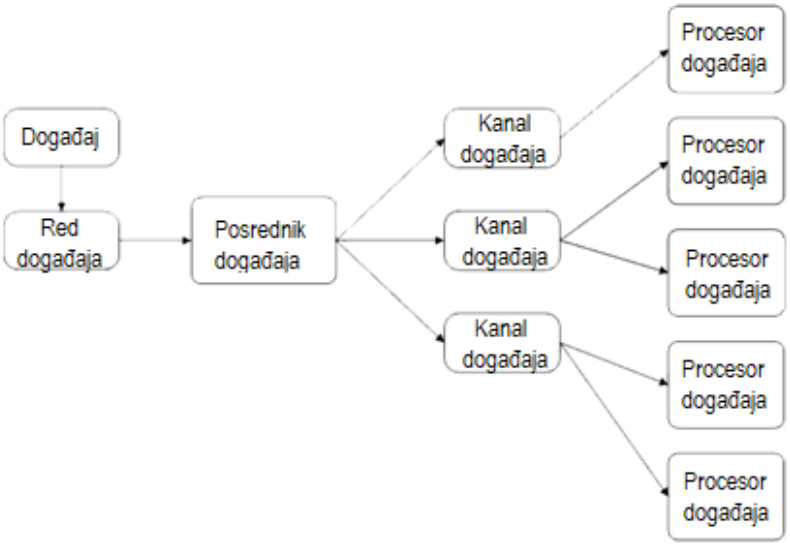
Arhitektura temeljena na događajima predstavlja distribuiranu arhitekturu softvera koja na asinkroni način povezuje više aplikacija (ili komponenti) kako bi one rukovale događajima. S obzirom na to da se ovdje radi o distribuiranoj arhitekturi, njezine komponente su labavo povezane tako da niti jedna komponenta nema informaciju o tome kako je došlo do nekog događaja i koji će događaj nastupiti nakon njezine obrade (Ingeno J., 2018). Ova vrsta arhitekture vrlo je popularna jer se pomoću nje mogu realizirati vrlo skalabilne aplikacije visokih performansi. Osim što se arhitektura temeljena na događajima primjenjuje samostalno, ona može biti korištena i u kombinaciji s drugim oblicima kao što je mikroservisna arhitektura, no o tome će nešto više riječi biti kasnije (Richards M. i Ford N., 2020).

5.4.1. Vrste topologija

U arhitekturi temeljenoj na događajima definirane su dvije vrste topologija koje se koriste pri implementaciji. Radi se o topologiji posrednika (eng. mediator topology) i topologiji brokera (eng. broker topology). Prema Richardsu M. (2015), topologija posrednika najčešće se koristi u slučajevima kada je potrebno orkestrirati izvođenje određenih koraka pri javljanju nekog događaja. Ta orkestracija se odvija preko središnjeg posrednika prema kojem je i dobila ime. S druge strane, topologija brokera koristi se kada se događaji žele povezati u lanac, a da se pritom ne koristi nikakav posrednik. Korištenje posrednika inače omogućuje veću kontrolu nad radnim procesom obrade događaja, dok broker daje visoku razinu odziva i dinamičke kontrole nad događajima (Richards M. i Ford N., 2020). U nastavku će biti navedene osnovne karakteristike svake od tih topologija.

5.4.1.1. Topologija posrednika

Topologija posrednika realizira se kroz red čekanja koji sadrži događaje i posrednika događaja koji prosljeđuje događaje na obradu. Prema tome, događaji se nakon stvaranja stavljaju u red čekanja kojih može biti više (Ingeno J., 2018). Događaji koji se ovdje obrađuju sastoje se od više koraka i ne mogu se provoditi bez neke razine orkestracije. Arhitekture komponenti koje se koriste za realizaciju ove topologije su: redovi čekanja događaja, posrednik događaja, procesori događaja i kanali događaja (komuniciranje posrednika s procesorima) (Richards M., 2015). Ova je struktura jasno prikazana na sljedećoj slici.

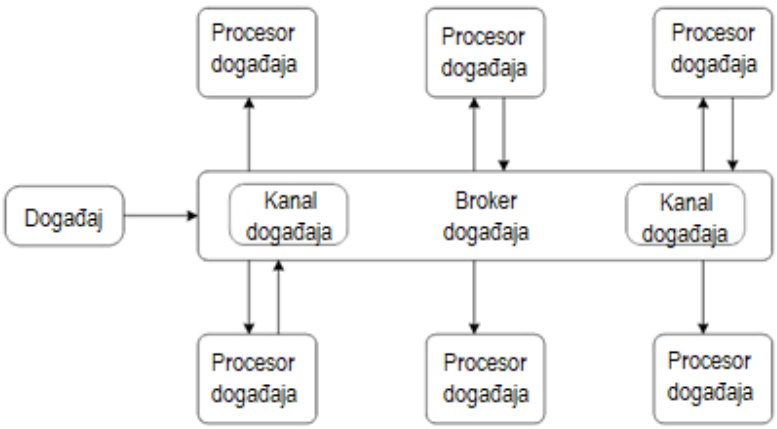


Slika 10 Topologija posrednika (Izvor: Ingeno J., 2018)

5.4.1.2. Topologija brokera

Topologija brokera ne sadrži središnjeg posrednika niti red čekanja, već se koristi broker poruka (eng. message broker) za slanje poruka kroz procesore. Primjer ovakve topologije je RabbitMQ o kojem će više riječi biti kasnije u radu. Ova topologija se koristi kada je tok obrade jednostavan (Richards M. i Ford N., 2020). Broker se još naziva i eng. event bus. Broker sadrži sve kanale koji se koriste za komunikaciju te mogu biti različitih vrsta, a to su: redovi poruka (eng. message queues), teme poruka (eng. message topics) ili kombinacija dviju navedenih (Ingeno J., 2018).

Komponente koje se koriste u ovoj vrsti topologije su: inicijalni događaj, broker događaja, procesor događaja i događaj koji se obrađuje. Struktura ove topologije prikazana je na sljedećoj slici.



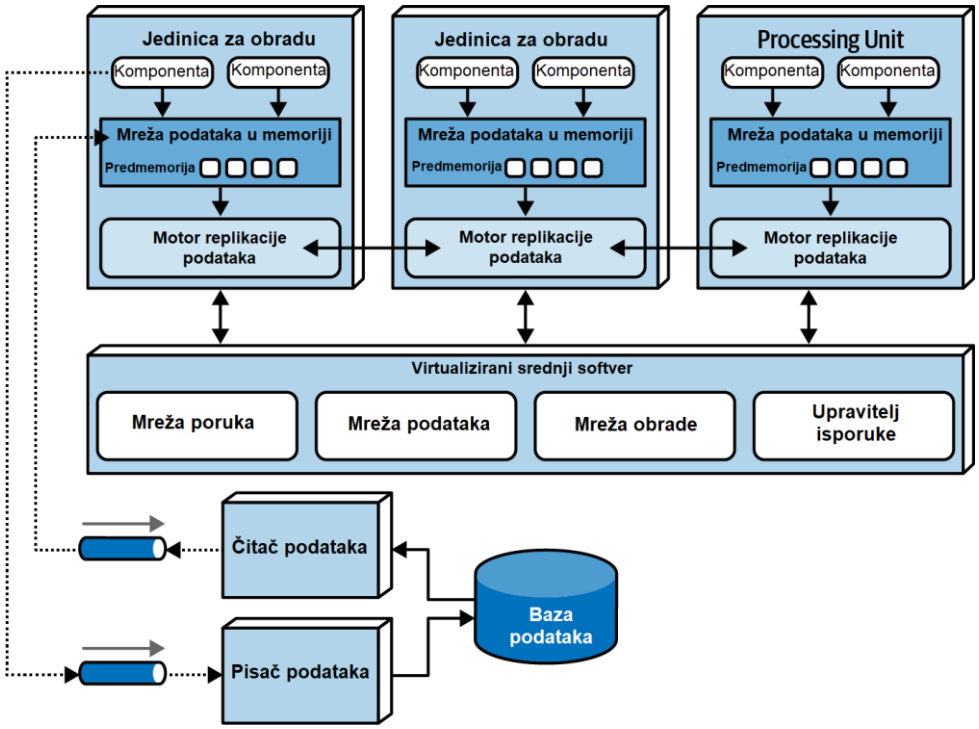
Slika 11 Topologija brokera (Izvor: Ingeno J., 2018)

5.5. Arhitektura temeljena na prostoru

Arhitektura temeljena na prostoru (eng. space-based architecture) još se naziva i uzorkom arhitekture u oblaku (eng. cloud architecture pattern). Cilj ovog uzorka arhitekture je minimizirati čimbenike koji ograničavaju skaliranje aplikacija. Uzorak arhitekture temeljene na prostoru je vjerojatno najmanje poznat od ovih koji su do sada obrađeni, no rješava problem koji je vrlo čest u Web aplikacijama (Richards M., 2015).

Arhitektura temeljena na prostoru dobila je naziv prema konceptu prostora n-torki (eng. tuple space) koji se temelji na ideji distribuirane dijeljene memorije. Uska grla koja se javljaju s povećanjem broja korisnika, a s time i zahtjeva, najčešće se rješavaju skaliranjem Web servera. Ovo rješenje zapravo nije potpuno, već samo odgađa uska grla na nižu razinu sve do kada se ne dođe do potrebe za skaliranjem baze podataka, s čime raste i cijena. Arhitektura temeljena na prostoru cilja upravo na rješavanje problema skalabilnosti, elastičnosti i visoke konkurentnosti. Pokušaj rješavanja ovih problema na razini arhitekture bolja je opcija nego pokušavati skalirati bazu podataka ili drugih dijelova u arhitekturi koja za to nije predviđena (Richards M. i Ford N., 2020).

Glavne komponente ove arhitekture su jedinica za obradu (eng. Processing unit) i virtualizirani srednji softver (eng. virtualized middleware). Jedinica za obradu sadrži komponente ili samo dijelove komponenti aplikacije, a konkretan sadržaj ovisi o vrsti aplikacije. Veće aplikacije rade podjelu jedinica za obradu prema funkcionalnostima aplikacije, dok je za manje nekad dovoljna samo jedna jedinica za obradu. Na sljedećoj slici koja prikazuje ovu



Slika 12 Arhitektura temeljena na prostoru (Izvor: Richards M. i Ford N., 2020)

vrstu arhitekture vidljivo je da jedinica za obradu sadrži mrežu podataka u memoriji (eng. in-memory data grid) i proizvoljnu asinkronu trajnu pohranu za nadilaženje (eng. persistent store for failover). Osim toga, u jednoj jedinici za obradu mehanizam za replikaciju se koristi kako bi se napravila replika promjene koja se dogodila na nekoj drugoj jedinici za obradu. Virtualizirani srednji softver služi za odrađivanje kućanskih poslova i reguliranje komunikacije kroz svoje komponente (Richards M., 2015).

Čest kompromis koji se mora riješiti pri poboljšanju performansi aplikacija jest želi li se dati prioritet prostoru (eng. space) ili vremenu (eng. time) izvršavanja. Prema tome, osim ove arhitekture postoji i arhitektura temeljena na vremenu (eng. time-based architecture), no o njoj se neće detaljnije govoriti.

6. Arhitekture temeljene na servisima

6.1. Servisno-orijentirana arhitektura

Servisno-orijentirana arhitektura (eng. Service-oriented architecture) i mikroservisna arhitektura se obje smatraju arhitekturom temeljenom na servisima. To znači da ovi arhitektonski uzorci imaju veliki naglasak na servisima koji predstavljaju glavnu komponentu koja se koristi pri implementaciji (ne) poslovnih funkcionalnosti (Richards M., 2016). Upravo je to razlog zbog čega će prije poglavlja o mikroservisima biti još nešto riječi i o servisno-orijentiranoj arhitekturi (u nastavku SOA).

Servisom se smatra softver, ali i hardver koji je korišten da pruži podršku automatiziranju poslovnih funkcija, a najčešći razlog korištenja servisa je mogućnost ponovnog iskorištavanja (eng. reusability). Barry D. K. (2012) dijeli servise na atomske (eng. atomic) i složene (eng. composite). Servis se smatra atomskim ako je dobro definirana i samostalna funkcija koja je u mogućnosti izvršavati svoje zadatke bez ovisnosti o drugim (atomskim ili složenim) servisima. Složeni servis, s druge strane, je onaj koji se sastoji od drugih (atomskih ili složenih) servisa. Servisi od kojih se sastoji složeni servis mogu ovisiti o kontekstu (eng. context) ili stanju u kojem se dugi servisi u istom složenom servisu nalaze.

Ono što je zajedničko svim arhitekturama temeljenim na servisima je distribuirana arhitektura. O distribuiranoj arhitekturi je već bilo riječi, no vrijedno je spomenuti da komponente u ovoj arhitekturi komuniciraju i pristupaju jedne drugima koriste protokol za pristupanje udaljenim servisima (eng. remote-access protocol). Primjer ove vrste protokola je danas popularan REST (eng. Representational State Transfer) oblik servisa i danas manje korišteni SOAP (eng. Sumple Object Access Protocol). Neki od protokola temeljeni na porukama su JMS (eng. Java Message Service), AMQP (eng. Advanced Message Queuing Protocol) i MSMQ (eng. Microsoft Message Queuing). Prednosti distribuirane nad monolitnom arhitekturom, kao što su npr. bolja skalabilnost, mogućnost odvajanja komponenti i bolja kontrola, su također već spomenute (Richards M., 2016).

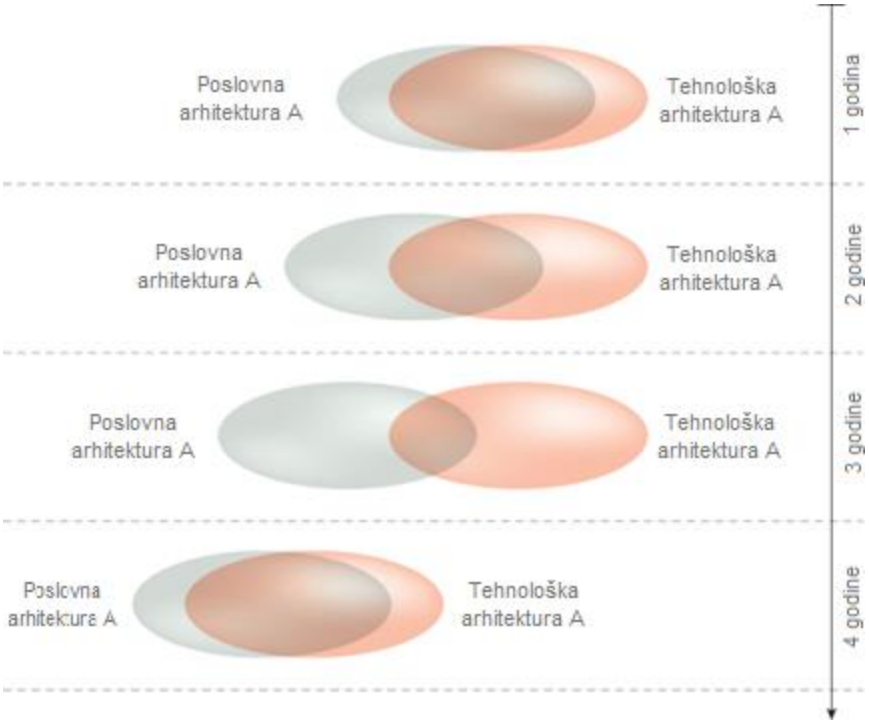
SOA je prema svemu navedenom način dizajniranja, implementiranja i sastavljanja servisa tako da pružaju podršku poslovnim funkcijama. SOA je već ustaljen koncept koji se javio još 1990-ih kod Microsoft DCOM-a i ORB-a (eng. Object Request Broker) koji se temelji na specifikacijama CORBA-e (eng. Common Object Request Broker Architecture). Prva ideja vezana uz SOA arhitekturu je zapravo još starija od toga i seže sve do koncepata skrivanja informacija (eng. information hiding) kojim se kreira sloj sučelja nad postojećim sustavom (Barry D. K. , 2012).

6.1.1.Karakteristike servisno-orijentirane arhitekture

Kako bi se za sustav smatralo da primjenjuje servisno-orijentiranu arhitekturu, potrebno je da ispunjava određene zahtjeve. Ti zahtjevi, ili svojstva, ispunjavaju osnovne zahtjeve za automatizirano rješenje koje se sastoji od servisa i primjenjuje servisno-orijentirane principe dizajna. U nastavku će se ukratko obraditi svaka od četiri karakteristike.

6.1.1.1. Vođena poslovanjem

Ideja dizajniranja arhitekture za određene tehnologije često je vođena potrebom za ispunjavanjem poslovnih zahtjeva. No, ne uzmu li se u obzir dugotrajni ciljevi poduzeća prilikom dizajniranja arhitekture, poslovna arhitektura i arhitektura korištenog softvera s vremenom se počinju udaljavati. Kada je arhitektura tehnologije vođena poslovanjem (eng. business-driven), vizija, ciljevi i potrebe poslovanja postaju središte za dizajn modela arhitekture. Time se postiže puno veća garancija da se arhitektura poslovanja i tehnologije neće s vremenom udaljavati, kao što je to prikazano na sljedećoj slici (Erl T., 2016).



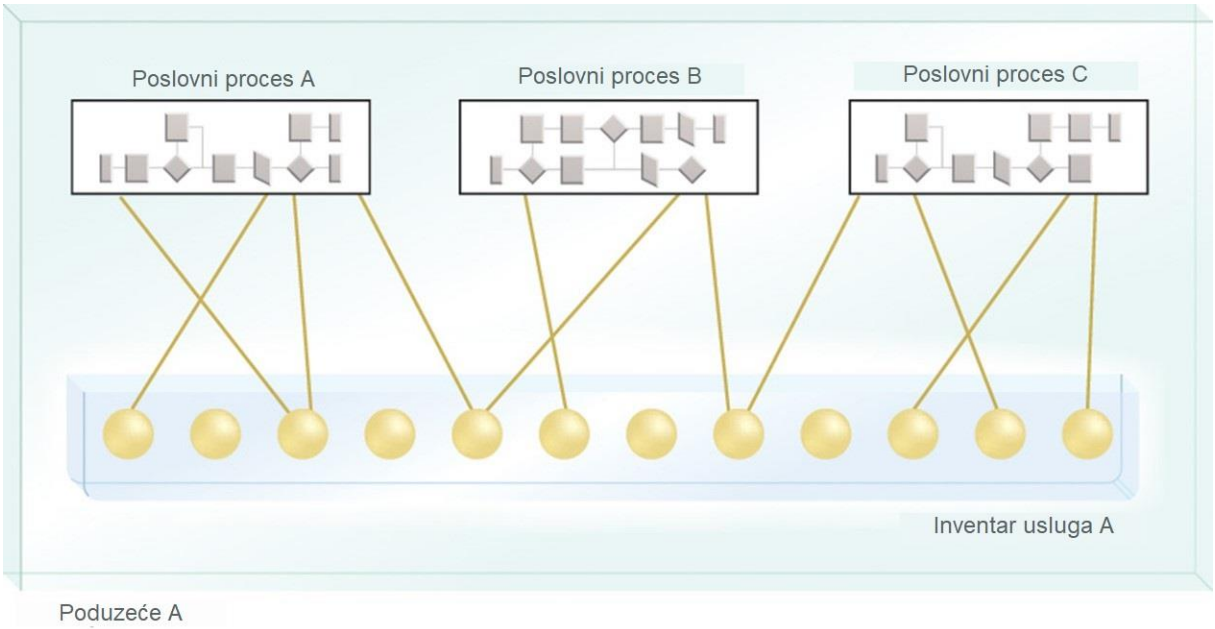
Slika 13 Udaljavanje arhitektura poslovanja i tehnologije (Izvor: Erl T., 2016)

6.1.1.2. Neutralan dobavljač

Mnoga poduzeća imaju klijente oko kojih se temelje njihovi proizvodi. Kada se servisno orijentirana tehnologija dizajnira samo za jednog klijenta (dobavljača), može se dogoditi da to rješenje nenamjerno poprimi određene karakteristike koje su vrlo specifične. Jasno je da to može utjecati na kasniji razvoj i potrebu „odgovora“ na inovacije novih dobavljača. Ovakva rješenja mogu sama ograničiti svoj životni vijek jer ih u jednom trenutku više nije moguće prilagođavati, već ih je potrebno zamijeniti. Cilj bi u ovakvim slučajevima trebao biti dizajnirati arhitekturu koja odgovara potrebama klijenta za kojeg se radi, no koja ostaje neutralna prema ostalim (Erl T., 2016).

6.1.1.3. Usmjereni na poduzetništvo

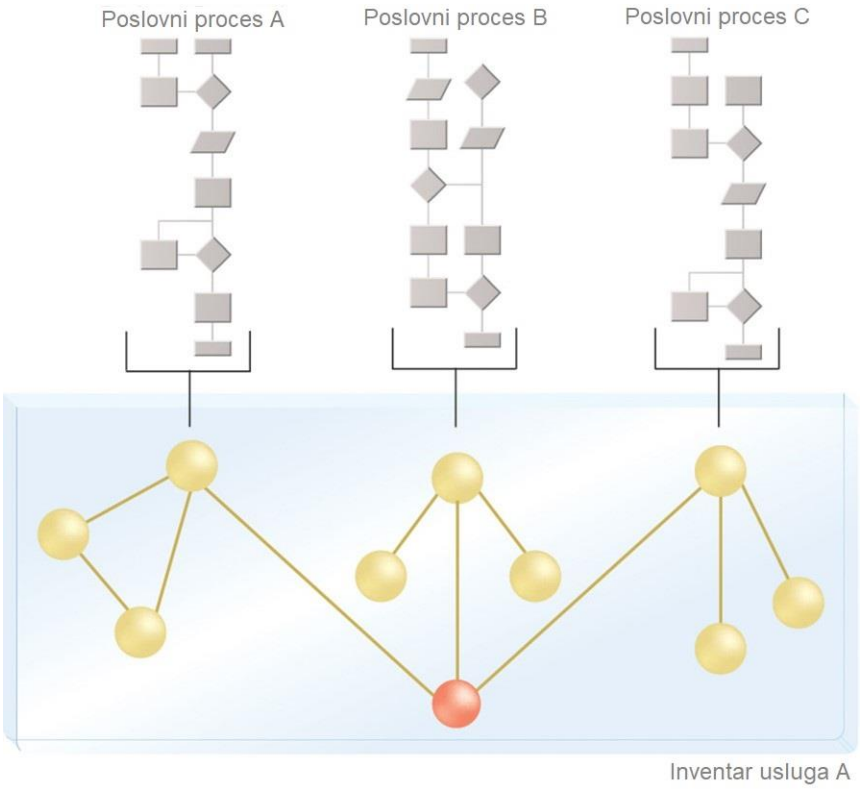
Loše dizajnirane arhitekture predstavljaju opasnost od kreiranja silosa unutar određenih poduzetništva zbog čega se komponente distribuiranih sustava ne mogu ponovo koristiti za rješenja kod drugi poduzetništva. U ovom slučaju potrebno je prilikom dizajniranja arhitekture uzeti u obzir činjenicu da će programsko rješenje biti isporučeno kao skup servisa koji su međusobno dijeljeni. Ovime se postiže mogućnost ponovnog iskorištavanja dijela servisa kao što je to prikazano na sljedećoj slici (Erl T., 2016).



Slika 14 Servisi bez neželjenih silosa (Izvor: Erl T., 2016)

6.1.1.4. Usmjereni na sastav

Osim što je bitno da dijelovi programskog rješenja (servisi) budu ponovno iskoristivi, poželjno je i da budu fleksibilni te da time pružaju mogućnost kombiniranja različitih struktura kako bi se postigli različiti oblici rješenja. Da bi se postiglo navedeno, servisi moraju biti sastavljivi. To znači da mora biti moguće na različite načine kombinirati servise, bez obzira jesu li oni prvobitno zamišljeni da tako funkcioniraju. Da bi se to realiziralo, potrebno je da arhitektura tehnologije bude spremna omogućiti niz jednostavnih i kompleksnih dizajna sastava (Erl T., 2016).



Slika 15 Različiti servisi naknadno zajedno konfigurirani (Izvor: Erl T., 2016)

6.1.2. Najčešći oblici servisno orijentirane arhitekture

Servisno orijentirana arhitektura može biti dizajnirana na različite načine, no prema autoru Erl T. (2016), postoje četiri najčešća oblika koja će biti spomenuta.

Prvi je jednostavan oblik Servisne Arhitekture (eng. Service Architecture) gdje se cijela arhitektura sastoji samo od jednog servisa. Drugi oblik je Arhitektura Kompozicije Servisa (eng. Service Composition Architecture) koji koristi skup servisa koji su međusobno spojeni u jedan složeni servis. Sljedeći oblik je Arhitektura Inventara Servisa (eng. Service Inventory Architecture) koja podržava kolekciju povezanih servisa koji su samostalno standardizirani i

vođeni. Posljednji oblik je Servisno Orijentirana Arhitektura Poduzeća (eng. Service-Oriented Enterprise Architecture) koja predstavlja arhitekturu poduzeća i do neke mjere je orijentirana na servise (Erl T., 2016). Sljedeća slika prikazuje hijerarhiju i međuovisnost navedenih oblika.

Slika 16 Slojevit model SOA oblika (Izvor: Erl T., 2016)

6.1.3. SWOT analiza

Da bi se analizirala SOA, navesti će se karakteristike SWOT (eng. Strenghts, Weaknesses, Opportunities and Threats) analiza prema Schmutz G. et al (2010).

Snage (eng. Strenghts):

- Niski naknadni troškovi
- Fleksibilna arhitektura
- Usklađenost sa standardima
- Podržano od strane svih vodećih softverskih tvrtki

Slabosti (eng. Weaknesses):

- Visoka cijena pokretanja i infrastrukture
- Potrebno je dobro poznavanje SOA strategije i upravljanja

Prilike (eng. Opportunities):

- Samostalni sustavi mogu biti jednostavno implementirani i orkestrirani

Prijetnje (eng. Threats):

- Manjak usredotočenosti na bitne poslovne procese

Vidljivo je da su sve prednosti vezane uz glavne karakteristike distribuiranih stilova arhitekture, ali postoje visoki zahtjevi na strani cijene i kompetencija potrebnih za dizajn i implementaciju.

6.2. Mikroservisna arhitektura

Mikroservisi, točnije mikroservisna arhitektura, u zadnje su vrijeme postali vrlo popularna opcija te su doživjeli značajan zamah. Prema Lewis J. i Fowler M. (2014), naziv „mikroservisna arhitektura“ koristi se za opisivanje načina dizajna softvera koji se sastoji od međusobno neovisnih i isporučivih servisa. Sama mikroservisna arhitektura nema definiciju kojom bi se jednostavno mogla opisati, već se definira nizom karakteristika koje su za nju specifične. Ove se karakteristike tiču organizacije poslovnih mogućnosti, automatizirane isporuke, inteligencije pristupnih točaka i decentralizirane kontrole jezika i podataka. U ovom će se poglavlju dublje ući u svojstva i specifičnosti mikroservisne arhitekture.

6.2.1. Pojava mikroservisa

Budući da ne postoji jednoznačna definicija za opis mikroservisa, korisno je osvrnuti se na ideju koja je pokrenula njihov razvoj. Naziv „mikroservis“ vjerojatno se koristio i prije nego što je definirana mikroservisna arhitektura, no ono na što se danas misli pod tim pojmom bio je rezultat sastanka nekolicine arhitekata softvera. Oni su primijetili neke sličnosti kod načina na koji određeni skup poduzeća gradi svoja programska rješenja. Radilo se o projektima čiji je problem bio to što je konačni sustav jednostavno bio prevelik. Prema tome, postavili su si sljedeće pitanje: „Kako izgraditi sustav koji je lagano izmjenjiv i održiv?“, a prvi naziv bio je mikro aplikacije (eng. micro apps) (Nadareishvili I. et al., 2016).

6.2.1.1. Opći koncepti mikroservisa

Nadareishvili I. et al (2016) su prema navedenom izdvojili tri koncepta koji su bitni za mikroservisni stil. Mikroservisi se smatraju idealnim za velike sustave, a sama veličina sustava je čest problem koji se javlja. Teško je definirati kod koje veličine se sustav smatra velikim, srednjim ili malim te se zbog toga gleda situacija u kojoj sustav postaje prevelik. Sustavi koji izrastu van svojih predviđenih granica predstavljaju problem vršenja naknadnih promjena nad njima. Prema tome, mikroservisi rješavaju problem skaliranja.

Drugi koncept govori da se mikroservisi smatraju orijentiranim ciljevima (eng. goal-oriented). Slučaj koji je bio opisan nije bio usredotočen na konkretno rješenje, već cilj koji se želi postići. Mikroservisna arhitektura prema tome ne definira skup specifičnih praksi, već shvaćanje da se sličan cilj želi postići na određeni način. Primjenom sličnog stila na više

projekata mogle bi se izdvojiti određene karakteristike, ali bitno je razumjeti da je i dalje središte pažnje na rješavanju problema prevelikih sustava.

Zadnji koncept usredotočuje se na zamjenjivost. Mogućnost zamjene jednog mikroservisa drugim mikroservisom predstavlja vrhunac ove arhitekture. Uvijek se težilo mogućnosti jednostavne zamjene komponenata umjesto održavanja onih koje postaju vrlo kompleksne i to je upravo ono što na vrlo jasan način dijeli mikroservise od ostalih stilova arhitekture (Nadareishvili I. et al., 2016).

6.2.2. Svojstva mikroservisa

Ono što softver temeljen na mikroservisima dijeli od monolitne arhitekture jest velik broj laganih servisa koji imaju sljedeća svojstva. Prvo svojstvo je mogućnost neovisne isporuke (eng. independently deployed), o čemu je već bilo riječi. Ti se servisi mogu nadograditi, popravljati ili uklanjati bez utjecaja na ostatak sustava.

U slučaju veće količine prometa, moguće je neovisno skalirati pojedine servise (eng. independently scalable) podizanjem dodatnih instanci te to također nema utjecaja na ostatak aplikacije. Labava povezanost (eng. loosely coupled) govori da degradacija ili bilo kakve promjene nad pojedinim servisom ne bi trebale utjecati na ostatak sustava. Neovisnost je bila spomenuta u kontekstu isporuke i skaliranja i vidljivo je da upravo ona potiče labavu povezanost. Mikroservisi su vođeni domenom (eng. domain-driven) kada su podijeljeni u module i grupe konteksta koje se temelje na poslovnoj domeni kojoj pripadaju. Vrlo je poznat princip jedne odgovornosti (eng. single-responsibility principle) i upravo su mikroservisi jedinice koje su vođene time da su odgovorne samo za jedan poslovni zadatak. Kod definiranja mikroservisa bitno je donijeti odluku o tome koliko će jedan mikroservis biti velik s obzirom na princip jedne odgovornosti (Raje G., 2021).

