Razvoj i integracija web aplikacija na bazi strujanja podataka

Iva, Levak

SVEUČILIŠTE U ZAGREBU

FAKULTET ORGANIZACIJE I INFORMATIKE

V A R A Ž D I N

Iva Levak

Razvoj i integracija web aplikacija na bazi strujanja podataka

DIPLOMSKI RAD

Varaždin, 2021.

SVEUČILIŠTE U ZAGREBU

FAKULTET ORGANIZACIJE I INFORMATIKE

V A R A Ž D I N

Iva Levak

JMBAG: 0016116427

Studij: Informacijsko i programsko inženjerstvo

Razvoj i integracija web aplikacija na bazi strujanja podataka

DIPLOMSKI RAD

Mentor:

Prof. dr. sc. Dragutin Kermek

Varaždin, lipanj 2021.

Iva Levak

Izjava o izvornosti

Izjavljujem da je moj 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.

Autorica potvrdila prihvaćanjem odredbi u sustavu FOI-radovi

_____

Sažetak

Cilj diplomskog rada je upoznati se s razvojem i integracijom web aplikacija na bazi strujanja podataka, primijeniti aspekte strujanja podataka i razviti web aplikaciju na bazi strujanja podataka. Strujanje podataka danas se sve više koristi radi optimalnog rada aplikacije pri brzoj i neovisnoj obradi velike količine kontinuiranih i beskonačnih podataka, te trenutne analize velike tih podataka. Uz strujanje podataka pojavljuju se usko povezani pojmovi integracije sustava, reaktivnih sustava, sustava poruka i web servisa. U radu je stoga obrađeno strujanje podataka i sve uske poveznice strujanja podataka. U svrhu rada izrađena je aplikacija na bazi strujanja podataka, a u izradi su korišteni alati Apache Spark i Apache Kafka. Kreirana aplikacija služi za pregled aerodroma, aviona i letova aviona. Rad aplikacije se bazira na radu više neovisnih komponenti povezanih Kafka posrednikom. Uz pregled aviona, aerodroma i letova aviona, kreirana je funkcionalnost preporuka aviona za praćenje putem metode za pronalaženje alternirajućih najmanjih kvadrata.

Ključne riječi: strujanje podataka; web servisi; integracija; sustavi poruka; reaktivni sustavi; visoke performanse.

Sadržaj

1. Uvod 1
2. Integracija 2
2.1. Izazovi integracije sustava 3
2.2. Sloj integracije 4
2.2.1. Integracija portala 4
2.2.2. Agregacija entiteta 5
2.2.3. Integracija procesa 7
2.3. Povezivanje sustava 8
2.3.1. Podatkovna integracija 9
2.3.2. Funkcionalna integracija 10
2.3.3. Prezentacijska integracija 11
2.4. Topologije Integracije 11
2.4.1. Povezanost od točke do točke 12
2.4.2. Broker 12
2.4.3. Sabirnica poruka 13
2.4.4. Publish/Subscribe 14
3. Web servisi 15
3.1. Karakteristike web servisa 16
3.2. Arhitektura web servisa 18
3.3. Servisno orijentirana arhitektura 20
3.4. Vrste web servisa 22
3.4.1. SOAP web servisi 22
3.4.2. RESTful web servisi 23
4. Sustavi poruka 25
4.1. Karakteristike sustava poruka 26
4.2. Koncepti sustava poruka 28
4.2.1. Kanali 29
4.2.2. Poruke 30
4.2.3. Isporuka u više koraka 30
4.2.4. Usmjeravanje 30
4.2.5. Transformacija 31
4.2.6. Krajnje točke 31
5. Reaktivni sustavi 32
5.1. Karakteristike reaktivnih sustava 33
5.2. Uzorci reaktivnih sustava 35

5.2.1. Uzorak Reactor 36
5.2.2. Uzorak Multireactor 36
5.2.2.1. Vert.x 36
5.2.3. Uzorak Actor Model 37
5.2.3.1. Akka 37
6. Strujanje podataka 39
6.1. Karakteristike strujanja podataka 40
6.2. Komponente arhitekture za strujanje podataka 42
6.3. Primjeri sustava sa strujanjem podataka 44
7. Izrada projekta 46
7.1. Alati 46
7.1.1. Apache Kafka 46
7.1.2. Apache Spark 47
7.1.3. Java DB i Apache Derby 49
7.1.4. OpenSky Network 49
7.2. Opis korisničke aplikacije 50
7.3. Opis slučajeva korištenja 51
7.4. Baza podataka 53
7.5. Arhitektura i rad projekta 55
7.6. Izrada aplikacija 57
7.6.1. Biblioteke 57
7.6.2. Postavke Kafka servera 61
7.6.3. Izrada proizvođača, potrošača i Kafka Streams 62
7.6.3.1. Proizvođač 62
7.6.3.2. Potrošač 64
7.6.3.3. Kafka Streams 65
7.6.4. Izrada Spark posla 66
7.6.4.1. ALS algoritam 67
7.7. Primjeri korištenja korisničke aplikacije 69
8. Zaključak 74
Popis literature 76
Popis slika 81
Popis tablica 82

1. Uvod

Svako poduzeće danas ima određenu razinu informatizacije procesa, radi efikasnijeg i efektivnijeg poslovanja te konkurentnosti na tržištu. Ustaljeno mišljenje je kako bez ili s jako malom razinom informatizacije procesa poduzeće ne može konkurirati na tržištu. Informatizacijom se poslovi obavljaju u kraćem vremenskom roku, s manje propusta i boljim rezultatima. Jako je bitna i izvedba određenih informacijskih rješenja. Stoga se pomiče fokus s postojanja informatizacije na kvalitetu izvedbe rješenja. Iako većina poduzeća sadrži određeni postotak informatizacije, korišteni sustavi su nepovezani i time se smanjuje njihova učinkovitost. U tom okviru javlja se pojam integracije sustava, koja se odnosi na spajanje više podsustava ili sustava u jednu cjelinu. Vrlo je vjerojatno da poduzeće ima barem jednom potrebu za integracijom sustava. Stoga se unaprijed treba razmišljati o izradi sustava koji će biti moguće integrirati s ostalima. Osim potrebom za integracijom sustava uvijek se stavlja naglasak na velikoj potrebi za široko dostupnim informacijama raspoloživim u tren oka. Napretkom tehnologije sve se više informacija prikuplja iz različitih senzora, korisničkih klikova, dnevnika i sl. Pretpostavlja se da će u 2025. godini biti kreirano 175 zetabajta podataka [26]. Podaci koji se proizvode često su beskonačni i kontinuirani. Prvo pitanje koje se postavlja je kako prenijeti i koristiti veliku količinu podataka uz optimalni rad aplikacije. S druge strane, velika se korist ostvaruje analizom podataka. Korisničko iskustvo korištenja aplikacije se može povećati analizom prikupljenih podataka o korisnicima pri čemu što je veći uzorak podataka, to su rezultati analitike upotrebljiviji i direktniji. S time se postavlja drugo pitanje: kako analizirati veliku količinu podataka u stvarnom vremenu? Analiza i prijenos velike količine podataka su usko povezani pri čemu se iziskuju brzi rezultati i optimalni rad aplikacije. U današnjem brzo rastućem svijetu informacije postavlja se u fokus obrada, analiza i prijenos podataka u realnom vremenu. Sami korisnici očekuju od sustava konstantan rad, što znači da sustav mora biti otporan na greške i elastičan. U tu svrhu javljaju se web servisi, sustavi poruka, REST sustavi i strujanje podataka. Strujanje podataka kao tehnologija omogućuje prijenos podataka preko interneta tako da se oni obrađuju kao neprekidan niz što rješava pitanja beskonačnosti podataka. Uz to dopušta neovisnu obradu podataka i prijenos velike količine podataka u realnom vremenu.

Ovaj rad će se osvrnuti na integraciju i razvoj aplikacija na bazi strujanja podataka. Najprije će biti objašnjena teorijska pozadina integracije sustava, Web servisa, sustava poruka, reaktivnih sustava i strujanja podataka. Fokus će biti na najboljim praksama izrade brze i učinkovite aplikacije kako bi se kasnije u radu mogla izraditi aplikacija.

2. Integracija

U početku kreiranja informacijskih sustava nije se vidjela potreba za povezivanjem sustava s drugim sustavima ili za komunikacijom više sustava. Takvi sustavi kreirani su kao samostojeće aplikacije, a često su nazivani otocima. Razvojem tehnologije i interneta pojavljuje se potreba za povezivanjem sustava zbog kompleksnosti poslovanja i sve veće informatizacije poslovanja. Informacijski sustavi, pa i sama informatizacija poslovanja se smatraju ključnim za uspjeh poslovanja i ostvarivanje konkurentnosti na tržištu. Radi se o pokušaju što veće informatizacije poslovnih procesa zbog manjeg utroška vremena i resursa te zbog bržeg razvoja poduzeća i povećanja njegove konkurentnosti. Sustavi postaju opsežniji i kompleksniji, zahtijevaju puno podataka i što veću iskoristivost programskog koda. Iako su sustavi kompleksniji i dalje je velika pažnja usmjerena na što manji utrošak vremena i resursa. Kao rješenje javlja se integracija aplikacija.

Integracija (lat. *integratio*: obnova cjeline) kao pojam označava spajanje u jedno odnosno okupljanje odijeljenih elemenata u jedinstven sustav [45]. Pa tako kao što i samo ime govori **integracija sustava** je fizičko ili funkcionalno spajanje više podsustava u jednu cjelinu, bili oni aplikacije, softveri ili komponente pri čemu se treba osigurati njihov efikasni međusobni rad. Podsustavi se mogu odnositi na interne aplikacije neke organizacije ili na spajanje gotovih sustava s drugim sustavima.

Kasnije će se spominjati orkestracija i koreografija unutar slojeva integracije pa je najprije potrebno pojasniti te pojmove. **Koreografija i orkestracija** su dva pristupa upravljanja interakcijama i dijeljenjem podataka između sustava. Često se spominju u okviru web servisa. Općenito unutar samog orkestra glavnu ulogu vođenja ima dirigent te on dirigira cijelim orkestrom. Tako i unutar **orkestracije** postoji jedinica, često nazivana orkestratorom ili upraviteljem, koja upravlja svim interakcijama koje čekaju na odgovor. Orkestracija se organizira oko te jedinice koja pruža instrukcije za interakciju, odnosno koristi se centralizirani pristup. Sustavi su unutar orkestracije međusobno ovisni pa ako dođe do greške na jednom podsustavu može doći do greške cijelog sustava. Orkestracija zahtjeva aktivnu kontrolu veza i transakcija, a orkestrator je odgovoran za pozivanje i kombiniranje usluga. Nasuprot tome u plesu, svaki član plesne grupe zna koreografiju i koji su ispravni slijedni koraci u plesu. Pokreću se većinom neovisno jedan od drugome i u isto vrijeme. Tako i unutar **koreografije** sustava, sustavi ne trebaju čekati instrukcije za interakciju već djeluju neovisno. Koreografija, nasuprot orkestracije koristi decentralizirani pristup. Koreografija određuje interakcije razmjenom poruka, pravilima interakcija i sporazumima između dviju ili više krajnjih točaka. U koreografiji ako se dogodi pogreška na jednom sustavu, ostali mogu nesmetano nastaviti raditi.

2.1. Izazovi integracije sustava

Integracija sustava podrazumijeva integraciju bilo koja dva podsustava pa čak i dva zasebna sustava u jednu cjelinu. U najgorem slučaju takve dvije podcjeline će se razlikovati u protokolima koje koriste, u platformama u kojima su kreirane i programskim jezicima u kojima su pisane. Ne može se uvijek sa sigurnošću reći da će se za svaki podsustav koristiti ista platforma, isti protokol i isti programski jezici ili slično. Takva pojava se naziva **heterogenost** [59]. Heterogenost komunikacijskih protokola označava korištenje različitih komunikacijskih kanala i protokola [59]. Heterogenost sustava predstavlja sustav koji koristi više različitih aplikacija [59]. Pri integraciji se učestalo može susresti s heterogenošću, a sam stupanj heterogenosti raste eksponencijalno pri povećanju broja podcjelina koje se žele integrirati. Heterogenost sustava najčešće se ne može izbjeći zbog toga što se želi integrirati različite funkcionalnosti različitih aplikacija, odnosno želi se da jedna aplikacija nadopunjuje drugu bilo to s podacima, funkcionalnostima ili s nečim trećim. Danas postoje tehnologije i principi s kojima se zaobilaze mogući problemi nastali različitosti sustava.

Integracija sustava uključuje jednu razinu apstrakcije jer apstrahiranjem jednog internog dijela sustava iz drugih omogućuje promjenu tog sustava bez da se utječe na druge [59]. Takva sposobnost ograničavanja širenja promjena je ključna za integraciju kod koje veze mogu biti opsežne, a izmjena jedne aplikacije jako kompleksna.

Uz pojavu heterogenosti, javlja se i različitost formata datoteka u koje aplikacije spremaju podatke. Zato se koriste opisni formati podataka koje svaki sustav može lako kreirati i obraditi. **XML, JSON i drugi formati podataka postaju standard** jer je njima olakšana komunikacija između dva sustava.

Kao što je spomenuto, integracija sustava je povezivanje više različitih sustava u jednu cjelinu. Takva povezanost može rezultirati međusobnoj ovisnosti sustava što predstavlja izazov ako se dogodi kvar ili bilo kakvo odstupanje jer bi to značilo prestanak rada cijelog sustava. Zato **je pravilan rad jednog podsustava** bitan za cjelinu. Pravilni rad sustava je važan već kod integriranja samo dva sustava i njegova važnost raste eksponencijalno sa svakim integriranjem novog sustava. Kako dva sustava najčešće komuniciraju podacima, jako je bitno i da svaki podsustav radi optimalno kako bi se pravilno i pravovremeno izvršavale operacije. Ovakav izazov se najčešće rješava labavom povezanošću dvaju različitih sustava pri čemu se stvara manja ovisnost sustava, a o tome će se govoriti nešto više kasnije.

Integrirati se mogu lokalno udaljeni sustavi što otvara pitanje **sigurnosti** i samog **integriteta podataka.** Jedan sustav ovisi o podacima drugog sustava i zato je potrebno osigurati siguran prijenos podataka u svakom trenutku**.**

Kod integracije jako je bitna semantika podataka. Semantika se odnosi na aspekte značenja koji su izraženi u jeziku, kodu ili nekom drugom obliku predstavljanja [45]. Tako se može dogoditi da podaci koji su zapravo jednaki, ne znače istu stvar. Takva pojava se naziva **semantičko neslaganje** [36]. Primjerice jedan sustav može tretirati udaljenost između dva grada kao zračnu udaljenost, dok drugi sustav tretira tu udaljenost kao cestovnu udaljenost. Takvo semantičko neslaganje je teško prepoznati i popraviti.

2.2. Sloj integracije

Integracija sustava, odnosno povezivanje više sustava zahtjeva infrastrukturu koja može prosljeđivati podatke između sustava. Često se želi da takvo rješenje napravi malo više od prosljeđivanja podataka, odnosno želi se dodati novi sloj funkcionalnosti na postojeće funkcionalnosti aplikacije [54]. Nadodani sloj dopušta automatiziranje složenih poslovnih procesa ili jedinstveni pristup informacijama koje su rasprostranjene po mnogim drugim sustavima [54]. Takav sloj se naziva **slojem integracije.**

Postoje tri pristupa sloju integracije koji su predstavljeni kao uzorci, a koji su [54]**: integracija portala, integracija procesa i agregacija entiteta**.

Ove tri vrste integracijskog sloja međusobno se ne isključuju tj. mogu djelovati paralelno. Sljedeći odjeljci opisuju svaki pristup integraciji.

2.2.1. Integracija portala

*„**Integracija portala** povezuje sustave tako da se pruža jedinstven prikaz korisniku. Integracija portala može dramatično povećati produktivnost jer korisnik ne mora pristupati više neovisnih aplikacija. Umjesto toga, moguće je pristupiti jednoj aplikaciji sa sveobuhvatnim pogledom u svim sustavima u vizualno dosljednom formatu.“* [54]

S obzirom na to da se povezuju sustavi tako da se pruža jedinstven prikaz korisniku, može se reći da je implementacija jednostavna i lagana. No to postavlja druge probleme jer se i dalje oslanja na korisnika za izvođenjem određenih poslovnih procesa i pravila. Bez obzira na to integracija portala često je dobar prvi korak k integraciji. [54]

Primjerice scenarij je takav da poslovanje zahtijeva od krajnjeg korisnika da pristupa informacijama iz različitih, odvojenih sustava. Korisnik sustava želi potvrditi narudžbu o rezervaciji apartmana. Najprije mora pogledati na drugom sustavu je li apartman slobodan u vrijeme rezervacije. Nakon što je potvrđeno da je vremenski slobodan, potrebno je potvrditi rezervaciju kod vlasnika objekta na drugom sustavu. Stalno izmjenjivanje i pretraživanje informacija po različitim sustavima može postati zamorno i sklono greškama [54]. Kako bi se riješio ovakav problem dovoljno je napraviti integraciju portala, odnosno integrirati sustave u

jednu aplikaciju tako da su informacije iz različitih sustava dostupne na zajedničkom korisničkom sučelju. Korisnik tada može pročitati na istom ekranu kakvo je stanje drugih rezervacija i potvrditi rezervaciju na istom ekranu.

Ovakva vrsta integracije sustava je popularna zbog svoje jednostavnosti, brze implementacije i nenametljivosti za razliku od ostalih integracija. Poslovni proces i dalje obavlja korisnik odnosno odlučuje o koracima procesa umjesto da je poslovni proces uključen unutar same aplikacije. Zbog toga što je prosuđivanje na temelju informacija i dalje dužnost korisnika, izbjegava se semantičko neslaganje sustava. Poslovni procesi stoga ne trebaju biti shvaćeni da bi se izvršila integracija portala što nadodaje na fleksibilnost ovakvog rješenja. Ponekad ovakva integracija nije dovoljna, no dobar je početak jer dopušta bolje razumijevanje poslovnih procesa kako bi se moglo kasnije integrirati same procese [54].

2.2.2. Agregacija entiteta

Integracija portala i dalje postavlja velik dio poslovnih odluka na korisnika sustava, umjesto informatizacije tih odluka i procesa. Ovakav problem rješava agregacija entiteta tako da pruža jedinstveno odredište i izvorište podataka aplikaciji. [54]

Agregacija entiteta pruža logičku reprezentaciju objedinjenih podatkovnih entiteta kroz više spremišta podataka, što pruža unificirani način komunikacije aplikacije s podacima. Agregacijom entiteta se pojednostavljuje izrada aplikacije kojoj je potreban uvid u više spremišta podataka. S druge strane, treba pripaziti na integraciju agregacijom entiteta zbog semantičkog neslaganja više sustava. [54]

Agregacija entiteta nam može poslužiti kada su različite informacije o istom objektu rasprostranjene na više spremišta podataka. Primjerice podaci o zaposleniku mogu biti rasprostranjeni tako da je unutar sustava odjela za ljudske resurse sadržan određeni skup podataka o zaposleniku, dok je u odjelu računovodstva drugi skup podataka o zaposleniku. U aplikaciji se javlja potreba za objedinjavanjem ovakvih podataka u jedinstven objekt „zaposlenik“ kako bi aplikacija mogla raditi s tim podacima i prikazivati ih. Koristeći agregaciju entiteta ovo je moguće ostvariti tako da se pruža logična reprezentacija entiteta na razini cijelog poduzeća s fizičkim vezama koje podržavaju pristup i ažuriranje odgovarajuće instance u spremištima podataka [54]. Glavna razlika između integracije portala i agregacije entiteta je što integracija portala pruža jedinstven prikaz korisniku, dok agregacija entiteta pruža jedinstven prikaz aplikaciji.

U slučaju agregacije entiteta se može pojaviti semantičko neslaganje. Plaća zaposlenika u jednom repozitoriju može sadržavati porez dok u drugom ne. Isto tako moguće je postojanje različitih informacija za istu varijablu nekog objekta kao što je nepodudaranje

financijskog stanja istog korisnika na dva različita sustava. Uz semantičko neslaganje mogu se pojaviti i netočni podaci. Zato je jako bitno da se agregacija entiteta izvede pravilno.

Agregacija entiteta uključuje dva koraka. Prvi korak je definiranje reprezentacije na razini cijelog poduzeća koja pruža unificiranu reprezentaciju entiteta. Drugi korak uključuje implementiranje dvosmjerne fizičke veze između definirane reprezentacije entiteta iz prvog koraka i odgovarajuće instance u pozadinskim spremištima. [54]

Postoje dva arhitektonska pristupa provedbe agregacije entiteta: **izravna obrada i replikacija**. Pristup **izravne obrade** dohvaća podatke iz odgovarajućih pozadinskih spremišta u stvarnom vremenu i povezuje informacije u jedinstveni pogled [54]

Pristup **replikacije** zahtjeva zasebni fizički repozitorij unutar sloja agregacije entiteta koji pohranjuje podatke u skladu s prikazom entiteta na razini cijelog poduzeća. Podaci u svakom pozadinskom spremištu repliciraju se u spremište entiteta agregacije. Replikacija zahtjeva implementaciju procesa podrške kako bi se nametnula poslovna pravila koja validiraju replicirane podatke. Replikacija podataka bi se trebala obaviti u oba smjera između spremišta podataka i repozitorija sloja agregacije entiteta. [31]

Pristup replikacije koristi se kada [54]:

- nije potrebna povezanost sa spremištima u realnom vremenu,
- kompliciranа spajanja više instanci jednog entiteta kroz različite repozitorije moraju omogućiti pružanje dosljednog prikaza i
- potrebne su visoke performanse rješenja.

Tijekom dizajniranja sloja agregacije entiteta treba uzeti u obzir **reprezentaciju podataka, identifikaciju podataka, operacije nad podacima i upravljanje podacima. Reprezentacija podataka** uključuje reprezentaciju entiteta i shemu usuglašavanja (eng. *reconciliation*) odnosno podrazumijeva definiranje entiteta na razini cijelog poduzeća, s atributima i ključnim vezama prema drugim entitetima i usklađivanje različitih definicija shema pozadinskih repozitorija [54]. Uz definiranje prikaza podataka, treba se ustanoviti format u kojem je reprezentacija podatka pohranjena. Osim toga potrebno je uvođenje odgovarajućeg mehanizma koji **jedinstveno identificira svaki entitet** kroz sve repozitorije kao što je referenca entiteta [54]. **Operacije nad podacima** uključuju način na koji se izvršavaju operacije kreiranja, čitanja, ažuriranja i brisanja nad podacima, što se može pratiti korištenjem reference entiteta [54]. **Upravljanje podacima** uključuje uspostavljanje vlasništva nad održavanjem i uspostavljanje procesa promjene upravljanja [54].

2.2.3.Integracija procesa

Integracija procesa bazira se na orkestraciji interakcija između više sustava [54]. Često je slučaj da automatizirani poslovni procesi zahtijevaju integraciju pri čemu je preporučljivo modelirati procese izvan aplikacija zbog labave povezanosti aplikacija. Integracija procesa prati stanje svake instance poslovnog procesa te omogućuje centralizirano izvještavanje [54].

Integracijom procesa može se spojiti više različitih sustava koji su dio jedne poslovne funkcije. Primjerice unutar organizacije postoji sustav za obradu naloga narudžbe s određenim proizvodima što može zahtijevati sudjelovanje sustava s podacima o kupcima, inventarni sustav s podacima o stanju na skladištima, sustav otpreme koji sadržava podatke o otpremi određenih proizvoda i sustav financija. Integracijom tih procesa automatizirala bi se obrada narudžbenice i otpremnice. Procesi bi se izvodili slijedno, odnosno sloj integracije procesa bi pozivao sustave za izvođenje svakog sljedećeg koraka. Kod integracije procesa u obzir treba uzeti moguća kašnjenja, promjene na pojedinim procesima, povezanost sustava i izvršavanje samih procesa.

Jako je bitno prije dizajniranja sloja integracije procesa da su **procesi unutar poslovne funkcije razumljivi i dokumentirani**. Najprije je potrebno definirati poslovni model procesa, nakon čega je moguće kreirati upravitelja procesa koji je povezan sa svakim sustavom i može izvršavati te korake. Na svaki zahtjev upravitelj procesa kreira novu instancu procesa na bazi modela procesa koja sadrži stanje procesa i dodatne informacije [54]. Upravitelj procesa određuje koji će se idući korak izvesti nakon što je određena instanca izvršila svoj zadatak što dozvoljava da više različitih aplikacija rade individualno bez znanja o idućem koraku i omogućuje da ako dođe do prestanka rada jedne aplikacije ne prestaje rad cijelog sustava [54]**. Integracija procesa** pruža odvojenost između definicije procesa u modelu, izvršavanja procesa u upravitelju procesa i implementacije operacija unutar zasebnih aplikacija [54]. Upravitelj procesa često je kreiran tako da izlaže vanjsko sučelje pri čemu korisnik, poslovni partner ili druga aplikacija mogu pokrenuti poslovni proces [54]. Tako se jedna poslovna funkcija može koristiti kao dio druge veće poslovne funkcije.

Upravitelj procesa se prema svemu navedenom ponaša kao svaka aplikacija koja implementira poslovnu logiku, no za razliku od tradicionalne aplikacije on iskorištava već napravljene funkcije unutar drugih sustava [54]. Upravitelj procesa mora biti u stanju povezati poruke vanjskih sustava s instancama poslovnog procesa kojem su namijenjene, podržati dugotrajne transakcije, obraditi iznimke nastale u pojedinačnim koracima poslovnog procesa i osigurati kompenzaciju za poništavanje obavljenih radnji u poslovnom procesu ako dođe do

kvara u istome [54]. Osim potreba upravitelja procesa javlja se potreba za korelacijom poruke i ispravne instance poslovnog procesa.

Dizajn integracije procesa treba podržavati **atomarne i dugotrajne transakcije** [54]. Najbolji primjer atomarne transakcije su transakcije u bazama podataka. **Atomarne** transakcije imaju sljedeća svojstva: atomarnost, konzistentnost, izolaciju i trajnost [54]. **Dugotrajne** transakcije su transakcije koje se događaju tijekom duljeg perioda, ne mogu biti nikako atomarne. Ako su dugotrajne transakcije uključene u transakciju, one po prirodi nisu transakcijske, a mogu sadržavati grupirane, atomarne ili dugotrajne transakcije [54]. Zbog same orkestracije i čekanja na izvođenje idućeg koraka, mora se omogućiti prekidanje transakcije u slučaju ako je ona dugotrajna pri čemu se ponovnim paljenjem treba nastaviti od zadnje izvršene akcije u transakciji.

Integracija procesa se učestalo koristi pa su zato već kreirani i standardi koji opisuju modele procesa primjerice jezik za upravljanje poslovnim procesima (eng. ***B**usiness **P**rocess **M**odeling **L**anguage - BPML*), jezik izvršavanja poslovnih procesa (eng. ***B**usiness **P**rocess **E**xecution **L**anguage - BPEL*) i sučelje koreografije web servisa (eng. ***W**eb **S**ervices **C**horeography **I**nterface - WSCI*) [54].

2.3. Povezivanje sustava

Danas je manji broj sustava izoliran, dok je veći broj onih koji su povezani s drugim aplikacijama i servisima. Malo poduzeće u realnosti ne treba kreirati aplikaciju koja se integrira s ostalima. Kako poduzeće raste javit će se potreba integracije i zato treba od početka razmišljati kako kreirati aplikaciju koja će se moći kasnije integrirati s ostalima. Arhitektura aplikacije treba biti konstruirana tako da sama aplikacija djeluje kao početni kostur na koji se kasnije može dodavati funkcionalnosti i dodatno ju proširivati. Tako se ušteđuje vrijeme i novac.

Danas se sve više arhitektura aplikacije bazira na troslojnoj arhitekturi sastavljenoj od sloja prezentacije, sloja podataka i sloja poslovne logike. Sloj prezentacije prikazuje informacije krajnjem korisniku i prima korisničke ulazne podatke, sloj poslovne logike sadrži poslovne funkcije koje koriste poslovne podatke dok sloj podataka sadrži i sprema podatke u spremišta [53]. Prezentacijski i podatkovni sloj su povezani preko sloja poslovne logike. Podatkovni sloj prikazuje korisniku podatke te prima korisnički unos i prosljeđuje taj unos sloju poslovne logike koji obrađuje te podatke i izvršava određene funkcije. Ako je potrebno dohvaća stare podatke kroz zahtjeve podatkovnom sloju ili šalje ažurirane i nove podatke podatkovnom sloju koji ih sprema.

Integracija je moguća na svakom sloju, no ovisi o tehnologiji koja će biti izabrana za integraciju, o samoj funkciji i strukturi sloja. Kako postoje tri različita sloja arhitekture aplikacije tako se dijele i tri različite vrste integracije: Ovisno o troslojnoj arhitekturi postoje tri vrste povezivanja s integracijskim slojem [54]:

- **prezentacijska integracija –** integracijski sloj može izvući informacije iz aplikacijskog prezentacijskog sloja**;**
- **funkcionalna integracija –** integracijski sloj može komunicirati sa slojem poslovne logike kroz aplikacijsko ili servisno sučelje;
- **podatkovna integracija –** integracijski sloj može premještati podatke u i izvan podatkovnog sloja**.**

U nastavku stoji objašnjenje zašto se koristi koja vrsta povezivanja, uz priložene uzorke koji se koriste pri integraciji.

2.3.1. Podatkovna integracija

Podatkovna integracija (eng. *data integration*) označava integraciju aplikacija zajedničkim korištenjem ili dijeljenjem podataka. Puno aplikacija sprema velike količine podataka u spremišta podataka kao što su neprekinute datoteke (eng. *flat file*) i relacijske, hijerarhijske ili objektno orijentirane baze podataka. Aplikacije koje pak trebaju te informacije mogu im pristupiti izravno iz tih spremišta podataka. Povezivanje aplikacija putem spremišta relativno je jednostavan zadatak. Obično se koriste FTP (eng. ***F****ile* ***T****ransfer* ***P****rotocol*) protokol, zakazane BAT datoteke (eng. *batch file*), sustavi za upravljanje bazama (eng. ***D****ata****B****ase* ***M****anagment* ***S****ystem - DBMS*) , alati za izvlačenje, pretvaranje i učitavanje (eng. ***E****xtrating,* ***T****ransforming and* ***L****oading - ETL*) te integracijski poslužitelji [54].

Kod **podatkovne integracije** u obzir treba uzeti toleranciju vremena čekanja jer neki oblici integracije podataka uključuju kašnjenje između ažuriranja podataka koje koristi više aplikacija [52]. Pri dohvaćanju podataka, sustav može povući podatke iz baze podataka ili ih baza podataka može sama pustiti kada se promjene dogode. Povlačenje podataka je manje nametljivo, no puštanje ili slanje podataka minimizira vrijeme čekanja. Osim tolerancije vremena čekanja, postoji problem sinkronizacije podataka s obzirom na to da više aplikacija ima dopušteno ažuriranje podataka.

Integracija na sloju podataka moguća je kroz tri uzorka koji predstavljaju tri vrste integracije podataka.

Prvi od njih je **zajednička baza podataka** (eng. *Shared Database)*. Sustavi se mogu integrirati tako što se omogući dijeljenje jedne instance baze podataka između više aplikacija koristeći uzorak zajedničke baze podataka [31]. Sve različite aplikacije koje se integriraju bi

tada direktno pohranjivale i čitale podatke iz jedne baze podataka, što omogućuje smanjivanje vremena čekanja. Čitanje podataka iz baze podataka je bezopasno, no postavlja se problem upisivanja u bazu podataka što može rezultirati greškama unutar aplikacija [54]. Potrebno je također napraviti jedinstvenu shemu baze podataka koju će koristiti sve aplikacije. Iako mehanizmi integriteta transakcije štite bazu podataka od višestrukih istodobnih ažuriranja, oni ne mogu zaštiti bazu od upisivanja loših podataka pa se zato se u većini slučajeva u bazi implementira skup ograničenja za podatke [31].

Zatim postoji uzorak **održavanja kopije podataka** (eng. *Maintain Data Copies*). Takav pristup odnosi se na kreiranje više kopija baze podataka i njihovo distribuiranje kroz sustav [52]. Ovakav uzorak zahtjeva sinkronizaciju i replikaciju podataka pri čemu bi više aplikacija sadržavalo kopiju istih podataka [52].

Zadnji uzorak je **prijenos datoteka** (eng. *File Transfer*). U ovom uzorku, jedna aplikacije kreira datoteku i prenosi ju tako da ju druga može koristiti. Datoteke mogu biti kreirane u redovitim intervalima zbog sinkronizacije dva ili više sustava, a same datoteke se ne zadržavaju u aplikaciji koja ih kreira. [31]

2.3.2. Funkcionalna integracija

Kao što i ime govori pri funkcionalnoj integraciji spaja se integracijski sloj sa slojem poslovne logike. **Funkcionalna integracija** omogućuje drugim aplikacijama i servisima iskorištavanje poslovne logike koja je učahurena u nekom sustavu [54]. Funkcionalna integracija povezuje aplikacije kroz sučelja i specifikacije, no to ne mora značiti da svaka aplikacija ima sučelja i specifikacije [54]. Za funkcionalnu integraciju bitno je da je poslovna funkcija dostupna unutar poslovne logike izvorne aplikacije i da je sučelje (eng. ***A****pplication* ***P****rogramming* ***I****nterface – API*) izvorne aplikacije dostupno na daljinu, a ako je to ne moguće može se koristiti međuprogram koji će prevoditi dolazne poruke drugih aplikacija u zahtjev lokalno dostupnog API-a [54]. Unutar različitih izvora često se spominje aplikacijska integracija kao drugi naziv za funkcionalnu integraciju.

Funkcionalna integracija moguća je kroz tri alternative: **integracija distribuiranim objektima, integracija međuprogramom (eng.** *middleware*) **orijentiranog prema porukama i integracija orijentirana servisima** [54].

Integracija distribuiranim objektima proširuje model objektno orijentiranog programiranja na distribuirana rješenja pri čemu objekti unutar jedne aplikacije komuniciraju s objektima u drugoj udaljenoj aplikaciji na isti način na koji bi lokalno komunicirali s drugim objektom [54]. Takva komunikacija označava da druga aplikacije upravlja životnim vijekom objekta neke aplikacije. U okviru integracije distribuiranim objektima često se spominje **poziv**

udaljene procedure (eng. *Remote Procedure Call - RPC*). Poziv udaljene procedure primjenjuje princip učahurivanja. Ako jedna aplikacije treba promijeniti podatke druge, to obavlja jednostavnim pozivom ili ako jedna aplikacija treba informacije od druge aplikacije, traži direktno tu aplikaciju [31].