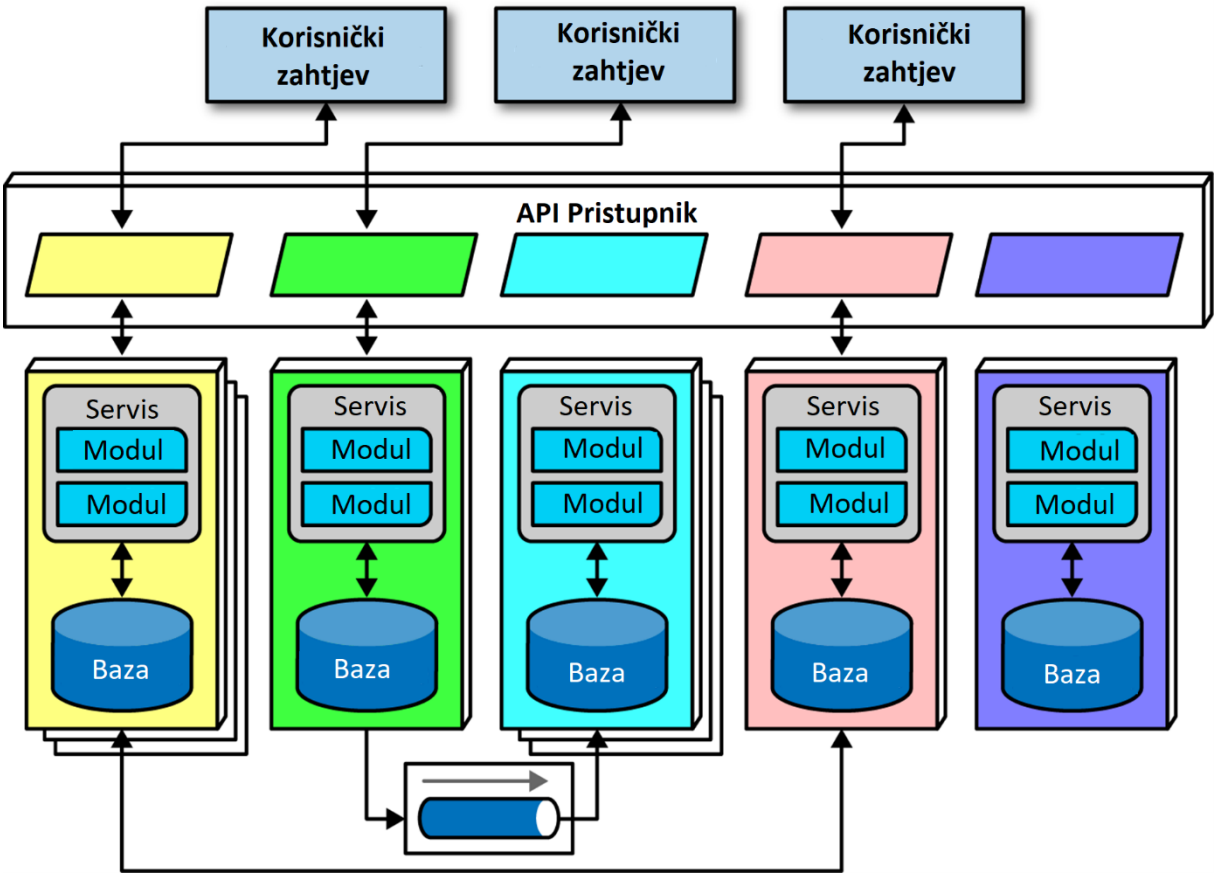
Ograničeni kontekst (eng. bounded context) govori da funkcionalnosti koje su međusobno povezane moraju biti kombinirane u jedinstvenu poslovnu sposobnost, a da svaki mikroservis ima zadatak implementirati tu sposobnost. Time se postiže savršeno preklapanje poslovnih sposobnosti i strukture sustava, zbog čega je lakše pronaći mikroservis u kojem je implementirana određena funkcionalnost i po potrebi učiniti određene promjene na istoj. Iako je o veličini servisa već bilo riječi, bitno je spomenuti da u slučaju kada mikroservis dosegne veličinu koja se smatra prevelikom, potrebno ga je podijeliti u dva ili više manjih servisa. Podjelom na manje servise zadržava se svojstvo granularnosti i usredotočenost na pružanje samo jedne poslovne funkcionalnosti. Komunikacija između mikroservisa odvija se pomoću „lakih“ (eng. lightweight) mehanizama, a najčešće se radi o HTTP protokolu.

Sva ova svojstva vrlo su popularna i smatraju se dobrom praksom što rezultira znatnom količinom refaktoriranja sustava u praksi da bi se prilagodilo ovoj paradigmi. Sam prijelaz na mikroservisnu arhitekturu je dosta „osjetljiva“ tema. S druge strane, postoje poduzeća koja od

samog početka svoj poslovni model pokreću na temelju izgradnje sustava koji prate mikroservisnu paradigmu (Dragoni G. et al., 2017).

6.2.3. Struktura mikroservisa

Topologija ili, kako je ovdje nazvana, struktura mikroservisa u pravilu se sastoji od nekoliko specifičnih dijelova koji će biti jednim dijelom obrađeni u ovom poglavlju, a dijelom prikazani kroz praktični dio rada. Jedan od primjera strukture mikroservisne arhitekture prikazan je na sljedećoj slici.



Slika 17 Struktura mikroservisne arhitekture (Izvor: Richards M. i Ford N., 2020)

6.2.3.1. API pristupnik

Krene li se od vrha analizirati prethodnu sliku, vidljivo je da se korisnikov zahtjev prvo susreće s API pristupnikom. Ovaj sloj najčešće se nalazi između sustava i klijenta, no nije nužan. S ove pozicije u sustavu, API pristupnik može se ponašati kao posrednik (eng. proxy) koji obavlja kućanske poslove ili čak štiti ostatak sustava. Ono za što ga nije preporučljivo koristiti je oblik posrednika ili orkestratora (Richards M. i Ford N., 2020).

API pristupnik može se usporediti s „facade“ uzorkom dizajna. Facade uzorak dizajna pruža vanjskim klijentima unificirani pogled na složenu unutrašnjost sustava, dok API pristupnik pruža unificirani pogled na vanjske resurse prema unutarnjim dijelovima aplikacije. Pojednostavljeno sučelje omogućuje da mikroservisi kroz njega budu pristupačniji, razumljiviji te lakši za testiranje, budući da im se svima može pristupiti „s jednog mjesta“. Ovime se također omogućuje kontrola opsega zahtjeva za vanjske korisnike. Različiti klijenti rade pod različitim okolnostima te se prema tome razlikuju potrebe njihovih zahtjeva. Primjerice, za mobilne klijente mogu biti omogućene tzv. grube ili krupne pristupne točke (eng. coarse-grained) kojima se smanjuje brbljavost (eng. chattiness) mobilnih klijenata (drugim riječima, broj potrebnih slanja zahtjeva). To se postiže tako da se više zahtjeva sklopi unutar jednog koji je optimiziran za željenog klijenta. Time se slanjem jednog zahtjeva, od strane klijenta na jednu točku API pristupnika, postiže više poziva (različitih) pristupnih točaka (više) mikroservisa te se svi odgovori spajaju u jedan. Glavna prednost ovog oblika jest da mobilni uređaj tada iskusi posljedicu latencije mreže samo jednom. S druge strane, postoje detaljno definirane (eng. fine-grained) pristupne točke za stolne klijente koji mogu koristiti mrežu visokih performansi i slati veći broj zahtjeva bez posljedica (Shahir Daya et al, 2016).

Kao takav, API pristupnik predstavlja dobru podlogu za dodavanje programske logike. Ipak, potrebno je imati na umu da, ako se želi držati pravila i filozofije ove vrste arhitekture, API pristupnik nije preporučeno koristiti kao posrednika (eng. mediator) ili orkestratora (eng. orchestrator). Sva programska logika koja se dodaje unutar API pristupnika treba biti unutar ograničenog konteksta (eng. bounded context), a dodavanjem orkestracije ili posredništva to bi se pravilo prekršilo (Richards M. i Ford N., 2020).

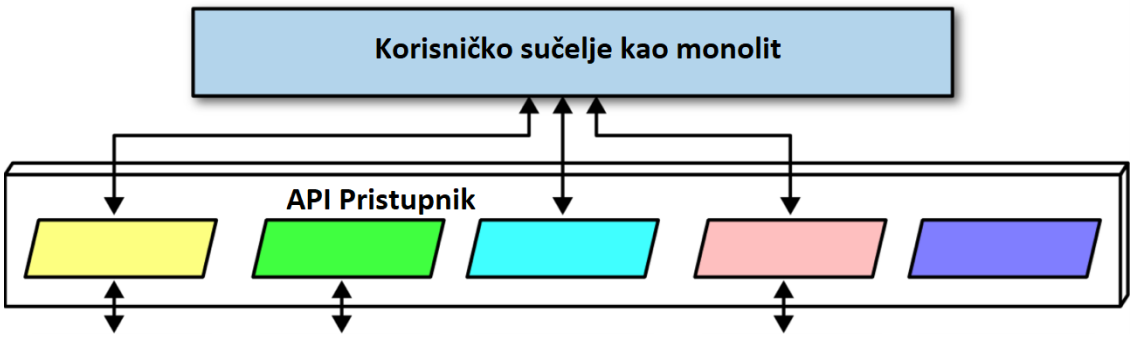
6.2.3.2. Izolacija podataka

Ograničavanje konteksta i opsega pojedinog mikroservisa prepoznaje se i na razini baze podataka. Velika većina arhitektonskih stilova koriste jednu bazu podataka za svu pohranu (eng. persistence). Međutim, kako se kroz mikroservise proteže ideja o izbjegavanju bilo kakvog čvrstog vezivanja (eng. coupling), isto se odnosi i na dijeljenje shema i podataka u bazi koja predstavlja zajedničku točku integracije. Arhitekt sustava mora imati na umu izolaciju podataka te pažljivo modelirati usluge koje nisu uvijek nužno identične entitetima u bazi, budući da se svi podaci protežu kroz više baza podataka. Ključan je faktor definirati kako će se rukovati podacima koji su distribuirani kroz cijelu arhitekturu. Ovaj zadatak može postati poprilično izazovan, ali paralelno otvara vrata novim mogućnostima. Više ne postoji vezanost uz jednu bazu podataka te svaki mikroservis može odabrati alat koji je najprikladniji za njegov scenarij, radilo se o faktoru cijene, vrste pohrane ili nečem sasvim drugom. Svaki od timova koji radi na implementaciji pojedinog mikroservisa dobiva na fleksibilnosti i slobodi prilikom odluke, a da se pri tome ne utječe na druge (Richards M. i Ford N., 2020).

6.2.3.3. Korisničko sučelje

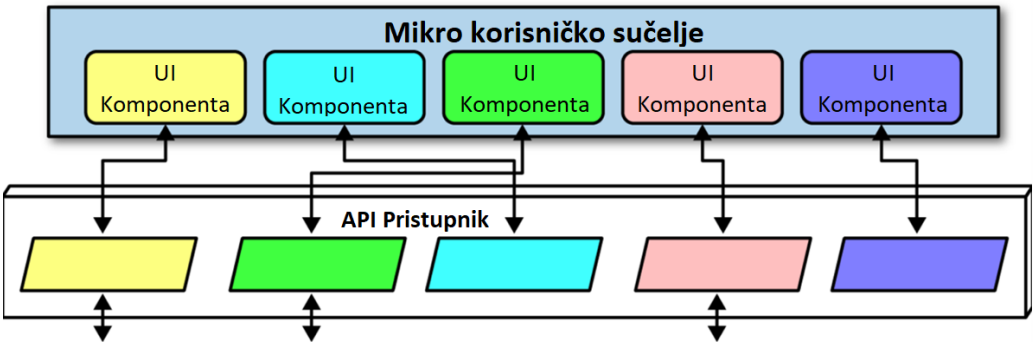
Ovdje će se vratiti na početak strukture te će se prijeći na drugu stranu API pristupnika, a to je korisnička strana od koje svi zahtjevi izvorno potječu. Labava povezanost i modularnost se ovdje ponovo javljaju ako se uzme primjer Web aplikacije s korisničkim sučeljem (eng. Frontend) i pozadinskom logikom (eng. Backend) odvojenom u mikroservisima. Izvorna ideja i vizija mikroservisa uključivale su korisničko sučelje kao dio ograničenog konteksta jer se slijedilo načelo Dizajna usmjerenog na domenu (eng. Domain-Driven Design). U praksi su se ipak počeli javljati problemi praktičnosti i ostalih zahtjeva koji su otežali postizanje tog cilja. Prema tome, postoje dva najčešća oblika korisničkih sučelja kod mikroservisnih aplikacija (Richards M. i Ford N., 2020).

Prvi oblik predstavlja korisničko sučelje kao monolit koji može biti predstavljen kao mobilna, Web ili aplikacija na stolnom računalu. Ovaj monolit tada koristi API pristupnik, kojem šalje zahtjeve te tako komunicira s mikroservisima, kako bi dohvatio potrebne resurse i zadovoljio zahtjeve korisnika. Kao primjer se može uzeti jednostavna aplikacija izgrađena koristeći Javascript Web okvir (eng. framework)



Slika 18 Korisničko sučelje kao monolit (Izvor: Richards M. i Ford N., 2020)

Drugi oblik su takozvana mikro korisnička sučelja (eng. microfrontends). Ovaj pristup koristi komponente na razini korisničkog sučelja. Svaki servis šalje podatke komponenti koja je zadužena za taj dio korisničkog sučelja te korisničko sučelje koordinira rad komponenti. Najčešća implementacija ovog oblika realizirana je kroz React programski okvir, a mogu se koristiti i ostale alternative okvira otvorenog koda (eng. Open source) koji podržavaju ovaj uzorak (Richards M. i Ford N., 2020).



Slika 19 Mikro korisničko sučelje (Izvor: Richards M. i Ford N., 2020)

6.2.4. Usporedba mikroservisne i servisno-orijentirane arhitekture

U prethodnim poglavljima obrađeni su distribuirani stilovi arhitekture kao i servisno-orijentirana arhitektura. Postoji velik broj sličnosti i karakteristika koje se protežu kroz ove pojmove, no i dalje oni ne predstavljaju istu stvar. Da bi se dodatno istaknula razlika između servisno-orijentirane i mikroservisne arhitekture, u ovom će se poglavlju prikazati njihova usporedba. Na prvi je pogled jasno da su oba pristupa usredotočeni na modularnost i rastavljanje velikih sustava na velik broj manjih servisa. No, poznavanje samo ovih karakteristika nije dovoljno za razlikovanje SOA-e i mikroservisa.

6.2.4.1. Komunikacija

Oba pristupa temelje se na servisima koji razmjenjuju podatke preko mreže, a komunikacija može biti sinkrona ili asinkrona. U servisno-orijentiranoj arhitekturi servisi mogu biti rastavljeni do te mjere da je dovoljno poslati informaciju o događaju kao što je primanje nove narudžbe. Svaki servis unutar SOA-e moći će drugačijom logikom obraditi zaprimljeni događaj. Razlog jake razdvojenosti je taj da ni jedan servis ne zna što je okinulo događaj te to što je za dodavanje novih servisa na isto mjesto potrebno samo dodati reakciju na isti događaj (Wolff E., 2016).

6.2.4.2. Orkestracija

Na razini komunikacije nisu bile vidljive velike razlike, no to ne vrijedi i za orkestraciju. Na integracijskoj razini javljaju se razlike između SOA-e i mikroservisa. Ako se govori o SOA, rješenje kojim se realizirala integracija servisa također mora biti zaduženo za njihovu orkestraciju. U mikroservisnoj arhitekturi ne postoji razina inteligencije u integracijskom rješenju, već je komuniciranje s drugim servisima jedini zadatak mikroservisa. SOA koristi orkestraciju da bi dobila na dodatnoj fleksibilnosti pri implementaciji poslovnih procesa, a jedini slučaj u kojem takav pristup može dobro funkcionirati jest kada su servisi i korisnička sučelja stabilna i ne zahtijevaju česte izmjene (Wolff E., 2016).

6.2.4.3. Fleksibilnost

Da bi se postigla potrebna fleksibilnost u mikroservisnoj arhitekturi, pomaže činjenica da svaki mikroservis može vrlo jednostavno biti izmijenjen i stavljen u produkciju. Kada poslovni procesi u SOA nisu dovoljno fleksibilni, SOA forsira promjenu iz servisa u isporučiv monolit. Naglasak mikroservisa je na izolaciji tako da u idealnom slučaju postoji jedan mikroservis koji u potpunosti rukuje komunikacijom s korisnikom i nema potrebe za pozivanjem drugih servisa. Potreba za novim funkcionalnostima je tada ograničena na jedan mikroservis, a poslovna logika kod SOA-e je distribuirana kroz orkestraciju pojedinih servisa (Wolff E., 2016).

6.2.4.4. Razina

Najbitnija razlika javlja se na razini na kojoj se primjenjuje pojedina arhitektura. SOA u obzir uzima cijelo poduzeće i definira kako brojni sustavi komuniciraju unutar njega. Mikroservisi, s druge strane, predstavljaju arhitekturu jednog sustava i alternativa su drugim tehnologijama za modularizaciju. Moguće je implementirati mikroservisni sustav s nekom tehnologijom za modularizaciju i staviti ga u produkciju kao isporučiv monolit bez distribuiranih servisa. SOA se proteže kroz cijeli poslovni sustav poduzeća i ne postoji alternativa distribuiranom pristupu. Prema tome, odluka o primjeni mikroservisne arhitekture može se donositi i na razini jednog projekta, dok SOA utječe na cijelo poduzeće (Wolff E., 2016).

6.2.4.5. Sažeti prikaz odnosa

U sljedećoj tablici nalazi se usporedba odnosa SOA-e i mikroservisne arhitekture:

Tablica 1 Usporedba SOA i mikroservisne arhitekture

Predmet interesa	SOA pristup	Mikroservisni pristup
Glavna ideja	Servisi, dogovor između pružatelja i korisnika usluga servisa	Dobro definirana sučelja servisa, neovisno isporučivi servisi, REST resursi
Metoda	Objektno orijentirana analiza i dizajn (OOAD), dizajn specifičan za pojedini servis	Dizajn vođen domenom (eng. Domain-driven design), agilni principi
Principi arhitekture	Slojevitost, labava povezanost, modularnost, neovisan tijek obrade	Idealni principi arhitekture u oblaku (djelomično se preklapa sa SOA)
Pohrana podataka	Informacijske usluge, implementacije pružatelja usluga	Perzistentnost poligota (SQL, NoSQL..)
Isporuka	Nije u opsegu logičke definicije (eng. out of scope)	Lagani kontejneri (Docker, Dropwizard..), XaaS opcije preko oblaka

Upravljanje sustavom		Mršavo, ali sveobuhvatno upravljanje sustavom/uslugama (tj. DevOps)
Rukovanje porukama, transformacija, prilagođavanje	ESB (eng. Enterprise Service Bus)	API pristupnik, lagani sustavi poruka (npr. RabbitMQ)
Sastav/kompozicija	Uzorci koreografije i orkestracije	Orkestracija kroz Obično Staro Programiranje (eng. Plain Old Programming)
Pregled servisa	Uzorak registara servisa (uključujući repozitorij servisa)	Proizvoljni registri i repozitoriji servisa, otkrivanje servisa (eng. service discovery)

(Izvor: Zimmerman O., 2016)

6.2.5. Zamke mikroservisne arhitekture

Mikroservisna arhitektura sa svojim karakteristikama zvuči vrlo primamljivo i kao nešto novo javlja se trend korištenja mikroservisa čak i kada za njima nema potrebe. Nove arhitekture često postaju trendovi zato što u početku ne postoji dovoljno primjera koji bi prokazali jasnu sliku njihove koristi u rješavanju određenih problema. U ovom će se poglavlju obraditi neki slučajevi u kojima mikroservisna arhitektura nije dobar odabir – upravo s ciljem da se pokaže da mikroservisi nisu rješenje za svaki problem.

6.2.5.1. Kretanje od mikroservisa

Iako će u praktičnom dijelu ovog rada arhitektura aplikacije biti dizajnirana s ciljem da se radi o mikroservisima, to u praksi nije uvijek potrebno. Kada se aplikacije razvijaju „od nule“, one su u početku vrlo male i često je moguće u vrlo kratkom roku ponoviti postupak. Monolit predstavlja veliku stijenu, a aplikacije su u svojem početku samo kamenčići. Bitno je uočiti trenutak kada sustav u svojem razvoju krene u smjeru monolita te tada reagirati na pravilan način (Shahir D. et al., 2016).

6.2.5.2. Mikroservisi bez DevOps-a

Mikroservisi predstavljaju veliku količinu dijelova koje je potrebno uskladiti. Usklađivanje na strani nadziranja i isporuke nije lagan zadatak. Potrebna je osoba koja će automatizirati cijeli postupak isporuke tako da ju je moguće izvršiti pritiskom jednog gumba, kao i razne okidače koji prilikom dodavanja novog kôda automatski usklađuju sav kôd u produkciji. DevOps inženjeri, koristeći najbolje prakse, omogućuju kontinuiranu isporuku i brinu o kvaliteti isporučenog softvera, tj. mikroservisa (Shahir D. et al., 2016).

6.2.5.3. Upravljanje infrastrukturom

Mikroservisi u pravilu koriste veći broj baza podataka, brokera poruka i sl. Sve to je potrebno održavati, grupirati u klastere i općenito se brinuti o radu. U početku rada s mikroservisima vrlo je korisno kada se o tome ne treba voditi velika briga. Postoje razne usluge u oblaku kao što su IBM Bluemix ili Cloud Foundry koje brinu o tim aspektima (Shahir D. et al., 2016).

6.2.5.4. Previše mikroservisa

Svaki mikroservis koristi i treba resurse za rad. Alati za upravljanje i isporuku imaju ograničenja koja se mogu lako prekoračiti ako se novi mikroservisi dodaju bez imalo opreza. S ovog pogleda, rastavljanje servisa tek kada se jave dijelovi koji imaju sukobljene zahtjeve za skaliranje, životni ciklus ili podatke, smatra se boljom opcijom. Kada su mikroservisi premali, a ima ih puno, to samo odgađa zahtjevan posao na njihovu integraciju (Shahir D. et al., 2016).

6.2.6. Pregled mikroservisne arhitekture

Richards M. (2015) proveo je analizu i ocjenu nekih od karakteristika mikroservisne arhitekture s obzirom na ostale koje su spomenute u početku ovog rada. Sve karakteristike osim jedne su dobile visoku ocjenu.

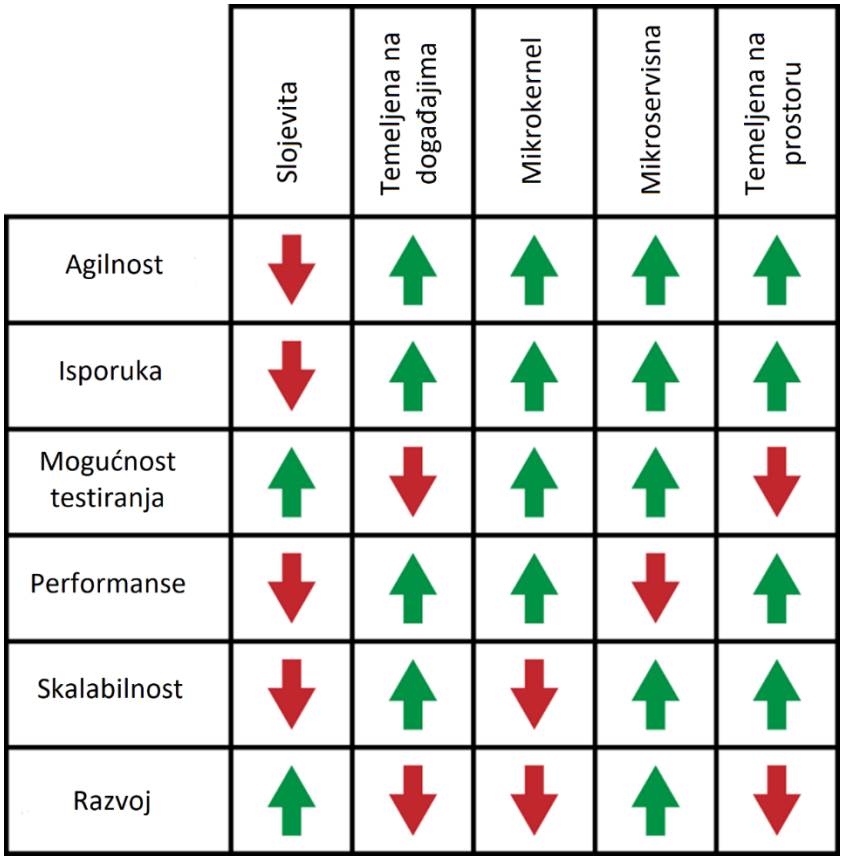
Prvo svojstvo koje je uzeto u obzir je agilnost koja se odnosi na mogućnost arhitekture da brzo odgovori na promjenjivu okolinu. Već je mnogo puta spomenuta činjenica da neovisne jedinice omogućuju brzu i jednostavnu isporuku zbog svoje labave povezanosti te je time opravdana visoka ocjena ovog aspekta.

Jednostavnost isporuke proizlazi iz mogućnosti da se radi tzv. vruća isporuka (eng. hot deployment) pojedinih servisa u bilo koje doba dana. Time se ujedno i smanjuje rizik koji dolazi s procesom isporuke jer se u slučaju neuspjeha mogu lakše ispraviti eventualne greške. Greške također ne utječu na cijeli sustav, već samo na onaj dio koji se isporučuje tako da ostatak sustava može nastaviti normalno raditi.

Kao i većina prednosti, jednostavnost testiranja proizlazi iz izoliranosti jedinica sustava. Jasno je da je manje jedinice lakše detaljnije testirati. Mogućnost skaliranja neće se ponovo objašnjavati, ali se također smatra velikom prednošću. Jednostavnost razvoja je posljednja karakteristika koju je autor ocijenio visokom ocjenom u odnosu na ostale arhitekture. Funkcionalnosti koje se implementiraju su izolirane u odvojene servisne komponente čime se smanjuje mogućnost širenja utjecaje na ostale komponente i potrebe za koordinacijom programera i njihovih timova.

Negativnom stranom mikroservisa smatraju se performanse. Moguće je implementirati sustave koji se temelje na mikroservisnoj arhitekturi i održati dobre performanse, no cilj samog

uzorka nije usredotočen na postizanje visokih performansi. Usporedba navedenih karakteristika u odnosu na ostale stilove arhitekture prikazana je na slici (Richards M., 2015).



Slika 20 Međusobna usporedba stilova arhitekture (Izvor: Richards M., 2015)

6.3. Conwayev zakon

Ono što se danas naziva Conwayevim zakonom, prvi je definirao američki računalni znanstvenik Melvin Edward Conway, prema kojem je zakon i dobio ime. Conwayev zakon prema tome glasi:

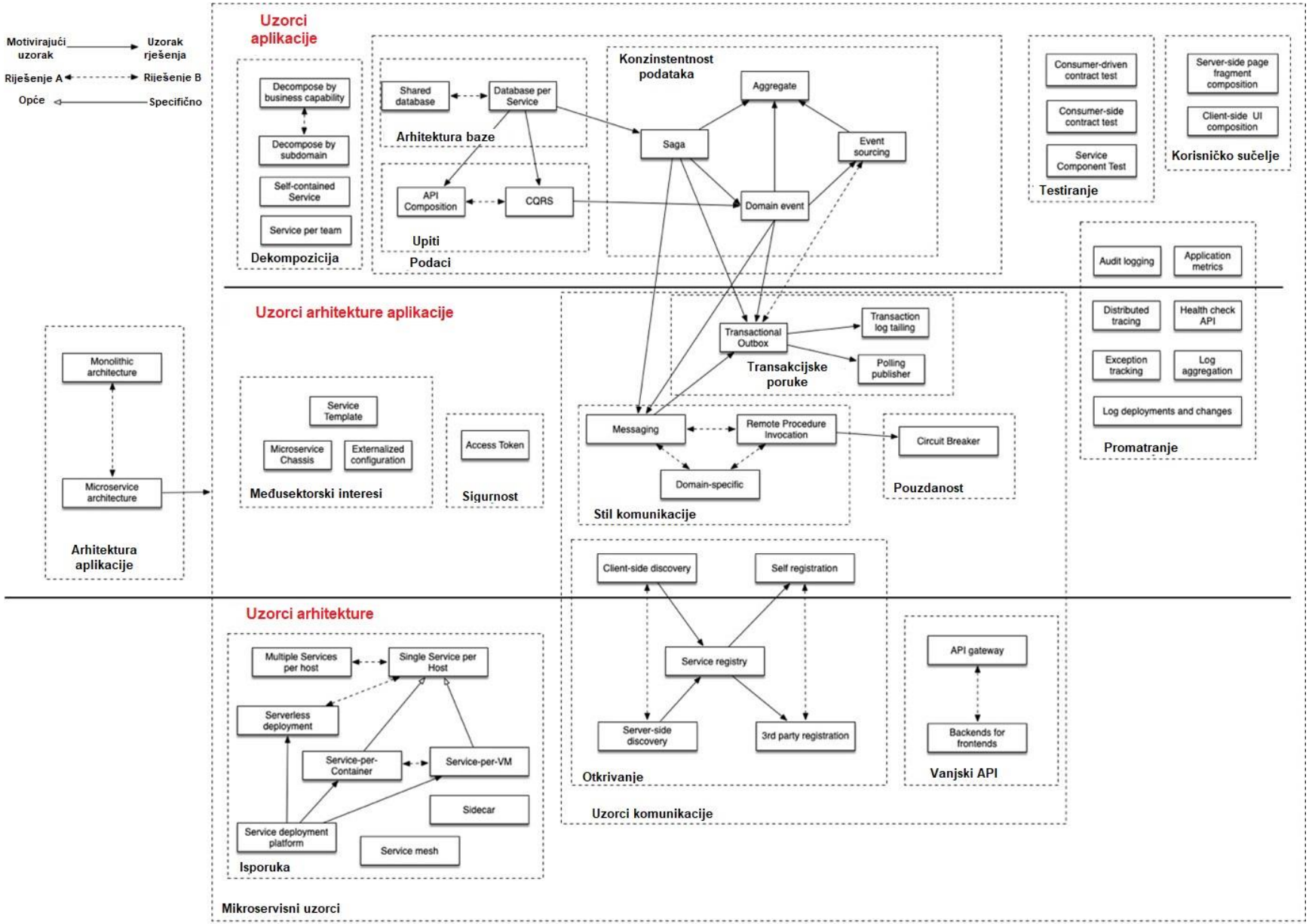
„Svaka organizacija koja dizajnira sustav (u širem smislu) proizvest će dizajn čija je struktura kopija komunikacijske strukture organizacije“ (Wolff E., 2016).

Ovaj zakon nije primjenjiv samo na softver, već bilo koju vrstu dizajna. Conway je tvrdio da je ovaj odnos nužna posljedica komunikacijskih potreba ljudi koji obavljaju posao. Naime, komunikacijska struktura koja se spominje u Conwayevom zakonu ne mora identično odgovarati shemi organizacije. Primjer su neformalne komunikacijske strukture koje je također potrebno uzeti u razmatranje unutar ovog konteksta. Geografska distribucija timova, koja danas u IT industriji nije rijetkost, rezultira komunikacijom na daljinu koja zbog raznih razloga može biti otežana. Prema tome, lakše je komunicirati s osobom koja je u istom uredu ili zgradi kao i vi (Herbsleb J. D. i Grinter R. E., 1999).

Veza mikroservisne arhitekture i Conwayevog zakona leži u tome što su mikroservisi vrlo pogodni za definiranje raspodjele funkcionalnosti prema Conwayevom zakonu. Budući da svaki mikroservis može funkcionirati neovisno o ostalima, distribucija poslova postaje lakim zadatkom. Međutim, usmjeri li se arhitektura sustava prema komunikacijskoj strukturi, rezultat će biti struktura koja promjenom jednog mjesta (mikroservisa) zahtjeva promjene i na drugim mjestima. Zbog toga su arhitektonske promjene mikroservisa otežane i cijeli proces postaje manje fleksibilnim. Premještanjem funkcionalnosti iz jednog mikroservisa u drugi može imati znatne posljedice za ostale timove (koji rade na razvoju drugih mikroservisa) u smislu prilagodbe i održavanja. Prema tome, važnost načina na koji se radi raspodjela mikroservisne arhitekture i timova koji rade na njezinoj implementaciji i održavanju postaje važnijim čim se radi o većoj organizacijskoj strukturi (Wolff E., 2016).

6.4. Uzorci u mikroservisnoj arhitekturi

Do sada je bilo riječi samo o uzorcima dizajna na arhitekturnoj razini, gdje je mikroservisna arhitektura jedan od predstavnika. Spusti li se razmatranje razinu niže i ograniči na područje mikroservisa, može se istaknuti nekolicina uzoraka koji su pogodni za njihovu primjenu. Primjena pojedinog uzorka ovisi o slučaju i svrsi koja se želi postići izgradnjom aplikacije te nije nužno pod svaku cijenu težiti njihovoj primjeni jer se tada stvara takozvani anti-uzorak (eng. anti-pattern). Na sljedećoj slici prikazana je šira slika odnosa mikroservisne arhitekture i često primjenjivih uzoraka u njenom kontekstu. Budući da je njihov broj velik, a nekih se dotiče i u ostatku rada, u nastavku ovog poglavlja će biti obrađeni samo oni značajniji i nespomenuti.

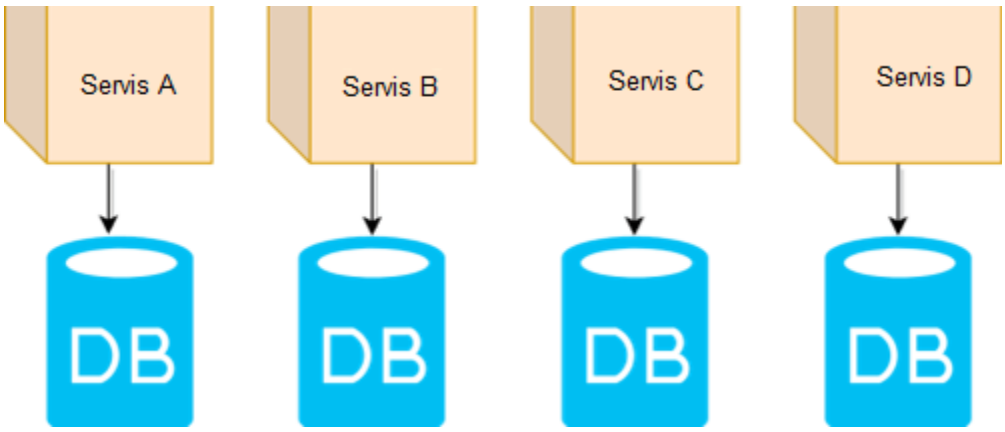


Slika 21 Uzorci vezani uz mikroservisnu arhitekturu (Izvor: Richardson K., 2019)

6.4.1. Upravljanje podacima

Podaci su od velike važnosti kada se govori o ispravnom i uspješnom radu aplikacije, tj. programskog sustava. U klasičnim aplikacijama ovdje ne dolazi do tolike nesigurnosti i problema odabira s arhitekturalne strane. No, kod mikroservisne arhitekture to nije slučaj. Naime, održavanje konzistentnosti podataka i izvršavanje upita nad bazom podataka postaje zahtjevnije kada su dijelovi aplikacije toliko razdvojeni. Prema tome, postoji par uzoraka, od kojih su neki i međusobno povezani, koji će biti obrađeni u ovom dijelu. To su: Database per Service uzorak, Shared Database uzorak, Saga uzorak, API Composition uzorak i CQRS uzorak.

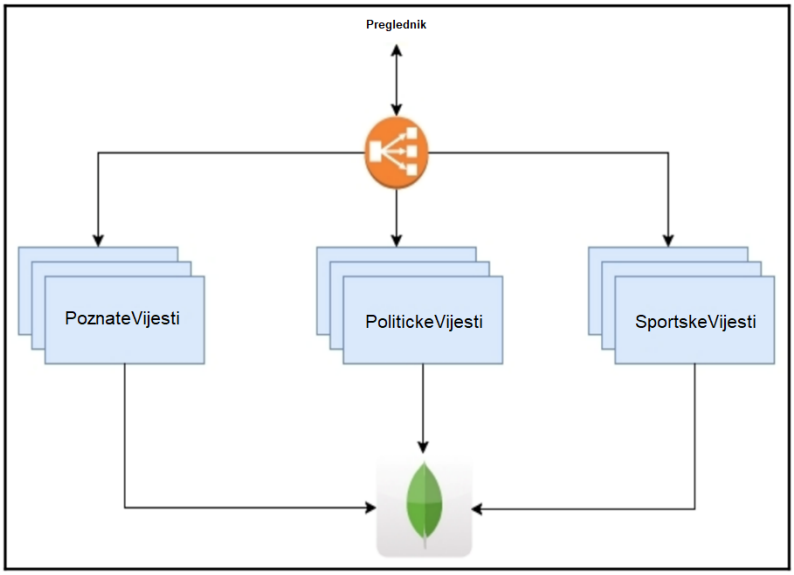
Database per service uzorak, kao što mu i samo ime govori, svodi se na korištenje zasebne baze podataka za svaki mikroservis. Do potrebe za ovim uzorkom dolazi zbog labave povezanosti mikroservisa. Prema tome, različiti mikroservisi imaju različite zahtjeve u pogledu pohrane podataka. Primjerice, zbog modela podataka koji koristi mikroservis A, najprikladnija baza podataka za njega bila bi relacijska. S druge strane, neki mikroservis B nema potrebu za relacijskom bazom, već je za njega idealna NoSQL baza (npr. MongoDB) zato što većinom rukuje nestrukturiranim podacima. Mikroservis C mogao bi imati neku treću varijantu te je zbog toga vidljiva potreba za posebnom bazom za svaki mikroservis (Richardson C., 2018). Dakle, rješenje problema je podatke (bazu) učiniti privatnom za svaki servis i pristup omogućiti samo preko njegovog API-ja. Time baza podataka mikroservisa postaje dio njegove implementacije te joj, kao i implementaciji, nije moguće direktno pristupiti od strane drugih servisa. Odvajanje baze podataka i pretvaranje iste u privatnu može se postići na više razina. Prva razina, tj. način za postizanje navedenog jest kreiranje privatnih tablica unutar jedne baze podataka (eng. private-tables-per-service). Ovako i dalje postoji jedna baza podataka, ali prividno svaki mikroservis ima svoju bazu jer ima omogućen pristup samo onim tablicama koje su u njegovom „vlasništvu“. Razina iznad je korištenje zasebne sheme za svaki servis (eng. schema-per service). U ove dvije razine uočljiv je nedostatak vezan uz početni primjer, a to jest da, s



Slika 22 Sustav s bazom podataka po servisu (Izvor: Engineering D. A., 2021)

obzirom na to da se radi o istoj bazi podataka, svi servisi su i dalje prisiljeni koristiti istu vrstu (npr. relacijska). Reklo bi se da pravo odvajanje baze dolazi tek na zadnjoj razini, a to je fizičko odvajanje baza podataka na zasebne servere (eng. database-server-per-service). Fizičko odvajanje daje mogućnost skaliranja po klasterima, bez obzira na servis koji ju koristi, no u detalje klastera baza podataka se neće ulaziti (Messina A. et al., 2016).

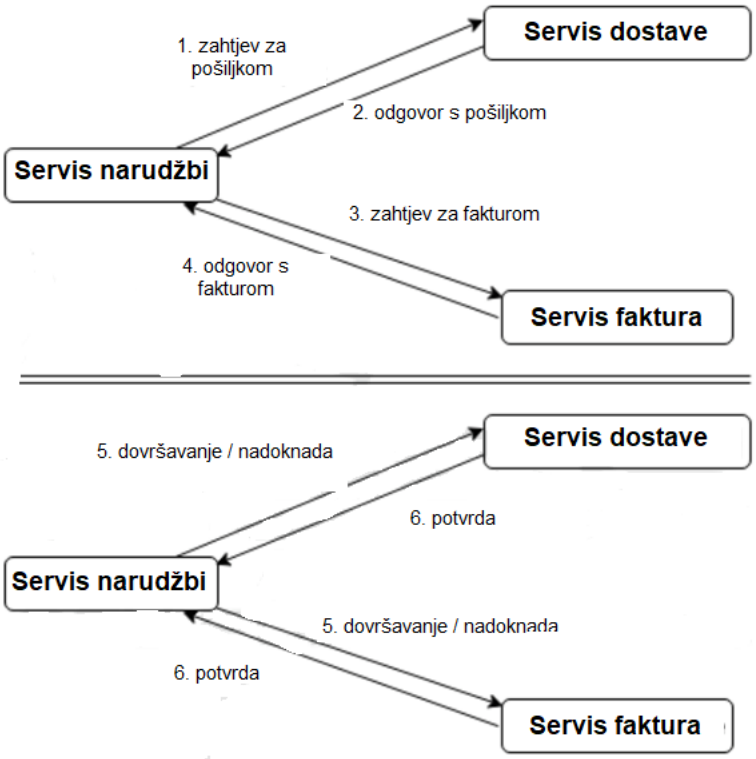
Shared Database uzorak također ima vrlo opisno ime te se odnosi na korištenje jedne baze podataka koja je dijeljena između svih mikroservisa. Nakon govora o prošlom uzorku, postavlja se pitanje zašto bi se ovaj (anti) uzorak uopće koristio? Ovaj uzorak radi sve na drugačiji način od prethodnog, ali je namijenjen i za druge situacije. Primarno, ovaj se uzorak koristi kroz proces migracije s monolitne arhitekture na mikroservisnu. Prijelaz s monolitne arhitekture nije lagan, pogotovo kada se radi o starijim (eng. legacy) sustavima. Noviji projekti, nazvani i „zelenim projektima“ (eng. green projects), često imaju dodirnih točaka sa starijim sustavima. Prema tome, korištenje ovog uzorka na novijim projektima se zasigurno smatra anti-uzorkom, no može biti privremeno rješenje za starije sustave u fazi tranzicije. U idealnom svijetu, potreba za ovim uzorkom ne bi postojala, ali zbog količine starijih sustava, ovaj uzorak pomaže velikom broju projekata da resetiraju i prilagode svoju arhitekturu. Ovdje također dolazi do velikog rizika u radu sustava budući da svi mikroservisi (kojih u većim sustavima može biti stvarno mnogo) rade promjene na istoj bazi podataka. Na projektnom timu je da uvidi rizike, mogućnosti i potrebu za korištenjem ovog uzorka na konkretnom scenariju (Pacheco V. F., 2018).



Slika 23 Struktura sustava s jednom bazom podataka (Izvor: Pacheco V. F., 2018).

Do potrebe za sljedećim uzorcima dolazi zbog korištenja jedne baze podataka po mikroservisu. Naime, jedna baza podataka, a prema tome i jedan servis ne sadrži uvijek sve

potrebne podatke da bi se ispunio jedan zahtjev. Zbog toga dolazi do potrebe da se dohvate podaci iz više baza podataka i na kraju ukomponiraju u cjelinu. Uzorak koji omogućuje da transakcije imaju raspon kroz više mikroservisa (budući da svaki servis ima svoju bazu, suštinski propagiraju kroz više baza), naziva se *Saga* uzorkom (Richardson C., 2018). Naziv ovog uzorka predstavlja slijed operacija koje se mogu ispreplitati s drugim operacijama. Prednost je u tome što svaka operacija, koja predstavlja jedan dio sage može biti poništena (eng. undo) i prema tome se postiže garancija da će sve operacije biti uspješno izvršene ili će se u suprotnom pokrenuti tzv. kompenzacijske radnje koje stanje podataka vraćaju na staro. (Štefanko M. et al., 2019). Ovakva djelomična obrada omogućuje da npr. kada se zahtjev propagira kroz više servisa i u jednom trenutku dođe do greške, svi servisi koji su prethodno

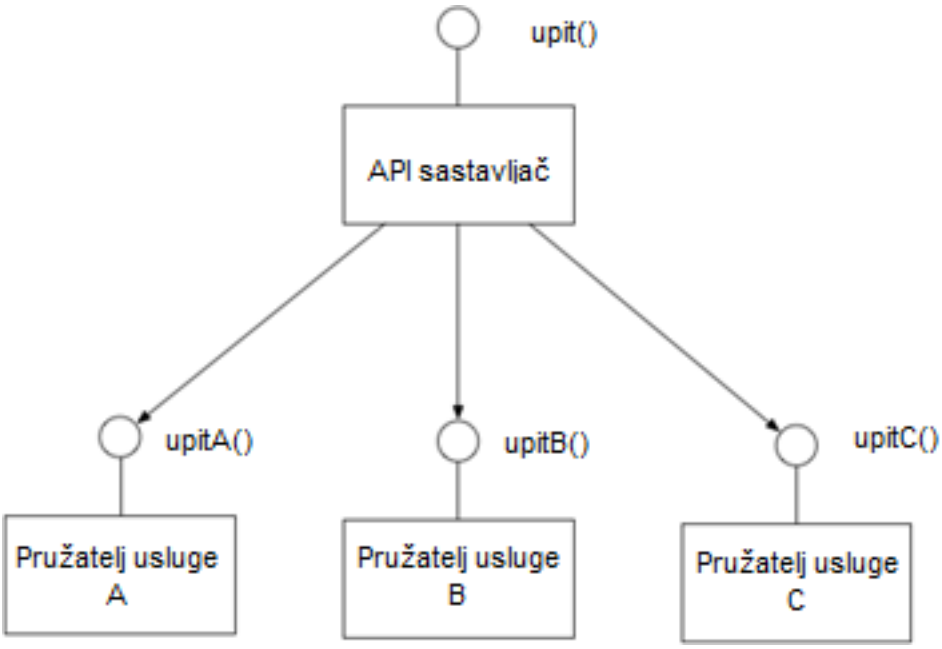


Slika 24 Saga uzorak (Štefanko M. et al., 2019).

napravili promjene nad svojim podacima, imaju mogućnost vraćanja na stanje prije početka obrade tog zahtjeva. Saga se na prvi pogled može činiti vrlo sličnom ACID transakcijama (eng. Atomicity, Consistency, Isolation and Durability) (Atomarna, Konzistentna, Izolirana i Trajna), no razlika je u svojstvu izoliranosti. Sage su često distribuirane i svode se na niz zahtjeva od kojih svaki servis mora pružiti garanciju ACID svojstava zbog čega je njihov temelj na međusobnoj pouzdanosti servisa (Richardson C., 2018).

API Composition uzorak još je jedan uzorak koji se bavi problemom koji se javlja korištenjem jedne baze podataka po mikroservisu. Problem na čije se rješavanje on usredotočuje jest problem izvršavanja upita koji treba podatke koji se nalaze u više različitih

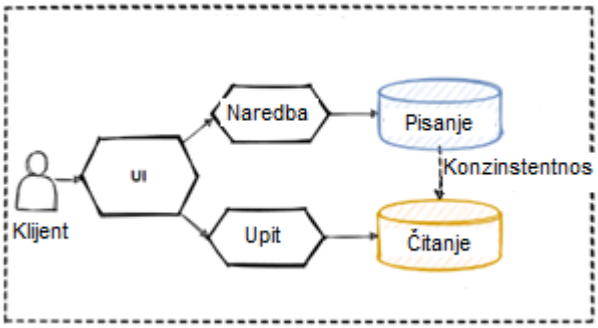
servisa (baza podataka). Ovaj uzorak sadrži dva glavna aktora, a to su API sastavljač (eng. API composer) i pružatelj usluge (eng. provider service). Pružatelji usluge su mikroservisi od



Slika 25 API Composition uzorak (Izvor: Richardson C., 2018)

kojih svaki ima mogućnost izvršavanja upita koji čini dio glavnog upita. API sastavljač prema tome ima pristupnu točku koja sakuplja podatke pozivanjem pojedinačnih pružatelja usluga i sastavlja ih u konačan odgovor (Richardson C., 2018). U mikroservisnoj arhitekturi API sastavljač je predstavljen jednim uzorkom o kojem je već prije bilo riječi, a to je API pristupnik. Pogleda li se dijagram ovog uzorka na prethodnoj slici, postaje jasno zašto je API pristupnik onaj koji obavlja posao API sastavljača u mikroservisnoj arhitekturi.

CQRS ili Odvajanje odgovornosti za naredbu upita (eng. Command Query Responsibility Segregation) je uzorak koji se može koristiti umjesto API Composition uzorka. Samo ime ovog uzorka govori o ideji koja stoji iza njega, a to je odvajanje odgovornosti zapisivanja i čitanja podataka. Postoje scenariji u kojima korištenje API Composition uzorka postaje vrlo zahtjevno (nemoguće) za implementiranje ili dolazi do pada performansi njegovim korištenjem. Tada je alternativa korištenje CQRS uzorka. Ideja ovog uzorka je implementacija dviju zasebnih baza podataka od kojih jedna služi samo za pisanje (spremanje) podataka (često je normalizirana) te druga koja služi za čitanje (izvršavanje upita) podataka (denormalizirana). Jasno je da je potrebno pronaći mehanizam kojim će se ove dvije odvojene baze podataka držati sinkroniziranim da bi se pri izvršavanju zahtjeva dohvaćali ispravni podaci (Pacheco V. F., 2018).



Slika 26 CQRS uzorak (izvor Ozkaya M., 2021)

6.4.2. Pouzdanost

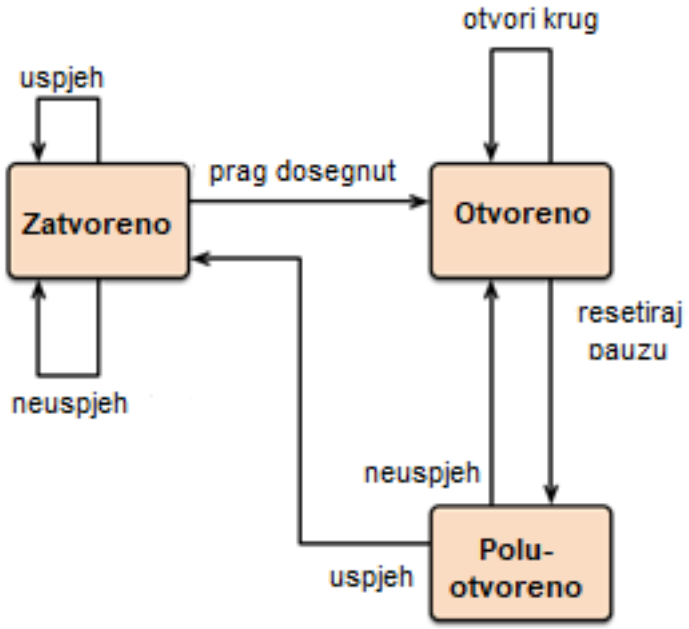
Sljedeća kategorija u kojoj će se obraditi jedan, ali vrlo važan, uzorak jest kategorija pouzdanosti. Budući da mikroservisi često surađuju i međusobno komuniciraju, postoji mogućnost da servis čiju uslugu treba mikroservis A, u jednom trenutku nije dostupan. Dostupnost mikroservisa nije moguće garantirati u svakom trenutku, kao ni kvalitetu mrežne povezanosti te zbog toga može doći do nemogućnosti međusobne komunikacije dvaju mikroservisa. S obzirom na to da mikroservisi unaprijed ne mogu znati kada će doći do problema u komunikaciji, a isti može negativno utjecati na rad, potreban je mehanizam kojim će se spriječiti širenje greške u takvim situacijama.

Naziv ovog uzorka dolazi od osigurača (eng. circuit breaker) za koji je jasno da nije prisutan samo u IT svijetu. Osigurač služi za automatsko isključivanje kada se javi preopterećenje ili kratki spoj u sustavu, upravo da bi se zaštitile električne instalacije. U mikroservisnom sustavu, greška ili nedostupnost jednog mikroservisa predstavlja kratki spoj, a električna instalacija je cijeli sustav, tj. integritet aplikacije. Dogodi li se greška u jednom mikroservisu, on prestaje biti u mogućnosti obrađivati pristigle zahtjeve, no ako se ne prestane slati zahtjeve na isti servis, stanje će se samo pogoršati. *Circuit breaker* uzorak nastoji otkriti kvar u ostalim mikroservisima na temelju kašnjenja ili nedostupnosti odgovora te spriječiti daljnje slanje zahtjeva kako bi se mikroservisu pružilo vrijeme potrebno za oporavak (Pacheco V. F., 2018).

Osigurač se postavlja između pošiljatelja i primatelja zahtjeva te ovisno o statusu, koji je uvjetovan s više parametara, propušta zahtjev ili pruža alternativu. Neki od parametara koji se ovdje uzimaju u obzir su broj neuspjelih pokušaja slanja zahtjeva i trajanje pauze (eng. timeout). Ako se određeni broj puta dogodi da slanje zahtjeva nije rezultiralo odgovorom, osigurač će se aktivirati i kroz određeni period zabraniti slanje zahtjeva. Nakon tog vremena će se ponovo pokušati poslati zahtjev te ukoliko je uspješan, servis normalno nastavlja slati zahtjeve, a u suprotnom ponovo pauzira (Richardson C., 2018). Opisan scenarij može se podijeliti u tri stanja u kojima se osigurač može nalaziti, a prema Falahah S. et al. (2021) to su:

1. *zatvoreno* (eng. closed) – ne postoji greška i zahtjevi se normalno šalju; kada dođe do N neuspješnih zahtjeva, prelazi se u sljedeće stanje
2. *otvoreno* (eng. open) - osigurač blokira slanje zahtjeva na nedostupan servis; postoji mogućnost definiranja zadane operacije koja se poziva u tom slučaju
3. *polu-otvoreno* (eng. half-open) – servis neko vrijeme nije slao zahtjeve i pokušat će ponovo kontaktirati servis s greškom; u slučaju uspješnog odgovora vraća se u zatvoreno stanje, a u suprotnom u otvoreno stanje

Ovaj scenarij prikazan je i na sljedećoj slici:



Slika 27 Circuit breaker uzorak (Izvor: Bandurski P., 2021)

6.4.3. Sigurnost

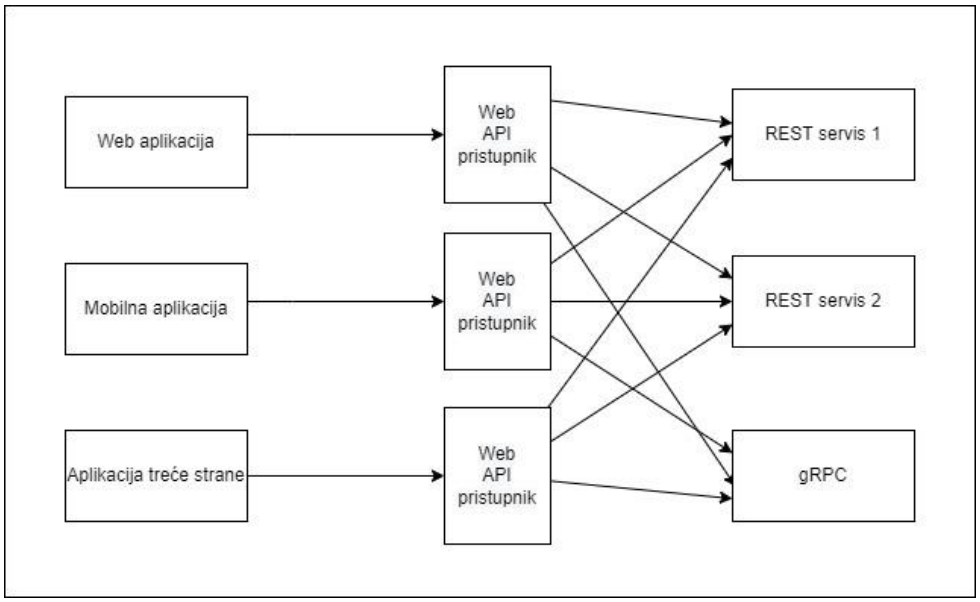
U mikroservisnoj arhitekturi odgovor na pitanje sigurnosti postaje nešto zahtjevniji. Zbog odvojenih mikroservisa, potrebno je definirati tko će i kako biti zadužen za autentikaciju korisnika, tj. zahtjeva. Dakle, razlika u implementaciji sigurnosnog mehanizma u mikroservisnoj arhitekturi, u odnosu na monolitnu arhitekturu temelji se na dva aspekta koja je definirao Richardson C. (2018). Prvi se naziva sigurnosnim kontekstom u memoriji (eng. in-memory security kontekst), a govori o tome da zbog rasprostranjenosti mikroservisa, nije moguće držati korisnikov identitet u memoriji, kao što je to slučaj u monolitnim aplikacijama. Drugi aspekt monolitne arhitekture koji nije primjenjiv u mikroservisnoj arhitekturi jest

centralizirana sesija (eng. centralized session), koja nije moguća iz istog problema vezanog uz pristup memoriji kao i kod rukovanja kontekstom.

Access token uzorak pruža mogućnost prijenosa identiteta korisnika kroz mrežu, a s time i mikroservise. O samoj implementaciji žetona koji se ovdje koriste bit će više riječi u poglavlju vezanom uz OAuth i JSON Web Token. Dakle, postoji mehanizam koji će potvrditi identitet korisnika, no potrebno ga je pravilno upotrijebiti. Svaki servis može provjeriti ispravnost žetona, no time otvaramo vrata unutrašnjosti mreže prema svim zahtjevima, pa čak i onima koji na kraju ne ispadnu autoriziranim. Također, svaki mikroservis može na različiti način definirati i vršiti provjeru autorizacije. Budući da je već definiran API pristupnik koji predstavlja centralnu točku pristupa sustavu, idealna implementacija je postavljanje mehanizma autorizacije na API pristupnik. Time se postiže da neispravni, tj. neautorizirani zahtjevi ne ulaze u sustav, kao i činjenica da se smanjuje rizik sigurnosnog propusta zbog samo jedne točke autorizacije u cijelom sustavu. Logika se s time miče sa svakog mikroservisa, a po potrebi se žeton prosljeđuje unutar ispravnog zahtjeva u slučaju da neki od mikroservisa i dalje treba izvršiti neku radnju s njime (npr. poslati zahtjev na neki treći mikroservis za koji je također potreban autoriziran identitet korisnika) (Richardson C., 2018).

6.4.4. Pristup sustavu

Jedan od dvaju uzoraka koji spadaju u ovu skupinu već je obrađen, a radi se od API pristupniku. API pristupnik je najčešći način realizacije jedne točke pristupa u mikroservisni sustav, no postoji i alternativni uzorak koji će se ovdje ukratko spomenuti. Nedostatak koji se može javiti kod korištenja API pristupnika jest u tome da je sve „stavljeno u isti koš“. Bez obzira odakle zahtjev dolazio, on prolazi kroz isti API pristupnik pa prema tome i svi timovi koji rade na ostatku sustava (mikroservisima u njemu) prilagođavaju jedno te isti API pristupnik. Ovaj se problem rješava korištenjem *Backend for frontends* (BFF) uzorka umjesto API pristupnika. Ideja uzorka je u suštini ista, no radi se podjela pristupnih točaka, tj. stražnjeg djela aplikacije (eng. backend) za svako od različitih sučelja (eng. frontend). Ovaj uzorak puno je jasniji ako se pogleda sljedeća slika.



Slika 28 Varijacija BFF uzorka

Na slici je vidljivo da se zapravo radi o API pristupnicima, no oni su podijeljeni ovisno o klijentskoj strani. Tako imamo poseban pristupnik za mobilne uređaje, poseban za Web aplikacije, ali i za neke aplikacije treće strane. Potreba za podjelom ovisi o sustavu koji se implementira, no omogućuje da svaki tim radi na zasebnoj pristupnoj točki čime se olakšavaju izmjene i općenit rad – čim je riječ o većim timovima. BFF uzorak jasno definira odgovornosti i pridonosi izoliranosti sustava čime se podiže i njegova pouzdanost jer greška u jednom pristupniku (za jednu vrstu klijenta) nema utjecaja na druge. Također, svaki API je moguće zasebno skalirati i oni su sami puno „lakši“ i jednostavniji za održavanje i pokretanje. Na kraju, korištenje BFF uzorka nije uvijek opcija, pogotovo ako običan API pristupnik zadovoljava sve potrebe sustava (Richardson C., 2018).

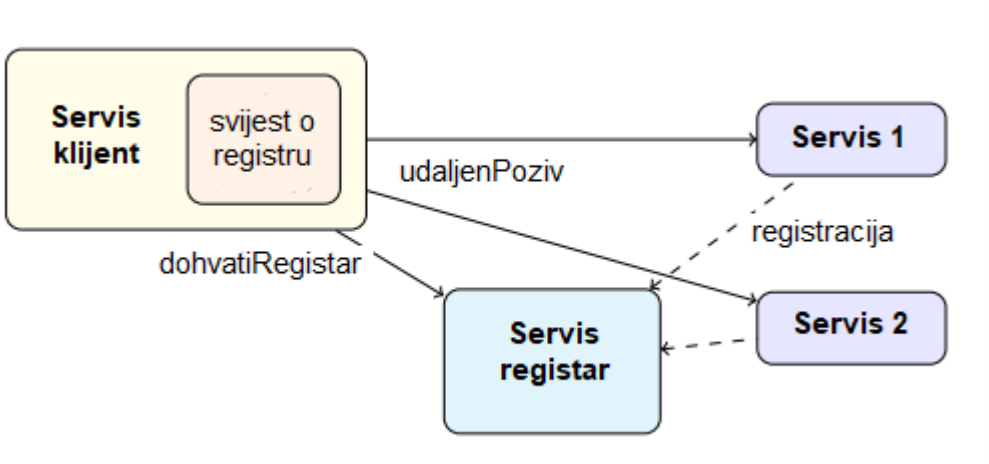
6.4.5. Otkrivanje servisa

Posljednja, no ništa manje značajna, skupina koja će se u ovom poglavlju obraditi je otkrivanje servisa (eng. service discovery). Nekoliko je puta kroz rad spomenuto da mikroservisi omogućuju lagano podizanje više instanci kako bi se tako moglo balansirati opterećenjem i preusmjeravati zahtjeve na one mikroservise koji su ih u mogućnosti obraditi, da bi se rad cijelog sustava što više optimizirao. Otkrivanje mikroservisa jednim dijelom zavire u njihovu isporuku, kao i održavanje, koji su vrlo zahtjevni. No, u njih se neće dublje ulaziti u ovom radu. Ipak, uzme li se primjer mikroservisa koji je isporučen u Docker kontejneru i isporučen na nekom poslužitelju, jasno je da je potrebno poznavati njegovu IP adresu i port da bi mu se pristupilo. Problem dolazi kod scenarija u kojem se želi pokrenuti nova instanca istog mikroservisa. Kako u tom slučaju razlikovati mikroservise?

Najveća greška koja može biti učinjena u ovom slučaju jest da se mikroservisu, kroz proces implementacije, dodijeli fiksan port (npr. želi se da svi mikroservisi za autentikaciju budu na portu 8888). Više instanci jednog mikroservisa će u ovom scenariju biti moguće samo ako se svaka instanca pokreće na zasebnom poslužitelju, tj. s drugačijom IP adresom. Zbog toga je dodjela porta instanci mikroservisa u pravilu dinamička te se eventualno može ograničiti bazen (eng. pool) dopuštenih portova za korištenje. Vidljivo je da, u većim sustavima, ljudi teško mogu pratiti gdje se nalazi koji mikroservis. Dolazi do problema kako to učiniti programski, u okruženju gdje mikroservisi moraju znati kamo poslati zahtjev da bi sustav funkcionirao. Upravo ovaj problem rješava otkrivanje servisa koje ima zadatak pratiti sve aktivne mikroservise (jednoznačno određene IP adresom i portom) kako bi se zahtjevi bezbrižno i ispravno mogli propagirati kroz sustav (Nadareishvili I. et al., 2016). Korištenje otkrivanja servisa pridonosi fleksibilnosti sustava s obzirom na to da omogućuje jednostavno dodavanje i uklanjanje instanci mikroservisa da bi se skalirao sustav. Otkrivanje servisa, u arhitekturi sustava, najčešće se nalazi na razini API sloja budući da se time stvara centralno mjesto pristupa sustavu.