Integracija međuprogramom orijentiranog prema porukama povezuje sustave preko asinkronih redova poruka temeljenih na međuprogramu orijentiranog na poruke [54]. Povezivanje sustava redovima poruka implicira da ti sustavi komuniciraju porukama . Bitno je naglasiti da je komunikacija asinkrona i trajna čime se osigurava izbjegavanje gubitka poruka zbog kvara na sustavu. Paradigma zahtjev-odgovor se ostvaruje tako da sustav koji traži funkcionalnost pošalje poruku zahtjeva putem reda poruka sustavu koji pruža funkcionalnosti. Sustav koji pruža funkcionalnost preuzima poruke po primitku iz reda poruka, procesuira poruku, izvršava potrebne radnje i kreira odgovor koji šalje nazad putem reda poruka. Sustavi poruka će biti objašnjeni nadalje u radu u poglavlju Sustavi poruka.

Servisno orijentirana integracija povezuje sustave tako da komuniciraju preko web servisa [54]. Zbog labave povezanosti i heterogenosti komunikacije web servisima, ova vrsta integracije uključuje standarde poput XML, SOAP, HTTP i drugih standarda kako bi se osigurala interoperabilnost između krajnjih točaka. Povezivanje web servisima će biti objašnjeno nadalje u tekstu u poglavlju web servisi.

2.3.3. Prezentacijska integracija

Aplikacije se mogu spojiti tako da ih se povezuje preko prezentacijskog niza bajtova [54]. Prezentacijska integracija uključuje simuliranje korisničkih klikova pri integraciji sustava. **Spajanje prezentacijskog sloja** predstavlja najmanje invazivni način povezivanja više aplikacija jer ovaj oblik integracije ne zahtijeva nikakve promjene na aplikaciji domaćina.

Nedostatak je što **je simuliranje interakcije korisnika glomazno i neučinkovito -** simulacija zahtijeva raščlanjivanje podataka ili funkcionalnosti iz toka bajtova, što poništava operacije prezentacijske logike [54]. Zbog simulacije korisnika dvije aplikacije su povezane tako da mogu koristiti samo one funkcionalnosti koje su dostupne običnom korisniku. Uz to svaka promjena na prezentaciji domaćina uvjetuje i promjenu aplikacije koja ju koristi. Ovakav način integracije pozitivan je zbog jeftinog načina integracije.

2.4. Topologije Integracije

Aplikacija koja je kreirana u vidu integracije može u isto vrijeme biti pružatelj usluge drugim aplikacijama i primatelj usluge drugih aplikacija kroz jednostavno **spajanje od točke do točke**. Ako je sustav pružatelj neke funkcionalnosti, tada se vjerojatno koristi **sabirnica**

poruke (eng. *message bus*) ili **posrednik poruka** (eng. *message broker*). Komponente aplikacije mogu slati poruke drugim aplikacijama koje su **pretplatnici** aplikacije koja **objavljuje** uslugu. Može se zaključiti da kako bi se kreirao sustav koji je ujedno i pružatelj usluge i korisnik usluge potrebno je najprije razumjeti **integracijske topologije**; uzorak Broker, povezivanje od točke do točke, sabirnicu poruka i objavljivanje/pretplatu (eng. *Publish/Subscribe*).

2.4.1. Povezanost od točke do točke

Povezanost od točke do točke (eng. *point-to-point*) predstavlja **najlakšu** integraciju dva sustava. Kao što i samo ime govori, omogućava prijenos poruke od jedne do druge točke, odnosno samo jedan primatelj zaprima poruku pošiljatelja.

Kako bi prijenos poruke od točke do točke bio moguć pošiljatelj mora unaprijed znati kome šalje poruku te također ponekad mora prije slanja poruke transformirati poruku u željeni format primatelja [54]. Pri povezivanju od točke do točke svaki sustav određuje adresu svih ostalih čvorova s kojima treba komunicirati, pa tako pri promjeni adrese primatelja, svi sustavi koji komuniciraju s istim moraju se ažurirati što može rezultirati s velikim vremenskim i ostalim troškovima, koji se povećavaju kako veličina integracijske mreže raste.

Pri povezivanju od točke do točke najčešće se zahtijeva konfiguracija usmjeravanja i transformacija podataka prije slanja zbog ciljnog sustava na koji se šalje poruka što može rezultirati duplikacijom koda na svakom sustavu koji zahtjeva transformaciju i preusmjeravanje [54]. Slabosti ove povezanosti se izbjegavaju dodavanjem sloja između točaka koji sadržava **posrednika (eng.** *broker*).

2.4.2. Broker

Uzorak Broker ima različite varijante, ali originalno uzorak dolazi iz knjige „Softverska arhitektura orijentirana uzorcima“ (eng. *Pattern Oriented Software Architecture* - POSA). Uzorak Broker koristi se i kod kreiranja aplikacije i kod integriranja aplikacije. **Broker** uzorak se centrira oko posrednika koji je postavljen između klijenta i servera, a služi za koordinaciju i komunikaciju komponenata.

Unutar POSA knjige **Broker** se definira kao uzorak koji se koristi za strukturiranje distribuiranih softverskih sustava s odvojenim komponentama koje surađuju pozivima udaljenih servisa [21]. Cilj takvog uzorka je graditi sustave koji su skup odvojenih i međudjelujućih komponenti tako da se poveća elastičnost, otpornost i promjenjivost. Unutar POSA Broker uzorka sudjeluju klijent, server, posrednik , posrednički poslužitelj (eng. *proxy*) na klijentskoj strani, posrednički poslužitelj na serverskoj strani i uzorak Bridge [21].

Namjera posrednika u uzorku Broker je odvojiti sustav pošiljatelja zahtjeva (nadalje izvorni sustav) od sustava primatelja zahtjeva (nadalje ciljni sustav). U komunikaciji gdje su izvorni resursi poruka koja se šalje od izvornog sustava i ciljna poruka koju prima ciljni sustav , postoje sljedeće tri odgovornosti [54]: **usmjeravanje, registracija krajnjih točaka i transformacija**. **Usmjeravanje** je određivanje lokacije ciljnog sustava. **Registriranje krajnje točke** je mehanizam registracije sustava unutar posrednika zbog olakšanog usmjeravanja i utvrđivanja postojanja sustava. S obzirom na to da različiti sustavi imaju različite ulazno/izlazne podatke, **transformacija** se koristi kako bi se podaci pretvorili iz jednog formata u drugi format.

Kao što je navedeno; postoji više varijanti uzorka Broker, a u okviru distribucije pojavljuju se **direktni Broker, indirektni Broker i Broker poruka** [54]. **Direktni Broker** uspostavlja inicijalnu komunikaciju između krajnjih točaka, pri čemu dvije krajnje točke komuniciraju direktno preko korištenja posredničkog poslužitelja na klijentskog strani i posredničkog poslužitelja na serverskoj strani [54]. Nasuprot tome**, indirektni Broker** je posrednik pri čemu kroz njega prolazi sva komunikacija između krajnjih točaka što omogućuje klijentu nepoznavanje detalja ciljnog sustava pružajući centralnu kontrolu toka poruke [54]. **Broker poruka** komunicira isključivo koristeći poruke i neizravnu komunikaciju [21,53]. Broker poruka kao fizička komponenta može primati poruke s više izvora, odrediti točno odredište i preusmjeriti poruku na ispravan kanal.

Broker poruka često je korišteni uzorak integracije, a često je referiran hub-and-spoke arhitekturi zbog strukture orijentirane oko centralnog posrednika [54]. Broker se zbog korisnosti u distribuiranim sustavim učestalo koristi pa je korišten u DCOM (eng. *Microsoft **D**istributed **C**ommon **O**bject **M**odel*), CORBA (eng. ***C**ommon **O**bject **R**equest **B**roker **A**rchitecture*), UDDI (eng. ***U**niversal **D**escription **D**iscovery and **I**ntegration*) i Microsoft Biztalk serveru [54].

2.4.3. Sabirnica poruka

Sabirnica je u okviru elektrotehnike, skup vodova koji prenose određenu skupinu podataka, primjerice adresna sabirnica prenosi adrese, podatkovna sabirnica podatke i upravljačka sabirnica upravljačke signale. Sabirnica poruke pruža zajednički mehanizam komunikacije između različitih sustava.

Sabirnica poruke je analogna komunikacijskoj sabirnici u kompjuterskom sustavu, koja služi kao žarišna točka za komunikaciju između procesora, glavne memorije i perifernih uređaja, pa se time može reći da se sabirnica poruka sastoji od niza dijelova, a koji su **zajednička komunikacijska infrastruktura, adapteri i zajednička struktura naredbi** [31]**.**

Kako su komunikacijske sabirnice sastavljene od vodova može se reći da su vodovi zajednička komunikacijska infrastruktura fizičkih sabirnica kompjutera, pa slično tome

potrebno je kreirati **zajedničku komunikacijsku infrastrukturu sabirnice poruka** koja će poslužiti jednakoj svrsi kao i kod fizičke sabirnice [31]. Zajednička infrastruktura može se postići korištenjem **usmjerivača poruka** ili pomoću mehanizma **Publish/Subscribe** [54]. Tijekom integracije pojavljuje se heterogenost sustava pa zbog toga treba pronaći način povezivanja različitih sustava sa sabirnicom poruka što se najčešće izvodi s komercijalnim ili prilagođenim **adapterima za kanale** [31]**.** Kako bi dva sustava mogla razumjeti sabirnice poruka, potrebno je da se standardizira ili uspostavi **zajednička struktura naredbu**. Zbog ovih značajki sabirnice poruka, nova aplikacija može se pretplatiti na poruke sabirnice bez promjene na ostalim, već spojenim aplikacijama. Spajanje nove aplikacije je jednostavno zbog zajedničke strukture naredbi, a ako je potrebno može se iskoristi adapter za učahurivanje bilo kojeg prijevoda koji je inicijalno potreban za spajanje sa sabirnicom poruka [54]. Ovo je i glavna prednost sabirnice poruka jer se ne kreiraju dodatni troškovi tijekom dodavanja novih aplikacija.

Može se zaključiti da je **sabirnica poruka** sastavljena od adaptera za kanale, zajedničkog skupa naredbi i zajedničke infrastrukture, a služi za razmjenu poruka koja dopušta različitim sustavima komuniciranje kroz dijeljenje sučelja.

2.4.4.Publish/Subscribe

Uzorak **Publish/Subscribe** opisuje suradnju u kojoj se jedan sustav pretplaćuje na promjene poruka ili poruke događaja koje proizvodi, odnosno objavljuje drugi sustav. Postoje tri proširenja uzorka Publish/Subscribe; temeljen na listi, temeljen na emitiranju i temeljen na sadržaju [54]. Uzorak **Publish/Subscribe temeljen na listi** temeljen je na popisu pretplatnika za određeni subjekt te ako se izvrši događaj, subjekt mora obavijestiti svakog pretplatnika s popisa [54]. Pri **Publish/Subscribe temeljenog na emitiranju**, izdavač događaja stvara poruku i emitira je na lokalnu mrežu pri čemu čvor osluškivanja servisa obrađuje poruku ako odgovara linija izdavača [54]. Uzorak **Publish/Subscribe temeljen na sadržaju** u neku ruku uključuje oba navedena uzorka, a uključuje usmjeravanje poruka do svoj konačnog odredišta iščitavanjem odredišta iz sadržaja poruke [17]. Sustavi temeljeni na sadržaju fleksibilniji su jer su pretplate povezane s određenim informacijskim sadržajem je sadržaj poruke promatran kao jedan dinamički logički kanal [17].

3. Web servisi

Danas se pojam web servisa najčešće veže uz Amazon web servise, Google web servise i Google platformu u oblaku. Nije slučajnost da se taj pojam veže uz tehnološki vodeće tvrtke. Vodeće tehnološke tvrtke su shvatile potrebu za web servisima i podigle ljestvicu baveći se potrebom za razvojem aplikacija. Postoje razne definicije web servisa, no kako bi se shvatilo što su web servisi najprije će biti objašnjene ključne značajke web servisa.

Iako sam pojam web servis ima razna, neprecizna i nova značenja, kao što i samo ime govori, web servis je web aplikacija koja omogućuje razmjenu poruka internetom od aplikacije do aplikacije.

Web servisi koriste poruke za razmjenu informacije putem **protokola HTTP** (eng. ***H**yper**T**ext **T**ransport **P**rotocol*), **ili HTTPS** (eng. ***H**yper**T**ext **T**ransfer **P**rotocol **S**ecure*) naizmjenice slanjem zahtjeva i odgovora, odnosno temelje se na paradigmi zahtjev – odgovor. Kako bi se ta paradigma najjednostavnije ostvarila, rad web servisa uključuje poslužitelje i klijente pri čemu klijenti šalju zahtjev prema poslužitelju, na što dobivaju odgovor od njega. Prema tome, može se reći da je web poslužitelj zapravo softver čija je glavna funkcija odgovoriti na klijentski HTTP zahtjev. Web servisi su stoga klijentske i poslužiteljske aplikacije koje komuniciraju preko HTTP ili HTTPS protokola.

Za svu komunikaciju web servisa najčešće je korišten **XML** (eng. *E**X**stensible **M**arkup **L**anguage*). Klijent poziva web uslugu slanjem XML poruke a zatim čeka odgovarajući XML odgovor. Kako je sva komunikacija odvijana putem XML-a, web servisi nisu usko povezani ni s jednim operativnim sustavom ili programskim jezikom pa tako primjerice Java aplikacija može razgovarati s Perl aplikacijom. Uz XML često se koristi i JSON (eng. ***J**ava**S**cript **O**bject **N**otation*). JSON je često alternativa za XML pa prema tome može se zaključiti da služi istoj svrsi. Razlika je najviše u strukturi samog zapisa. Tako se može reći da je web servis softverska komponenta koja sebe čini dostupnom na internetu i koristi standardizirani XML ili drugi sustav razmjena poruka.

Jedna od glavnih značajki web servisa o kojoj će se govoriti nešto više u idućim odjeljcima je interoperabilnost. **Interoperabilnost** označava sposobnost komuniciranja s više sustava, odnosno sposobnost komuniciranja različitih komponenata [56]. Uz interoperabilnost često se spominje i **labava povezanost** koja omogućuje da klijent pošalje zahtjev i nastavlja

raditi drugi koristan posao dok ne dobije odgovor. U drugu ruku čvrsta povezanost odnosi se na prestanak rada komponente sve dok se ne dobije odgovor.

S obzirom na sve navedeno može se zaključiti da je web servis bilo koja **klijentska ili poslužiteljska** aplikacija, softver ili tehnologija u oblaku koja komunicira putem standardiziranog **protokola HTTP ili HTTPS**, a omogućuje **interoperabilnost, labavu povezanost, komunikaciju i razmjenu podataka** putem slanja poruka internetom pri čemu se najčešće koristi **XML ili JSON.**

3.1. Karakteristike web servisa

Kroz objašnjavanje web servisa, njihovog rada i samog pojma u prošlom odjeljku već su spomenute neke od osnovnih karakteristika web servisa. U ovom poglavlju navedene su sve osnovne karakteristike web servisa.

Karakteristike web servisa su [56, 51]:

- XML bazirani,
- neovisni o programskoj platformi i programskim jezicima,
- pružaju bržu i jednostavniju integraciju aplikacija,
- omogućuju ponovnu iskoristivost,
- omogućuju labavu povezanost,
- pružaju interoperabilnost,
- omogućuju sinkronu i asinkronu funkcionalnost i
- pružaju podršku RPC.

XML je jezik izrađen 1998. godine, a dizajniran je za samoopisivanje informacija. Podskup je SGML-a (eng. ***S**tandard **G**eneralized **M**arkup **L**anguage*) te je kao jezik meta-oznaka stvoren da se pomoću njega definiraju vlastiti identifikatori, etikete, i oznake prema potrebama strukture podataka. Pomoću njega moguće je definirati skup oznaka koje će biti sastavni dio specifikacije, a sadrži zapis stablaste strukture podataka. [20]

Web servisi koriste XML za reprezentaciju podataka u transportnom sloju i sloju prezentacije. Aplikacije web servisa najčešće koriste neki standard razmjene poruka, pa su tako web servisi većinom **bazirani na XML standardu**. Korištenjem XML-a miču se ovisnosti o mreži, operacijskom sustavu ili platformi jer je XML uobičajeni jezik kojeg svi mogu razumjeti i obraditi. Zbog toga aplikacija pisana u Java jeziku može poslati XML zahtjev pri čemu servis vrača XML odgovor.

S obzirom na to da se za zahtjev i odgovor u komunikaciji web servisa koristi standard, aplikacije koje dohvaćaju podatke web servisom nisu direktno ovisne o tom standardu. Zato

aplikacije koje su pisane u različitim jezicima mogu komunicirati razmjenjivanjem podataka kroz web servise čiji je rezultat **neovisnost o programskoj platformi i programskim jezicima**.

Labava povezanost (eng. *loose coupling*) je takva povezanost komponenti sustava da su iste otporne u slučaju kašnjenja i kvarova na mreži, a odnosi se na povezivanje klijenta i servisa [35]. Klijent i servis su povezani tako da nisu ovisni o međusobnim promjenama tako da promjena web servisa ne utječe na način pozivanja istog u klijentu. Opisi samih servisa se nalaze u sučeljima što okruženju dodaje sloj apstrakcije koji omogućava da veza bude fleksibilna i prilagodljiva. Neovisnost o programskoj platformi i programskim jezicima te labava povezanost omogućavaju **bržu i jednostavniju integraciju aplikacija**.