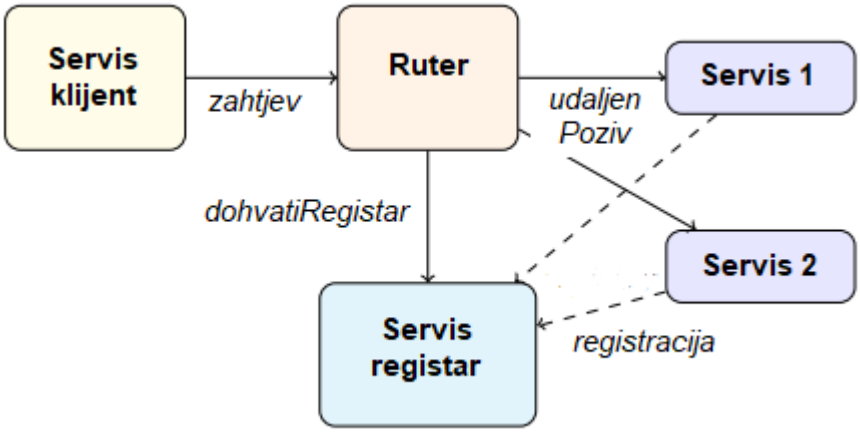
Glavna komponenta potrebna za realizaciju ovog uzorka je registar servisa (eng. service registry), slično kao i kod SOA-e, koji služi kao središnja baza podataka za praćenje instanci mikroservisa, kao i njihove lokacije u mreži (adrese). Prema tome, mikroservisi imaju zadatak, u principu pri pokretanju, registrirati se kod registra servisa, tj. javiti mu svoje puno ime (IP adresa i port). Ovaj proces se naziva još i samoregistracijom (eng. self registration). Mikroservisi, koji se u ovom slučaju mogu nazvati klijentima, koriste registar servisa kako bi saznali adrese ostalih mikroservisa na koje je potrebno poslati zahtjev.

Postoje dva oblika otkrivanja servisa koji autori navode, a razlikuju se prema načinu dohvaćanja adrese mikroservisa. Ti se oblici nazivaju otkrivanje na strani klijenta (eng. client-side service discovery) i otkrivanje na strani poslužitelja (eng. server-side service discovery). Na sljedećoj slici prikazan je scenarij otkrivanja na strani klijenta. Već je spomenuto da se pri implementaciji očekuje da mikroservisi nemaju fiksnu lokaciju pa je prema tome servis „klijent“ (onaj koji želi poslati zahtjev) implementiran tako da prvo traži adresu mikroservisa od registra servisa. Nakon što su mu adrese svih potrebnih mikroservisa poznate, može nastaviti s radom i slati zahtjeve. U teoriji je ovo jednostavno, no potrebno je takvo ponašanje implementirati za svaku vrstu mikroservisa u cijelom sustavu, što može postati dosta kompleksan zadatak.



Slika 29 Otkrivanje servisa na strani klijenta (Izvor: Montesi, F. i Weber J., 2016).

Zbog toga postoji drugi oblik otkrivanja servisa, prikazan na sljedećoj slici, a to je otkrivanje na strani poslužitelja. Ovdje je, u usporedbi s prethodnom slikom otkrivanja servisa na strani klijenta, prisutan novi akter – ruter (eng. router). Sva logika i zadatak informiranja o adresi odredišnog mikroservisa zahtjeva je preseljena sa servisa „klijenta“ na ruter koji sada ima zadatak kontaktirati registar servisa. Jasno je da je adresa rutera fiksna budući da je dovoljan samo jedan i servisi „klijenti“ ipak moraju znati kamo poslati zahtjev. S druge strane, nedostatak ovog pristupa je dodatan servis (ruter) koji mora biti implementiran i isporučen zato što, kao i svi ostali servisi, aktivno troši resurse za rad (Montesi, F. i Weber J., 2016).



Slika 30 Otkrivanje servisa na strani poslužitelja (Izvor: Montesi, F. i Weber J., 2016).

Upravo zato u praksi, kod većih sustava, ovi se oblici znaju kombinirati te postoji nekoliko poznatijih implementacija ovog uzorka, a jedna od njih će biti korištena u praktičnom dijelu ovog rada.

7. Žetoni i poruke

Ovo će se poglavlje pobliže dotaknuti nekih od već spomenutih koncepata, koji su često korišteni u okviru mikroservisa. Osim što se općenito koriste u mikroservisnoj arhitekturi, žetoni, poruke i kontejneri bit će korišteni i za realizaciju praktičnog dijela ovog rada. Zbog toga se ovim poglavljem kreće prema praktičnom dijelu te će ovi koncepti biti razrađeni kroz tehnologije i standarde u sklopu kojih se koriste. Cilj nije ulaziti u dubinu realizacije pojedine tehnologije ili standarda, već stvoriti širu sliku koja će obrazložiti njihovu primjenu.

7.1. JSON Web Token i OAuth standard

Žetoni su već spomenuti u poglavlju vezanom uz sigurnost mikroservisne arhitekture, a kada je riječ o žetonima, često se javljaju dva pojma – JSON Web Token (JWT) i OAuth. Ovo su dva različita pojma koja mogu, ali ne moraju biti korišteni u kombinaciji kako bi se zadovoljili određeni aspekti sigurnosti.

JWT je otvoreni standard koji predstavlja sigurnosni žeton temeljen na JSON (eng. JavaScript Object Notation) formatu koji omogućuje jednostavno dijeljenje sigurnosnih informacija o identitetu kroz različite osigurane domene (RFC-7519, 2015). JWT koristi Base64 shemu kodiranja kako bi se zaštitio sadržaj koji je zapravo predstavljen običnim tekstom. Zbog toga se u JWT žetonu može poslati bilo koja informacija. No, bitna činjenica jest da, iako podaci jesu kodirani i time je garantiran njihov izvor, oni nisu kriptirani tako da je moguće presresti JWT i urediti ga od treće strane. Prema tome, korištenje JWT-a bez HTTPS (eng. Hypertext Transfer Protocol Secure) protokola nije preporučeno.

JWT je podijeljen na 3 dijela koji se nazivaju zaglavlje (eng. header), teret (eng. payload) i potpis (eng. signature). Spojeni točkom ('.'), ova tri dijela predstavljaju JWT u cijelosti. Slika

Slika 31 Generiranje JWT žetona

prikazuje primjer jednog generiranog žetona koristeći Web aplikaciju jwt.io (https://jwt.io/) uz uvid u njegove dijelove prije i poslije procesa kodiranja. Zaglavlje žetona definira vrstu žetona (JWT) i algoritam kojim je on kriptiran. Za kriptiranje mogu se koristiti brojni algoritmi, a najčešći su HS256 (eng. HMAC using SHA256) i RS256 (eng. RSA using SHA256). Središnji dio (teret) može sadržavati bilo koju informaciju potrebnu u kontekstu pojedine aplikacije. Potpis je dobiven spajanjem kodiranog zaglavlja i tereta uz navedenu tajnu koja je potrebna za dekodiranje s druge strane. Spajanjem ovih triju dijelova, na slici je jasno vidljiva dobivena struktura žetona (RFC-7519, 2015).

Kao standard za način prenošenja JWT-a postala je tzv. autentifikacija nositelja (eng. bearer authentication). Naziv ove sheme provjere autentičnosti prati misao „omogući pristup nositelju ovog žetona“. Svi žetoni nisu obavezno JWT, no i OAuth standard, o kojem će više riječi biti u nastavku, također koristi ovu shemu. Pravilo koje se ovdje slijedi jest da se žeton šalje u zaglavlju (eng. header) HTTPS zahtjeva (može i HTTP, no nije preporučljivo) u formatu: `Authorization: Bearer <žeton>` (Bearer Authentication, bez dat.).

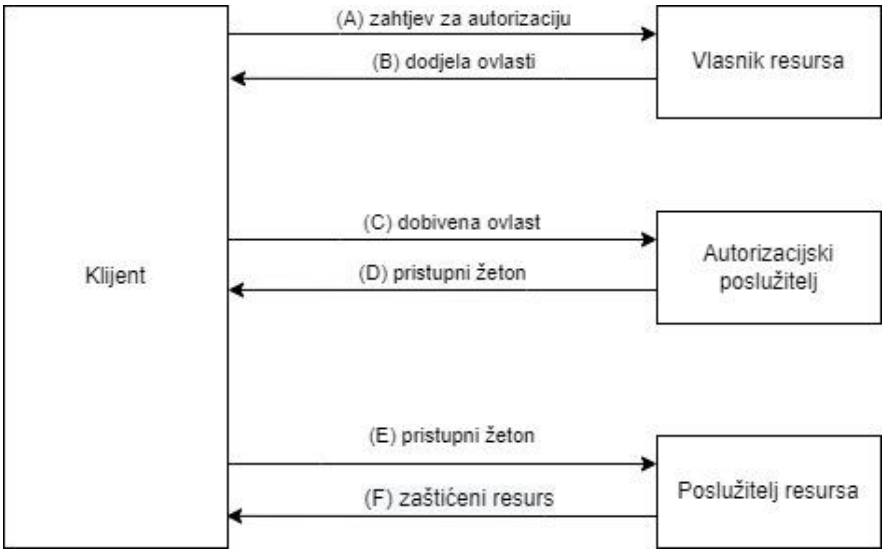
Drugi pojam koji se uvodi u ovom poglavlju jest OAuth standard za autorizaciju, tj. dodjelu prava pristupa resursima. OAuth omogućuje aplikaciji da dohvaća resurse u ime njegovog vlasnika, a sredstvo koje se koristi za „prenošenje ovlasti“ je žeton. Postoje OAuth 1.0 i OAuth 2.0 protokoli, od kojih je OAuth 2.0 novija verzija te će se naglasak na nju staviti u nastavku (RFC-6749, 2012). OAuth 2.0 zapravo pojednostavljuje protokol koji je bio definiran u OAuth 1.0, što govori i činjenica da Google, Facebook, Twitter i brojni drugi koriste OAuth 2.0. Ono što aplikacijama treće strane, na primjer Facebook, omogućuje jest pristup korisnikovim podacima kao što su ime i prezime i sl. Unutar OAuth protokola prema RFC-6749 (2012) definirane su četiri ključne uloge, a to su:

1. *vlasnik resursa* (eng. resource owner) – entitet (krajnji korisnik ili aplikacija treće strane) koji ima pristup i vlasništvo nad resursom
2. *poslužitelj resursa* (eng. resource server) – poslužitelj na kojem se nalazi zaštićen resurs
3. *klijent* (eng. client) – aplikacije bilo koje vrste, koja pristupa resursu u ime vlasnika resursa
4. *autorizacijski poslužitelj* (eng. authorization server) – poslužitelj koji izdaje i upravlja žetonima

RHCF 6749 (2012) također definira četiri načina dobivanja ovlasti (eng. authorization grant) nad resursom, no oni će biti samo spomenuti. Dakle, dostupni načini dobivanja ovlasti su:

1. Autorizacijski Kod (eng. Authorization Code)
2. Implicitna dodjela (eng. Implicit grant)
3. Vjerodajnice Lozinke Vlasnika Resursa (eng. Resource Owner Password Credentials)
4. Vjerodajnice Klijenta (eng. Client Credentials)

Jednostavan prikaz osnovnog načina dohvaćanja resursa prikazan je na sljedećoj slici i bit će opisan u nastavku. Na dijagramu su vidljivi svi navedeni sudionici i tijek njihove međusobne komunikacije. Postupak inicira klijent (aplikacija) koji traži autorizaciju od vlasnika resursa (autorizacija može biti izvršena i preko autorizacijskog poslužitelja). Nakon što je uspješno autoriziran, klijent dobiva ovlast ovisno o korištenom načinu dobivanja ovlasti. Koristeći dobivenu ovlast, klijent zatim traži pristupni žeton od autorizacijskog poslužitelja. Autorizacijski server provjerava priloženu ovlast od strane klijenta te ako je ispravna, vraća mu pristupni žeton. Pristupni žeton klijent zatim može koristiti pri slanju zahtjeva na poslužitelj resursa, kako bi imao pravo na zaštićen resurs (Boyd R., 2012; RFC-6749, 2012) Pristupni se žeton, kao i JWT, u pravilu šalje koristeći već spomenutu shemu autentifikacije nositelja, unutar zaglavlja HTTP zahtjeva.

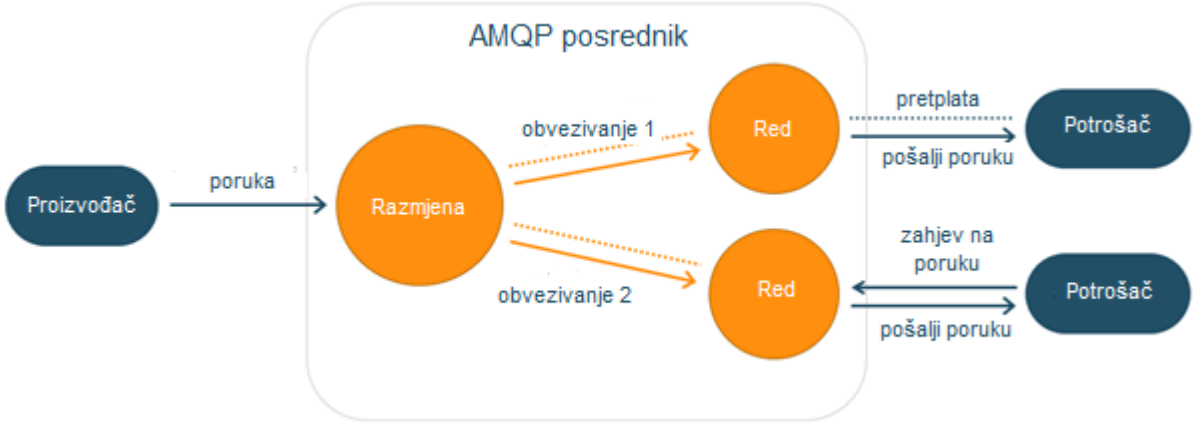


Slika 32 Osnovni OAuth proces dohvaćanja zaštićenog resursa

Također, vrijedno je spomenuti da se uz pristupni žeton, često koristi i žeton za osvježavanje (eng. refresh token). Razlog tome su pristupni žetoni koji unutar sebe nose informaciju o isteku valjanosti, tj. imaju rok trajanja kako bi se izbjegla moguća zlouporaba. Valjanost pristupnog žetona se prema tome produžuje slanjem zahtjeva koji sadrži žeton za osvježavanje na autorizacijski poslužitelj. Isto tako, autorizacijski poslužitelji često nude opciju ručnog deaktiviranja pristupnog žetona ručnim slanjem zahtjeva za deaktivaciju. Nakon što klijent završi s korištenjem pristupnog žetona, da bi se izbjegao potencijalan sigurnosni propust, žeton je poželjno deaktivirati (Boyd R., 2012).

7.2. RabbitMQ

Predstavnik komunikacije preko poruka, a i tehnologija koja će u tu svrhu biti korištena u praktičnom dijelu rada, je RabbitMQ posrednik za razmjenu poruka (eng. message broker). RabbitMQ je jedan od poznatijih i raširenijih posrednika koji koristi napredni protokol reda čekanja poruka (eng. Advanced Message Queuing Protocol – AMQP). AMQP je dokazao bolje performanse i skalabilnost, koji su ključni u sustavima, a pogotovo onim s mikroservisnom arhitekturom. AMQP posrednik sastoji se od tri komponente: razmjena (eng. exchange), red poruka (eng. message queue) i obvezivanje (eng. binding). AMQP struktura i način rada prikazani su na sljedećoj slici.



Slika 33 AMQP posrednik (Izvor: Smartbear, bez dat.)

Poslužitelj ili, u ovom slučaju, proizvođač (eng. publisher) šalje poruku koja dolazi do razmjene koja ju prosljeđuje na jedan od redova poruka. Uloga obvezivanja je da jednoznačno poveže pojedini red poruka s razmjenom, koristeći specifičan ključ za povezivanje. Ovime se postiže to da pri zaprimanju poruke, razmjena zaprima i ključ te prema tome zna na koji red poruka treba proslijediti dobivenu poruku (RabbitMQ, bez dat..; Akshay K. B. i Chaitra B. H., 2020).

RabbitMQ je vrlo pouzdan zato što je isporučen i rad vrši neovisno o mikroservisima pa eventualni problemi unutar mikroservisa ne utječu na razmjenu poruka ostatka sustava. Neovisna isporuka čini ga i neovisnim o platformi na kojoj se koristi te je vrlo čest slučaj da dva mikroservisa koji komuniciraju, nisu pisani u istom programskom jeziku. Također, odlikuje visokom razinom dostupnosti jer se koristi veći broj klastera koji sadrže kopije redova poruka te time osiguravaju sigurnosne kopije podataka. Budući da je temeljen na AMQP-u, RabbitMQ ima sposobnost usmjeravanja poruka preko razmjene i zahvaljujući različitim vrstama razmjene, koje će biti opisane u nastavku, nudi vrlo fleksibilno rješenje za preusmjeravanje poruka (Akshay K. B. i Chaitra B. H., 2020).

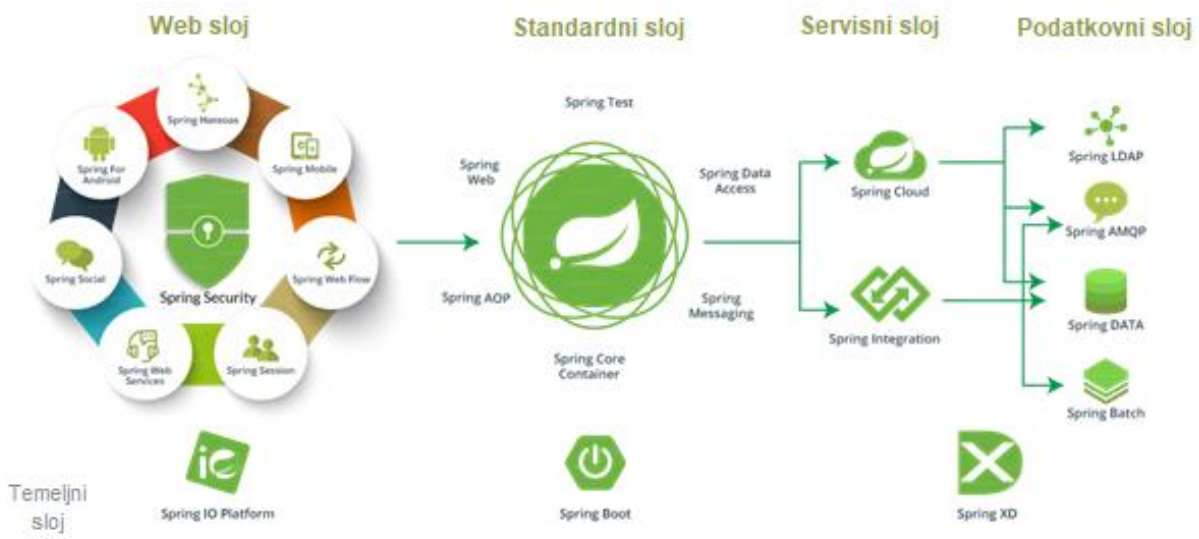
Prema službenoj RabbitMQ dokumentaciji (RabbitMQ, bez dat.), postoje četiri vrste razmjena koje su dostupne unutar ovog posrednika za razmjenu poruka, a to su:

1. *izravna razmjena* (eng. direct exchange) – uspoređuje pristigli ključ s ključevima vezanim uz redove i šalje poruku na red koji ima odgovarajući ključ
2. *raširena razmjena* (eng. fanout exchange) – poruka se prosljeđuje na svaki povezani red čekanja, bez obzira na ključ
3. *razmjena tema* (eng. topic exchange) – uspoređuje pristigli ključ sa zadanim izrazom te šalje poruku u redove na kojima je izraz zadovoljen
4. *razmjena zaglavlja* (eng. headers exchange) – preusmjerava na temelju zadovoljenih vrijednosti iz zaglavlja poruke

S opcijom korištenja Reprezentacijskog prijenosa stanja (eng. Representational state transfer – REST) i poruka, potrebno je odlučiti koju od opcija kada iskoristiti. Prva glavna razlika je način komunikacije koji je sa strane poruka asinkron i omogućuje da poruke „pričekaju“ da ih se pročita. Time se postiže neovisnost pošiljatelja (proizvođača) i primatelja (potrošača) poruka, tj. zahtjeva. REST servisi će često imati loše performanse u slučaju velikog broja istovremenih zahtjeva, dok to za vrstu AMQP posrednika poruka nije slučaj. AMQP posrednici mogu podnijeti veliku propusnost te se prema tome lakše rješavaju pitanja vezana uz skaliranje sustava (Akshay K. B. i Chaitra B. H., 2020).

8. Spring programski okvir

S obzirom na to da je praktični dio ovog rada usredotočen na realizaciju kroz Spring Cloud programski okvir, u ovom će se poglavlju predstaviti „obitelj“ Spring programskih okvira koji su u današnje vrijeme praktički postali standard za Web razvoj u programskom jeziku Java. Sve ono što spada pod naziv „Spring programski okvir“ nemoguće je pokriti u kratkim crtama upravo zato što Spring pokriva velik broj projekata koji se stalno razvijaju. Svaki od ovih projekata specijaliziran je za određen aspekt, kao npr. sigurnost, podaci, Web aplikacije. Prema ovakvom pristupu razvojnog okvira vidi se da je građen modularno i omogućuje korisnicima da po potrebi uključuju projekte prema želji (Spring Projects, bez dat.). Teško je u jednoj slici obuhvatiti sve projekte, no sljedeća slika pomoći će pri vizualizaciji toga koliko je „svijet Spring-a“ zapravo velik.



Slika 34 Spring ekosistem (Izvor: Chand S., 2022)

S teorijske strane, Spring predstavlja lagano (eng. lightweight) rješenje za izgradnju aplikacija spremnih za poduzeća (eng. enterprise-ready) koja je brzo postala zamjena klasičnoj Java Enterprise Edition (Java EE). Spring se prema tome temelji na uzorku injekcije ovisnosti (eng. dependency injection) koji omogućuje realizaciju srži ovog programskog okvira, a to je inverzija kontrole (eng. Inversion of Control – IoC). IoC kontejner je odgovoran za instanciranje, konfiguriranje i sastavljanje objekata, tj. zrna (eng. bean), kao i vođenje brige o njihovom životnom ciklusu. Ovime je implementacija odvojena od realizacije i omogućuje laku izmjenu implementacije u kodu čime se podiže modularnost cijelog programa (Spring, bez dat.). U nastavku će pobliže biti objašnjeni koncepti koje koristi Spring razvojni okvir.

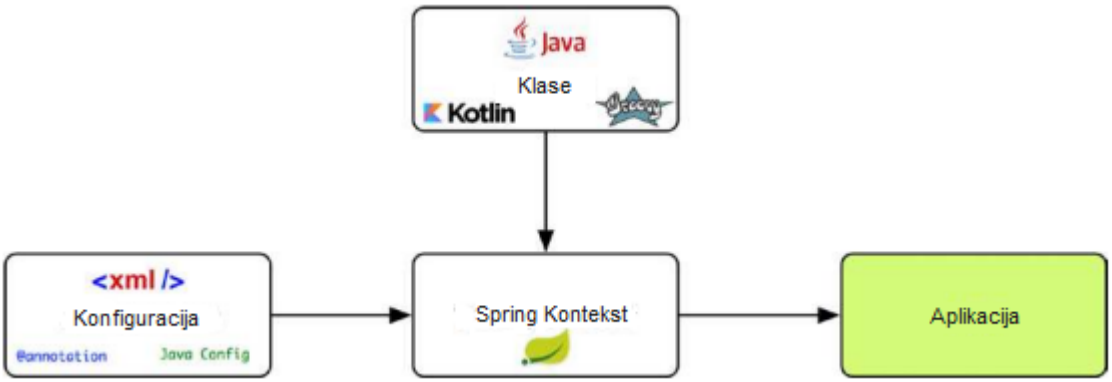
8.1. Koncepti Spring programskog okvira

Na početku ovog poglavlja važno je spomenuti da će se svi primjeri, konfiguracije i slično obraditi na oba načina, koristeći anotacije i preko XML-a. Osnovni koncepti nešto će se više koristiti XML-om, dok će anotacije primarno biti korištene u sklopu Spring Boot projekta, budući da on na neki način proširuje i automatizira osnove Spring programskog okvira. Nešto više riječi o samom Spring Boot projektu bit će u jednom od narednih poglavlja.

Dakle, ako se kreće od samih osnova Spring programskog okvira, ako se zna samo što on radi, tada programski okvir ostaje crna kutija, što za dobre programere ne bi trebalo biti zadovoljavajuća razina poznavanja područja rada. Prema tome, Gutierrez F. (2019) navodi nekoliko osnovnih principa Spring programskog okvira, koji pružaju nešto detaljniji uvid u sam programski okvir.

- *Omogućavanje izbora na svakoj razini* – Spring pruža mogućnost programerima da što kasnije odgode odluke o dizajnu, primjerice pružatelji perzistencije mogu se u bilo kojem trenutku izmijeniti kroz konfiguraciju, bez promjene programskog koda
- *Prilagođavanje različitih perspektiva* – Spring uvelike podržava fleksibilnost razvoja i ne postavlja granice kako bi stvari trebale funkcionirati; prema tome podržava velik niz različitih primjena Spring aplikacija
- *Održavanje kompatibilnosti unatrag* – prilikom razvoja i novih verzija, Spring nastoji održati kompatibilnost sa starijim verzijama, o čemu govori i samo nekolicina prijelomnih promjena između verzija, kao i pomno odabrane JDK verzije i biblioteke treće strane koje se koriste
- *Briga o dizajnu API-ja* – razvojni tim koji radi na Spring projektima ulaže puno vremena i truda u izradu API-ja koji su intuitivni do te razine da će ih biti moguće koristiti kroz više verzija i godina
- *Postavljanje visokih standarda kvalitete koda* – razvojni tim također vodi brigu o kvalitetnoj, ažurnoj i preciznoj dokumentaciji (eng. Javadocs) te je zbog toga Spring jedan od rijetkih projekata koji se mogu pohvaliti „čistom“ strukturom koda i nepostojanjem cirkularnih ovisnosti

Spring aplikacija koristi obične Java klase (eng. Plain Old Java Objects – POJO) zbog čega ju je jednostavno izgraditi i proširivati. Ključan korak u kreiranju Spring aplikacija je postavljanje konfiguracije koja POJO klase pretvara u Spring zrna. Na sljedećoj slici prikazan je Spring kontekst koji, koristeći POJO klase i konfiguraciju, kreira Spring zrna koja u suštini pokreću Spring aplikaciju.



Slika 35 Spring kontekst (Izvor: Gutierrez F., 2019)

Dakle, Spring aplikacija sastoji se od velikog broja Spring zrna (zapravo POJO klase) koja predstavljaju njezine komponente od kojih svatko ima određen zadatak, tj. svrhu zbog koje je kreirano. Spring kontekst aplikacije (eng. application context), koji se češće naziva samo Spring kontekstom, predstavlja kontejner koji sadrži sve te komponente (zrna). Osim toga, Spring kontekst je zadužen za instanciranje, konfiguriranje i sastavljanje samih zrna. Spring kontejner „zna“ što i kako treba učiniti upravo prema konfiguraciji koju je potrebno pripremiti od strane programera. Kao što je već spomenuto, ta konfiguracija može biti u obliku XML datoteka, Java anotacija ili čak kroz sam Java programski kod. Korištenje XML-a je tradicionalan način postavljanja konfiguracije, dok je korištenje anotacija za konfiguriranje (eng. annotation-based configuration) predstavljeno u Spring verziji 2.5, a konfiguracija kroz Java programski kod (eng. Java-based configuration) u Spring verziji 3.0. (Walls C., 2018.; Spring, bez dat.)

U Springu se ovaj kontejner naziva još i IoC kontejnerom ili IoC načelom. IoC je zapravo dobro poznati uzorak injektiranja ovisnosti te je upravo to način na koji Spring kontejner „veže“ zrna. Umjesto da komponente stvaraju i upravljaju životnim ciklusom drugih zrna o kojima ovise, IoC kontejner injektira zrna koja sadrži u zrna koja ga zahtijevaju. Uobičajeni načini realizacije su preko konstruktora i metoda pristupa (eng. setter method). Bez ulaženja u detalje i krajnje slučajeve, sljedeći isječci programskog koda prikazuju definiranje konfiguracije kroz XML te klasu koja sadrži injekciju putem konstruktora i metode pristupa.

```
<bean id="slusacLetova" class="foi.org.SlusacLetova" >
<constructor-arg ref="letoviServis" />
<property name="porukeServis" ref="porukeServis" />
</bean>
```

```
public class SlusacLetova {
      private LetoviServis letoviServis;
      private PorukeServis porukeServis;

      //injektiranje konstruktorom
      public SlusacLetova (LetoviServis letoviServis) {
         this.letoviServis = letoviServis;
     }

      //injektiranje koristeci setter
      public void setPorukeServis(PorukeServis porukeServis) {
         this. porukeServis = porukeServis;
     }
      //ostatak poslovne logike…
}
```

Korištenjem konfiguracije kroz Java programski kod, preko XML konfiguracije postiže se veća sigurnost kada se govori o tipovima podataka i mogućnosti refaktoriranja programskog koda. Primjerice, ako se unutar XML-a navede Spring zrno pogrešnog tipa u usporedbi s onim koje se traži, doći će do iznimke prilikom pokretanja Spring aplikacije te će se ista srušiti. Također, promjene u programskom kodu, zahtijevaju da iste budu učinjene u odgovarajućim XML datotekama, što rezultira duplom količinom posla. No zahvaljujući razvoju Spring programskog okvira, postoji nešto što se naziva automatskom konfiguracijom i predstavlja mogućnog Spring programskog okvira da samostalno, bez potrebne konfiguracije konfigurira komponente, tj. Spring zrna. Ova se mogućnost temelji na dvjema tehnikama koje Spring naziva automatsko vezivanje (eng. autowiring) i skeniranje komponenti (eng. component scan). Koristeći skeniranje komponenti, Spring može samostalno otkriti komponente koje se nalaze na putanji aplikacije (eng. application classpath) i registrirati ih kao zrna u Spring kontekstu. S druge strane, automatsko vezivanje, kao što naziv i govori, automatski injektira komponente u zrna koja ih zahtijevaju (Walls C., 2018). Ovo su koncepti koji se događaju u pozadini Spring programskog okvira i teško ih je demonstrirati, no sljedeći isječak koda prikazuje upravo način na koji se vrši automatsko vezivanje, kao još jedan oblik injektiranja ovisnosti u zrna.

```
      @Autowired
      private LetoviServis letoviServis;
```

Vidljivo je da se ovdje koristi anotacija *@Autowired*, koja odgovara nazivu koncepta. Istu je moguće koristiti i kod korištenja konstruktora, no u novijim verzijama Spring programskog okvira za time nema potrebe jer Spring to prepoznaje automatski. Dobro je spomenuti da se korištenje injektiranja kroz konstruktor smatra najboljom praksom zbog razine sigurnosti koju pruža pri injekciji te bi se trebalo nastojati istu koristiti pored ostale dvije opcije.

Kada je riječ o automatskoj konfiguraciji, važno je osvrnuti se na Spring zrna, a posebice pravila njihovog imenovanja kojih se, kao i svih konvencija, uvijek preporučuje pridržavati iz više razloga. No, prije toga je potrebno približiti pojam samih zrna unutar Spring programskog okvira. Općenito, već je više puta spomenuto da IoC kontejner sadrži i upravlja jednim ili više zrnima unutar Spring aplikacije. Prema službenoj dokumentaciji (Spring, bez dat.), konkretni metapodaci koji dolaze uz svako zrno, iz konfiguracija, su:

- *Puno ime klase* – sadrži ime paketa u kojima se stvarna klasa (implementacija) definiranog zrna nalazi, kao i njezino ime
- *Elementi konfiguracije ponašanja* – navode kako se zrno treba ponašati unutar kontejnera; obuhvaća njegov opseg (eng. scope), povratne pozive životnog ciklusa (eng. lifecycle callbacs) itd.
- *Reference na ostala zrna* – nazvane još i ovisnostima (eng. dependency) ili suradnicima (eng. collaborator), predstavljaju druga zrna potrebna za obavljanje posla
- *Ostala konfiguracija za postavljanje* – primjerice svojstva kao što je ograničenje veličine skupa (eng. pool) veza kojim zrno upravlja moraju biti postavljena nakon kreiranja samog zrna

Ove metapodatke kasnije se prevodi u niz svojstava koja zajedno čine definiciju jednog zrna. Ta svojstva redom su:

1. Klasa
2. Naziv
3. Opseg (eng. scope)
4. Argumenti konstruktora
5. Svojstva
6. Način automatskog vezivanja (eng. autowiring mode)
7. Način lijene inicijalizacije (eng. lazy initialization)
8. Metoda inicijalizacije (eng initialization method)
9. Metoda destrukcija (eng. destruction method)

Svako zrno u pravilu ima jedinstven identifikator unutar kontejnera koji ga sadrži. Pravilo imenovanja kojeg se Spring programski okvir drži je isti onome koji se koristi u imenovanju kod programskog jezika Java. Dakle, radi se o nazivima koji počinju malim slovom i svaka nova riječ ima veliko početno slovo (eng. camel-case), kao primjerice `letoviServis` i `porukeServis`. Pridržavanjem ovih pravila i postizanjem konzistentnih naziva unutar projektne konfiguracije, ona postaje jednostavnija za čitanje i razumijevanje.

Ova pravila posebno je važno poznavati kada se Spring programskom okviru prepušta posao automatske konfiguracije. Naime, prilikom skeniranja komponenti i dodavanja odgovarajućih zrna u kontejnere, Spring će uzeti ime klase i primijeniti navedena pravila. Budući da nazivi klasa počinju velikim slovom, prvo će se slovo pretvoriti u malo te će nazivi zrna za klase izgledati kao u sljedećem primjeru:

Tablica 2 Primjeri imenovanja zrna

Naziv klase	***Naziv zrna***
LetoviServis	letoviServis
KonfiguracijaPorukaZaLetove	konfiguracijaPorukaZaLetove

Ipak, u većim i kompleksnijim sustavima može doći do potrebe za više identifikatora kao načinom referenciranja zrna. Ako je potrebno zadovoljiti neku ovisnost, čije ime ne odgovara postojećem identifikatoru zrna, postoji mogućnost dodavanja tzv. aliasa na definiciju zrna. Ovih aliasa može biti više, a pomažu kontejneru da pravilno zadovolji ovisnosti ostalih zrna. Dodavanje aliasa na prethodni XML primjer bilo bi odrađeno na sljedeći način. U primjeru za zrno s identifikatorom `slusacLetova` dodaje se alias pod nazivom `slusacLetovaAlias` te će Spring, kada naiđe na potrebu injektiranja ovisnosti pod nazivom `slusacLetovaAlias`, znati o kojem se zrnu radi.

```
<alias name="slusacLetova" alias="slusacLetovaAlias" />

<bean id="slusacLetova" class="foi.org.SlusacLetova" >
<constructor-arg ref="letoviServis" />
<property name="porukeServis" ref="porukeServis" />
</bean>
```

Posljednji i vrlo važan dio za poznavanje, kada se govori o osnovama Spring programskog okvira i zrna koji čine njegovu osnovu, jest njihov opseg. Svako zrno koje se nalazi u kontejneru ima određen opseg prema kojem je ono prvenstveno i kreirano. Moguće vrste opsega zrna navedene su u sljedećoj tablici.

Tablica 3 Opsezi zrna

Opseg	***Opis***
Singleton	Jedna instanca za svaki IoC kontejner
Prototype	Jedna instanca za svaku ovisnost, tj. potrebu injektiranja
Request	Jedna instanca kreirana za svaki HTTP zahtjev
Session	Jedna instanca kreirana za svaku HTTP sesiju
Application	Jedna instanca za životni ciklus konteksta servleta (ServletContext klasa)
Websocket	Jedna instanca za životni ciklus sesije Web utičnice (eng socket) (WebSocket klasa)

(Izvor: Spring, bez dat.)

Zadani opseg prema kojem IoC kontejner kreira zrna jest s*ingleton*, tako da u slučaju kada ništa nije navedeno pri definiciji zrna, koristi se taj opseg. Singleton je prema tome i najčešće korišten opseg te je dostupan u svim oblicima Spring aplikacija, baš kao i *prototype* opseg. Ograničenje dostupnosti na samo web aplikacije, u kojima je Spring kontekst „svjestan weba“ (eng. web-aware), primijenjeno je na sve ostale opsege (request, session, application i websocket). To ima smisla budući da se uvjeti za kreiranje novih instanci zrna odnose na koncepte (HTTP zahtjev, sesija…) koji postoje samo unutar weba, tj. web aplikacija. Prethodno korišten primjer definiranja zrna kroz XML, proširen opcijom koja sadrži naveden opseg zrna izgledao bi na sljedeći način (vrijednost opsega može odgovarati bilo kojem od navedenih u tablici).

```
<alias name="slusacLetova" alias="slusacLetovaAlias" />

<bean id="slusacLetova" class="foi.org.SlusacLetova" scope="singleton">
<constructor-arg ref="letoviServis" />
<property name="porukeServis" ref="porukeServis" />
</bean>
```

8.2. Spring Boot

Spring se kroz vrijeme znatno razvijao, a najteži dio pri korištenju Spring razvojnog okvira u njegovim počecima zasigurno je bilo postavljanje projekta. Brojna konfiguracija koja bi morala biti pravilno postavljena kroz razne XML (eng. Extensible Markup Language) datoteke u projektu, uzrokovala bi glavobolju svakom novom korisniku Spring-a. Dakle, svako uključivanje novog Spring projekta bi, uz inicijalne postavke, tražilo neku vrstu konfiguracije kroz XML. Kako bi se rad s XML datotekama smanjio, ubrzo su se javile i anotacije koje bi u suštini omogućile da se ista stvar napravi kroz Java kod, no i dalje je bilo potrebno proći kroz mukotrpno postavljanje aplikacije. Sve do kada se nije javio Spring Boot projekt čiji je cilj pojednostaviti i korisnicima omogućiti što brže postavljanje i podizanje Spring aplikacija, od čega dolazi i sam naziv (eng. boot up). Spring Boot praktički uklanja potrebu za ikakvom konfiguracijom pri inicijalnom pokretanju te netko tko nije koristio Spring bez Spring Boot projekta ne može ni zamisliti količinu konfiguracije koja je odrađena u pozadini. Spring Boot se nekad čini magičnim, a neki od osnovnih trikova prema Walls C. (2018) su:

- *automatska konfiguracija* – već spomenuto, Spring Boot pruža osnovnu konfiguraciju za većinu projekata
- *početne ovisnosti* – Spring Boot dodaje potrebne biblioteke prema potrebi kroz tzv. Spring Boot startere (eng. starters) koji pakiraju više ovisnosti u samo jednu
- *sučelje naredbenog retka* – aplikacije se po želji mogu razvijati bez tradicionalnog kompiliranja
- *Pokretač (eng. The Actuator)* – omogućuje analizu Spring aplikacije iznutra, kroz razne statističke podatke, za vrijeme izvođenja

Kada je riječ o automatskoj konfiguraciji, ona u Spring Boot projektu dolazi u obliku `@SpringBootApplication` anotacije te predstavlja glavnu komponentu jedne Spring Boot aplikacije. Ova anotacija zapravo obuhvaća tri druge anotacije, a to su:

1. `@Configuration` – govori da anotirana klasa sadrži definicije zrna
2. `@ComponentScan` – govori da svi paketi ispod onog u kojem se nalazi anotirana klasa (uključujući i taj paket) trebaju biti skenirani za pronalazak zrna, zato se često radi o *main* klasi kao anotiranoj koristeći ovu anotaciju; moguće je eksplicitno navesti iznimke skeniranja
3. `@EnableAutoConfiguration` – govori da Spring treba nastojati provoditi automatsku konfiguraciju za zrna koja nisu eksplicitno definirana i injektirati ih prema prethodno navedenim pravilima

Spomenute iznimke skeniranja mogu se odnositi na cijele pakete ili konkretne klase te je kao parametar anotacije potrebno proslijediti listu naziva takvih klasa ili paketa (koriste se regex izrazi). Isto tako je moguće eksplicitno navesti bazne pakete (eng. base package), tj. njihove pune nazive, koji će biti skenirani i dodavati složenije filtere. Osim toga, do sada je kreiranje zrna bilo prikazano samo kroz XML, a analogna anotacija `<bean/>` oznaci jest `@Bean` te omogućuje da se na razini metode pruži potrebno zrno. Budući da se radi o metodi, ovaj pristup može biti prigodan kada se prije instanciranja zrna trebaju podesiti neki dodatni parametri na objektu (Gutierrez F., 2019.). Sljedeći primjer prikazuje pokretanje Spring Boot aplikacije korištenjem `@SpringBootApplication` anotacije uz isključivanje jedne konkretne klase, navođenje bazne klase za skeniranje i definiranje novog zrna na razini aplikacije (napomena da nazivi parametara anotacija variraju ovisno koristi li se primjerice `@SpringBootApplication` ili `@ComponentScan` anotacija).

```
@SpringBootApplication(
    //bazni paketi koji će se skenirati
      scanBasePackages="foi.org.skeniran.paket",
      //klasa koja neće biti skenirana, tj. bit će zanemarena
      exclude=ZanemarenaKlasa.class
      )
public class DemoAplikacija {
      public static void main(String[] args){
            SpringApplication.run(DemoAplikacija.class, args);
      }
      //definirano novo zrno na razini metode, uz potrebnu konfiguraciju
      @Bean
    RabbitTemplate rabbitTemplate(ConnectionFactory connectionFactory) {
        RabbitTemplate predlozak =
                        new RabbitTemplate(connectionFactory);
        predlozak.setReplyTimeout(3000);
        return predlozak;
    }
}
```

U nastavku će se navesti, pojasniti i demonstrirati uporaba još nekih od češće korištenih anotacija, prema Spring Boot dokumentaciji (Spring Boot, bez dat.). Spomenuto je automatsko skeniranje paketa i pronalaženje zrna, no dolazi do pitanja kako Spring programski okvir zna što je zrno, a što ne (u slučaju kad ne koristimo XML). U ovu svrhu javlja se anotacija @Component, kojom je potrebno anotirati klasu koja predstavlja zrno, kako bi ju automatsko skeniranje registriralo kao zrno unutar kontejnera. Osim @Component, postoje slične anotacije, kao što su primjerice `@Service`, `@Repository` i `@Controller`. U suštini, sve od ovih četiriju anotacija bit će uzete u obzir pri automatskom skeniranju, dok neke od njih pružaju dodatne informacije ili postavke te su namijenjene za korištenje u specifičnom kontekstu. Primjerice, `@Service` označava da određeno zrno sadrži poslovnu logiku i u pravilu se koristi na poslovnom sloju aplikacije, dok je `@Repository` namijenjen za podatkovni sloj, a `@Controller` se koristi za kontrolere.

Postoji slučaj kada unutar aplikacije dolazi do potrebe za više zrna istog tipa (npr. razlikuju se u određenoj konfiguraciji) te dolazi do problema kada je potrebno injektirati ovisnost jer Spring kontejner ne zna koje konkretno zrno treba pružiti. Ova situacija će kao rezultat vratiti iznimku i aplikacija neće raditi. Budući da se ne može očekivati potreba za samo jednim zrnom istog tipa u svakoj mogućoj Spring aplikaciji, za ovaj problem postoji rješenje u obliku `@Qualifier` anotacije. Koristeći ovu anotaciju, pružajući joj identifikator zrna kao parametar, moguće je jednoznačno navesti koje zrno je potrebno injektirati na mjesto tražene ovisnosti. Ova anotacija može se koristiti pri bilo kojoj vrsti injekcije, tj. na razini metode, svojstva ili parametra, a sljedeći primjer prikazuje korištenje iste pri injekciji koristeći konstruktor.

```
@Component
public class MojeZrno {
      private LetoviServis letoviServis1;
      private LetoviServis letoviServis2;

      public MojeZrno(@Qualifier("let1") LetoviServis letoviServis1,
                  @Qualifier("let2") LetoviServis letoviServis2) {
            //ostatak konstruktora…
    }
}
```

Osim `@Qualifier` anotacije, postoji još jedna opcija za definiranje injektiranog zrna u slučaju kada ih postoji više. Ta anotacija jest `@Primary`, no radi nešto drugačije od `@Qualifier` anotacije i reklo bi se da ima nešto specifičniju primjenu. Koristeći `@Primary` anotaciju na definiciji zrna, označit će to zrno kao zadano (eng. default). Prema tome, kada se javi slučaj u kojem postoji više zrna istog tipa za traženu ovisnost, ukoliko nije eksplicitno navedeno zrno (npr. koristeći `@Qualifier` anotaciju), a postoji zrno anotirano s `@Primary` anotacijom, ono će biti injektirano i neće se javiti iznimka.

Zanimljivost koja se ne tiče samo Spring programskog okvira, već dodiruje i Jakarta EE anotacije, vezana je uz injektiranje. Već je spomenuta `@Autowired` anotacija, no postoje još dvije anotacije koje se također javljaju pri injekciji ovisnosti te je dobro izdvojiti razlike koje postoje između njih. Druge dvije anotacije su `@Resource` (`javax.annotation.Resource`) i `@Inject` (`javax.inject.Inject`). Ono što čini razliku (i sličnosti) između ovih anotacija jest prioritet svojstva prema kojem se vrši injekcija. Dakle, injektiranje može biti izvršeno prema nazivu, tipu ili kvalifikatoru, a sljedeća tablica prikazuje navedene anotacije i redoslijed prioriteta injektiranja za svaki (Baeldung, 2022).

Tablica 4 Usporedba prioriteta injektiranja za anotacije

Prioritet	*@Resource*	*@Inject*	*@Autowire*
1.	naziv	tip	tip
2.	tip	kvalifikator	kvalifikator
3.	kvalifikator	naziv	naziv

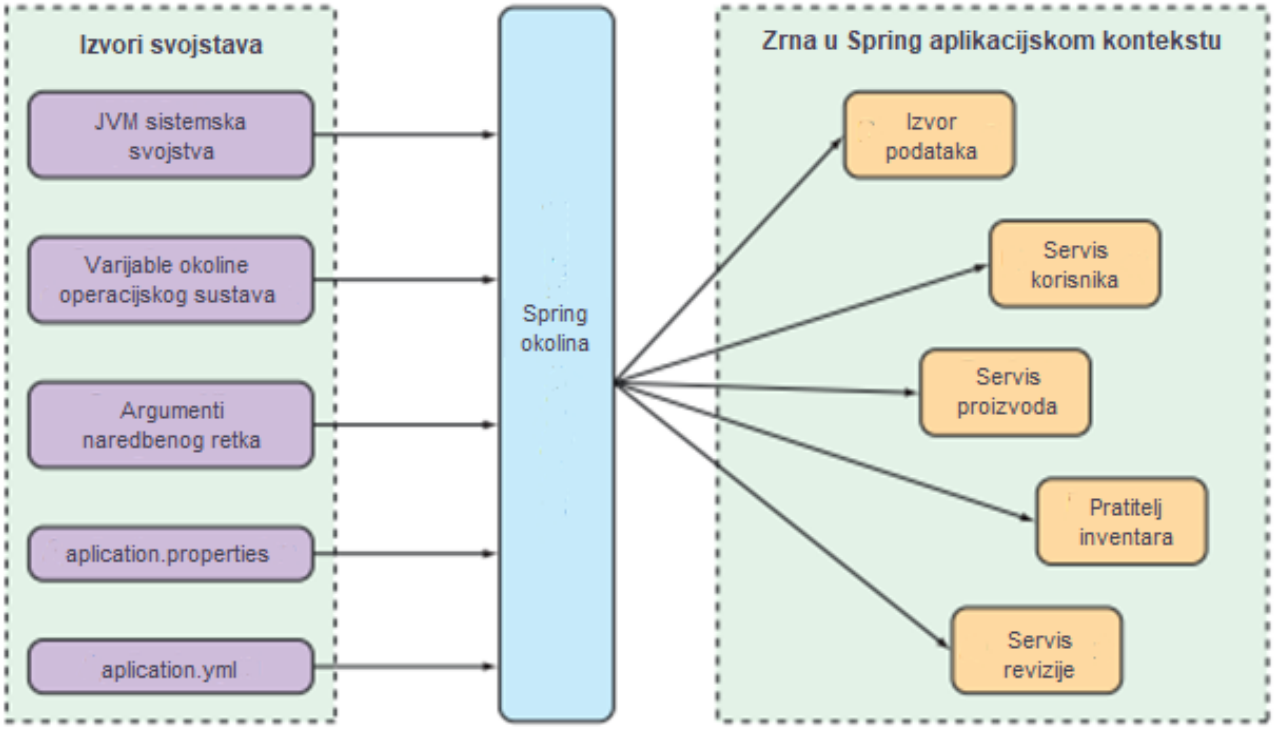
Iz prethodne tablice vidljivo je da `@Inject` i `@Autowire` poštuju iste putanje izvršavanja pri injektiranju te je jedina razlika to što je `@Autowire` dio Spring programskog okvira. Anotacija `@Resource` prioritizira naziv pa tek nakon njega tip i kvalifikator. Sve tri anotacije moguće je koristiti na razini metode pristupa (eng. setter) i svojstava.

Posljednja karakteristika Spring Boot programskog okvira bit će vezana uz svojstva i njihovo korištenje u kontekstu Spring Boot aplikacija. Korištenje eksterne konfiguracije često je i potrebno, ne samo u Spring aplikacijama, no Spring jednostavnost rukovanja istima podiže na višu razinu.

Spring Boot nudi nekoliko mjesta na kojima konfiguracijska svojstva mogu biti spremljena, a to su (Gutierrez F., 2019):

- *application.properties* datoteka
- *application.yml* datoteka (eng. Yet Another Markup Language / YAML Ain't Markup language – YAML)
- *Varijable okoline*
- *Argumenti naredbenog retka*

Neovisno o izvoru, sva svojstva učitana su od strane Spring programskog okvira i predstavljaju jedinstven izvor informacija za sva zrna u sustavu. Sva zrna imaju mogućnost konfiguracije i korištenja učitanih svojstava za vrijeme njihovog „života“. Učitavanje svojstava iz svih izvora konfiguracije i omogućavanje njihove dostupnosti od strane okoline Spring programskog okvira prikazano je na sljedećoj slici.



Slika 36 Izvori spring svojstava (Izvor: Walls C., 2018)

Učitanim svojstvima moguće je pristupiti na nekoliko načina. Prvi način je koristeći `@Value` anotaciju i pružajući joj naziv svojstva kao vrijednost parametra. Spring Boot će direktno injektirati vrijednost svojstva, ukoliko je to moguće. Drugi način je koristeći `org.springframework.core.env.Environment` sučelje koje sadrži metode za dohvaćanje svojstava, također prema nazivu. Posljednji način koristi `@ConfigurationProperties`

anotaciju koja automatski veže vrijednosti svojstava na svojstva strukturiranih objekata (Spring Boot, bez dat,; Gutierrez F., 2019). Sljedeći primjer prikazuje korištenje svakog od navedenih načina dohvaćanja vrijednosti svojstava.

```
@Configuration
@ConfigurationProperties(prefix="konfiguracija")
public class MojaKonfiguracija {
    private int port;

    @Value("${server.naziv}")
    private String nazivServera;

    @Autowired //koristi se ovaj oblik injektiranja zbog jednostavnosti
    private Environment okolina;

    private void konfiguracijskaMetoda() {
         //poslovna logika…
        final String lozinka = okolina.getProperty("korisnik.lozinka");
        //poslovna logika…
    }
    …
}
```

Ako sljedeća svojstva postoje u bilo kojem od izvora konfiguracije, isti će biti injektirani i raspoloživi za korištenje unutar zrna u ovom primjeru. Sljedeći primjer predstavlja izvor svojstava unutar `application.properties` datoteke:

```
konfiguracija.port=8008
server.naziv=testiranje
korisnik.lozinka=tajna123
```

Kroz njezin razvoj, aplikacija se može nalaziti u više faza i okolina. Uzme li se kao primjer situacija u kojoj još ne postoji produkcijska verzija te ne postoje razvojna (eng. development) i testna okolina. Moguće je da u svakom scenariju postoje različita svojstva i zrna (primjerice početno učitavanje podataka je različito). Ako postoji zadan skup svojstava i zrna koji se izmjenjuju ovisno o okolini, Spring pruža jednostavan način za njihovu izmjenu bez

potrebe eksplicitnog navođenja naziva na svakom mjestu gdje se zrna ili svojstva koriste. Funkcionalnost koja je vezana uz ovaj slučaj su Spring profili.

Ako se radi o zrnima, osim anotacija koje označavaju da je u pitanju zrno, moguće je dodati `@Profile` anotaciju, uz naziv profila kao vrijednost parametra kako bi se definirao profil u kojem se ovo zrno treba učitati. U slučaju svojstava, potrebno je kreirati novu datoteku svojstava (ekstenziije `.properties` ili `.yml`) te ju nazvati prema sljedećem predlošku: `application-<naziv profila>.(properties|yml)`. Primjerice, datoteka svojstava za testnu okolinu imala bi naziv `application-test.properties`, a jedno zrno moglo bi biti definirano na sljedeći način.

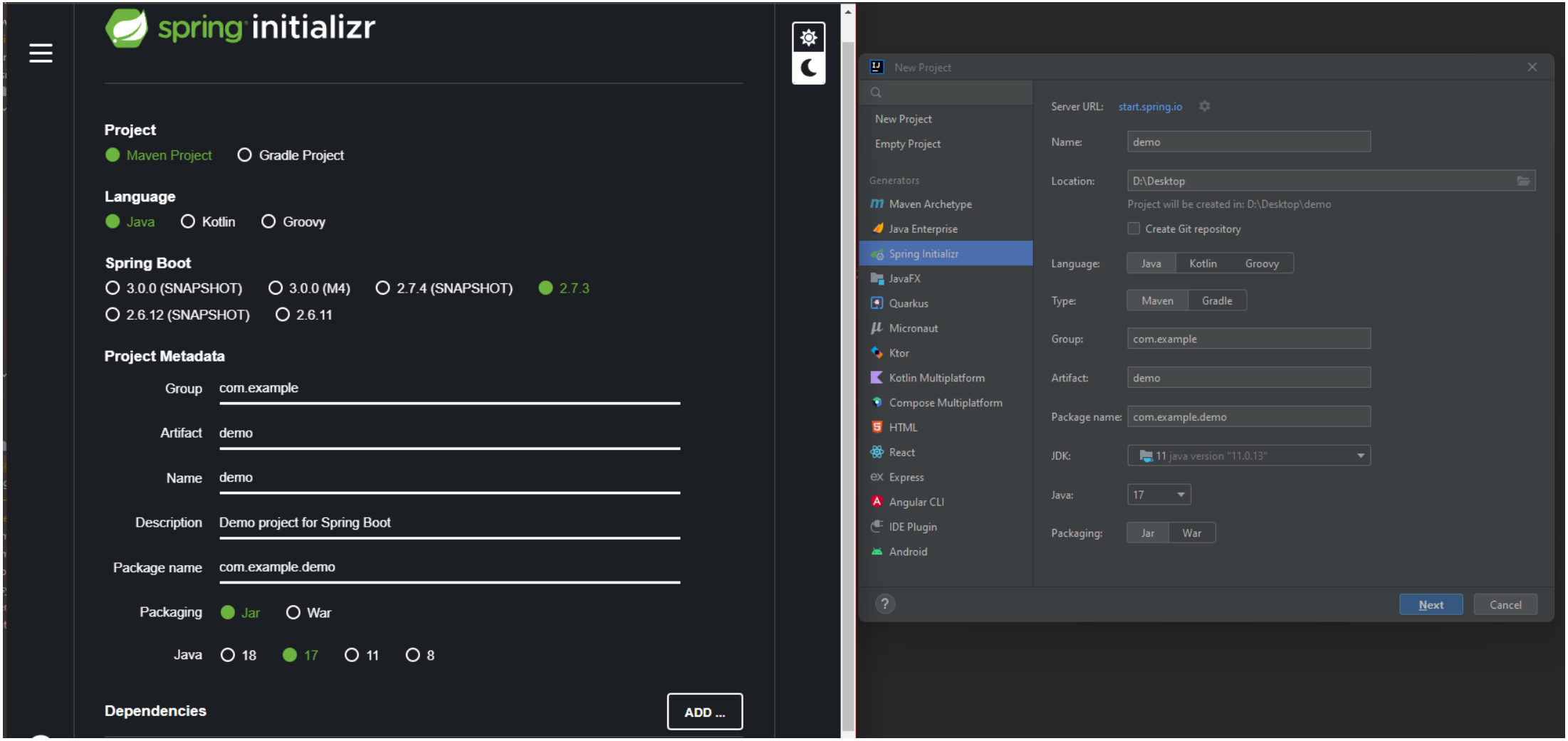
```
@Configuration
@Profile(„test“)
public class TestnaKonfiguracija implements Konfiguracija{
    //…
}
```

Kako bi Spring automatski i pravilno učitao odgovarajući profil, potrebno je u jednom od izvora svojstava konfiguracije postaviti vrijednost `spring.profiles.active` svojstva na naziv onog profila koji se želi koristiti (Spring Boot, bez dat,; Walls C., 2018).

8.3. Spring Initializr

Da Spring tim nastoji olakšati razvoj Spring aplikacija nije vidljivo samo u funkcionalnostima Spring Boot projekta. Spring nudi takozvani Spring Initializr u obliku web aplikacije i REST API-ja, koji omogućuje brzo i jednostavno postavljanje kostura Spring aplikacije uz automatski dodatne sve potrebne ovisnosti za samo one projekte koji su odabrani za korištenje. Da je Spring vrlo popularan dokazuje i opcija korištenja Spring Initializr funkcionalnosti direktno kroz razvojna okruženja, a Walls C. (2018) navodi sljedeće načine korištenja Spring Initializr funkcionalnosti kao jedne od popularnijih opcija.

- Koristeći web aplikaciju (https://start.spring.io/)
- Iz naredbenog retka
    - koristeći `curl` naredbu
    - koristeći Spring Boot sučelje za naredbeni redak
- Prilikom izrade novog projekta koristeći
    - Spring Tool Suite
    - Intellij IDEA integrirano razvojno okruženje
    - NetBeans integrirano razvojno okruženje



Slika 37 Korištenje Spring Initializr-a kroz web aplikaciju (lijevo) i IntelliJ IDEA (desno)

8.4. Spring ovisnosti

Upravljanje ovisnostima u većim i složenijim projektima jedan je od ključnih aspekata performansi projekta. S obzirom na to da postoji velik broj Spring projekata te svaki od njih zahtjeva određene ovisnosti kako bi sam po sebi mogao ispravno raditi, ručno dodavanje i upravljanje svakom od ovisnosti može uzeti puno vremena i skrenuti pažnju s važnijih dijelova projekta. Spring Boot kao rješenje ovog problema nudi funkcionalnost tzv. Spring startera. Spring starteri omogućuju jednostavno dodavanje jar datoteka u projekt i znatno ubrzavaju i olakšavaju izgradnju Spring projekata.

Alat za automatizaciju izgradnje projekta korišten u nastavku bit će Maven, no Spring starter može biti primijenjen i korištenjem npr. Gradle-a. Spring starteri predstavljaju deskriptore ovisnosti (eng. dependency descriptor) koji se, kao i ostale ovisnosti, uključuju u POM datoteke projekta. Ono što svaki od ovih deskriptora sadrži su sve potrebne ovisnosti za određen Spring projekt. Nazivi svih startera prate isti uzorak: *spring-boot-starter-** (* označava vrstu aplikacije i varira ovisno o projektu koji se uključuje. Prema tome, naziv startera za Spring Web projekt glasio bi `spring-boot-starter-web`. Budući da postoji velik broj Spring projekata, postoji i više od 30 Spring Boot startera (Baeldung, 2022). Sljedeći primjer prikazuje uključen Spring Data starter unutar POM datoteke.