W3C (***W**orld **W**ide **W**eb **C**onsortium*) definira web servis kao softverski sustav namijenjen pružanju interoperabilnosti između računala preko mreže [19]. Može se zaključiti da je interoperabilnost najvažnija značajka web servisa i oni sami postoje sa svrhom pružanja interoperabilnosti [44]. Cilj interoperabilnosti je stvoriti takvu okolinu u kojoj je omogućena nesmetana komunikacija između bilo kojeg broja različitih aplikacija i komponenata. **Interoperabilnost** je stoga mogućnost sustava da radi nesmetano bez ikakvih ograničenja tijekom implementacije s bilo kojim drugim sustavom ili tijekom pristupanja bilo kojem drugom sustavu [24]. Postoje četiri vrste interoperabilnosti: tehnička, semantička, procesna i pravna [24].

Web servisi omogućavaju **sinkronu i asinkronu komunikaciju**. Sinkrona komunikacija je komunikacija koja se odvija u realnom vremenu, a sam pojam „sinkrono“ označava mogućnost postojanja i izvršavanja događaja u isto vrijeme. Realno vrijeme označava vrijeme koje se odvija upravo u ovom trenutku bez kašnjenja, odnosno ostvaruje se komunikacija koja je slična komunikaciji dvije osobe u stvarnom svijetu. **Sinkrona funkcionalnost** u web servisima označava povezivanje klijenta i izvršavanje rada web servisa [56]. Komunikacija u realnom vremenu prepoznatljiva je u aplikacijama za čavrljanje (eng. chat) gdje su dobivene poruke dostupne istog trenutka kada su poslane. Tako će klijent koji je poslao zahtjev web servisu u sinkronoj komunikacija prije izvršavanja drugih operacija čekati na odgovor web servisa. Sinkronim web servisima klijent pristupa kroz poznate web protokole, a sinkroni web servisi su omogućeni i kroz RPC orijentirane poruke.

Asinkrona komunikacija je razmjena poruka pri čemu se podaci mogu prenositi s kašnjenjem i prekidima nasuprot sinkronom prenošenju podataka u stalnom toku u realnom vremenu. **Asinkrona funkcionalnost** web servisa omogućava klijentu pozivanje servisa pri čemu on ne mora čekati na odgovor web servisa već se mogu obavljati i druge operacije paralelno [56]. Asinkrone operacije se najčešće koriste kako bi se osiguralo da se aplikacija

ne zaustavi ako određena dođe do neizvršavanja ili prekida pozvane operacije web servisa. Asinkroni klijenti rezultat dobivaju kasnije tijekom izvršavanja koda, dok sinkroni čekaju na odgovor kako bi nastavili s izvršavanjem. Asinkroni web servisi su ključan faktor u omogućavanju labavog povezivanja sustava [35].

Web **servisi podržavaju RPC** pružajući osobne usluge, jednake onima tradicionalne komponente ili prevođenjem zahtjeva u poziv komponente [28]. RPC je sposobnost softvera da pokrene izvršavanje procedure na različitom adresnom mjestu, bilo na drugom uređaju ili na mreži [28]. Takav poziv se tretira kao lokalni bez da se mora specificirati daljinska interakcija. Web servisi omogućavaju klijentima da pozivaju procedure, funkcije i metode na daljinskim objektima koristeći XML bazirani protokol. SOAP RPC poruke sadrže XML koji predstavlja poziv metode ili odgovor na metodu [28]. Tako će se XML dio poruke na poslužitelju pretvoriti u poziv metode i odgovor će biti zapisan u obliku odgovora korisniku.

3.2. Arhitektura web servisa

Arhitektura web servisa se sastoji od tri sloja. Na najvišoj razini nalaze se **aplikacijski servisi**, zatim je tu **servisna infrastruktur**a i na najnižoj razini su **standardi i protokoli**. Za lakše razumijevanje arhitekture priložena je sljedeća slika.

<table>
<tr><td colspan="4">ARHITEKTURA WEB SERVISA:</td></tr>
<tr><td colspan="4">1. Aplikacijski servisi</td></tr>
<tr><td>App servis</td><td>App servis</td><td>App servis</td><td>…</td></tr>
<tr><td colspan="4">2. Servisna infrastruktura</td></tr>
<tr><td rowspan="3">Zajednički servisi:

Sigurnost, naplata, provjera korisnika</td><td colspan="3">Sustav za upravljanje dostupnošću servisa: monitoriranje, QoS, sinkronizacija, upravljanje konfliktima</td></tr>
<tr><td colspan="3">Sustav upravljanja podacima o servisima: direktoriji, broker-i, repzitoriji, transformacija podataka</td></tr>
<tr><td colspan="3">Sustav upravljanja transportom: message quing, filtering, routing, orkestracija resursa</td></tr>
<tr><td colspan="4">3. Standardi i protokoli (najniža razina)</td></tr>
<tr><td colspan="2">Programski standardi : WSDL, UDDI, XML</td><td colspan="2">Komunikacijski protokoli: SOAP, HTTP, TCP/IP</td></tr>
</table>

Slika 1: Arhitektura web servisa (Izvor: vlastita izrada, prema: [27])

Kao što se može iščitati sa slike, u **servisnu infrastrukturu** spadaju zajednički servisi za sigurnost, naplatu i provjeru korisnika, sustav za upravljanje dostupnošću servisa, sustav upravljanja podacima o servisima i sustav upravljanja transportom. Osnovni **programski**

standardi za web servise su **WSDL, UDDI i XML**. WSDL je nezaobilazan standard koji se koristi za objavljivanje servisa koje koristi pružatelj dok se UDDI može zaobići. Kao dodatni, noviji programski standardi javljaju se WADL i JSON.

UDDI je posrednik između korisnika i pružatelja, a služi za registraciju i pretraživanje servisa te pristup WSDL dokumentima [51]. Servisno orijentirana arhitektura se oslanja na mogućnost identificiranja servisa i njihovih mogućnosti, na postojanje direktorija koji opisuje servise dostupne u svojoj domeni, a tu ulogu preuzima registar servisa. S obzirom na to da je UDDI postao standard za registar servisa servisno orijentirane arhitekture, pod registrom servisa automatski se misli na UDDI. Ukratko UDDI služi korisnicima za otkrivanje servisa koji je objavljen u registru.

WSDL (eng. ***W****eb* ***S****ervices* ***D****escription* ***L****anguage*) je nezaobilazan standardni jezik za opis web servisa koji specificira što web servis radi, potrebne formate podataka i protokole [22]. Detaljnije, ono je XML sintaksa koja opisuje koje ulazne podatke je potrebno dati servisu i kakav izlaz se očekuje odnosno opisuje njegove funkcionalnosti, način pristupa i pozivanja servisa. Svaki servis mora sadržavati WSDL sučelje u kojem je zapisano sve što je potrebno za upotrebu tog servisa. WSDL definira 4 tipa operacija koji web servisi mogu izvršiti [22]:

- **jedan smjer (eng.** *one-way*) - operacija koja prima poruku ali ne odgovara na nju;
- **zahtjev-odgovor (eng.** *request-response*) - operacija koja prima poruku i odgovara na nju (najčešći tip);
- **traženi odgovor (eng.** *solicit response*) - operacija koja šalje zahtjev za odgovorom i čeka na odgovor;
- **obavijest (eng.** *notification*) - operacija koja šalje poruku, ali ne čeka na odgovor.

Danas se za opis strukture XML dokumenta koristi XML definicija sheme (eng. ***X****ML* ***S****chema* ***D****efinition - XSD*) koji podržava tip podataka, koristi XML sintaksu, proširiv je i omogućava grupiranje oznaka [24]. Komponente XML Schema-e omogućavaju validaciju strukture XML dokumenta u ovisnosti na XSD te omogućavaju postavljanje i provjeru nekih ograničenja nad vrijednostima podataka [28]. DTD (eng. **D**ocument **T**ype **D**efinition) je jedan od standarda XML-a, stariji od XSD-a i za razliku od XSD ne sadrži tip podataka. Pa tako je i sam XML dokument valjan ako odgovara pravilima definiranim od strane korisnika ili je strukturi sukladan XSD-u, DTD-u ili nekom drugom standardu.

Komunikacijski protokoli koji se koriste su **SOAP, HTTP i TCP/IP**. TCP/IP protokol služi za transport i usmjeravanje kroz mrežu, HTTP protokol se koristi kao ovojnica, budući da vatrozidi propuštaju poruke unutar ovog protokola [24]. Uz navedene osnovne komunikacijske protokole koristi se i **REST**.

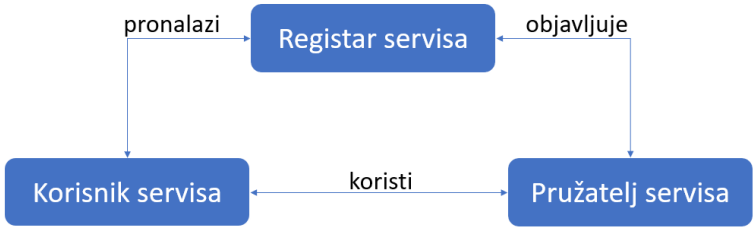
SOAP (eng. *Simple Object Access Protocol*) je komunikacijski protokol neovisan o platformi, a baziran je na XML-u koji se koristi za razmjenu informacija između aplikacije preko HTTP protokola [41]. SOAP definira formate poruka za komunikaciju s web servisima i propisuje povezivanje na postojeće komunikacijske protokole. S obzirom na to da se informacije prosljeđuju putem XML-a, SOAP je neovisan o programskom jeziku, platformi i transportu.

REST (eng. ***RE**presentational **S**tate **T**ransfer*) je alternativa za SOAP pa je isto tako neovisan o platformi. Za razliku od SOAP protokola REST nije protokol već stil arhitekture i ne ovisi o XML-u kako bi se izradio zahtjev već se oslanja na URI (eng. ***U**niform **R**esource **I**dentifier*) [51]. Iako je u nekim scenarijima potrebno prenijeti više informacija, većina web servisa koji koriste REST se isključivo koriste URL-om. Također se ne mora koristiti XML kako bi se kreirao odgovor, dapače može se koristiti primjerice CSV, JSON ili RSS. Zaključno, REST može koristiti bilo koji pisani oblik koji je moguće podijeliti u manje cjeline unutar aplikacije, a omogućuje manje formate poruka.

3.3. Servisno orijentirana arhitektura

Danas se servisno orijentirana arhitektura (eng. ***S**ervis **O**riented **A**rhitecture - SOA*) prepoznaje kao prva asocijacija na web servise. Servisno orijentirana arhitektura teži k tome da se aplikacija može poslužiti već dostupnim servisima na internetu. Naime servisno orijentirana arhitektura je arhitektura koja zadovoljava interoperabilnost, labavu povezanost, stalnu raspoloživost, ponovnu iskoristivost, heterogenost i sigurno upravljanje njenim komponentama. To su ujedno i razlozi njenog korištenja.

Servisno orijentiranu arhitekturu je moguće implementirati bez pomoći već testiranih i postojećih komponenata, no to zahtjeva puno rada, zato postoje tehnologije koje se mogu koristiti kako bi se kreirala takva arhitektura [51]. Za kreiranje servisno orijentirane arhitekture najčešće se koriste web servisi. Kako bi shvatili zašto se koriste web servisi najprije je objašnjen model servisno orijentirane arhitekture. Model servisno orijentirane arhitekture se sastoji od tri glavne uloge; **registra usluga, tražitelja usluga i pružatelja usluga** što se može vidjeti i na sljedećoj slici .



Slika 2: SOA model (Izvor: vlastita izrada)

SOA model se sastoji pored navedenih uloga i od tri glavne radnje; **objavljivanje**, **pronalaženje** i **korištenje/spajanje** (eng. *binding*). Korisnik servisa inicira zahtjev servisa pri kontaktu s registrom servisa. Korištenjem direktorija registra servisa, korisnik otkriva servis koji sadrži određeni ugovor servisa (eng. *interface*). Registar servisa ima informacije o tome koji pružatelj servisa podržava koji ugovor jer su isti registrirani unutar registra servisa kroz objavljivanje svojih podržanih ugovora registru. Registar servisa vraća korisniku informaciju o pružatelju servisa nakon čega korisnik koristi tu informaciju kako bi se spojio na određeni pružatelj servisa te šalje istom zahtjev za izvršavanjem svoje usluge. Registar servisa je opcionalan u ovome modelu, no predlaže ga se koristiti kod velikog broja servisa dok su pružatelj i korisnik servisa neophodni.

Može se zaključiti da pod korisnika servisa spada bilo koja softverska aplikacija ili servis koji zahtjeva servis, pružatelji servisa su softverska aplikacija ili servis koja objavljuje servise u registru i čini ih dostupnim korisnicima, prihvaća zahtjeve korisnika, izvršava ih i vrača odgovor. Registar usluga je posrednik između korisnika i pružatelja usluga koji pruža informacije o pružatelju usluga korisnicima.

Prema ovome modelu i samoj definiciji web servisa može se zaključiti zašto su sami web servisi učestale komponente ovakve arhitekture. Danas postoje tehnologije i već izrađene komponente koje se uklapaju u model servisno orijentirane arhitekture, a povezane su s web servisima. Primjer toga su UDDI i WSDL. UDDI preuzima ulogu registra servisa, dok je WSDL standard prema kojemu se opisuje rad servisa kako bi se isti mogli objaviti.

Servisno orijentirana arhitektura, kao što je već navedeno, pruža puno koristi, a koje su [51, 56]:

- **interoperabilnost,**
- **raspoloživost i fleksibilnost,**
- **ponovna iskoristivost,**
- **labava povezanost,**

- **sigurno upravljanje komponentama i**
- **samozacjeljivanje**.

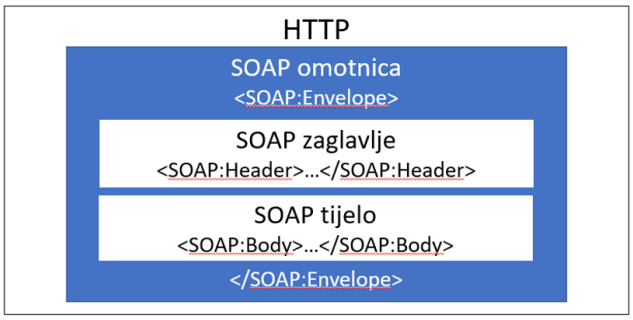
Najvažnija od navedenih je interoperabilnost. Interoperabilnost omogućuje da se servis jedne organizacije može koristiti servise druge organizacije, odnosno omogućuje komuniciranje više različitih sustava ili komponenata. **Interoperabilnost** je stoga sposobnost poslovnih procesa i informacijskih sustava koji ih podržavaju da razmjenjuju podatke, informacije i znanje. Osim interoperabilnosti SOA princip zagovara da servis treba biti **raspoloživ** svaki dan u bilo koje vrijeme osobi koja ga treba te da je **fleksibilan**, odnosno da je dinamičan, kompozicijski, mrežno adresabilan, lokalno transparentan i da podržava raznoliko vlasništvo [28]. SOA se sastoji od više servisa povezanih tako da budu otporni u slučaju kašnjenja i kvarova na mreži kroz asinkrono komuniciranje komponenti što rezultira **labavoj povezanosti**. Naravno danas je jako bitna zaštita tajnosti, raspoloživosti i integriteta pri čemu se zaštita web servisa svodi na primjenu SLA (eng. ***S**ervice-**L**evel **A**greement*) i WSS-a (eng. ***W**eb **S**ervices **S**ecurity - WS-Security*) [28]. Ako pak dođe do zakazivanja servisa od kojih je aplikacija sastavljena ista ima mogućnost pronaći i povezati radeće komponente ili servise što se naziva **samozacjeljivanje**. Zaštita i samozacjeljivanje su objedinjeni i omogućavaju **sigurno upravljanje komponentama.**

3.4. Vrste web servisa

Web servisi se mogu implementirati na razne načine, no najpoznatije vrste web servisa su SOAP web servisi i RESTful web servisi. Kao što je već spomenuto SOAP, XML, UDDI i WSDL su osnovni standardi za web servise. Iz tih osnovnih standarda proizlazi SOAP web servis.

3.4.1. SOAP web servisi

SOAP web servisi, kao što i ime govori koriste SOAP komunikacijski protokol koji jednostavno dodaje zaglavlje svakoj XML poruci prije nego je prenesena putem HTTP protokola. Temeljni dio SOAP protokola je **SOAP poruka** koja se sastoji od **SOAP omotnice, zaglavlja i tijela** [41]**.** SOAP protokolom se omogućava interoperabilnost i rad web servisa, uz WSDL i UDDI. Osim navedenih dijelova SOAP poruke, u poruci se može i opcionalno nalaziti pogreška unutar samog tijela poruke [41]. Pogreška kao dio SOAP poruke omogućava rukovanje greškama što može uvelike pomoći ako ne postoji vlasništvo nad web servisom na koji se spaja aplikacija [28]. Na sljedećoj slici mogu se vidjeti dijelovi SOAP poruke.



Slika 3: Struktura SOAP poruke (Izvor: vlastita izrada, prema: [28])

Unutar zaglavlja poruke nalaze se informacije o transakciji kao što su primjerice sigurnosne oznake, oznake za ostvarivanje pouzdanosti, semantičke oznake o sadržaju poruke i dr. [41]. U tijelu SOAP poruke se nalazi korisnički definirana informacija odnosno XML podaci [41]. SOAP poruka povezuje korisnika s web servisom. Postoji i SOAP RPC čije poruke sadrže XML koji predstavlja poziv metode ili odgovor na metodu [28, 148]. SOAP poruka ne mora se slati samo preko HTTP protokola koji se najčešće koristi zbog sigurnosti. Naime može se slati preko drugih protokola kao primjerice SMTP (engl. ***S**imple **M**ail **T**ransfer **P**rotocol*), TCP/IP protokola ili FTP.