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-data-jpa</artifactId>
</dependency>
```

8.5. Spring Cloud

Spring Cloud jedan je od brojnih projekata unutar Spring obitelji, a usredotočen je na razvoj mikroservisnih aplikacija. Spring Cloud pruža već gotovu podršku (eng. Out of the Box - OOTB) za brojne slučajeve i mehanizme koji se koriste u mikroservisnoj arhitekturi. Praktični dio, koji slijedi, usredotočen je na korištenje Spring Cloud programskog okvira te će demonstrirati i proći kroz većinu ovih opcija. Christudas B. (2019) kao neke od njih navodi:

- distribuirana i verzionirana konfiguracija
- registracija i otkrivanje servisa
- usmjeravanje i balansiranje opterećenja
- osigurači (eng. circuit breakers)
- distribuirano slanje poruka

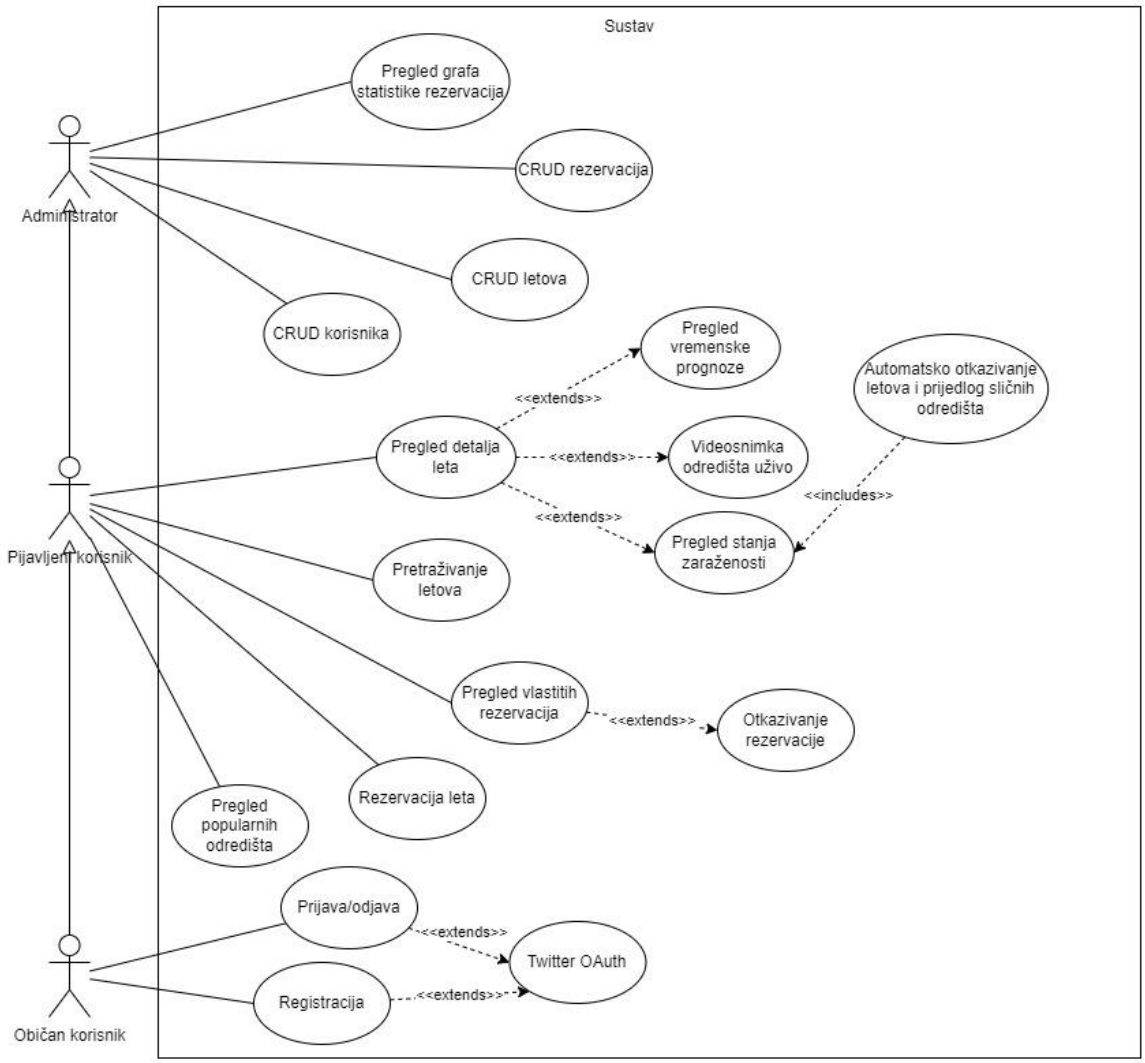
9. Praktični dio

Posljednja veća cjelina u ovom radu pokrit će implementaciju jednog mikroservisnog sustav, kroz primjenu dobrog dijela teorijski pokrivenih koncepata. Za implementaciju će se koristiti programski jezik Java i programski okvir Spring Cloud, ali i drugi potrebni okviri u obitelji Spring programskih okvira. Sljedeće poglavlje prikazat će pogled na kompletan sustav, kao cjelinu, dok će se svako od narednih poglavlja usredotočiti na jedan mikroservis i pokriti ključne elemente implementacije za svaki. Zadnje poglavlje, posvećeno Web aplikaciji koja se nalazi na korisničkoj strani, prikazat će krajnji rezultat rada sustava kao cjeline, s pogleda krajnjeg korisnika.

9.1. Razrada ideje

Ciljana Web aplikacija bavi se upravljanjem letova te će se prema tome koristiti naziv „Upravitelj letova“. Glavna svrha Upravitelja letova jest omogućiti krajnjim korisnicima pregled dostupnih letova i njihovih detaljnih informacija te njihovu rezervaciju i eventualno otkazivanje iste. Kao dodatak korisničkom iskustvu, aplikacija pruža uvid u očekivanu vremensku prognozu na odredištu, za vrijeme slijetanja, kao i uvid u trenutno vremensko stanje na odredištu kroz snimke uživo. Aplikacija također uzima u obzir epidemiološko stanje te pri visokoj razini zaraženosti otkazuje rizične letove.

U Upravitelju letova postoje dvije vrste korisnika, a to su običan korisnik i administrator. Prijava i registracija u aplikaciju moguće su klasično pomoću forme u aplikaciji ili koristeći OAuth prijavu preko Twitter-a. Administrator također ima pristup CRUD operacijama nad nekim od osnovnih resursa, a sve mogućnosti aplikacije prikazane su na sljedećem dijagramu slučajeva korištenja.

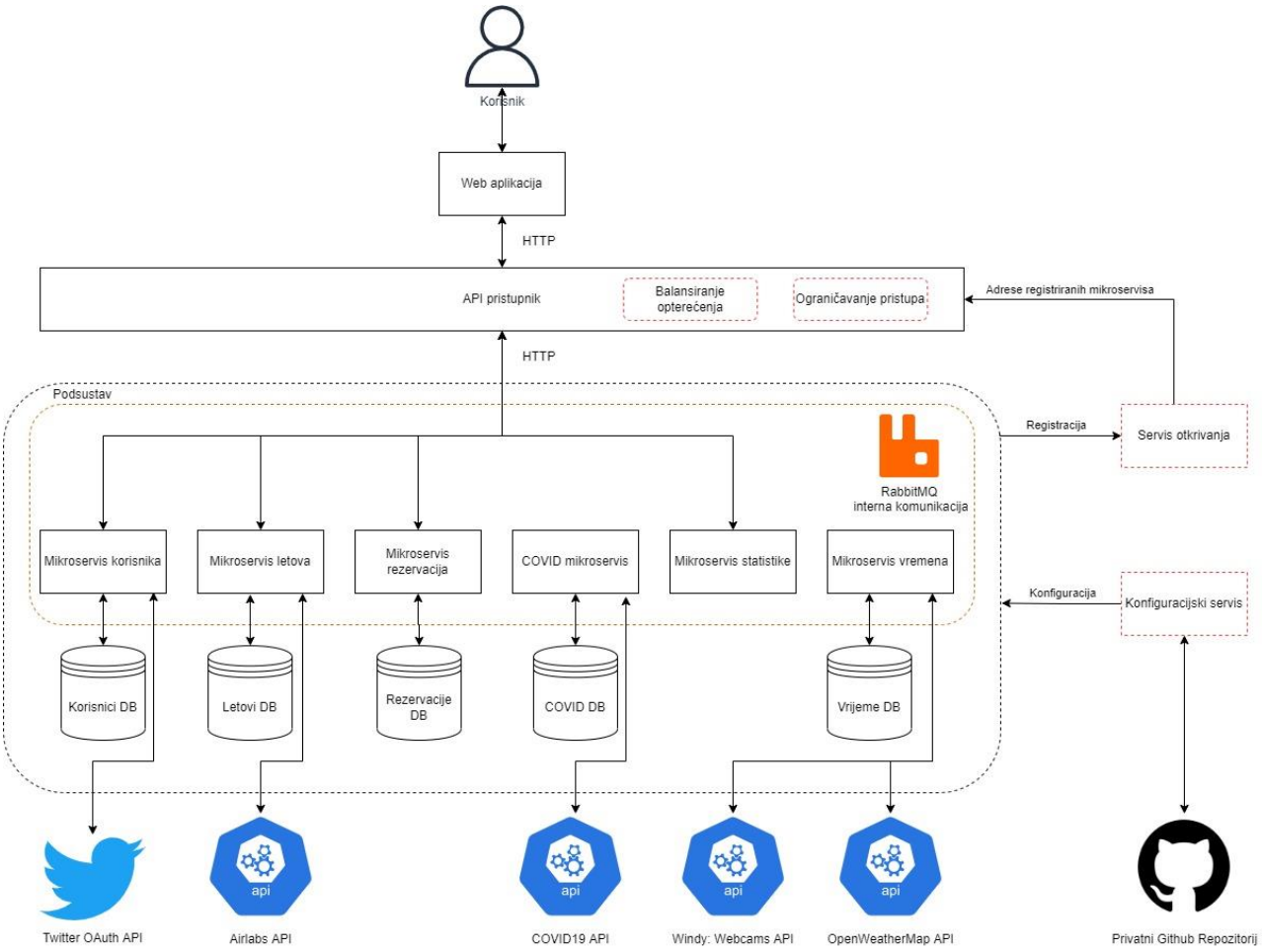


Slika 38 Dijagram slučajeva korištenja

9.1.1. Arhitektura sustava

Budući da je naglasak ovog rada na mikroservisnoj arhitekturi, bitniji dio od funkcionalnosti Upravitelja letova jest njegova arhitektura. Sljedeća slika prikazuje razrađenu arhitekturu Upravitelja letova.

Na dijagramu je vidljivo da krajnji korisnik komunicira samo s Web aplikacijom (Upravitelj letova) preko Web preglednika. Upravitelj letova za realizaciju svih funkcionalnosti koristi mikroservise u podsustavu tako da šalje potrebne HTTP zahtjeve. Pristup mikroservisima u podsustavu moguć je samo kroz API pristupnik jer je samo njegova adresa poznata Web aplikaciji Upravitelj letova. API pristupnik vrši balansiranje opterećenja dolaznih zahtjeva te ograničava pristup resursima ovisno o definiranoj razini autorizacije. API pristupnik koristi Servis otkrivanja kako bi saznao lokacije mikroservisa kojima treba proslijediti zahtjeve.



Slika 39 Arhitektura sustava

U podsustavu postoji šest mikroservisa od kojih je svaki zadužen za jednu domenu, što je lako razumljivo iz njihovih naziva. Svi mikroservisi registriraju se kod Servisa otkrivanja i koriste Konfiguracijski servis koji pomoću privatnog GitHub repozitorija pruža potrebnu konfiguraciju za svaki. Svaki mikroservis koji ima potrebu za pohranom podataka koristi vlastitu, MySQL bazu podataka. Napomena je da u ovoj konkretnoj implementaciji baze podataka nisu fizički odvojene, no to je čest slučaj u stvarnim mikroservisnim sustavima. Za internu komunikaciju, između mikroservisa unutar podsustava, koristi se RabbitMQ posrednik za razmjenu poruka. Neki od mikroservisa imaju potrebu za dohvaćanjem podataka sa servisa treće strane te su isti prikazani na dijagramu. Korišteni vanjski servisi i razlog njihovog korištenja prikazani su u sljedećoj tablici.

Tablica 5 Korišteni servisi treće strane

Naziv	Adresa dokumentacije	Razlog
Twitter OAuth API	https://developer.twitter.com/en/docs/authentication/oauth-2-0/authorization-code	OAuth funkcionalnost

Airlabs API	https://airlabs.co/docs/	Podaci o aviokompanijama, lukama i letovima
COVID19 API	https://documenter.getpostman.com/view/10808728/SzS8rjbc#4bad095c-e106-4eb1-9498-ec5954d2165d	Broj zaraženih
Windy API	https://api.windy.com/webcams/docs#/list/nearby	Identifikator kamere
OpenWeatherMap API	https://openweathermap.org/api/one-call-3	Vremenska prognoza

9.2. Otkrivanje servisa

Prije nego što se krene na mikroservise koji pridonose funkcionalnosti same aplikacije, potrebno je proći kroz one koji svojim postavljanjem omogućuju da ova arhitektura uopće funkcionira kao cjelina. Prvi i osnovni servis koji svojim postojanjem omogućuje ispravan rad ovog sustava će se nazivati „Servis otkrivanja“, a služi upravo onome o čemu je bilo riječi u jednom od poglavlja vezanih uz uzorke u mikroservisnoj arhitekturi – otkrivanje servisa (eng. service discovery). Ovaj je servis pogodan za to da ga se prvog predstavi te po tome što je njegova implementacija i više nego jednostavna. Točnije, više je nego jednostavna za korisnike Spring Cloud programskog okvira. Razlog tome je Spring Cloud Netflix čija implementacija pruža gotovu implementaciju otkrivanja servisa koja se u ovom slučaju naziva „Eureka“. Budući da je za konfiguriranje i korištenje ove implementacije potrebno samo par linija, ovaj servis neće sadržati posebne klase niti programsku logiku. Dakle, osim što je potrebno dodati Spring Cloud Netflix ovisnosti (eng. dependency) u *pom.xml* datoteku, osnovna konfiguracija dolazi u obliku anotacije i u slučaju Spring Boot projekta, dodaje se uz osnovnu klasu koja pokreće cijeli programski okvir:

```
@EnableEurekaServer
@SpringBootApplication
public class DiscoveryServiceApplication {

   public static void main(String[] args) {
     SpringApplication.run(DiscoveryServiceApplication.class, args);
   }
}
```

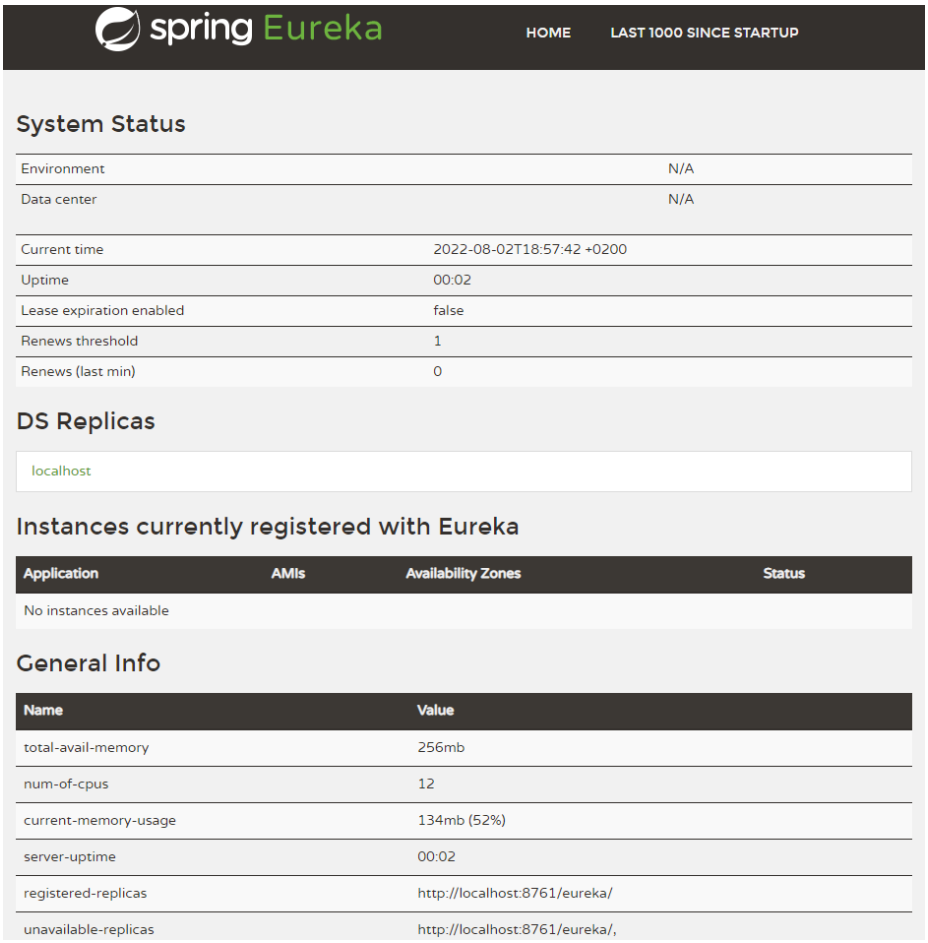
Iz prethodnog primjera vidljivo je da je u programski kôd dodana anotacija `@EnableEurekaServer` koja, kao što joj i naziv govori, omogućuje korištenje implementacije Netflix Eureka servisa otkrivanja. Preostale postavke niti nisu nužne, no pružaju višu razinu kontrole nad servisom otkrivanja. Ove postavke dolaze u obliku svojstava (eng. properties) i nalaze se u datoteci specifičnoj za svaki Spring Boot projekt – *application.properties*. Ovdje se

na jednom mjestu mogu definirati svojstva za projekt, koja Spring Boot jednostavno može čitati i pružiti za vrijeme izvršavanja.

Na sljedećem primjeru prikazana su četiri svojstva od kojih će se prva dva viđati u svakom projektu (mikroservisu), a druga dva su specifična za Eureka poslužitelj. Svojstva `server.port` i `spring.application.name` omogućuju da se striktno definiraju port i naziv aplikacije pri izvršavanju. Svojstvo `eureka.client.register-with-eureka` govori ovoj aplikaciji treba li sama sebe registrirati kod registra Eureka poslužitelja te je postavljena na `false` jer se u ovom slučaju ova aplikacija ne koristi ni za što drugo i nema potrebe za time. Drugo svojstvo, `eureka.client.fetch-registry`, je također postavljeno na `false` i govori ovoj aplikaciji da za nju nije potrebno dohvaćati informacije o ostalim servisima iz registra. Isključivanjem ovih svojstava izbjegava se izvršavanje nepotrebnih poslova u pozadini i Servis otkrivanja se drži jednostavnim i laganim.

```
server.port=8761
spring.application.name=servis-otkrivanja

eureka.client.register-with-eureka=false
eureka.client.fetch-registry=false
```

Eureka poslužitelj također dolazi s nadzornom pločom koja omogućuje praćenje statističkih podataka za vrijeme izvršavanja, kao i instanci registriranih mikroservisa. Sljedeća slika prikazuje izgled osnovnog dijela navedene nadzorne ploče te je u ovom slučaju popis registriranih instanci prazan jer niti jedan drugi servis još nije pokrenut.



Slika 40 Nadzorna ploča Eureka poslužitelja

9.3. Konfiguracijski servis

Drugi mikroservis koji nema značajnu ulogu za logiku sustava, već služi samo za konfiguraciju naziva se „Konfiguracijski servis“. Dok je otkrivanje servisa bilo obrađeno u teorijskom dijelu rada, konfiguracijski servis se nije spominjao te će se zato u ovom poglavlju većinom obrazložiti njegovo korištenje, s obzirom na to da je jednostavnost implementacije na razini Servisa otkrivanja.

U prethodnom poglavlju spomenuta je *application.properties* datoteka i njezina mogućnost pružanja vrijednosti svojstava za vrijeme izvršavanja aplikacije. Praktički svaka Spring aplikacija imat će potrebu za korištenjem svojstava, pa tako i one u ostatku ovog rada. U klasičnom bi slučaju svaka aplikacija (ovdje mikroservis) imala svoja svojstva zapisana u svojoj *application.properties* datoteci. Problem se javlja, primjerice, kod sljedećeg scenarija. Kako su mikroservisi u pravilu mali i jednostavni dijelovi veće arhitekture, u velikim sustavima će ih u jednom trenutku biti pokrenut velik broj (npr. 50-ak). Dođe li do potrebe za promjenom samo jednog svojstva, npr. veza s bazom podataka, bit će potrebno svaki mikroservis ugasiti,

napraviti izmjenu, ponovo ga izgraditi (eng. build), isporučiti i pokrenuti. To je relativno jednostavno učiniti jednom, no i do 50 puta baš i nije.

Zbog toga je dobro koristiti centralizirano mjesto na kojem su pohranjena sva svojstva i mogu biti izmijenjena bez potrebe za rukovanjem samim mikroservisima. U obliku Konfiguracijskog servisa, ovakvo mjesto može se nalaziti na datotečnom sustavu (eng. file system), na nekoj vrsti pohrane sa stražnje strane aplikacije (eng. backend) ili koristeći neku od opcija za verzioniranje programskog kôda kao npr. GitHub. Ova opcija i dalje ostavlja mogućnost korištenja *application.properties* datoteke, no pruža veću fleksibilnost mikroservisa. Spring također pruža implementaciju u obliku Spring Cloud Config poslužitelja i klijenata, koja će se uz opciju za vanjsku (eksternu) pohranu preko privatnog GitHub repozitorija, koristiti u ovom primjeru.

Dakle, sam Konfiguracijski servis, kao i Servis otkrivanja, nema posebnu logiku niti previše potrebnih klasa. Kao i kod Servisa otkrivanja, potrebno je uključiti gotovu implementaciju koristeći sljedeću anotaciju:

```
@EnableConfigServer
@SpringBootApplication
public class KonfiguracijskiServerApplication {

    public static void main(String[] args) {
        SpringApplication.run(KonfiguracijskiServerApplication.class,
                              args);
    }
}
```

Osim ove anotacije, Konfiguracijski servis također koristi svojstva koja su u njegovom slučaju nužna za ispravno funkcioniranje, točnije za povezivanje s udaljenim GitHub repozitorijem:

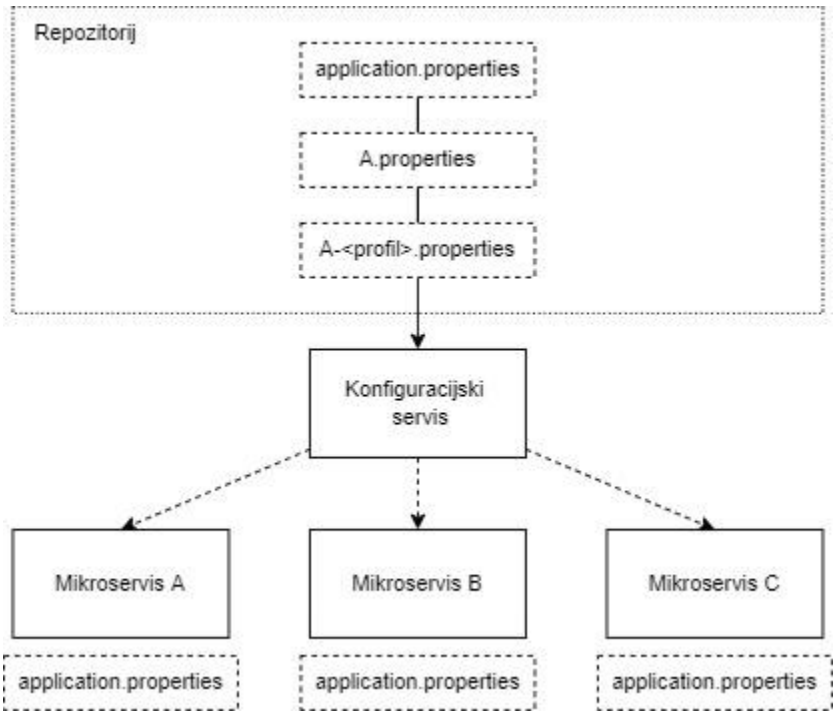
```
server.port=8888
spring.application.name=konfiguracijski-server

spring.cloud.config.server.git.uri=https://github.com/asankovic/konfiguracija-upravitelja-letova
spring.cloud.config.server.git.username=asankovic
spring.cloud.config.server.git.password=ghp_8xnQ68pn1ZSO8bSxJdf274KoFUN9V40
spring.cloud.config.server.git.default-label=main
spring.cloud.config.server.git.clone-on-start=true
```

Na prethodnom primjeru, prva dva svojstva imaju istu svrhu kao i kod Servisa otkrivanja. Ostatak svojstava potreban je da bi se Konfiguracijski servis pri pokretanju mogao povezati na GitHub repozitorij i s njega preuzimati vrijednosti potrebnih svojstava. Korištena svojstva imaju samoopisne nazive i lako su razumljiva, osim zadnja dva koja će biti pobliže objašnjena u nastavku. Svojstvo `spring.cloud.config.server.git.default-label` definira na kojoj zadanoj grani (eng. branch) na udaljenom repozitoriju treba tražiti datoteke, a ovisno o

vrijednosti `spring.cloud.config.server.git.clone-on-start` svojstva, servis (ne)će preuzeti (klonirati) sve podatke s repozitorija pri pokretanju. Prema tome, osnova grana je specificirana kao *main*, budući da je ona ujedno i jedina na ovom repozitoriju, a postavljanjem drugog svojstva na `true` usporit će se inicijalno pokretanje servisa (zbog podataka koji prvo trebaju biti preuzeti), ali zato će prvo dohvaćanje vrijednosti svojstva biti znatno brže.

9.3.1. Prioriteti konfiguracijskih datoteka

Jasno je da se podaci iz jedne *application.properties* datoteke mogu prenijeti na udaljeni repozitorij i time eventualno izdvojiti zajednička svojstva koja se javljaju kod više servisa. No kod većih sustava, s velikim brojem svojstava, korištenje jedne datoteke za spremanje svih svojstava nije baš sistematski ni pregledno. Prema tome, postoje pravila prema kojima Konfiguracijski servis pretražuje određene datoteke, ovisno o servisu koji vrši zahtjev. Ovakav pristup omogućuje izdvajanje specifičnih svojstava za određeni servis u zasebne datoteke i drži cijeli sustav mnogo preglednijim. Sljedeća slika prikazuje opcije izdvajanja datoteka svojstava za specifične mikroservise, kao i njihove prioritete pri čitanju. Na slici su prikazana tri mikroservisa od kojih svaki ima svoju *application.properties* datoteku i koristi Konfiguracijski server za čitanje svojstava s udaljenog repozitorija. Na strani repozitorija, datoteke sa svojstvima mogu se specificirati na tri načina. Prvi je korištenje klasične *application.properties* datoteke, drugi je korištenje datoteke s ekstenzijom *properties*, a nazivom koji odgovara nazivu mikroservisa na koji se odnosi, i treći koji osim naziva mikroservisa ima naveden i Spring profil. Datoteke koje su na slici postavljene niže, imaju veći prioritet. Prioritet dolazi do značaja kada



Slika 41 Prioritet konfiguracijskih datoteka

se u više datoteka javi svojstvo s istim ključem (nazivom). Prema tome, Konfiguracijski servis će uvijek vratiti vrijednost ključa koja se nalazi u datoteci s većim prioritetom. Također, datoteke koje se nalaze u repozitoriju imaju veći prioritet od lokalne datoteke.

Važno je napomenuti da će svi mikroservisi koji se obrađuju u nastavku koristiti Konfiguracijski servis, no zbog jednostavnijeg i preglednijeg prikaza, sva svojstva iz integriranog razvojnog okruženja (eng. integrated development environment - IDE) neće uvijek biti prikazana (ovisno o važnosti i ponavljanju).

9.4. API pristupnik

Sljedeći servis predstavlja centralizirani pristup cijelom mikroservisnom sustavu i dok sadrži nešto više logike od dvaju prethodno opisanih servisa, uloga mu je i dalje većinom posrednička. Radi se o API pristupniku koji je dobrim dijelom već obrađen, a ovdje će biti prikazana njegova jednostavna implementacija u programskom okviru Spring Cloud. Srž API pristupnika su definirane rute zahtjeva prema kojima API pristupnik prosljeđuje pristigle zahtjeve u dubinu sustava mikroservisa. Rute je moguće definirati programski ili kroz svojstva te je u ovom slučaju odabrana opcija korištenja *application.properties* datoteke (preko Konfiguracijskog servisa) zbog bolje preglednosti.

Sljedeći primjer prikazuje neke od osnovnih svojstava koja se koriste pri definiranju rute:

```
spring.cloud.gateway.routes[0].id=prijava-registracija
spring.cloud.gateway.routes[0].uri=lb://korisnici-ws
spring.cloud.gateway.routes[0].predicates[0]=Path=/prijava, /registracija
spring.cloud.gateway.routes[0].predicates[1]=Method=POST

spring.cloud.gateway.routes[1].id=korisnici
spring.cloud.gateway.routes[1].uri=lb://korisnici-ws
spring.cloud.gateway.routes[1].predicates[0]=Path=/korisnici/**
spring.cloud.gateway.routes[1].filters[0]=FiltarPotrebnaPrijava
```

Na primjeru su vidljive dvije rute, što se vidi prema vrijednosti indeksa uz svako svojstvo, budući da ih Spring sprema kao listu. Svaka ruta ima navedeni jedinstveni identifikator koji je obično opisnog karaktera. Sljedeće potrebno svojstvo je lokacija mikroservisa koja može odgovarati bilo kojoj URL adresi, no vidljivo je da to ovdje nije slučaj. Budući da je API pristupnik također spojen na Servis otkrivanja, moguće je specificirati naziv mikroservisa koji se nalazi u registru. Prefix *lb* daje do znanja API pristupniku da se nad ovom vrstom mikroservisa vrši automatsko balansiranje opterećenja. Spring Cloud zadano koristi Spring Cloud LoadBalancer implementaciju u ovu svrhu, no čest odabir je i Ribbon LoadBalancer implementacija. Prema tome, adresa *lb://autentikacija-ws* govori da se sve putanje navedene u sljedećem svojstvu traže na mikroservisima koji su registrirani pod imenom *autentikacija-ws*

te da je nad njima potrebno balansirati opterećenje. Za definiranje putanja moguće je koristiti jednostavnije i složenije izraze (eng. expression language) koje podržava Spring, kao što je to slučaj u drugoj ruti. Osim putanje, svojstva omogućuju ograničavanje zahtjeva prema brojnim svojstvima. U slučaju prve rute radi se o dopuštenju samo zahtjeva poslanim HTTP POST metodom, no moguće je ograničiti i prema vrijednosti kolačića (eng. cookie), zaglavlja, vremena, domaćina (eng. host), upita (eng. query) i brojnih drugih.

Druga ruta također ima definirani ručno implementirani i prilagođeni filtar. Naime, jedina logika ovog mikroservisa leži u dvama filterima koji su ovisno o potrebi dodani na određene rute. Radi se o filterima koji ograničavaju pristup ovisno o potrebnoj razini autorizacije, a to su filtar koji zahtjeva prijavljenog korisnika i drugi koji zahtjeva korisnika s administratorskim pravima. Oba filtera funkcioniraju većinom na sličan način te, koristeći informacije koje se nalaze u JWT-u, određuju ima li zahtjev pravo pristupa resursu. Detalj o sadržaju žetona obradit će se u sljedećem poglavlju vezanom uz mikroservis zadužen za sav rad s korisnicima, kao i njihovu autorizaciju. Sljedeći isječak izvornog kôda prikazuje dio koji predstavlja način implementacije i osnovni tijek logike u filteru koji provjerava dolazi li zahtjev od prijavljenog korisnika. U ovom se dijelu također vrlo kratko koristi reaktivno programiranje kroz Spring WebFlux.

```
@Component
public class FiltarPotrebnaPrijava extends
    AbstractGatewayFilterFactory<FiltarPotrebnaPrijava.Config> {

    private static final String AUTHORIZATION_ZAGLAVLJE = "Authorization";
    private final Environment okolina;

    public FiltarPotrebnaPrijava(Environment okolina) {
        super(Config.class);
        this.okolina = okolina;
    }

    @Override
    public GatewayFilter apply(Config config) {
        return (exchange, chain) -> {
            if (!autenticiranKorisnik(exchange))
                return zabraniPristup(exchange.getResponse());
            return chain.filter(exchange);
        };
    }
    //…
}
```

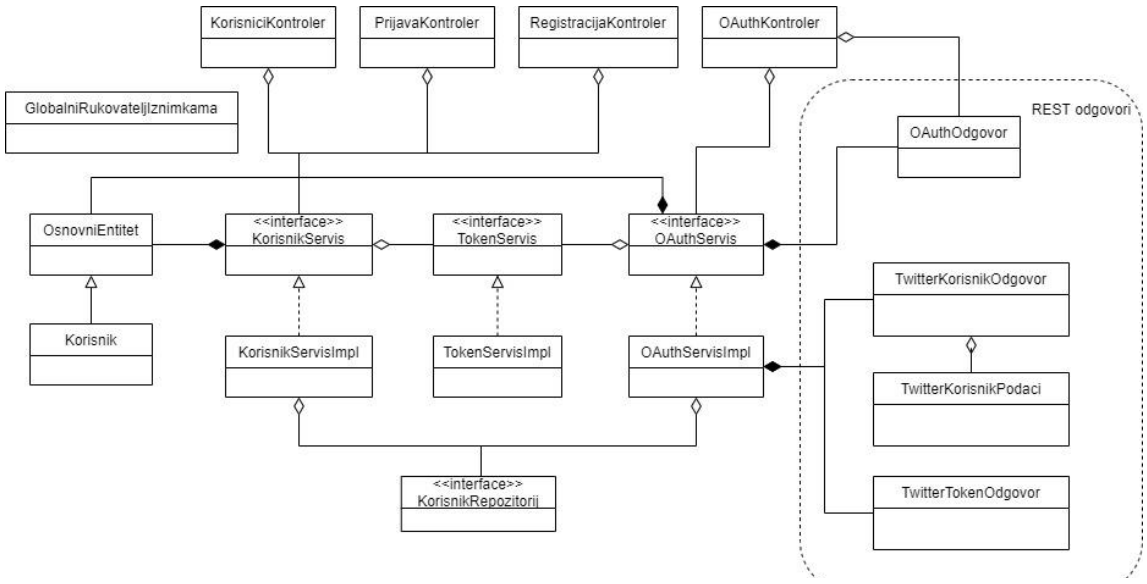
9.5. Mikroservis korisnika

Prvi mikroservis u podsustavu, koji će biti obrađen u ovom poglavlju, jest mikroservis korisnika i autentikacije. Idealno bi rukovanje korisnicima i njihovom prijavom i pravima bilo odvojeno, no zbog količine posla koji treba biti obavljen u pogledu svih ovih operacija, nije bilo razloga za odvajanjem istih u dva premalena mikroservisa.

Za početak, ovdje će biti prikazan način na koji se mikroservisi povezuju na Servis otkrivanja i Konfiguracijski servis te se isti pristup podrazumijeva u svim obrađenim mikroservisima u nastavku. Prema tome, ponovo postoji nekoliko svojstava koja se koriste u ovu svrhu, a prikazana su na sljedećem primjeru.