3.4.2. RESTful web servisi

RESTful web servisi su web servisi koji su bazirani na REST arhitekturi, a koriste HTTP protokol za dohvaćanje i slanje podataka. Ključni elementi RESTful servisa su resursi, HTTP metode, zaglavlja zahtjeva, tijelo zahtjeva, tijelo odgovora i status odgovora [34]. U REST arhitekturi, HTTP protokol se ne ponaša samo kao transportni protokol već kao i sučelje koje nudi svoje metode kako bi se izvršio RESTful zahtjev pa zato RESTful web servisi nisu ovisni o XML porukama i WSDL opisima [34]. REST arhitektura se organizira oko resursa koji su najčešće dostupni preko URI pri čemu se na osnovnu adresu dodaju korisni podaci kasnije korišteni za dobivanje određenog rezultata [34].

Svaka komponenta u REST arhitekturi je tretirana kao resurs kojeg se dohvaća korištenjem standardnih HTTP metoda. Najkorištenije HTTP metode su ***GET, POST, DELETE i PUT*** [34]. *GET* metoda omogućava čitanje pojedinog resursa, *POST* metoda se koristi za kreiranje novog resursa, *DELETE* za brisanje resursa, a metoda *PUT* za ažuriranje već kreiranog resursa ili kreiranje novog resursa [34]. RESTful web servisi najčešće koriste JSON za reprezentaciju resursa. S obzirom na to da su RESTful web servisi izgrađeni na temelju HTTP-a, odgovori mogu bili u JSON, XML ili bilo kojem drugom formatu koji podržava korisničke potrebe.

RESTful web servisi su **bez stanja** odnosno na klijentu je da osigura da se svi potrebni podaci dostavljaju poslužitelju kako bi poslužitelj pravilno odgovorio na zahtjev [34]. Poslužitelj ne bi trebao spremati bilo kakve informacije između zahtjeva klijenta i prema tome svaki odgovor poslužitelja je neovisan o prošlom. Ako to predstavlja problem moguće je **uvesti koncept predmemorije** (eng. *cache*) [34]. S obzirom na to da su odgovori međusobno neovisni, ponekad klijent želi ponovo poslati isti zahtjev. To povećava promet na mreži što se želi izbjeći predmemorijom. Predmemorija bi bila stoga implementirana na klijentu kako bi se spremili zahtjevi koji su već poslani na poslužitelj. Tako će umjesto da se ponovno isti zahtjev pošalje poslužitelju, poslati u predmemoriju i time će se dobiti potrebne informacije. Tako se mogu podići performanse uštedom količine mrežnog prometa.

Za razliku od SOAP web servisa, ne postoje standardi za RESTful web servise. To je zato što je SOAP protokol dok je REST stil arhitekture. REST nije standard sam po sebi, ali se često koriste standardi poput HTTP, URI, JSON i XML. RESTful web servisi su lakši za učenje i efikasniji jer koriste manje poruke od SOAP web servisa. Posebno se treba osvrnuti na to da su RESTful web servisi sinkroni. [29]

4. Sustavi poruka

Prije definicije sustava poruka, bit će objašnjena razmjena poruka, asinkrona i sinkrona komunikacija. Najbolji primjer za objašnjavanje istog su pozivi telefonom. U početcima su bili omogućeni samo telefonski pozivi, pri kojem pozivatelj može komunicirati s drugom osobom samo ako je ta osoba dostupna i kraj telefona u isto vrijeme. To je sinkrona komunikacija. Sinkrona komunikacija je stoga komunikacija koja se odvija u realnom vremenu. Nakon nekog vremena izumljena je glasovna poruka. Glasovna poruka je omogućila da pozivatelj ako ne dobije direktni odgovor od primatelja poziva, može ostaviti poruku kako bi ju kasnije primatelj poslušao. Poruke su dostupne u kronološkom slijedu kako su ostavljene i ako ju primatelj odsluša ona se obriše. Ovo je primjer asinkrone komunikacije koja označava razmjenu poruka pri čemu se podaci mogu prenositi s kašnjenjem i prekidima. Navedeni primjer također uključuje red poruka odnosno glasovni poziv se pretvara u glasovnu poruku koja je zatim spremljena u red za kasnije korištenje; što je zapravo način kako funkcionira razmjena poruka.

Razmjena poruka se u kontekstu informacijskih tehnologija odnosi na razmjenu poruka između dva sustava. Poruke koje šalju sustavi su zapravo paketi podataka, a šalju ih kanalima odnosno redovima poruka koji povezuju programe. **Kanal** se ponaša kao kolekcija poruka, ali kolekcija koja nije prisvojena samo od jednog programa, već je dijeljena kroz više povezanih sustava i može se koristiti od strane više aplikacija [31]. Zbog izmjene radnja primanja i slanja poruka, poznati su pošiljatelj poruke odnosno aplikacija koja šalje poruku na kanal i primatelj poruke odnosno aplikacija koja prima poruku čitajući je iz kanala.

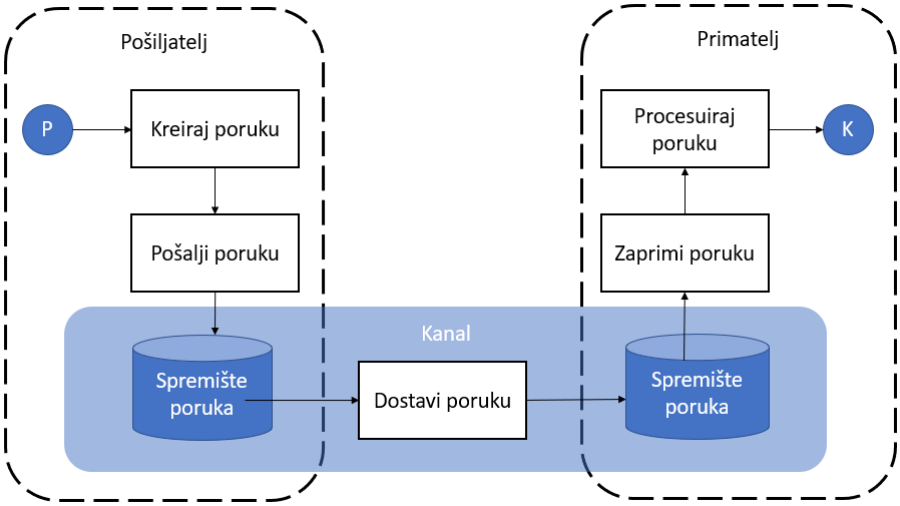
Poruka koja se šalje je neki oblik strukturiranog podatka, bilo to običan niz znakova, bajtova ili objekt [31]. Poruka se sastoji u pravilu od zaglavlja i tijela. Zaglavlje poruke sadrži meta informacije o samoj poruci o pošiljatelju, primatelju i slično. Sustav poruka čita meta podatke, dok su za primatelja i pošiljatelja skoro nebitni [31]. Tijelo poruke sadržava podatke koje treba prenijeti do druge aplikacije te za razliku od zaglavlja, sustav poruka ignorira tijelo poruke [31]. U razgovoru se često na spomen poruke misli na samo tijelo poruke.

S obzirom na navedeno zaključuje se da je **razmjena poruka** tehnologija koja omogućava **brzu, asinkronu i povjerljivu** komunikaciju između dva programa koji se nazivaju primateljem i pošiljateljem, a koristi **poruke** u obliku paketa podataka **s tijelom i zaglavljem** kako bi se osiguralo prenošenje podataka o primatelju i podataka koji se žele proslijediti drugoj aplikaciji.

Ranije je spomenuto kako je sustav poruka uključen u razmjenu poruka. **Sustav poruka** ili međuprogram orijentiran porukama (eng. ***M****essage-****O****riented* ***M****iddleware -MOM*) upravlja razmjenom poruka tako što pruža red poruka na koji se poruka šalje i omogućuje

sigurno slanje poruke drugoj aplikaciji [31]. Svrha sustava poruka je upravljanje i premještanje poruka. Kako bi se mogao koristiti sustav poruka, potrebno ga je konfigurirati s kanalima koji određuju puteve komunikacije između aplikacija [31]. Sustav poruka nadilazi ograničenja poput nespremnosti aplikacija na primanje poruka ili nepravilnog rada mreže tako da konstantno pokušava slati poruke dok slanje poruke ne bude uspješno [31].

Proces razmjene poruka sustavom poruka uključuje pet koraka. Pošiljatelj prije nego pošalje poruku mora ju **kreirati** te uključiti podatke u nju. Nakon što ju je uspješno kreirao sprema se na **slanje** poruke u kanal. Sustav poruka po primitku poruke **dostavlja** poruku iz sustava pošiljatelja na sustav primatelja. Ako je spreman, primatelj poruka **zaprima** i čita poruku iz kanala. Po primitku čitave poruke, primatelj ju **procesuira** tako što izvlači podatke iz nje. Sljedeći dijagram prikazuje proces razmjene poruka koji je opisan. [31]



Slika 4: Dijagram procesa razmjene poruka (Izvor: vlastita izrada, prema: [31])

Postoje dva koncepta slanja poruka. Jedan koncept je „**pošalji i zaboravi“** pri čemu aplikacija po slanju poruke na kanal nastavlja obavljati druge poslove dok kanal šalje poruku u pozadini. Poruka koja je poslana se ne sprema. Drugi koncept je „**spremi i pošalji“** pri čemu se poruka koja je poslana sprema na računalo pošiljatelja, a sustav prosljeđuje poruku s jednog računala na drugo te sprema poruku još jednom na računalo primatelja. [31]

4.1. Karakteristike sustava poruka

U ovom odjeljku kratko će biti objašnjene karakteristike sustava poruka. Tijekom funkcijske integracije spomenuta su rješenja koja su uključivala i sustave poruka, tako da će biti opisane njegove koristi, ali i moguće mjere opreza kod korištenja sustava poruka.

Karakteristike sustava poruka su [31]:

- udaljena komunikacija,
- integracija divergentnih aplikacija,
- asinkrona komunikacija,
- labava povezanost,
- minimiziranje prigušivanja prijemnika,
- pouzdana isporuka i
- posredništvo.

Dva objekta unutar iste aplikacije mogu jednostavno koristiti iste podatke iz memorije, no slanje poruka drugoj udaljenoj aplikaciji je puno kompliciranije i zahtjeva kopiranje podataka s jednog sustava na drugi [31]. Razmjena poruka stoga omogućuje različitim udaljenim aplikacijama da komuniciraju i razmjenjuju podatke. Ukratko razmjena poruka omogućuje **udaljenu komunikaciju.** Zbog prirode posrednika, potrebno je podatke pretvoriti u niz bajtova što se naziva serijalizacijom te pri čitanju poruka pretvoriti niz bajtova u podatke što se naziva deserijalizacijom.

Već se spominjalo ranije kod web servisa da aplikacije koje se žele spojiti mogu biti različitih programskih jezika, tehnologija, platforma ili formata podataka koje koriste. Za rješavanje ovog problema najlakše je dodati posrednika između te dvije aplikacije koji bi upravljao komunikacijom. Sustav poruka se ponaša kao takav posrednik i može biti **jedinstven prevoditelj između divergentnih aplikacija** tako što im omogućava da komuniciraju putem čestih paradigmi poruka [31].

Asinkrona komunikacija omogućuje pošiljatelju poziva da nastavi s izvođenjem, nasuprot čekanju da primatelj zaprimi i obradi podatke. Tako se sprječava prestanak rada aplikacije ako dođe do pucanja veze i ako druga aplikacije nije spremna obraditi zahtjev ili je ugašena. U sinkronom načinu rada, čeka se da primatelj zahtjeva primi zahtjev i obradi ga što može biti problem ako se šalje više zahtjeva uzastopno. Slanje uzastopnih poruka u sinkronoj komunikaciji zahtijeva od primatelja da između slanja zahtjeva čeka da se prijašnji obradi što usporava rad sustava. Asinkrona komunikacija dozvoljava primatelju slanje poruka u kojoj god brzini on to zahtijevao, kao i što dozvoljava pošiljatelju da obrađuje zahtjeve onoliko brzo koliko mu je potrebno [31]. Sustavi poruka stoga omogućavaju aplikacijama da **se brzo izvršavaju i da bespotrebno ne troše vrijeme na čekanje** [31].

Zbog postavljanja posrednika između više aplikacija i asinkrone komunikacija, sustavi poruka omogućuju **labavu povezanost sustava** [35]. Ako dođe do greške na jednoj aplikaciji, to se neće odraziti na ostale povezane aplikacije. Pošiljatelj uz to ne čeka odgovor primatelja te nastavlja izvršavati svoje operacije. S obzirom na to da su pošiljatelj i primatelj zasebne

aplikacije povezane putem asinkronih poruka, promjene izvršene na jednoj od njih se neće odraziti na drugu.

Kod poziva udaljenih procedura može se preopteretiti prijemnik što prouzrokuje pogoršanje performansi ili prestanak rada primatelja, no asinkronom komunikacijom primatelj može sam kontrolirati brzinu kojom obrađuje zahtjeve pa je **neželjeni učinak uzrokovan prigušivanjem minimiziran** [31].

Uz to razmjena poruka pruža **pouzdaniju isporuku**. Naime razmjena poruka uključuje koncept „spremi i pošalji“ pri čemu se spremljena poruka automatski ponovno šalje primatelju sve dok se ne uspije poslati [31]. Sustavi poruka ovise o stanju mreže i stanju primatelja. Ako se dogodi do kvara na mreži ili primatelju, neće biti moguće zaprimanje poruke. Neke aplikacije su posebno dizajnirane za **rad bez mreže**, no ipak kada je mreža dostupna sinkroniziraju se s poslužiteljima [31]. Asinkrona komunikacija razmjene poruka i koncept „spremi i pošalji“ omogućavaju aplikacijama da šalju poruke u red čekanja, čekajući dok se aplikacija ponovno poveže s mrežom.

Spomenuto je da sustavi poruka djeluju kao **posrednici između programa** koji šalju i primaju poruke. Sustav poruka se može zato koristiti od strane aplikacije kao registar drugih aplikacija ili servisa dostupnih za integraciju [31]. Ako aplikacija prekine s radom ili na neki drugi način prekine vezu sa sustavom poruka može se kasnije ponovo povezati i biti dostupna drugima preko sustava za razmjenu poruka. Sustav za razmjenu poruka može pružiti veliki broj distribuiranih veza na zajednički resurs poput baze podataka uz što **može iskoristiti redundantne resurse** zbog visoke dostupnosti, uravnoteženja opterećenja, preusmjeravanja zbog neuspjelih mrežnih veza i podešavanje performansi i kvalitete usluge [31].

Sustavi poruka imaju i neke negativne strane. Primjerice kanali poruka garantiraju dostavu poruka, ali ne garantiraju da će poruke koje su poslane u nizu i ostati u nizu [31]. S obzirom na to da se danas poruke zbog brzine prijenosa šalju u manjim paketima, to može biti katastrofalno na korištenje istih. Stoga treba paziti na način kako kasnije iz razlomljenog paketa kreirati ponovo niz slijednih poruka. Isto tako, ne može svaka aplikacija raditi asinkrono pa je potrebno uskladiti asinkronost i sinkronost aplikacije [31].

4.2. Koncepti sustava poruka

Kako bi bolje razumjeli sustave poruka, u ovom poglavlju bit će objašnjeni osnovni koncepti ove tehnologije. Osnovni koncepti sustava poruka su: **kanali, poruke, isporuka u više koraka, usmjeravanje, transformacija i krajnje točke.**

4.2.1. Kanali

Treba imati u vidu da to što postoji dostupan sustav poruka za povezivanje aplikacija ne znači da će te aplikacije biti automatski povezane, odnosno da će se kanali sami kreirati. Aplikacije ne mogu samo slijepo slati informacije u sustav poruka kao i što ne mogu slijepo uzimati podatke na koje naiđu. Ona aplikacija koja šalje poruku zna kakve podatke šalje, nasuprot tome aplikacija koja želi zaprimiti poruku zna kakve podatke treba primiti [31]. Aplikacije koje koriste razmjenu poruka prema tome prenose podatke putem **kanala** koji je zapravo virtualna cijev koja povezuje pošiljatelja i primatelja poruke.

Prema [31]:

„*Sustav poruka nije predviđen za nasumično slanje i uzimanje bilo kakvih informacija. Sustav poruka je skup poveznica koje omogućuju aplikacijama komuniciranje prijenosom podataka* ***na unaprijed određene, predvidljive načine*****. „**

Kada aplikacija ima podatke koje želi poslati, ne pošalje ih samo u sustav već dodaje informaciju na njih o određenom kanalu poruke. S druge strane, kada aplikacija želi preuzeti podatke ne izabere neki slučajan skup podataka iz sustava poruka već ih dohvati preko informacije o određenom kanalu poruke. [31]

S obzirom na to da sustav poruka **ima različite kanale** za različite tipove informacija koje aplikacije žele iskomunicirati, aplikacija pošiljatelja podataka ne mora znati o aplikaciji primatelja. Ništa se ne prepušta slučajnosti. Informacije o kanalu su usmjerene prema direktnom tipu kanala koji će prenijeti podatke i prema primatelju koji će htjeti obraditi podatke s direktnog tipa kanala. Može se prema tome zaključiti da su kanali logičke adrese unutar sustava poruka, a njihova implementacija ovisi o samom sustavu poruka koji se koristi [31].

Postoje dvije različite vrste kanala poruka: **od-točke-do-točke** i **objavljivanje/pretplata**.