```
server.port=${PORT:0}
spring.application.name=korisnici-ws
eureka.client.service-url.defaultZone=http://localhost:8761/eureka
eureka.instance.instance-
id=${spring.application.name}:${spring.application.instance_id:${random.val
ue}}
spring.config.import=optional:configserver:http://localhost:8888
```

Ovdje je također vidljiv način dinamičke dodjele slobodnog porta u slučaju da on nije eksplicitno definiran. Razlog je, kao što je već spomenuto, mogućnost više instanci istog mikroservisa koji pritom čitaju ista svojstva. Vrijednosti svojstava *eureka.client.service-url.defaultZone* i *spring.config.import* moraju odgovarati adresama Servisa otkrivanja i Konfiguracijskog servisa, a ovdje je također navedeno da je Konfiguracijski servis opcionalan. Osim što ne mogu koristiti isti port, više instanci iste aplikacije također ne mogu biti registrirane pod istim identifikatorom. Zbog toga se koristi svojstvo *eureka.instance.instance-id* kojim je određen format identifikatora koji će biti jedinstven za svaku novu pokrenutu instancu.



Slika 42 Dijagram klasa mikroservisa korisnika

Dakle, ovaj mikroservis pruža pristupne točke za rukovanje korisnicima (kreiranje, dohvaćanje, uređivanje i brisanje) kao i točke za običnu prijavu i registraciju. Također postoji mogućnost prijave i registracije preko Twitter profila, što je realizirano koristeći OAuth 2.0. Dijagram klasa ovog mikroservisa prikazan je na prethodnij slici. Pristup izgradnji pojedinog mikroservisa u pravilu je jednak za sve i dobro se može vidjeti na dijagramu klasa. Na vrhu, tj. ulazu nalaze se kontroleri koji su logički podijeljeni. Kontroleri u pravilu nemaju zahtjevnu logiku, već se ona nalazi na sljedećoj „razini“ servisnih klasa koje su uvijek definirane sučeljem kako bi se izmjena njihove implementacije mogla lako izvršiti i time izbjeći čvrsto povezivanje klasa implementacijama. Najniža razina jest razina repozitorija koji komuniciraju s bazom podataka. Repozitoriji u slučaju sustava koji je izrađen za primjer ovog rada nikada nemaju implementaciju zahvaljujući mogućnosti Spring Data JPA programskog okvira koji je toliko napredan da sam pruža implementaciju na temelju naziva metoda, kao u sljedećem primjeru:

```
public interface KorisnikRepozitorij
            extends CrudRepository<Korisnik, Long> {

    Optional<Korisnik> findByKorisnickoImeAndOauthFalse(
                                        String korisnickoIme);

    Optional<Korisnik> findByKorisnickoIme(String korisnickoIme);
}
```

Ostatak su entitetne klase koje odgovaraju tablicama u bazi podataka te klase koje se koriste za mapiranje odgovora koji se šalju preko REST-a, u JSON formatu, koristeći Java Jackson JSON procesor. Kao što je vidljivo na dijagramu klasa, entitet koji se koristi u ovom mikroservisu je samo jedan, pod nazivom „Korisnik“ pa je prema tome i ERA dijagram sastavljen od samo jednog entiteta:

Slika 43 ERA dijagram mikroservisa korisnika

Prijava, registracija i CRUD (eng. Create, Read, Update, Delete) korisnika standardno su odrađeni, a OAuth proces bit će prikazan na strani korisničkog sučelja budući da je implementacija na strani ovog mikroservisa suštinski samo slanje zahtjeva na Twitter poslužitelj, koji prati tijek opisan u poglavlju vezanom uz OAuth. Dio koji će biti prikazan vezan je uz kreiranje žetona koji se koristi kroz cijeli sustav, a temelji se na JWT implementaciji. Isječak sljedeće metode prikazuje jednostavan način kreiranja žetona koji sadrži podatke potrebne na ostalim dijelovima sustava. Koristi se gotov Builder uzorak kojim se postavlja teret (ovdje nazvan subjektom), vrijeme isteka te algoritam koji se koristi. Tajna koja se koristi je zapisana u datoteci svojstava i poznata je svim mikroservisima koji trebaju vršiti proces autentikacije (npr. API pristupnik). Vrijedno je spomenuti da se u praksi ne bi koristio identifikator identičan onome u bazi podataka, već neki sigurniji pristup, no prihvatljivo je za potrebe ovog primjera (kao i kod pristupanja REST resursima).

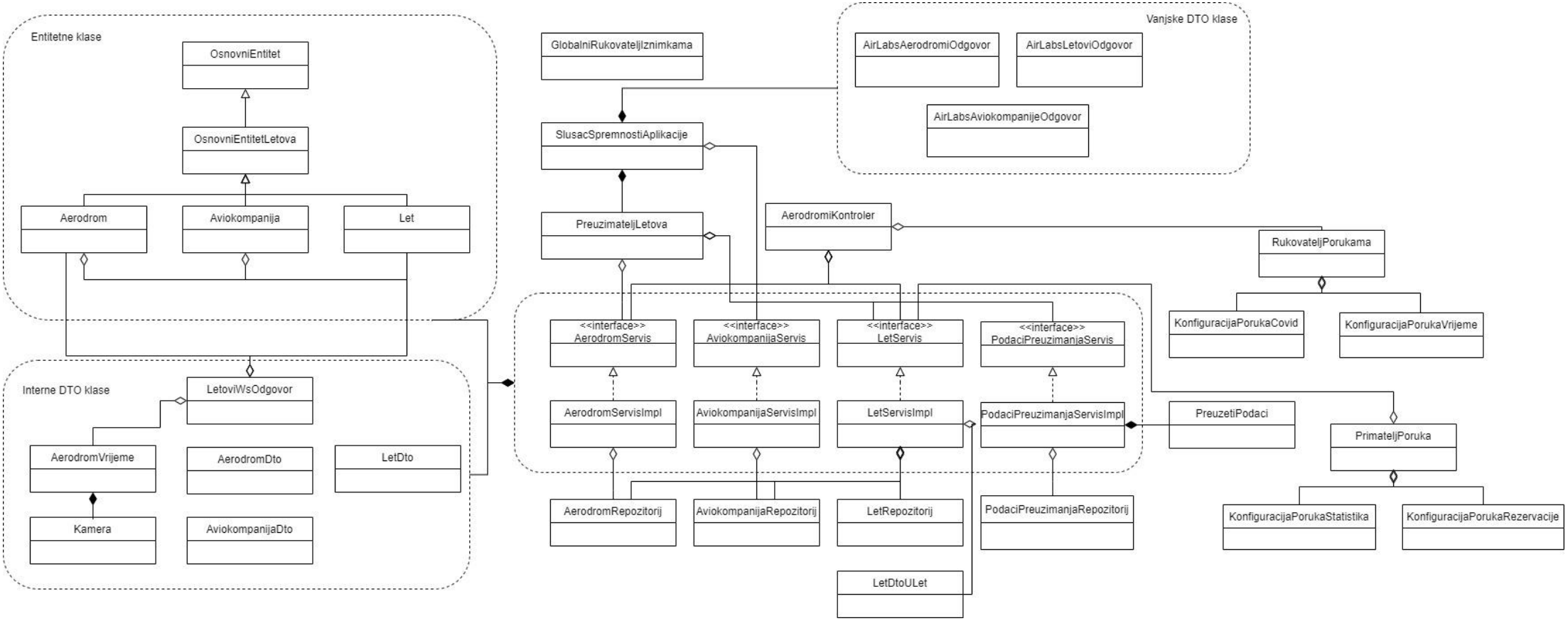
```
@Override
public String generirajTokenZaKorisnika(final Korisnik korisnik) {
    return Jwts.builder()
        .setSubject(korisnik.getId().toString() +
            SEPARATOR +
            korisnik.getKorisnickoIme() +
            SEPARATOR +
            korisnik.isAdministrator())
        .setExpiration(new Date(System.currentTimeMillis() +
            Long.parseLong(okolina.getProperty(TOKEN_TRAJANJE))))
        .signWith(SignatureAlgorithm.HS512,
            okolina.getProperty(TOKEN_TAJNA))
        .compact();
}
```

9.6. Mikroservis letova

Kada je riječ o funkcionalnosti ovog sustava, jedan od glavnih mikroservisa jest mikroservis letova. Ovaj mikroservis, kao što mu i samo ime govori, obavlja sve zadatke vezane uz letove. To uključuje dohvaćanje letova, aerodroma i aviokompanija sa servisa treće strane i njihovo spremanje u bazu podataka, kao i pružanje pristupnih točaka za mikroservis koji predstavlja korisničko sučelje. Mikroservis letova također komunicira s mikroservisima rezervacija, COVID-a, statistike i vremena preko poruka. Mikroservisi COVID-a i vremena pružaju ovom mikroservisu izvor popratnih informacija vezanih uz letove, dok mikroservisi rezervacije i statistike zahtijevaju podatke o letovima kako bi mogli izvršiti svoju funkciju.

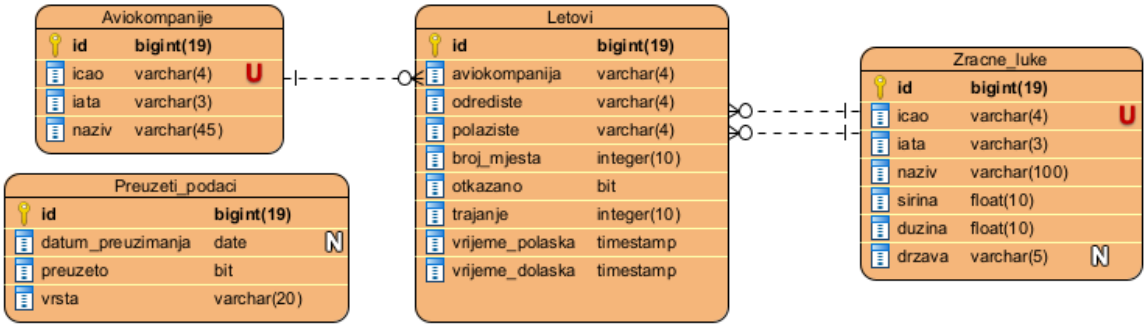
Dijagram klasa na sljedećoj slici prikazuje dosta visoku razinu kompleksnosti ovog mikroservisa u usporedbi sa svim dosadašnjima. Razlog tome najviše proizlazi iz činjenice da je zbog drugih servisa (bili oni dio sustava ili treće strane) potrebno raditi puno izdvajanja i rukovanja podacima, što rezultira veliki brojem entitetnih i DTO (eng. Data Transfer Object)

klasa. DTO klase u pravilu se koriste kada se žele ograničiti podaci koji se šalju (manji skup od onih koji su u entitetnim klasama) te iako većina ostalih mikroservisa izbjegava njihovo korištenje, ovdje to nije bilo moguće izbjeći.



Slika 44 Dijagram klasa mikroservisa letova

Unatoč složenosti dijagrama klasa, struktura baze podataka nije toliko zahtjevna pa ERA model izgleda kao na sljedećoj slici. U središtu aplikacije, a tako i modela, su letovi od kojih svaki sadrži podatke o aviokompaniji koja nudi usluge leta, aerodroma, tj. zračnih luka koje su polazište i odredište te popratne podatke kao što su vrijeme polaska, dolaska ili ukupno očekivano trajanje leta. Također, u posebnu tablicu bilježi se vrijeme preuzimanja zadnjih podataka o letovima, aerodromima i aviokompanijama, kako se isto ne bi nepotrebno izvršilo više puta.



Slika 45 ERA model mikroservisa letova

Glavni razlog zbog kojeg se želi izbjeći nepotrebno slanje zahtjeva više puta jest ograničenost broja zahtjeva od strane vanjskog servisa koji se koristi za dohvaćanje svih podataka u ovom mikroservisu. Zbog prirode odgovora koji vanjski servis pruža, potrebno je slati zahtjev za svaki aerodrom, što može rezultirati velikim brojem zahtjeva. Zbog toga se svi primljeni odgovori filtriraju prema tome sadrže li potpune podatke i jesu li države aerodroma jednake Hrvatskoj, Njemačkoj, Španjolskoj ili Finskoj. Za svrhu ovog primjera ovime se osigurava razumna količina podataka te se sprečava prekoračenje granice maksimalnog broja zahtjeva. Zadatak koji vrši preuzimanje letova prikazan je na sljedećem primjeru, a koristi Cron izraz kako bi započeo preuzimanje svakog dana u 03:00 sata ujutro, kada je opterećenje sustava u pravilu najniže.

```
@Scheduled(cron = "0 0 3 * * ?")
public void preuzimanjeLetova() {
    if (!okolina.getProperty(PREUZIMANJE_AKTIVNO_PROPERTY,
                             Boolean.class, false))
        return;

    if (podaciPreuzimanjaServis.jePreuzetoZaDatum(LETOVI_VRSTA,
                                                  danasnjiDatum()))
        return;

    final List<Aerodrom> spremljeniAerodromi =
                                aerodromServis.dohvatiSveAerodrome();
```

```
    for (Aerodrom aerodrom : spremljeniAerodromi) {
        final AirLabsLetoviOdgovor odgovor =
                    restTemplate.getForObject(AIRLABS_LETOVI_URL +
                                              aerodrom.getIcao()+
                                              LETOVI_PARAMETRI,
                                              AirLabsLetoviOdgovor.class);
        if (odgovor == null)
            continue;
        for (final LetDto letDto : odgovor.getLetovi())
            letServis.kreirajLet(letDto, spremljeniAerodromi);
        }

    podaciPreuzimanjaServis
            .zabiljeziPreuzimanje(new PreuzetiPodaci(
                                LETOVI_VRSTA, true, new Date()));
}
```

Drugi dio koji će biti istaknut jest korištenje Circuit breaker uzorka kroz njegovu Resilience4j implementaciju. Rečeno je da u teoriji osigurač može biti postavljen na sve opasne dijelove u aplikaciji i pružati zadani odgovor, no to ne bi imalo previše smisla. U ovom primjeru, pri komunikacijom s COVID mikroservisom preko RabbitMQ posrednika za razmjenu poruka, koristi implementirana je neka vrsta priručne (eng. cache) memorije koja pamti zadnji broj zaraženih od prošlog odgovora COVID mikroservisa. Pri nedostupnosti COVID mikroservisa, doći će do iznimke (eng. exception) i pozvat će se metoda navedena unutar anotacije koja pruža zadano ponašanje.

```
@CircuitBreaker(name = ZARAZENI_CIRCUIT_BREAKER, fallbackMethod =
                                            "dohvatiZadnjiBrojZarazenih")
public long dohvatiBrojZarazenih(final String drzava) {
    final long brojLjudi = Long.parseLong((String)
                            rabbitTemplate.convertSendAndReceive(
                                KonfiguracijaPorukaCovid.EXCHANGE,
                                KonfiguracijaPorukaCovid.ROUTING_KEY,
                                drzava));
    brojZarazenihCache.put(drzava, brojLjudi);
    return brojLjudi;
}

private long dohvatiZadnjiBrojZarazenih(final String drzava,
                                        final Exception e) {
    return brojZarazenihCache.getOrDefault(drzava,
                                           ZADANI_BROJ_ZARAZENIH);
}
```

Anotacija također zahtjeva ime osigurača jer prema tome povezuje ponašanje s ostalim parametrima koji su navedeni kao svojstva. Ova svojstva opisnih imena lako su razumljiva i omogućuju postavljanje parametara koji odgovaraju onima iz teorijskog dijela ovog uzorka pa ovdje neće biti pobliže objašnjeni.

```
resilience4j.circuitbreaker.instances.zarazeniCircuitBreaker.register-
health-indicator=true
```

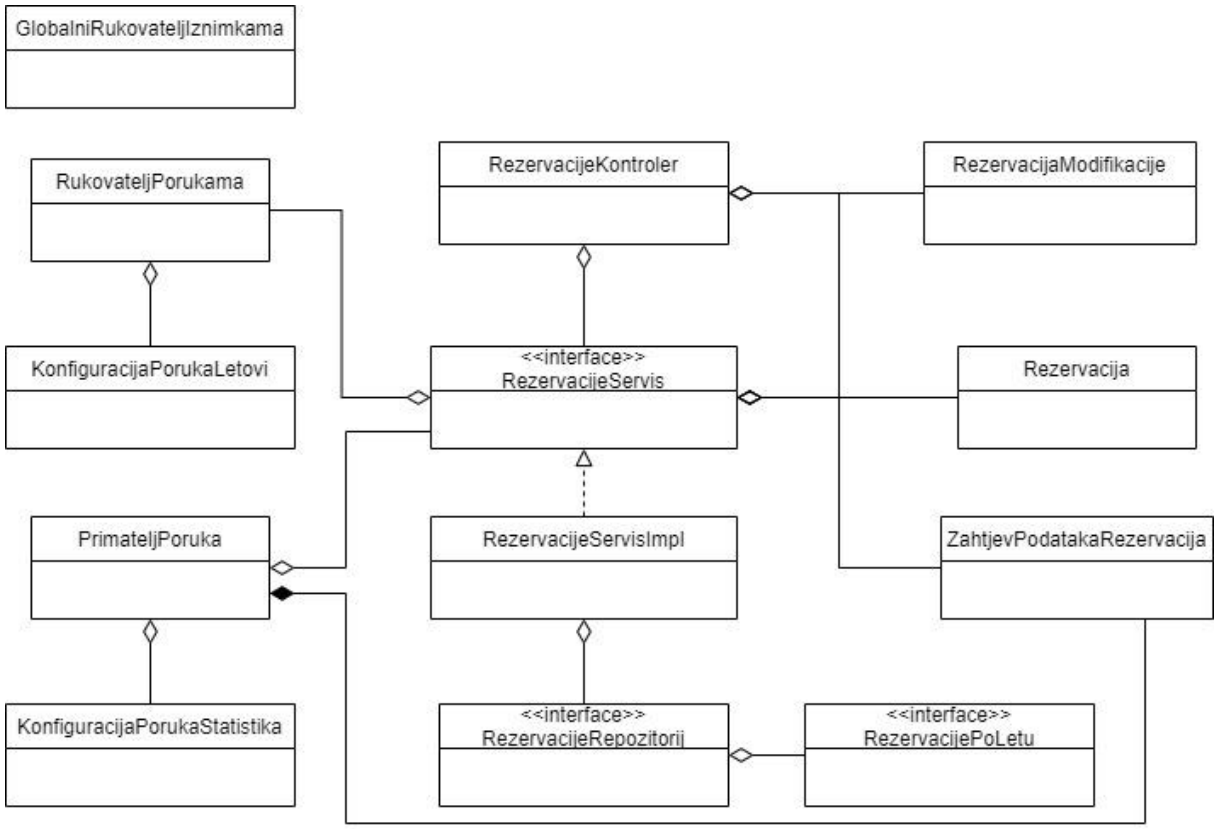
```
resilience4j.circuitbreaker.instances.zarazeniCircuitBreaker.event-
consumer-buffer-size=10
resilience4j.circuitbreaker.instances.zarazeniCircuitBreaker.failure-rate-
threshold=50
resilience4j.circuitbreaker.instances.zarazeniCircuitBreaker.minimum-
number-of-calls=2
resilience4j.circuitbreaker.instances.zarazeniCircuitBreaker.automatic-
transition-from-open-to-half-open-enabled=true
resilience4j.circuitbreaker.instances.zarazeniCircuitBreaker.wait-duration-
in-open-state=5s
resilience4j.circuitbreaker.instances.zarazeniCircuitBreaker.permitted-
number-of-calls-in-half-open-state=3
resilience4j.circuitbreaker.instances.zarazeniCircuitBreaker.sliding-
window-size=10
resilience4j.circuitbreaker.instances.zarazeniCircuitBreaker.sliding-
window-type=COUNT_BASED
```

9.7. Mikroservis rezervacija

Drugi mikroservis odgovoran za glavnu funkcionalnost sustava jest mikroservis rezervacija. Kao što mu i ime govori, on obavlja sve zadatke vezane uz rezervacije i pruža informacije o rezervacijama ostalim mikroservisima. Za razliku od mikroservisa letova, ovaj mikroservis nema posebnih zadataka te se u pravilu sve svodi na CRUD operacije preko pristupnih točaka koje su određene kontrolerom ili preko reda poruka.

Slika 46 ERA model mikroservisa rezervacija

Iako to na ERA dijagramu s prethodne slike nije vidljivo, tablica rezervacija povezana je s korisnicima i letovima, no radi se o različitim bazama podataka budući da svaki mikroservis ima svoju. Rad s podacima i dalje ostaje isti, samo je potrebno da podaci o korisnicima i letovima budu dohvaćeni od strane drugih mikroservisa. Mikroservis rezervacija zapravo ni ne treba znati detalje o korisniku niti letu jer mu to nije „posao“.



Slika 47 Dijagram klasa mikroservisa rezervacija

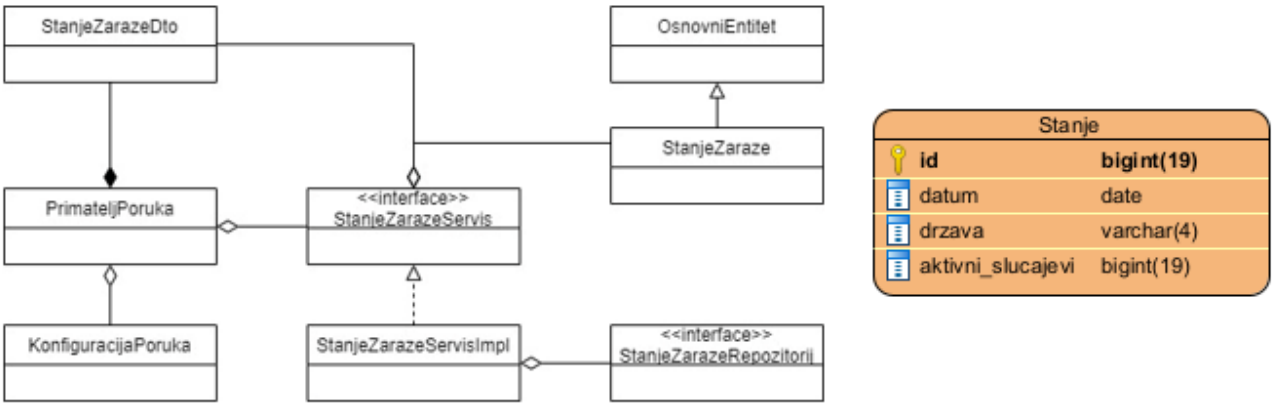
Budući da ovaj servis nema specifičnih dijelova, a sljedeća dva mikroservisa se u potpunosti oslanjaju na komunikaciju poruka, ovdje će biti prikazana RabbitMQ konfiguracija i rukovanje porukama. Kao što je opisano u poglavlju vezanom uz RabbitMQ, AMQP red poruka mora biti povezan s razmjenom preko jedinstvenog ključa. Prema tome, sljedeći isječak izvornog kôda prikazuje definiranu izravnu izmjenu koja se koristi za izmjenu poruka između mikroservisa rezervacija i letova. Spring AMQP omogućuje da se potom, koristeći anotaciju `@RabbitListener`, postavi metoda kao slušač na određenom redu, koja se izvršava pri pristigloj poruci. Primjer anotacije slušača i konfiguracije bio bi

```
//klasa konfiguracije
@Configuration
public class KonfiguracijaPorukaStatistika {

    public static final String QUEUE = "rezervacije-statistika-red";
    public static final String EXCHANGE = "rez-stat-razmjena";
    public static final String ROUTING_KEY = "rez-stat-kljuc";

    @Bean
    public Queue redStatistika(){
        return new Queue(QUEUE, false);
    }
```

```
    @Bean
    public DirectExchange razmjenaStatistika(){
        return new DirectExchange(EXCHANGE);
    }

    @Bean
    public Binding povezivanjeStatistika(@Qualifier("redStatistika")
                                         Queue red,
                                         @Qualifier("razmjenaStatistika")
                                         DirectExchange razmjena){
        return BindingBuilder.bind(red).to(razmjena).with(ROUTING_KEY);
    }
}