Od-točke-do-točke kanal osigurava da samo jedan primatelj koristi bilo koju poruku [32]. Ako kanal ima više primatelja, samo jedan od njih može uspješno preuzeti određenu poruku. Bez obzira na jedinstvenu obradu poruka, više aplikacija može biti spojeno na isti kanal. Zato se ovaj kanal naziva od-točke-do-točke. S druge strane, kanal objavljivanje/pretplata je kanal koji omogućuje obrađivanje jedne poruke više primatelja.

Kanal **objavljivanje/pretplata** ima jedan ulazni kanal koji se dijeli na višestruki izlazni kanal, po jedan za svakog primatelja [32]. Kada je poruka objavljena na kanalu, kanal isporučuje kopiju poruke na svaki od izlaznih kanala. Svaki izlazni kanal ima samo jednog primatelja ili pretplatnika, kojem je dopušteno samo jednom preuzeti poruku [31].

4.2.2.Poruke

Poruka se može gledati kao skup podataka koji se prenosi preko komunikacijskog kanala. Kada se šalje pošta mora se navesti kome se šalje i tko ju šalje u slučaju neuspješne pošiljke. Poruke sustava razmjena poruka također moraju sadržavati podatke o pošiljatelju i primatelju poruka. Zato se poruke sastoje od zaglavlja i tijela. Unutar **zaglavlja** se nalaze sve dodatne informacije o poruci koje će poslužiti sustavu poruka kako bi odredio koja mu je destinacija, izvor i slično [31]. U **tijelu** poruke, kao i kod pošte, se nalaze podaci koji se žele prenijeti. Treba imati na umu da se preveliki podaci trebaju podijeliti u manje podatke i poslati s jednom ili više poruka, a takav niz poruka od podijeljenih podataka naziva se sekvenca poruke [31].

Postoje tri vrste poruka s pogleda programiranja: naredbene poruke, poruke dokumenta i poruke događaja. Ako se želi pozvati procedura u drugoj aplikaciji šalje se **naredbena poruka**, ako se pak želi prenijeti set podataka šalje se **poruka dokumenta** te ako se želi obavijestiti aplikaciju o promjenama u drugoj aplikaciji šalje se **poruka događaja**. [31]

Primjerice SOAP poruke imaju mogućnost pozivanja metoda, čineći ju naredbenom porukom ako sadržava poziv. S druge strane, XML može poslužiti za opis i siguran prijenos podataka pa je tako XML poruka poruka dokumenta.

4.2.3. Isporuka u više koraka

Poruke koje šaljemo najčešće je potrebno transformirati. U najboljem slučaju poruku koju šaljemo nije potrebno transformirati, obogaćivati ili prevesti. Ipak najčešće je slučaj da nad porukom treba obaviti potrebne operacije nakon što je poslana, a prije nego što je zaprimljena [31]. Takav način slanja poruka naziva se isporukom u više koraka. Primjerice možemo dohvaćati podatke o autobusu koji uključuju i lokacije autobusa. Između ishodišne i krajnje točke poruke može postojati komponenta koja izračunava brzinu autobusa na temelju prijašnje spremljenog podatka i time obogaćuje sadržaj poruke.

4.2.4.Usmjeravanje

Pri integraciji velikog broja aplikacija, poruka može prolaziti kroz nekoliko kanala. Svaki pošiljatelj unutar velikog sustava zna samo informaciju o odredištu na koje se šalje poruka. Put poruke može biti skup kanala pri čemu svaka aplikacija postavlja poruku na određeni kanal. Tako svaka aplikacija može obraditi i preusmjeriti poruku na drugi kanal. Informacije o samom putu poruke nisu dostupne pošiljatelju. Pošiljatelj poruka stoga šalje poruku na **usmjerivač poruka,** aplikacijsku komponentu i filter arhitekture kanala i filtera koji će se navigirati kanalima i usmjeriti aplikaciju do konačnog primatelja [31].

4.2.5. Transformacija

Pri integraciji dviju aplikacija može se dogoditi nesklad objekta koji se šalje i objekta koji se zaprima. Primjerice moguće je da se šalju podaci o datumu šalju u engleskom formatu, a aplikacija koja prima podatke o datumima zahtjeva datum u hrvatskom formatu. Često se ne postavlja određeni ugovor između dvije aplikacije o formatu podataka. Osim toga navedene aplikacije u najgorem slučaju nemaju mogućnosti promjene. Takav problem se rješava transformacijom poruka između pošiljatelja i primatelja kako bi se osigurao isti format datoteka. Takva tehnologija unutar sustava poruka naziva se translatorom poruka [31]. Iako sustavi poruka dopuštaju različite formate podataka, format poruka je najčešće definiran sustavom poruka i treba biti u skladu s njime [31].

Već je ranije spominjan slučaj potrebe za **obogačivaćem sadržaja.** Osim različitih formata, može se dogoditi da primatelj poruke zahtijeva podatkovna polja koja poruka pošiljatelja ne sadrži, pa je potrebno nadodati sadržaj. S druge strane, može se dogoditi da primatelj traži manje podataka od poslanih tako da je potreban **filtar sadržaja** koji će ukloniti neželjene podatke iz poruke [31]. **Normalizator** pak prevodi više poruka koje stižu u mnogo različitih formata u zajednički format [31].

4.2.6.Krajnje točke

Čak i da sve se sve navedeno koristi tijekom integriranja aplikacija, integriranje neće rezultirati uspješno. Naime same aplikacije trebaju način spajanja na sustav poruka kroz sučelje. Potrebno je nadodati na aplikaciju sloj koji će znati kako aplikacija radi i kako sustav poruka radi. Takav sloj bi se tada ponašao kao most između aplikacije i sustava poruka [54]. Prema tome, sloj treba sadržavati set koordiniranih krajnjih točaka koje omogućavaju aplikaciji slanje i prihvaćanje poruka. Hohpe i Woolf [31] govore nešto više o uzorcima slanja, primanja i obrađivanja poruka.

5. Reaktivni sustavi

U ožujku ove godine, tvrtka ABOUT YOU, koja se bavi prodajom odjeće i obuće, odlučila je dati velike popuste kroz pet dana na online kupovinu preko web aplikacije. Iz osobnog iskustva autora, već u prvom danu aplikacija nije pravodobno i ispravno radila, što je vjerojatno izazvalo velike novčane gubitke i negodovanje korisnika. Prema iskustvu, puno korisnika je pokušalo obaviti kupnju pri čemu ili nisu uspjeli jer je došlo do pogreške aplikacije, ili su odustali nakon predugog čekanja na učitavanje pojedinih ekrana aplikacije. Zbog nedostupnosti web stranice počelo se postavljati pitanje je li tvrtka namjerno sabotirala rasprodaju jer korisnici očekuju od web aplikacije da radi u svakom trenutku, bez obzira na to koliko korisnika pristupa aplikaciji u isto vrijeme. Može se zaključiti da se takav incident odražava na sadašnje, ali i buduće stanje poduzeća. Studija o utjecaju performansi web aplikacije na ponašanje kupaca pokazala je da spore aplikacije smanjuju korisničku želju za kupnjom, kao i količinu koju je kupac spreman platiti za proizvod [37]. Stoga je jako bitno da je web aplikacija responzivna bez obzira na opterećenje. Responzivnost stranice se treba ostvariti i u slučaju greške i u slučaju opterećenja. Reaktivni sustavi su upravo sustavi vođeni responzivnošću i reakcijama na okruženje.

Do 2013. je pojam reaktivnog sustava varirao od implementacije do implementacije. Bilo je potrebno odrediti standard reaktivnih sustava pa je tako prva formalizacija termina reaktivnih sustava nastala 2013. kada je Jonas Boner kreirao „Reaktivni manifest“ okupljajući neke od najboljih mozgova u industriji distribuiranih sustava kako bi se razjasnila zbunjenost oko reaktivnosti i kreirao solidni temelj za održiv način razvoja [18].

Prema manifestu [18] **reaktivni sustav** je pristup arhitekturi sustava sastavljenoj od više individualnih servisa koji rade zajedno kao jedna cjelina, reagiraju na svoje okruženje i jedan na drugog što rezultira otpornošću na zatajenje komponenata i većoj elastičnosti pri suočavanju s uvijek promjenjivim zahtjevima radnih opterećenja.

Temelj reaktivnih sustava je prosljeđivanje poruke (eng. *message-passing*) koje stvara privremenu vezu između komponenata, dopuštajući im vremensku i prostornu razdvojenost, što omogućava distribuciju i mobilnost [33].

Kada se spominje pojam „reaktivni“ u kontekstu razvoja i dizajna softvera, najčešće se misli na reaktivno programiranje ili na reaktivni sustav. **Reaktivno programiranje** je podskup asinkronog programiranja i paradigma gdje dostupnost novih informacija pokreće logiku nasuprot izvršavanju dretvi koje pokreću kontrolni tok [33]. Reaktivno programiranje podržava razlaganje problema na više diskretnih koraka pri čemu se svaki može izvršiti asinkrono i neovisno te spajanje koraka kako bi se pružio određeni tijek rada, takav da je neograničen

svojim ulazima ili izlazima ako je to moguće [33]. Bitno je ne miješati pojmove reaktivnog programiranja i reaktivnog sustava, odnosno reaktivno programiranje za rezultat nema reaktivni sustav.

Reaktivno programiranje je **uvjetovano događajem** (eng. *event-driven*), za razliku od reaktivnih sustava koji su **uvjetovani porukama** (eng. *message-driven*) [33,23]. Ova razlika je jako bitna, jer poruke su same po sebi usmjerene, imaju jedno jasno odredište dok su događaji neusmjerene činjenice koje drugi trebaju promatrati [33]. Nadalje, poruka je stavka podataka koja se šalje odnosno prenosi mrežom i baza je komunikacije distribuiranih sustava, a događaj je signal koji komponenta emitira lokalno nakon postizanja određenog stanja [33].

5.1. Karakteristike reaktivnih sustava

Kao što je spomenuto, unutar reaktivnog manifesta su navedeni standardi kako bi se jednoliko razumio pojam reaktivnih sustava. Reaktivni manifest sadrži niz osnovnih principa potrebnih za kreiranje sustava koji ima mogućnost brzog odgovora na redove, bez obzira na preopterećenje ili greške.

Reaktivni sustavi moraju biti [18]:

- responzivni,
- otporni,
- elastični i
- uvjetovani porukama.

Unutar Oxford-ova rječnika responzivnost se definira kao brzo i pozitivno reagiranje na nekoga ili nešto [45]. Reaktivno programiranje je orijentirano upravo tome, pa i sami naziv reaktivno određuje da sustav reagira na nešto. **Responzivnost** je stoga glavna značajka reaktivnih sustava, a definira se kao pravovremeno reagiranje sustava, ako je to moguće [18]. Responzivnost omogućava reaktivnim sustavima brzo vrijeme odgovora čime možemo pružiti dosljednu kvalitetu usluge. Ako sustav pravovremeno reagira, moguće je brže pronaći i rukovati greškama. S druge strane, responzivnost poboljšava korisničko iskustvo korištenja aplikacije.

Danas ne postoji razlika između točnog odgovora koji nije dostupan kada je potrebno i odgovora pri kvaru. Zbog toga se mora osigurati responzivnost sustava i u slučaju kvara i u slučaju dinamičko promjenjivog opterećenja. Responzivnost sustava u slučaju kvara je zapravo **otpornost** sustava, dok je responzivnost sustava u slučaju dinamičko promjenjivog opterećenja zapravo **elastičnost** sustava.

Otpornost sustava je sposobnost rješavanja kvarova bez prestanka rada, pri čemu se to ne odnosi samo na visoko dostupne, kritične sustave, već na bilo koji sustav [18]. Otpornost se postiže kombiniranjem **replikacije, delegacije i izolacije** [33]. Ako dođe do kvara na jednoj komponenti to se ne smije odraziti ni na koju drugu, pa treba osigurati izolaciju komponente s kvarom kako bi sustav uspješno nastavio raditi. Interakcije s porukama omogućuju komponentama da se lokalno suoče s kvarom, a zahvaljujući asinkronosti komponente aktivno ne čekaju odgovor pa se kvar na jednoj komponenti neće odraziti na druge komponente. Oporavak neispravnih komponenti delegiran je na vanjsku komponentu tako da klijent komponente nije opterećen s rješavanjem kvara [18]. Replikacijom komponenta može se pružiti visoka dostupnost, tako da kada jedna komponenta ne uspije obraditi poruku, poruku može obraditi druga na istoj adresi [18].

Reaktivni sustavi moraju ostati responzivni pod različitim uvjetima opterećenja i pritiska. **Elastičnost** omogućuje reaktivnim sustavima da elegantno reagiraju na promjene stope unosa, dinamičkim povećanjem ili smanjenjem dodijeljenih resursa . To podrazumijeva dizajn u kojem nema sukobljenih točaka i uskih grla, a koji dozvoljava sustavu uništavanje ili repliciranje komponenata i distribuciju podataka među njima. [23]

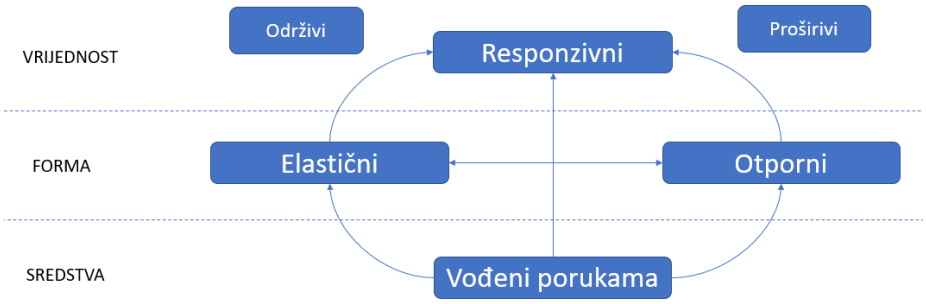
Kako bi se postigla elastičnost i otpornost, sustav mora biti **vođen porukama** [18]. Kao što je navedeno, prosljeđivanje poruke stvara privremenu vezu između komponenata, dopuštajući im vremensku i prostornu razdvojenost što omogućava distribuciju i mobilnost. Ovo razdvajanje uvjet je za potpunu izolaciju između komponenata i čini osnovu i za otpornost i elastičnost. Reaktivni sustavi oslanjaju se na asinkronu komunikaciju kako bi uspostavili granicu između komponenata što osigurava **labavo spajanje**, **izolaciju i transparentnost lokacije** [33]. Izolacija omogućena asinkronim prosljeđivanjem poruka i razdvajanje instanci izvršavanja od njihovih referenci je ono što se naziva transparentnost lokacije [18]. Asinkrono prenošenje poruka pruža sredstva za delegiranje grešaka kao poruka [18]. Eksplicitno asinkrono prosljeđivanja poruka omogućuje upravljanje opterećenjem, veću elastičnost i kontrolu opterećenja pomoću oblikovanja i nadgledanja redova poruka u sustavu te primjenom povratnog pritiska (eng. *back-pressure*) po potrebi [23]. Ovakav stil komunikacije osigurava da primatelji troše resurse samo dok su aktivni, što dovodi do manjeg opterećenja sustava te učinkovitijeg korištenja resursa.

Povratni pritisak se odnosi na slučaj kada se jedna komponenta bori da ostane u koraku s ostalima, a sustav u cjelini mora odgovoriti na svjestan način [18]. Treba poduzeti predostrožne mjere pri izgradi sustava i komponenata kako ne bi došlo do ispuštanja podataka što povlači sa sobom pitanje sigurnosti sustava. Zato bi se povratnim pritiskom trebalo omogućiti smanjenje opterećenja komponente.

Prema [18]:

„…povratni pritisak važan je mehanizam povratne sprege koji omogućuje sustavima da graciozno reagiraju na opterećenje, umjesto da se sruše pod njim.“

Reaktivni manifest također prikazuje odnos ovih principa, a isto je prikazano u sljedećoj slici.



Slika 5: Odnos principa reaktivnih sustava (Izvor: vlastita izrada, prema: [18])

Ukratko, kako bi sustav bio potpuno responzivan, mora biti responzivan i u slučaju kvara i u slučaju velikih opterećenja što su respektivno princip elastičnosti i otpornosti. Kako bi se ostvarila elastičnost i otpornost, sustav mora biti vođen porukama. Ako se svi navedeni principi ostvare, može se reći da je kreirani sustav upravo reaktivni sustav.

Ovi principi rezultiraju nizom prednosti reaktivnih sustava kao što su: labava povezanost, lagani za razvijanje i pogodni za promjene, smanjuju vrijeme čekanja, upravljaju greškama i ovisnostima, upravljaju opterećenjima, omogućavanju slanje, primanje i prosljeđivanje poruka u nepouzdanoj mreži i podržavaju transparentnost lokacije.

5.2. Uzorci reaktivnih sustava

Kako bi odgovorili na pitanje kako implementirati principe reaktivnih sustava, bit će opisani uzorci implementacije koji se mogu koristiti za transformiranje ili izradu aplikacije koja spada u reaktivni sustav.

Prije objašnjenja uzoraka, mora se osvrnuti na konkurentnost i paralelnost. Pri kreiranju reaktivnih sustava, želi se maksimalno iskoristiti hardver, što se postiže kroz konkurentnost i paralelnost [33]. Konkurentni rad se odnosi na započinjanje, izvršavanje i završavanje dva ili više zadataka u istom vremenskom okviru [33]. Paralelni rad je rad dva ili više zadatka u isto

vrijeme [33]. Sljedeći uzorci rade na poboljšavanju paralelnosti i konkurentnosti unutar aplikacije.

5.2.1. Uzorak Reactor

U uzorku Reactor sudjeluju**: petlja događaja, demultipleksor događaja, red zahtjeva i rukovatelj zahtjeva** [23]. Rukovatelj zahtjeva može biti i funkcija povratnog poziva. Glavna ideja ovog uzorka je postojanje **rukovatelja zahtjeva** koji je povezan s ulaznim i izlaznim operacijama, a koji će se pozivati čim se proizvede događaj i procesuira od strane petlje događaja. **Petlja događaja** se ponaša kao petlja koja reagira na događaje tako da konstantno provjerava dolazak novih događaja i prosljeđuje izvršavanje događaja određenom rukovatelju. **Demultipleksor** događaja primljene zahtjeve demultipleksira i otprema ih u **red zahtjeva**. Demultipleksiranje i otpremanje zahtjeva demultipleksora se može opisati kao radnja telefonskog operatera, koji u isto vrijeme može primiti veću količinu poziva, a zatim se javlja na pozive i prosljeđuje ih jedan po jedan odgovarajućem kontaktu.

Aplikacija generira novu ulazno/izlaznu operaciju šaljući zahtjev demultipleksoru događaja pri čemu se specificira rukovatelj događaja koji će biti pozvan kad se operacija dovršava. Zahtjevi koji se šalju demultipleksoru događaja mogu se poslati istovremeno i neblokirajuće tako da aplikacija generira zahtjev, a demultipleksor vraća kontrolu aplikaciji, odnosno aplikacija može nastaviti s izvršavanjem [21]. Kada se cijeli set ulazno/izlaznih operacija popuni, demultipleksor događaja postavlja nove događaje u red događaja. U tom trenutku petlja događaja prolazi kroz red događaja. Za svaki događaj poziva se dodijeljeni rukovatelj događaja. Rukovatelj će vratiti kontrolu petlji kada se izvrši radnja. Kada su sve stavke unutar reda događaja procesuirane, petlja će se ponovo zaustaviti i javiti demultipleksoru događaja što će pokrenuti novi ciklus ako je došlo do novih zahtjeva.

5.2.2.Uzorak Multireactor

Uzorak Multireactor je kreiran kako bi se mogli bolje iskoristiti dostupni računalni resursi i višejezgreno-višedretveni procesi [33]. Kao što i samo ime govori, **Multireactor** je zapravo prerađeni uzorak Reactor koji se u suštini ne sastoji samo od jedne petlje događaja, već od više. Broj petlji događaja zapravo ovisi o količini jezgri koje su na računalu [33]. **Vert.x** je set alata otvorenog koda za kreiranje reaktivne aplikacije na JVM (eng. ***J**ava **V**irtual **M**achine*) [46]. Svaka petlja događaja radi na pojedinačnoj dretvi.

5.2.2.1. Vert.x

Kod uzorka Reactor je problem u tome što se jedna dretva izvršava na jednoj jezgri računala. Pri uzroku Reactor, ako je potrebno skalirati aplikaciju kroz višejezgreni sustav, treba

se kreirati i upravljati više procesa. Umjesto jedne petlje događaja, svaka Vert.x instanca održava više petlji događaja [46]. Vert.x proširuje uzorak Multireactor s programskim konstruktima nazvanim vertikale. Dokumentacija Vert.x opisuje ga kao „*jednostavnu, skalabilnu implementaciju i model konkurentnosti izvan kutije*“ [46]. Vertikale su dijelovi koda koji su implementirani i izvršavani unutar Vert.x instance [46]. Standardne vertikale su pridružene petlji događaja kada se pokreću i izvršavaju na dretvi petlje događaja. Vertikale se mogu pokretati i zaustavljati asinkrono te komunicirati asinkrono kroz sabirnicu događaja [46]. Sabirnica događaja formira distribuirane ravnopravne (eng. *peer-to-peer*) sustave poruka koji obuhvaćaju više čvorova poslužitelja i više preglednika [46]. Sabirnica događaja podržava razmjenu poruka objavljivanjem/pretplatom, od točke do točke i paradigmu zahtjev-odgovor [46].

5.2.3. Uzorak Actor Model

Uzorak Actor Model je predstavljen 1973. kao arhitekturalni temelj za umjetnu inteligenciju [30]. Unutar uzorka Actor Model, **akteri** (eng. *actors*) su temeljna jedinica strukture. Na aktera se gleda kao na jedan računalni proces ili funkciju s adresom, odnosno programski kod kojem će se poslati poruka. Akteri mogu sadržavati više adresa ili jednu adresu, a jedna adresa može biti pridružena više aktera.

Akter obavlja poslove spremanja poruka, primanja poruka drugih aktera, prosljeđivanja poruka drugim akterima i kreiranje dodatnih aktera djece. Podatke koje jedan akter spremi nije moguće promijeniti direktno drugim akterima, što rezultira time da jedan akter ne može izvršavati procese koji mijenjaju stanje drugog procesa bez slanja poruke drugom akteru. [30]

Akteri imaju neke važne osobine koje ih posebno čine pogodnima za okruženje distribuiranih sustava, a koje **su labava povezanost, lakoća i održavanje vlastitog stanja** [33]. Osim toga prenošenje poruka između aktera je potpuno **asinkrono** i bez ograničenja na redoslijed poruka [30]. S obzirom na to da je interakcija između aktera ograničena na prosljeđivanje poruka, oni se mogu rasporediti po čvorovima kao što su poslužitelji, virtualni strojevi i spremnici [33].

Akka je jedan od okruženja Actor Model-a, a o njemu će se pričati u nastavku.

5.2.3.1. Akka

Akka je skup alata otvorenog koda *„… za dizajniranje skalabilnih, otpornih sustava koji obuhvaćaju procesorske jezgre i mreže*“ [48]. Akka pruža hijerarhijsku implementaciju aktera i sadrži biblioteke za upravljanje klastera aktera, raspodjelu i perzistenciju [33].

Unutar Akka, otpremnik poruka upravlja dretvama unutar sustava aktera, na način da definira koji će se izvršitelj usluga koristiti, veličinu spremišta dretvi i količinu poruka koje će

akter obraditi prije nego napusti dretvu [48]. Otpremnik poruka dodjeljuje određenom akteru dretvu ako on treba obraditi poruku te po obradi poruke akter napušta dretvu [48]. Prednost takve dodijele i napuštanja dretve je to da je dretva trošena samo kada postoji posao za aktera i to što akteri bez posla ostaju u memoriji, što ih čini dostupnima za izvršavanje u svakom trenutku [33].

6. Strujanje podataka

Strujanje podataka danas se sve više koristi. Pojam strujanje (eng. *streaming*) zapravo se danas često spominje u kontekstu snimanja i pregleda videozapisa uživo, pa i samim pretraživanjem o istome može se naići na puno stranica koje pružaju prenošenje videozapisa uživo. Uz videozapise, često se spominje i u okviru preslušavanja glazbe. Razlog tome je što su sami videozapisi i pjesme velika količina podataka koju je teško prenositi putem interneta zbog propusnosti. Strujanje podataka uistinu jest povezano s videozapisima i muzikom, ali nema veze s njihovom izradom ili korištenjem već s načinom prenošenja podataka putem interneta.

Ne postoji standard prema kojem se može jasno reći što je strujanje podataka, pa stoga postoji jako puno različitih nesukladnih definicija i različitih shvaćanja pojma strujanja podataka. K tome uz strujanje podataka (eng. *data streaming*) jako su povezani i analiza strujanja podataka (eng. *streaming analysis*), sustav baziran na strujanju podataka (eng. *streaming systems*), procesuiranje tokova (eng. *stream processing*) i podaci unutar strujanja podataka (eng. *streaming data*). Zbog toga se strujanje podataka zna definirati kao sustav, obrada ili kao podaci. Posebnu pažnju treba obratiti na razlikovanje pojmova obrade tokova, podataka strujanja, sustava baziranog na strujanju podataka i strujanje podataka.

Podaci strujanja podataka su podaci kontinuirano generirani iz više različitih izvora koji obično šalju podatke u manjem obujmu na odredišta. Oni su **neprestano rastući neograničen** skup podataka koji se obrađuje i analizira u skoro realnom vremenu [1]. Tok podataka (eng. *data stream)* se danas koristi kao naziv za apstrakciju koja predstavlja neograničen skup podataka. Podaci strujanja podataka uključuju širok spektar podataka, poput datoteka dnevnika koje generiraju kupci pomoću pojedinih mobilnih ili web aplikacija, datoteke kupnje generirane pomoću e-trgovine, datoteke aktivnosti igrača u igrici, informacije s društvenih mreža, dokumenti financijskih ili geoprostornih usluga s povezanih uređaja ili instrumenata u podatkovnim centrima. Takve podatke treba obrađivati uzastopno i postepeno, te ih koristiti za analiziranje. **Strujanje podataka** je tehnologija prijenosa podataka preko interneta koja omogućava da se kasnije podaci obrađuju kao neprekidan niz. **Obrada toka podataka ili strujno procesuiranje** uključuje čitanje podataka iz neograničenog skupa podataka, obrađivanje tih podataka i korištenje rezultata obrade za pravovremeno reagiranje na njih [42]. Ono je tehnologija koja korisnicima omogućuje neprekidan tok podataka i brzo otkrivanje uvjeta u malom vremenskom razdoblju od primanja podataka. Strujno procesuiranje je krovni pojam koji uključuje analitiku u realnom vremenu, analitiku strujanja podataka, kompleksnu obradu događaja itd.

Sustav baziran na strujanju podataka je stoga vrsta mehanizma za obradu podataka koji je dizajniran s razmišljanjem o beskonačnim nizovima podataka [42]. Sustav baziran na strujanju podataka može se gledati kao skup komponenti, izgrađenih za preuzimanje i obradu velike količine podataka s više izvora.

Strujanje podataka uključuje sve podatke koji su potrebni u velikoj količini i zahtijevaju brzu dostavu. Tako danas strujanje podataka koristi Spotify, Netflix, online igrice, Uber, TikTok i mnoge dr. aplikacije.

6.1. Karakteristike strujanja podataka

Strujanje podataka je postalo poznato zbog samih videozapisa. No iako se danas sve više koristi, ne postoji neka dosljedna definicija pa tako i same karakteristike sustava na bazi strujanja podataka. Tako više različitih izvora spominje različite karakteristike. U ovom poglavlju bit će navedene karakteristike strujanja podataka i sustava na bazi strujanja podataka.

Podaci koji se šalju su najčešće: velikog volumena, beskonačni, kontinuirani, niske razine i podijeljeni u manje dijelove. Podaci koji se obrađuju dolaze u red čekanja te se zbog obrade podataka u realnom vremenu očekuje da su ti podaci **kontinuirani** i **beskonačni** s obzirom na to da se generiraju i prikupljaju u realnom vremenu iz IoT senzora, serverskih dnevnika, klikova korisnika i oglašavanja u realnom vremenu. Strujanje podataka kao tehnologija upravo služi tome da se mogu velike količine podataka proizvesti, prenijeti i koristiti. Stoga su sami podaci s obzirom na njihovu kontinuiranost i beskonačnost najčešće **velikog volumena**. S druge strane, strujanje podataka nam koristi kako bi mogli nesmetano konzumirati i obrađivati podatke u realnom vremenu. Zato su podaci koji su prikupljani najčešće **niske razine**, u JSON formatu ili nekom drugom formatu zbog jednostavnosti obrade i korištenja podataka. Veća količina vode može se prenijeti brže s više cijevi, no u slučaju prijenosa podataka mora se pomno poslati podatke kroz „cijevi“ kako bi ih mogli zaprimiti i obraditi. Zato su podaci najčešće poslani u obliku u kojem su prikupljeni odnosno kao razlomljeni podaci ili se oni razlamaju na manje smislene dijelove ako je to potrebno.

Iz navedenog o podacima mogu se iščitati neke od **karakteristika samog strujanja podataka**. Strujanje podataka kao takvo ima visoku frekvenciju dolazaka podataka s obzirom na to da su podaci slani kontinuirano, u realnom vremenu i bez prestanka. Danas se pokušava uspostaviti odgovor na određene akcije u realnom vremenu odnosno samo strujanje podataka nastalo je zbog velike potrebe za brzim odgovorom na neposredne promijene unutar samih podataka. Ponekad kako bi odgovorili na podatke potrebna je analiza tih podataka te strujanje podataka stoga uključuje i analizu i obradu podataka u realnom vremenu. Kontinuiranost

podataka se više odnosi na slanje samih podataka u pogledu vremena, no sami podaci ne moraju nužno biti redoslijedno zaprimljeni. Sve komponente strujanja podataka trebaju raditi nesmetano te zbog toga se treba stvoriti neovisnost komponenti odnosno labavo povezivanje kako bi se ostvarila otpornost komponenti na greške. U suštini strujanje podataka omogućuje brzo i sigurno zaprimanje velike količine razlomljenih podataka, obrađivanje velike količine podataka te analizu velike količine podataka i to sve u realnom vremenu.

Može se zaključiti da strujanje podataka:

- ima visoku frekvenciju dolaska podataka,
- brzi odgovor na brzo mijenjajuće podatke,
- analiza u realnom vremenu i
- obrada u realnom vremenu.

Razvojem tehnologije pojavljuju se određeni zahtjevi za aplikacije. U početku nije bilo potrebno da aplikacija trenutno obrađuje podatke u realnom vremenu. Kako se pojavljuje umjetna inteligencija te poslovna inteligencija i analiza, došlo je do potrebe za brzim prijenosom podataka i njihovom brzom obradom. Današnja poduzeća jednostavno ne mogu čekati da se podaci **obrade putem „batch“ oblika.** Tako se danas sve aplikacije; od aplikacija otkrivanja malicioznih programa, platformi za burzu do aplikacija za dijeljenje dokumenata i aplikacija e-trgovine oslanjaju na dohvaćanje podataka u realnom vremenu. Uključivanjem strujanja podataka, aplikacije evoluiraju u aplikacije koje ne samo da integriraju podatke, već ih i obrađuju, filtriraju, analiziraju i reagiraju na njih u stvarnom vremenu, kako su zaprimljeni [42].

Karakteristike sustava strujanja podataka su:

- skalabilnost sustava,
- elastičnost sustava,
- otpornost na promijene i greške,
- reakcija na promijene u realnom vremenu,
- analiza u realnom vremenu,
- kratko vrijeme odgovora i
- zaprimanje velikog volumena kontinuiranih podataka u visokoj frekvenciji.

Sam sustav strujanja podataka nasljeđuje neke karakteristike strujanja podataka pa s time mora omogućiti visoku frekvenciju dolazaka podataka, mora brzo odgovoriti na mijenjajuće podatke te ih analizirati i obraditi u realnom vremenu. Kako bi to bilo moguće moraju se ispuniti određeni zahtjevi sustava.

Od sustava koji treba zaprimiti veliku količina podataka očekuje se da je skalabilan. Sustav je skalabilan ako može održati razinu svojeg izvođenja pri dodavanju novih funkcionalnosti ili rukovati s povećanjem zahtjeva i povećanjem podataka. Pri tome se najčešće od sustava strujanja podataka očekuje da mogu rukovati s većom količinom podataka i različitom frekvencijom dolazaka podataka što uključuje i smanjivanje i povećanje frekvencije dolazaka pri čemu sustav mora neometano raditi.

Osim što se zahtjeva da je sustav skalabilan, on mora biti otporan na pogreške i elastičan. Otpornost se ostvaruje labavim povezivanjem komponenti i sustavom vođenim porukama. Prema tome, ako dođe do kvara na jednoj komponenti, druga komponenta može nesmetano raditi. Tako se izoliraju greške, ali i same komponente što dozvoljava izolirane promjene na komponentama. Mora biti omogućeno i spremanje određenih podataka ako je došlo do pogreške. Primjerice ako sustav čita iz reda poruka, bilo bi kritično da se ne spremi podatak o zadnje pročitanoj poruci. Ne spremanje tog podatka rezultiralo bi ponovnim čitanjem svih poruka i njihovom ponovnom obradom.

Potrebno je također ostvariti savršen sklad brisanja podataka iz reda čekanja. Naime raznih je slučajeva potreba podataka koji se čitaju iz reda poruka. Moguće je da komponenta radi tako da nakon uzrokovanog gašenja ponovo čita sve dostigle poruke i obrađuje ih ili da je potrebno pročitati samo neposredne podatke. U tom slučaju ponovno čitanje svih podataka može izazvati zastoj ako se poruke ne brišu nakon nekog vremena, a opet nije niti dobro da se poruke brišu nakon što su obrađene. Stoga treba omogućiti određivanje vremena nakon kojeg će se određeni podatak obrisati.

Danas aplikacije teže k tome da mogu automatski reagirati na korisnikove radnje, primjerice analizirati korisnikova sviđanja i klikove i prema tome prikazivati sadržaj i oglase korisniku. Sustav za strujanje podataka baziran je na tome da se može odgovoriti u realnom vremenu na promijene u okruženju. Stoga sustav za strujanje podataka mora omogućiti procesuiranje podataka kako dolaze, nasuprot tome da se najprije prikupljaju podaci u kolekciju pa onda obrađuju što bi dovelo do zastoja ili nedovoljno brzog odgovora na promijene podataka. Pošto se podaci obrađuju trenutno po primitku to za sobom veže i kraće vrijeme odgovora na pojedini zahtjev. Procesuiranje novih podataka omogućuje analizu u realnom vremenu.