…

//metoda slušača u nekoj drugoj klasi
@RabbitListener(queues = KonfiguracijaPorukaStatistika.QUEUE)
public String zaprimiPoruku(final Message poruka){
    //poslovna logika…
}
```

9.8. COVID mikroservis

Jedan od mikroservisa u sustavu, koji nema REST pristupnih točaka, već se koristi samo interno u sustavu jest COVID mikroservis. Budući da je epidemiološka situacija trenutno dosta ozbiljan problem u svijetu, prilikom organizacije letova je također bitno voditi brigu o širenju zaraze. Ovaj mikroservis koristi servis treće strane kako bi prikupio trenutne podatke o broju zaraženih, koje pruža ostalim mikroservisima u sustavu preko komunikacije porukama. Kako vanjski servis pruža podatke na dnevnoj bazi, COVID mikroservis implementira mehanizam koji šalje zahtjev na vanjski servis samo ako podaci o statistici za trenutni dan i državu već nisu dohvaćeni. Shema baze podataka i dijagram klasa prikazani su u nastavku.

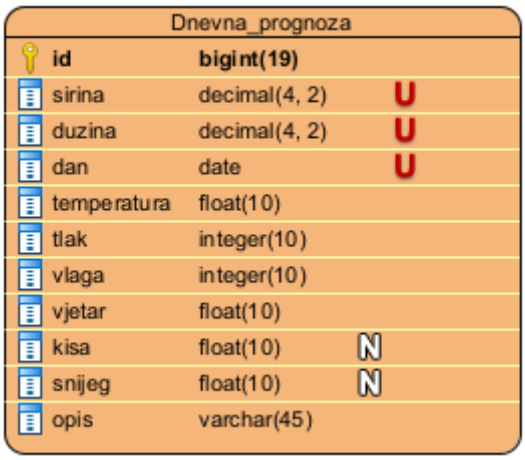


Slika 48 Dijagram klasa (lijevo) i ERA model (desno) COVID mikroservisa

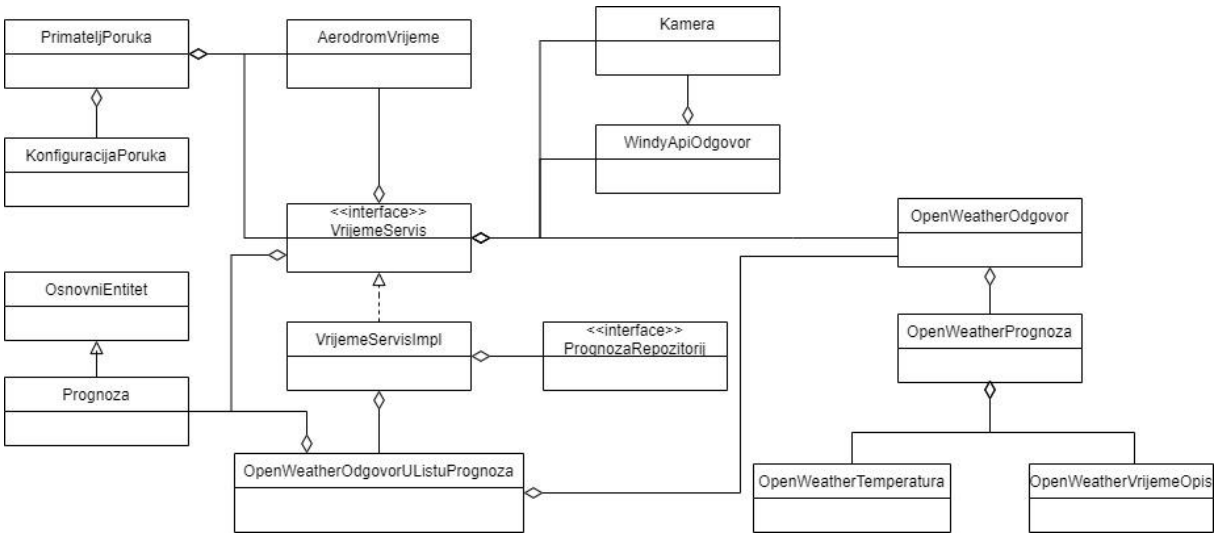
9.9. Mikroservis vremena

Vrlo slično prošlom mikroservisu, samo usredotočen na drugu domenu, jest mikroservis vremena. Mikroservis vremena također koristi samo komunikaciju porukama da bi ostalim mikroservisima (primarno mikroservisu letova) pružio vremenske podatke za određenu lokaciju i određeni datum. Ovaj mikroservis koristi dva vanjska servisa – jedan za vremensku prognozu, a drugi za identifikatore kamera kojima se prikazuje stanje uživo u blizini lokacije aerodroma (detaljniji prikaz u poglavlju vezanom uz korisničku stranu). Ovim dodatkom nastoji se poboljšati korisničko iskustvo pri odabiru, tj. odluci o rezervaciji leta.

ERA dijagram mikroservisa vremena također je vrlo jednostavan, dok dijagram klasa sadrži nešto više klasa zbog kompleksnijeg rukovanja odgovorima vanjskih servisa. Oba su prikazana na slikama u nastavku.



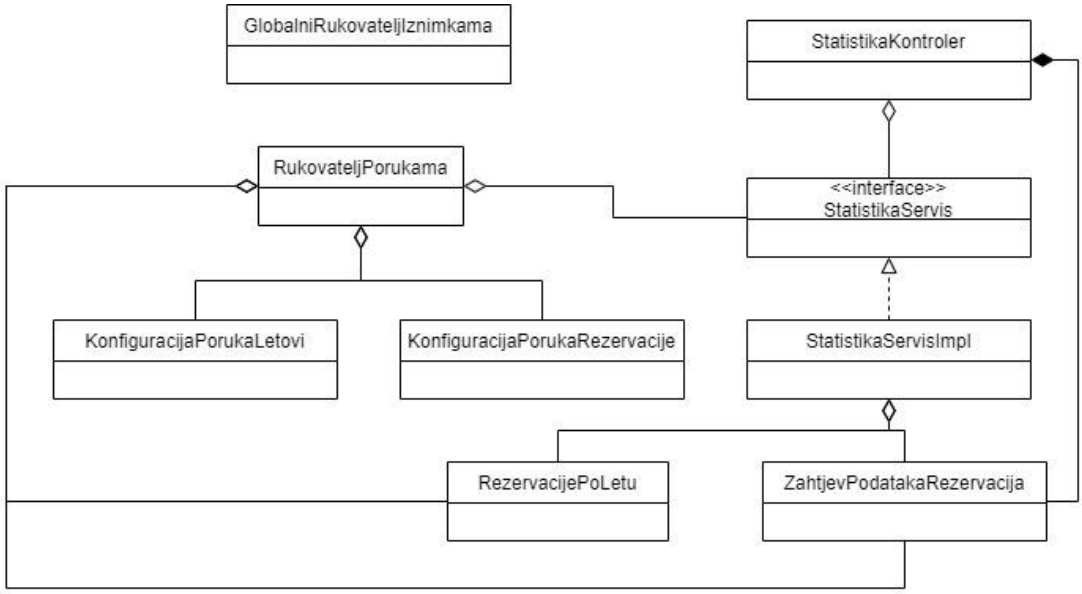
Slika 50 ERA model mikroservisa vremena



Slika 49 Dijagram klasa mikroservisa vremena

9.10. Mikroservis statistike

Posljednji mikroservis u unutrašnjosti sustava koji je zadužen za izračun i pružanje statističkih podataka o rezervacijama i letovima jest mikroservis statistike. Za razliku od prethodna dva, mikroservis statistike pruža pristupne točke koje konzumira aplikacija na korisničkoj strani. Ovaj mikroservis također je specifičan po tome što ne posjeduje bazu podataka jer samo vrši izračune na temelju podataka koje mu kroz komunikaciju porukama pružaju ostali mikroservisi u sustavu. Prema tome, ovaj mikroservis imat će samo dijagram klasa koji, kao što je vidljivo na sljedećoj slici, ne sadrži klase na razini repozitorija.

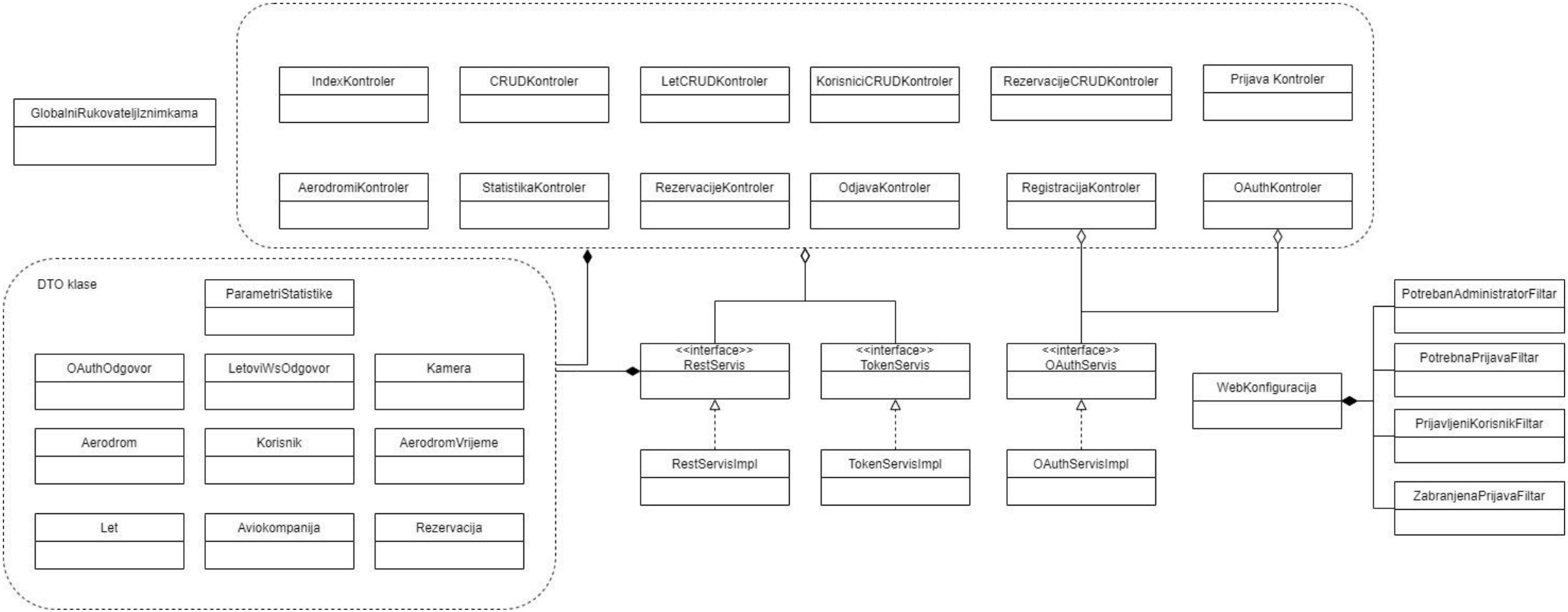


Slika 51 Dijagram klasa mikroservisa statistike

9.11. Web aplikacija

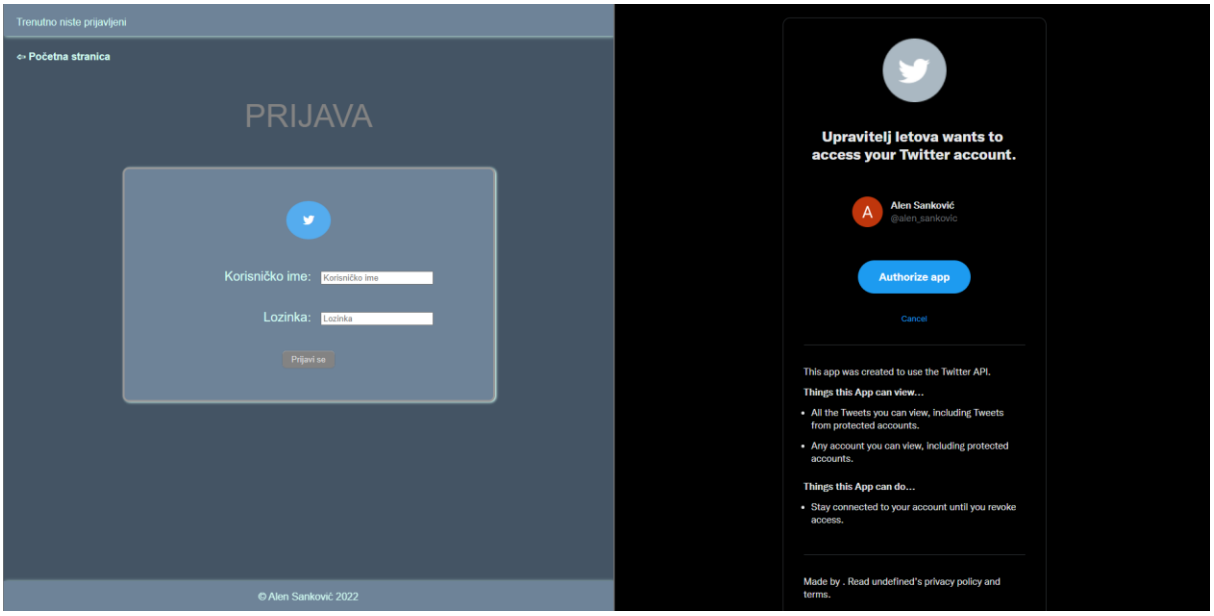
Dio koji spaja sve mikroservise u jednu funkcionalnu cjelinu u obliku Web aplikacije s korisničkim sučeljem jest Web aplikacija Upravitelj letova. Ova aplikacija koristi Model-Pogled-Kontroler (eng. Model-View-Controller – MVC) arhitekturu koristeći Spring MVC programski okvir. Već je spomenuto da se sva funkcionalnost realizira pozivima prethodno opisanih mikroservisa, dok se korisnička strana, tj. pogledi generiraju primjenom Thymeleaf predložaka umjesto standardnog JSP-a. Korisnička strana je jednostavna i ne sadrži ništa osim navedenih predložaka koji generiraju HTML elemente i minimalnog CSS-a (eng. Cascading Style Sheets) za oblikovanje izgleda sučelja. Ova aplikacija ne sprema podatke pa prema tome nema niti vlastitu bazu podataka, a dijagram klasa prikazan je na sljedećoj slici. Zbog toga što se

`RestServis` koristi za sve pozive mikroservisa i `TokenServis` za sve operacije vezane uz žetone, a to se događa u svakom kontroleru, navedeno je prikazano samo jednom vezom kako bi dijagram ispao pregledniji. Isto vrijedi i za entitetne klase kroz cijelu aplikaciju. S dijagrama je vidljivo da je većina fokusa na kontrolerima i podacima, uz kojih postoji par filtara koji ograničavaju pristup pogledima prema razini autorizacije.



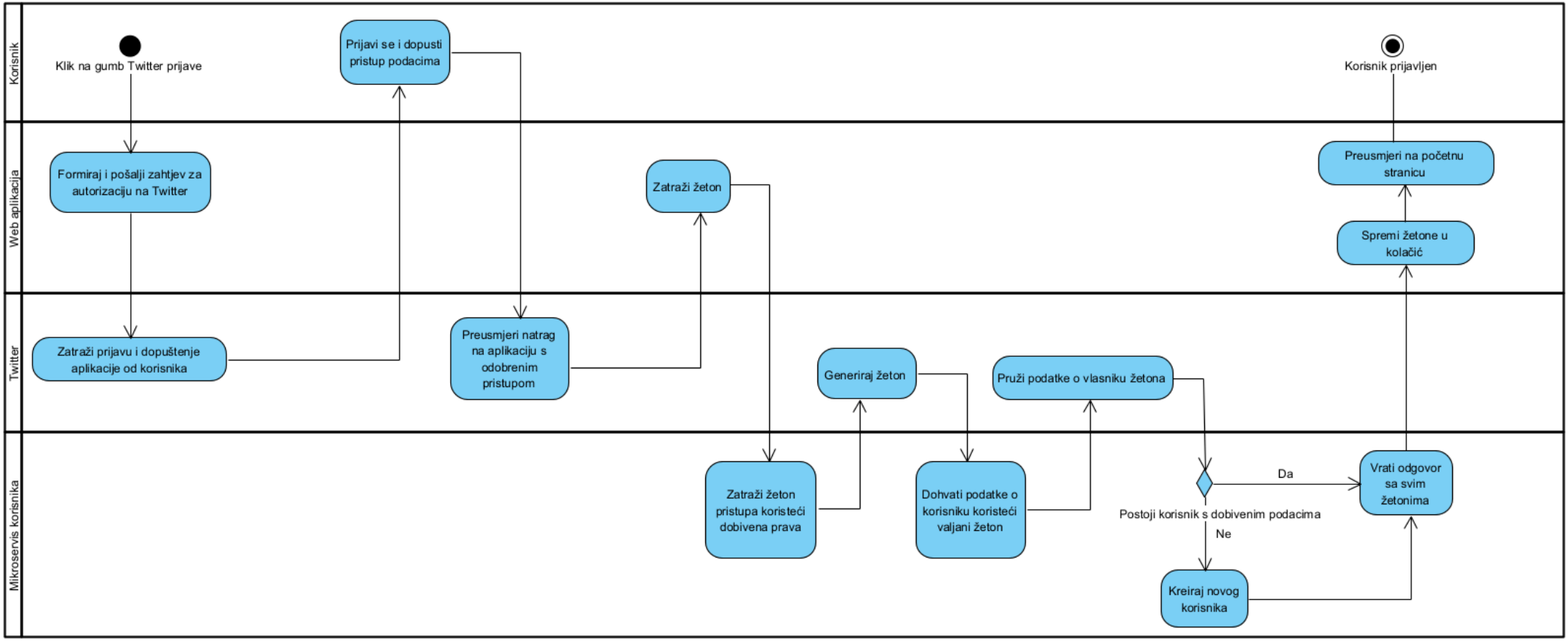
Slika 52 Dijagram klasa Web aplikacije

U nastavku će biti prikazani neki od važnijih pogleda i funkcionalnosti uz slike ekrana i povezanog izvornog kôda, a za svaki će također biti navedeno koji se mikroservisi koriste za realizaciju.



Slika 53 OAuth Twitter prijava u aplikaciju

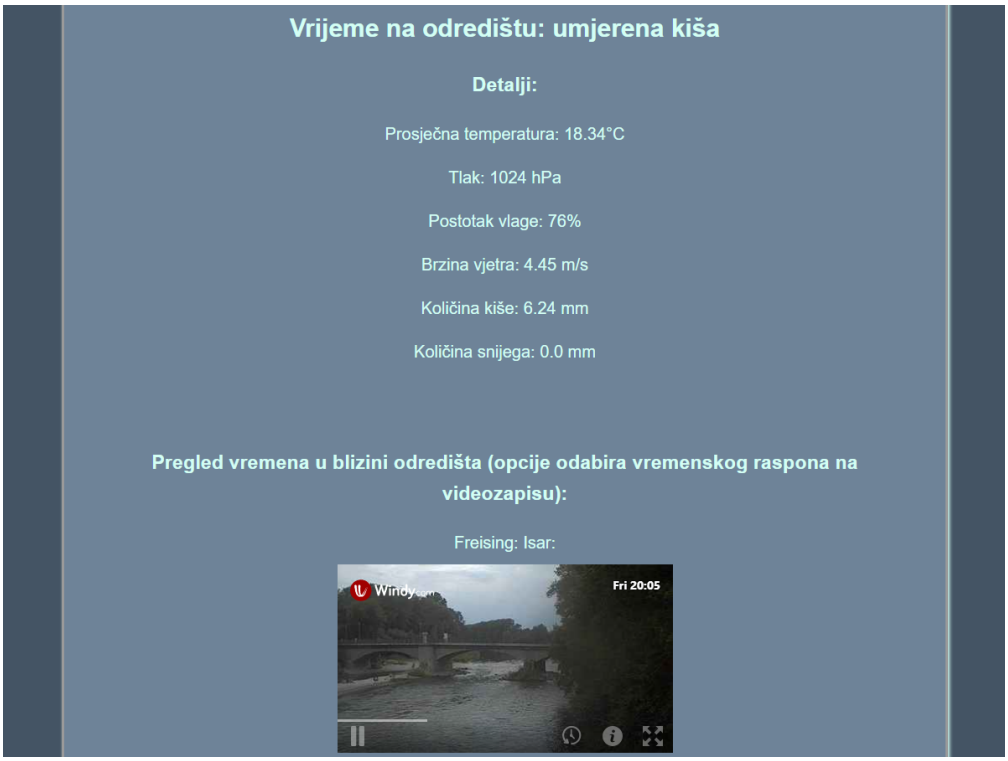
Prvi zanimljiv dio koji će biti istaknut jest korištenje prijave i registracije preko postojećeg Twitter korisničkog računa, koristeći OAuth. Na prethodnoj slici s lijeve strane je vidljiv ekran prijave koji, osim standardnog načina, nudi gumb za prijavu preko Twittera. Pritiskom na navedeni gumb i nakon prijave na Twitter-u, s desne strane vidljiv je ekran na kojem Upravitelj letova traži pristup podacima korisnika. Ti se podaci potom koriste za prijavu ili registraciju, ovisno o tome postoji li već korisnički račun povezan s tim Twitter profilom. Tijek zahtjeva i logike u pozadini prikazan je sljedećim dijagramom aktivnosti.



Slika 54 Dijagram aktivnosti OAuth prijave i registracije

Sljedeći, nešto složeniji prikaz jest prikaz detalja o odabranom letu, koji također daje mogućnost rezervacije istog. Rezervacija je moguća samo u slučaju prihvatljive razine zaraženosti, a alternativa koja se pritom prikazuje je vidljiva na sljedećoj slici.

Slika 55 Dio pogleda detalja leta s onemogućenom rezervacijom

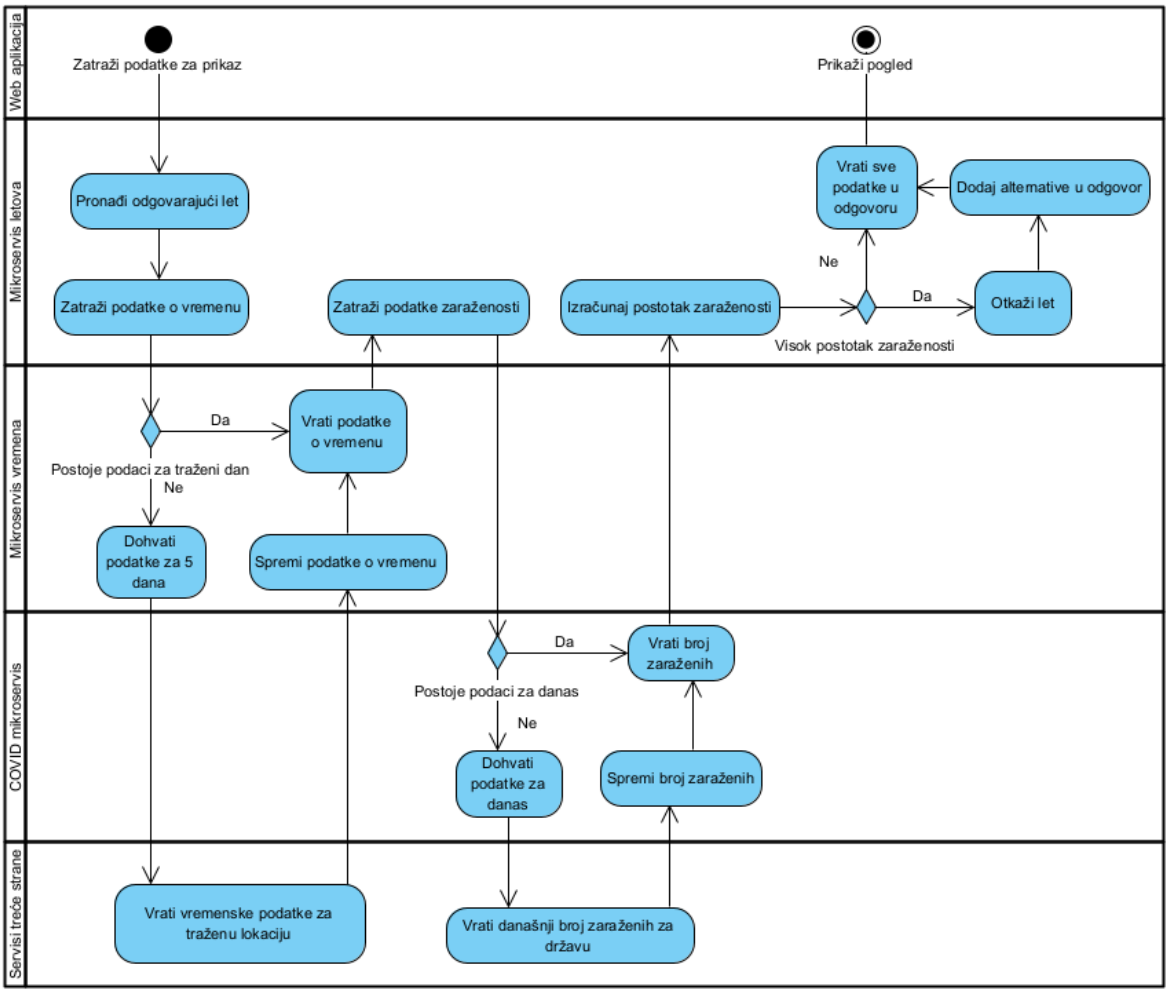


Slika 56 Realizacija prikaza snimaka kamera uživo

Ostatak pogleda ispunjen je s vremenskim podacima, kao i videosnimakama kamera uživo, a ukoliko COVID mikroservis ili mikroservis vremena nije dostupan, prikazan je alternativan tekst na tom mjestu

Dio korištenog predloška za prikaz kamera na ovom pogledu, kao i dijagram aktivnosti za njegovu realizaciju prikazani su u nastavku.

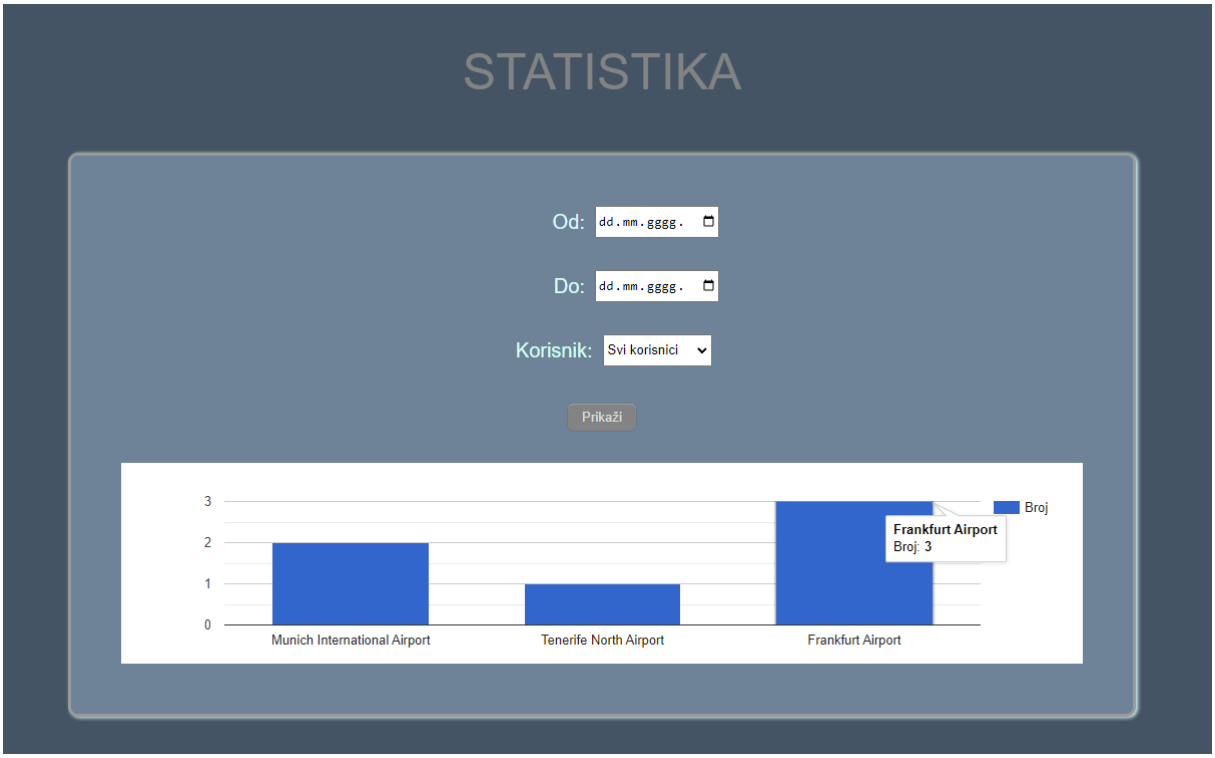
```
<div th:each="kamera : ${aerodromVrijeme.kamere}"
     style="width: 400px;
     height: 250px; margin: auto">

   <p th:text="*{kamera.lokacija} + ':'"></p>
   <a name="windy-webcam-timelapse-player"
       th:attr="data-id=*{kamera.id}" data-play="day"
       th:href="'https://windy.com/webcams/' + *{kamera.id}"
       target="_blank" th:text="*{kamera.lokacija}"></a>

   <script async type="text/javascript"
   src="https://webcams.windy.com/webcams/public/embed/script/player.js">
   </script>

</div>
```



Slika 57 Dijagram aktivnosti prikaza detalja leta

Osim toga, za realizaciju prikaza podataka statistike, koji se dobivaju s mikroservisa statistike, koriste se Google Charts, a izvorni dio i prikaz ekrana nalaze se u nastavku. Ovaj prikaz omogućuje odabir raspona i korisnika za koje se želi vidjeti grafički prikaz broja rezervacija po odredišnom aerodromu.

```
<script type="text/javascript" th:inline="javascript">
   google.charts.load('current', {'packages': ['corechart']});
   google.charts.setOnLoadCallback(drawChart);

   function drawChart() {
    var data = new google.visualization.DataTable();
    data.addColumn('string', 'Naziv');
    data.addColumn('number', 'Broj');
    data.addRows([[${statistika}]]);

    var chart = new google.visualization.ColumnChart(
                              document.getElementById('chart_div'));
   chart.draw(data);
   }

</script>
```



Slika 58 Primjer pogleda statistike za administratora

10.Zaključak

Kroz ovaj rad, uz osnove arhitekture sustava i njezinih oblika, obrađena je mikroservisna arhitektura, njezini počeci, karakteristike, kao i sličnosti i razlike s još jednom poznatom arhitekturom temeljenom na servisima – servisno-orijentiranom arhitekturom gdje je vidljivo da glavnu razliku predstavlja razina na kojoj se SOA i mikroservisna arhitektura koriste. Prikazana je važnost distribuiranog i modularnog pristupa koji u mikroservisnoj arhitekturi omogućuje neovisnu isporuku, bolju i neovisnu skalabilnost, labavu povezanost komponenti, princip jedne odgovornosti, jednostavnije testiranje i održavanje i slično. Osim brojnih prednosti, vidljivo je da mikroservisna arhitektura može znatno podići kompleksnost sustava zbog velikog broja komponenti, čime se i podiže težina njezinog održavanja.

Također, prikazana je potreba za uzorcima dizajna na različitim razinama aplikacije, koji dodatno karakteriziraju ovaj stil arhitekture. U svrhu upravljanja podacima, najčešći odabir uzorka pokazao se Database per service uzorak, koji odgovara jednoj bazi podataka za svaki mikroservis, a njegovim korištenjem dolazi do potrebe za drugim specifičnim uzorcima kao što su to Saga ili API Composition uzorak. Circuit breaker uzorak i korištenje žetona definirani su se kao standardi za oporavak od grešaka u mikroservisnom sustavu i način postizanja sigurnosti zbog specifične rasprostranjenosti komponenti. API pristupnik i otkrivanje servisa nezaobilazan su dio mikroservisne arhitekture zbog mogućnosti povezivanja i praćenja neovisnih komponenti čime se omogućuje rad sustava kao cjeline.

Implementacijom praktičnog dijela prikazana je jednostavnost korištenja Spring, točnije Spring Cloud, programskog okvira, u svrhu izrade mikroservisnog sustava. Kroz gotovu implementaciju koncepata kao što je API pristupnik, otkrivanje servisa, Circuit breaker uzorak i sl., koje nudi Spring Cloud, potvrđeno je da oni zaista imaju svrhu i koriste se u praksi. Primjenom RabbitMQ posrednika za razmjenu poruka prikazana je alternativa klasičnoj komunikaciji preko HTTP-a, dok je opravdana jednostavnost i kvaliteta korištenja JWT i OAuth standarda u arhitekturi kao što je to mikroservisna.

Mikroservisna arhitektura distribuiran je stil arhitekture čiji je cilj bio nadići tradicionalne oblike monolitnih arhitektura i njihove nedostatke. Ovaj stil arhitekture definitivno će biti korišten u bližoj budućnosti zbog svojih brojnih prednosti i zahtjeva koje ispunjava. Ipak, s brzim razvojem tehnologije i promjenom zahtjeva, uvijek postoji mogućnost dolaska novog stila arhitekture koji će zasjeniti mikroservisnu arhitekturu ili ona iz nekog razloga više neće biti dovoljno dobra za toliko rasprostranjenu primjenu.

Popis literature

Akshay K. B., Chaitra B. H., A Comparison between RabbitMQ and REST ful API for Communication between Micro-Services Web Application, in IRJET, vol. 7, no. 5, May 2020

Baeldung (2022), Intro to Spring Boot Starters, Preuzeto 30.08.2022 s https://www.baeldung.com/spring-boot-starters

Baeldung (2022), Wiring in Spring: @Autowired, @Resource and @Inject, Preuzeto 29.08.2022 s https://www.baeldung.com/spring-annotations-resource-inject-autowire

Bandurski P. (2021) Curcuit breaker uzorak [Slika] Preuzeto 17.7.2022. s https://dzone.com/articles/circuit-breaker-module

Barry D. K. (2012). Web Services, Service-Oriented Architectures, and Cloud Computing, 2nd Edition, morgan Kaufmann, Burlington, Massachusetts, United States

Bearer Authentication (bez dat.), Preuzeto 20.7.202 s https://swagger.io/docs/specification/authentication/bearer-authentication/

Boyd R. (2012), Getting Started with OAuth 2.0, O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol

C. V. Ramamoorthy i H. F. Li. 1977. Pipeline Architecture. ACM Comput. Surv. 9, 1 (March 1977), 61–102.

Chand S. (2022) Spring ekosistem [Slika] Preuzeto 30.8.2022. s https://www.edureka.co/blog/spring-tutorial/

Christudas B. (2019), Practical Microservices Architectural Patterns: Event-Based Java Microservices with Spring Boot and Spring Cloud, Apress, New York, USA

Dragoni, Nicola & Lanese, Ivan & Larsen, Stephan & Mazzara, Manuel & Mustafin, Ruslan & Safina, Larisa. (2017). Microservices: How To Make Your Application Scale.

Earl T. (2016). Service-Oriented Architecture: Analysis and Design for Services and Microservices, Pearson, London, United Kingdom

Engineering D. A. (2021), Sustav s bazom podataka po servisu [Slika] Preuzeto 15.7.2022. s https://dev.to/dogealgo/when-you-need-database-per-service-2710

Garlan D. (2008) Software Arhitecture, School of Computer Science Carnegie Mellon University 5000 Forbes Avenue, Pittsburgh, PA

Gutierrez F. (2019), Pro Spring Boot 2: an authoritative guide to building microservices, web and enterprise applications, and best practices, Second edition. New York, NY: Apress

Haselböck, Stefan & Weinreich, Rainer & Buchgeher, Georg. (2017). Decision Guidance Models for Microservices: Service Discovery and Fault Tolerance. 1-10

Herbsleb J. D. i Grinter R. E. (1999), „Architectures, coordination, and distance: Conway's law and beyond", IEEE software, sv. 16, br. 5

Ingeno J. (2018) Software Architect's Handbook, Packt Publishing, Birmingham, United Kingdom

Irakli Nadareishvili, Ronnie Mitra, Matt McLarty i Mike Amundsen (2016), O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol

Jayathilaka C. (2020), Agilni model [Slika] Preuzeto 16.6.2022. s https://medium.com/@chathmini96/agile-methodology-30ec4cdf3fc

Khalfallah H. B. (2020), Modularna i monolitna arhitektura [Slika] Preuzeto 22.6.2022. s https://betterprogramming.pub/inside-software-modularity-and-related-metrics-2e5af2b447dc

Lewis J. i Fowler, M. (2014), Microservices, Preuzeto 09.03.2022., Dostupno na: https://www.martinfowler.com/articles/microservices.html

Liebhart, Daniel & Welkenbach, Peter & Schmutz, Guido. (2010). Service Oriented Architecture: An Integration Blueprint.

M. Mosleh, K. Dalili and B. Heydari, "Distributed or Monolithic? A Computational Architecture Decision Framework," in IEEE Systems Journal, vol. 12, no. 1, pp. 125-136, March 2018

Messina, Antonio & Rizzo, Riccardo & Storniolo, Pietro & Urso, Alfonso. (2016). A Simplified Database Pattern for the Microservice Architecture. 10.13140/RG.2.1.3529.3681.

Montesi, Fabrizio & Weber, Janine. (2016). Circuit Breakers, Discovery, and API Gateways in Microservices, arxiv

Ozkaya M. (2021), CQRS uzorak [Slika] Preuzeto 17.7.2022. s https://medium.com/design-microservices-architecture-with-patterns/cqrs-design-pattern-in-microservices-architectures-5d41e359768c

Pacheco V. F. (2018), Microservice Patterns and Best Practices, Packt Publishing, Birmingham, UK

RabbitMQ (bez dat.), AMQP 0-9-1 Model Explained, Preuzeto 25.7.2022. s https://www.rabbitmq.com/tutorials/amqp-concepts.html

Raje G. (2021), Security and Microservice Architecture on AWS, O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol

RFC-6749 - The OAuth 2.0 Authorization Framework (2012), Preuzeto 22.7.202 s https://www.rfc-editor.org/rfc/rfc6749

RFC-7519 - JSON Web Token (JWT) (2015), Preuzeto 20.7.202 s https://www.rfc-editor.org/rfc/rfc7519

Richards M. (2015) Software Architecture Patterns, O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol

Richards M. (2016). Microservices vs. Service-Oriented Architecture, O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol

Richards M., Ford N. (2020) Fundamentals of Stoftware Arhitecture, O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol

Richardson C. (2018), Microservices Patterns, Manning Publications, Shelter Island, New York, United States

Richardson K. (2019), Pregled uzoraka vezanih uz mikroservisnu arhitekturu [Slika] Preuzeto 15.7.2022. s https://betterprogramming.pub/inside-software-modularity-and-related-metrics-2e5af2b447dc

Shahir Daya et al., Microservices from Theory to Practice: Creating Applications in IBM Bluemix Using the Microservices Approach.: IBM Redbooks, 2016.

Smartbear (bez dat.), AMQP posrednik, [Slika] Preuzeto 25.7.2022. s https://support.smartbear.com/readyapi/docs/testing/amqp.html

Spring (bez. dat), Spring Framework Documentation, Preuzeto 28.08.2022. s https://docs.spring.io/spring-framework/docs/current/reference/html/

Spring Boot (bez dat.), Spring Boot Reference Documentation, Preuzeto 29.08.2022 s https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#io

Spring Projects (bez dat.), Preuzeto 30.7.2022 s https://spring.io/projects

Suprapto, Falahah & Surendro, Kridanto & Sunindyo, Wikan. (2021). Circuit Breaker in Microservices: State of the Art and Future Prospects. IOP Conference Series: Materials Science and Engineering

Štefanko, Martin & Chaloupka, Ondřej & Rossi, Bruno. (2019). The Saga Pattern in a Reactive Microservices Environment. 483-490.

Van Casteren, Wilfred. (2017). The Waterfall Model and the Agile Methodologies : A comparison by project characteristics – short, Neobjavljeno [Internet]

Walls C. (2018), Spring Boot in Action (5th. ed.). Manning Publications Co., USA.

Wolff E. (2016), Microservices: Flexible Software Architecture. Addison-Wesley Professional

Zimmermann, Olaf. (2016). Microservices tenets: Agile approach to service development and deployment. Computer Science - Research and Development. 32.

Popis slika

Popis tablica

Prilozi

Izvorni programski kôd svih izrađenih mikroservisa i web aplikacije.