6.2. Komponente arhitekture za strujanje podataka

Arhitektura sustava za strujanje podataka je sastavljena od komponenti koje mogu obraditi i zaprimiti veliku količinu podataka iz više različitih izvora. Aplikacija koja je bazirana na strujanju podataka prima podatke odmah po generiranju, zadržava ih i obrađuje. Arhitektura

shodno s time može imati dodatne komponente u slučaju potrebe, poput alata za procesuiranje u stvarnom vremenu, upravljanje podacima i analitiku. Arhitektura sustava na bazi strujanja podataka mora biti u stanju uzeti u obzir jedinstvene karakteristike tokova podataka, koji mogu uključivati velike količine podataka, a u najboljem slučaju podaci su polustrukturirani.

Unutar arhitektura aplikacije „brzih“ podataka često se spominje **SMACK stog** koji redom uključuje Apache Spark alat za analizu podataka, Apache Mesos kao distribuirani sustav, Akka skup alata koji šalje i povlači podatke, Apache Cassandra NoSQL bazu podataka i Apache Kafka koji pruža funkcionalnost sustava poruka [40]. Prva slova ovih alata redom čine naziv SMACK. Nešto kasnije će se više govoriti o nekim poznatim Apache alatima koji služe kao potpora aplikacijama na bazi strujanja podataka.

S obzirom na učestalost korištenja svih sastavnica SMACK-a može se zapravo zaključiti kakve i koje komponente koristiti unutar aplikacije za strujanje podataka. Sustav za prenošenje podataka igra veliku ulogu u unosu podataka te u brzom i sigurnom distribuiranju podataka, te je takav sustav kritičan za povezivanje svih komponenti zajedno. Podaci koji se prenose trebaju biti trenutno dostupni, a sustav mora moći tolerirati duplicirane ili ispuštene poruke. Bazu podataka ili bilo kakvo drugo spremište podataka mora biti dizajnirano za skaliranje kada je ono potrebno, kako bi se išlo u korak s količinama i brzinom dolaska podataka. S obzirom na to da je potrebna obrada, transformacija i upotreba dobivenih podataka u realnom vremenu potrebni su određeni alati koji mogu procesuirati velike količine podataka u trenu.

S obzirom na to da postoji puno sustava na bazi strujanja podataka i načina kako ostvariti strujanje te kako se sve brže razvija tehnologija rad će se bazirati na osnovnim komponentama koje svaka arhitektura sustava baziranog na strujanju podataka mora sadržavati. Te su komponente **procesor tokova ili posrednik poruka, ETL alati, analiza podataka i spremište podataka** [38]**.**

Procesor tokova ili posrednik poruka je zapravo tehnologija koja prihvaća veliku količinu podataka i usmjerava ih na odredište. Pri tome tokovi podataka koji dolaze mogu se procesuirati, obrađivati i uređivati kako bi na odredištu mogli maksimalno iskoristiti dobivene podatke. Posrednik poruka je u ovom kontekstu zapravo sustav poruka. Česti posrednici poruka koji se koriste pri strujanju podataka su Apache Kafka i Amazon Kinesis Data Streams, koji su i sami poznati kao sustavi poruka na „steroidima“. U daljnjem tekstu će biti objašnjen rad Apache Kafka alata. Za razliku od prvih sustava poruka, posrednici poruka za strujanje podataka podržavaju vrlo visoke performanse, imaju ogroman kapacitet za poruke i usko su usredotočeni na strujanje podataka s malo podrške za transformaciju podataka ili raspoređivanje zadataka [38].

Podaci koji se koriste pri strujanju podataka su zapravo podaci dobiveni iz IoT senzora, serverskih i sigurnosnih dnevnika, oglašavanja u stvarnom vremenu i podaci dobiveni korisničkim korištenjem web i dr. aplikacija. Sami podaci koji se strujanjem prenose su najčešće sirovi, izvorni podaci niske razine. Kao takve, ovisno o potrebi, treba ih strukturirati, filtrirati, proširiti, objediniti, transformirati ili obraditi prije nego što se mogu koristiti. U tu svrhu služe **ETL alati** (eng. ***E****xtracting,* ***T****ransforming and* ***L****oading – ETL*), koji su ranije spomenuti u vidu podatkovne integracije sustava, ili platforma koja prima upite od korisnika, dohvaća događaje iz reda poruka i primjenjuje upit kako bi generirala rezultat [38]. Rezultat procesuiranja takvih podataka može biti poziv metode, poduzimanje određenih akcija, prezentacija podataka, kreiranje upozorenja na događaj ili u nekim slučajevima novi tok transformiranih podataka [38]. Neki od poznatijih ETL alata za strujanje podataka su Apache Storm i Apache Spark.

Kako bi pripremljeni podaci dobili na većoj vrijednosti, potrebno ih je **analizirati**. Dobar primjer korištenja analize u realnom vremenu je unutar mobilne aplikacije TikTok. Naime ona prikuplja podatke u realnom vremenu o zadržavanju određenog korisnika na videu određene tematike uz prikupljanje podataka o sviđanju korisnika, pregleda svih korisnika i sl. Analizom tih podataka odlučuje u realnom vremenu koji će idući sadržaj prikazivati korisniku. Može se reći sa sigurnošću da je takva analiza omogućila veću konkurenciju aplikacije i mjesto na tržištu. Iz primjera se može vidjeti prava vrijednost analize podataka u svojim aplikacijama. Postoje mnogi alati i biblioteke za analizu strujanja podataka, a najviše se spominju Amazon Athena, Amazon RedShift, Elasticsearch i Apache Cassandra [38].

Na samom kraju potrebno je i **spremište podataka.** Danas postoje razne tehnologije za spremanje podataka, no treba uzeti u obzir kakvi podaci se spremaju, koliko dugo trebaju biti spremljeni, koju količinu i kojom brzinom se spremaju. U okviru strujanja podataka često se spominju NoSQL baze podataka kao što su RocksDB, Apache Cassandra, no na popularnosti dobivaju i grafičke baze podataka poput Neo4j. Tijekom izrade aplikacije dobro je upitati se koje podatke je zapravo potrebno spremiti unutar baze podataka, a koje je dovoljno postaviti u red čekanja.

6.3. Primjeri sustava sa strujanjem podataka

Danas skoro svaka aplikacija koristi strujanje podataka. Primjer toga je Netflix koji strujanje podataka koristi za preporuke u realnom vremenu, Uber koji strujanje podataka koristi kako bi predložio i pokazao vozače u okolici naručitelja, Facebook koji strujanje podataka koristi kako bi analizirao sviđanja korisnika i prikazao mu određeni sadržaj ili Amazon koji koristi strujanje podataka kako bi prema kupovini ili pretraživanju korisnika preporučio

proizvode korisniku. Ovo poglavlje posvećeno je primjeru korištenja strujanja podataka na web stranicama.

Netflix je online platforma za gledanje filmova i serija. Netflix temelji svoj rad na podacima tako da su sve poslovne odluke i odluke o proizvodima bazirane na analizi podataka. U tu svrhu gotovo svaka aplikacija na Netflix-u koristi cjevovode podataka za prikupljanje, obradu i premještanje podataka u oblaku [57]. Prema [57] putem aplikacija se proizvede 24 gigabajta u sekundi tijekom vrhunca opterećenja. Netflix prikuplja podatke o ulaznim/izlaznim aktivnostima korisnika, aktivnostima pregleda videa, dnevnicima grešaka, događajima performansa i događajima pronalaženja grešaka i dijagnostike [58].

Pri pojavi Apache Kafka i ElasticSearch, unutar Netflix-a se stvara potreba za analitiku u realnom vremenu pa se na već napravljeni sustav koji uključuje Chukwa nadodao Apache Kafka sustav koji je primao podatke iz Chukwa. Usmjerivač, kojem Apache Kafka sustav šalje poruke, nakon toga filtrira podatke i šalje dalje na drugi Kafka sustav ili na ElasticSearch. Nadalje Netflix koristi Apache Flink i Apache Spark za procesuiranje tokova podataka i Apache Hadoop za procesuiranje podataka. Nakon pokušaja i pogrešaka, stvorila se potreba za zamjenom Chukwa alata s Apache Kafka sustavom. [57]

Prema navedenome može se zaključiti da je Netflix uvidio potrebu za strujanjem podataka, a primjene strujanja podataka unutar Netflix-a se odnose na prikupljanje, slanje, procesuiranje i analizu podataka.

Pored Netflixa, Uber isto koristi alate za strujanje podataka. Dapače, Uber ima jednu od najvećih implementacija Apache Kafka sustava na svijetu – procesuiraju se bilijune poruka i nekoliko petabajta podataka svaki dan [25]. Unutar Uber Kafka ekosustava nalaze se Kafka cjevovod, Apache Flink, Apache Hadoop, Apache Cassandra i ostali alati. Uber koristi Apache Flink za brzu analitiku u stvarnom vremenu, Apache Cassandru kao bazu podataka i Apache Hadoop za procesuiranje podataka. Proizvođači poruka unutar Uber ekosustava su aplikacije vozača, aplikacije naručitelja vožnje i ostali servisi koji proizvode podatke [25]. Uber koristi strujanje podataka zbog prijenosa velike količine podataka o položaju vozača i naručitelja, izračunavanje najbližih vozača od lokacije naručitelja, izračunavanje cijene vožnje i nadgledanje provedbe plaćanja korisnika kroz aplikaciju.

7. Izrada projekta

U ovom poglavlju bit će objašnjene funkcionalnosti aplikacije, sama izrada projekta i korištenje određenih alata. Naglasak izrade aplikacije je na strujanju podataka tako da se glavnina aplikacije odnosi na pozadinske funkcionalnosti poput prikupljanja, obrade i prijenosa podataka. Najprije su istaknuti alati koji su se koristili pri izradi projekta, a kasnije se ulazi u samu izrade aplikacije.

7.1. Alati

Za izradu aplikacije koristio se alat NetBeans zbog lakog i intuitivnog korištenja. Izrada korisničkog sučelja se bazira na okviru PrimeFaces. PrimeFaces je okvir otvorenog koda za JavaServer Faces, a koristi se kod kreiranja korisničkog sučelja i omogućava lagano i efikasno kreiranje i uređivanje stranice [47]. Pri izradi svih dijagrama koristio se online alat Visual Paradigm, a ostali alati su navedeni dalje u tekstu [55].

7.1.1. Apache Kafka

Apache Kafka je distribuirana platforma otvorenog koda za strujanje podataka. Apache Kafka je sustav za skalirano skupljanje, procesuiranje, spremanje i analiziranje podataka [7] Najpoznatiji je po svojim izvrsnim performansama, malom vremenu odgovora, toleranciji pogrešaka i velikoj propusnosti što omogućava obradu tisuća poruka u jednoj sekundi [9]. Koristi ju preko 60% poduzeća na listi Fortune 100 poduzeća s najvećim bruto prihodom u Sjedinjenim Američkim Državama [4]. Apache Kafka pruža mogućnost objave i pretplate na strujne podatke uz kontinuiran uvoz ili izvoz podataka iz drugih sustava, mogućnost spremanja podataka trajno i pouzdano i mogućnost obrade tokova događaja kako se javljaju ili retrospektivno [6].

Kafka je distribuirani sustav koji se sastoji od poslužitelja i klijenata koji komuniciraju putem TCP mrežnog protokola visokih performansi [7]. Klijenti mogu biti proizvođači ili potrošači poruka. Proizvođači poruka, kako samo ime govori su klijentske aplikacije koje proizvode ili objavljuju poruke na Kafka temu, a potrošači su klijentske aplikacije koje se pretplaćuju na temu, odnosno čitaju i procesuiraju Kafka poruke. Ranije je spominjana labava povezanost i koliko je ona bitna za sustave. Unutar Kafke, proizvođači i potrošači su međusobno potpuno neovisni što rezultira visokom skalabilnošću. Proizvođači ne trebaju čekati na potrošače i obrnuto.

Poruke unutar sustava su distribuirane i trajno pohranjene u Kafka teme. Analogija za Kafka temu je red čekanja poruka. Prednost tema kod Kafka sustava je što može imati nula, jednog ili više proizvođača koji zapisuju događaje, kao i nula, jednog ili više potrošača koji se pretplaćuju na te događaje. Unutar Kafke moguće je definirati koliko dugo se želi zadržati podatke unutar teme nakon čega će se podaci izbrisati. Time se poruke unutar sustava mogu pročitati jednom ili onoliko puta koliko je potrebno. Teme se mogu raščlaniti na određen broj particija. Particije se mogu gledati kao dodatni zapisnici ili analogno kao na niz cijevi kojima voda prolazi. Ako tema ima više particija nije garantirano da će one pri povlačenju biti vremenski poredane. Ono što se sa sigurnošću može reći je da će dva podatka s istim ključem biti na istoj particiji. Pri pretplati potrošača na temu s više particija, Kafka sustav određuje na koje particije će se potrošač pretplatiti. Prema tome, ako postoje četiri particije i dva potrošača, svaki potrošač će čitati iz svake druge particije.

Osim biblioteke za potrošača i proizvođača, Apache Kafka pruža Kafka Connect i Kafka Streams. Kafka Connect je alat za skalabilno i sigurno strujanje podataka između Kafke i drugih sustava poput baze podataka, REST servisa i sl. [5]. Kafka Connect za bazu podataka se može zamijeniti s Kafka Streams ili proizvođačem i potrošačem jer je Kafka Connect funkcionalno baziran na dohvaćanju i slanju podataka na Kafka sustav. Veliki naglasak Kafka Connect-a se postavlja na strujanje podataka između Kafke i drugih sustava. Kafka Streams je klijentska biblioteka za procesuiranje i analizu podataka unutar Kafka teme. Kafka Streams može se gledati kao posrednik obrade tokova između proizvođača i potrošača. Kuhara u restoranu može se gledati kao posrednika između kupca i proizvođača proizvoda koje koristi pri izradi jela. Kuhar umjesto da posluži sirove proizvode, transformira proizvode u određeno jelo. Tako Kafka Streams osim što prenosi poruku s jedne teme na drugu povezujući proizvođača i potrošača; obrađuje, transformira i analizira podatke dobivene s teme i šalje nove, ažurirane, transformirane ili filtrirane podatke za potrošača.

Bitno je spomenuti kako Apache Kafka radi uz Zookeeper server pa ga je potrebno instalirati i pokrenuti prije korištenja Apache Kafka alata. Kafka pruža podršku za Javu preko specificiranih biblioteka. Kasnije će se više govoriti o korištenju Kafka biblioteka i kreiranju potrošača, proizvođača i Kafka Streams.

7.1.2. Apache Spark

Apache Spark je objedinjeni analitički alat za veliku obradu podataka. Ono je platforma otvorenog koda koja omogućuje brzu i skalabilnu obradu velikih skupova podataka. Apache Spark podržava programiranje u programskim jezicima Java, Scala, Python i R putem aplikacijskih programskih sučelja [10]. Velika prednost Apache Spark-a je sposobnost obrade unutar radne memorije što osigurava poboljšavanje performansi i brzinu izvedbe [10].

Apache Spark nudi sljedeće module : Spark SQL, Spark Mlib, Spark Streaming, Spark GraphX i Spark Core [12].

Ovo je još jedna prednost alata Apache Spark jer objedinjuje provođenje SQL upita, izgradnju toka podataka, korištenje statističkih algoritama, rad s grafovima, korištenje metoda za duboku analizu podataka i dr. Apache Spark dopušta kombiniranje navedenih modula čime se izbjegava teško povezivanje više alata i dobiva na jednostavnosti kreiranja potrebnih funkcionalnosti sustava. Bitno je spomenuti da se Apache Spark bazira na lijenom izvođenju (eng. *lazy evaluation*). Apache Spark je organiziran oko gospodar-rob arhitekture (eng. *master-slave architecture*) prema tome postoji jedan centralni proces koji koordinira velik broj radnih čvorova koji su zasebni procesi [12]. Spark GraphX je novija komponenta Spark za grafove i paralelno računanje na grafovima [13].

Spark Core je osnovna baza cijelog Apache Spark projekta. Ona podržava osnovne ulazno-izlazne operacije, raspodjelu zadataka, upravljanje memorijom i slično. Osnova rada Spark Core-a se bazira na *RDD* (eng. *Resilient Distributed Dataset*) apstrakciji. *RDD* je nepromjenjiva podijeljena kolekcija elemenata nad kojima se mogu paralelno izvršavati operacije [16].

Spark SQL je biblioteka za rad sa strukturiranim podacima kao što su podaci CSV datoteke, baze podataka i sl. Spark SQL integrira relacijsko procesuiranje s funkcionalnim programiranjem. Sadržava *DataSet* koji je tipizirana kolekcija domenski specifičnih objekata koji se mogu transformirati u paralelne funkcijske ili relacijske operacije [11]. Sa Spark SQL mogu se vršiti upiti nad podacima pročitanim iz datoteka i drugih izvora koji ne moraju tipično biti baza podataka.

Spark Streaming omogućava skalabilni tok podataka velike propusnosti, otpornog na greške. Putem Spark Streaming omogućava nam se spajanje na bilo koji sustav koji koristi strujanje podataka. Spark Streaming dobiva podatke te ih dijeli u hrpe koje su onda procesuirane pomoću Spark mehanizma kako bi se generirao konačni tok podataka u obliku hrpa [15]. Dobiveni podaci se mogu kasnije obraditi na željeni način.

Spark Mlib je biblioteka za strojno učenje čiji je glavni cilj napraviti strojno učenje skalabilnim i jednostavnim za korištenje. Ova biblioteka uključuje algoritme strojnog učenja kao što su regresija, klasifikacija i suradničko filtriranje, perzistenciju za spremanje i pokretanje algoritama i modela, metode za linearnu algebru i statistiku, metode za transformaciju, smanjenje dimenzionalnosti i selekciju te alate za kreiranje, evaluiranje i podešavanje cjevovoda strojnog učenja. [14]

7.1.3.Java DB i Apache Derby

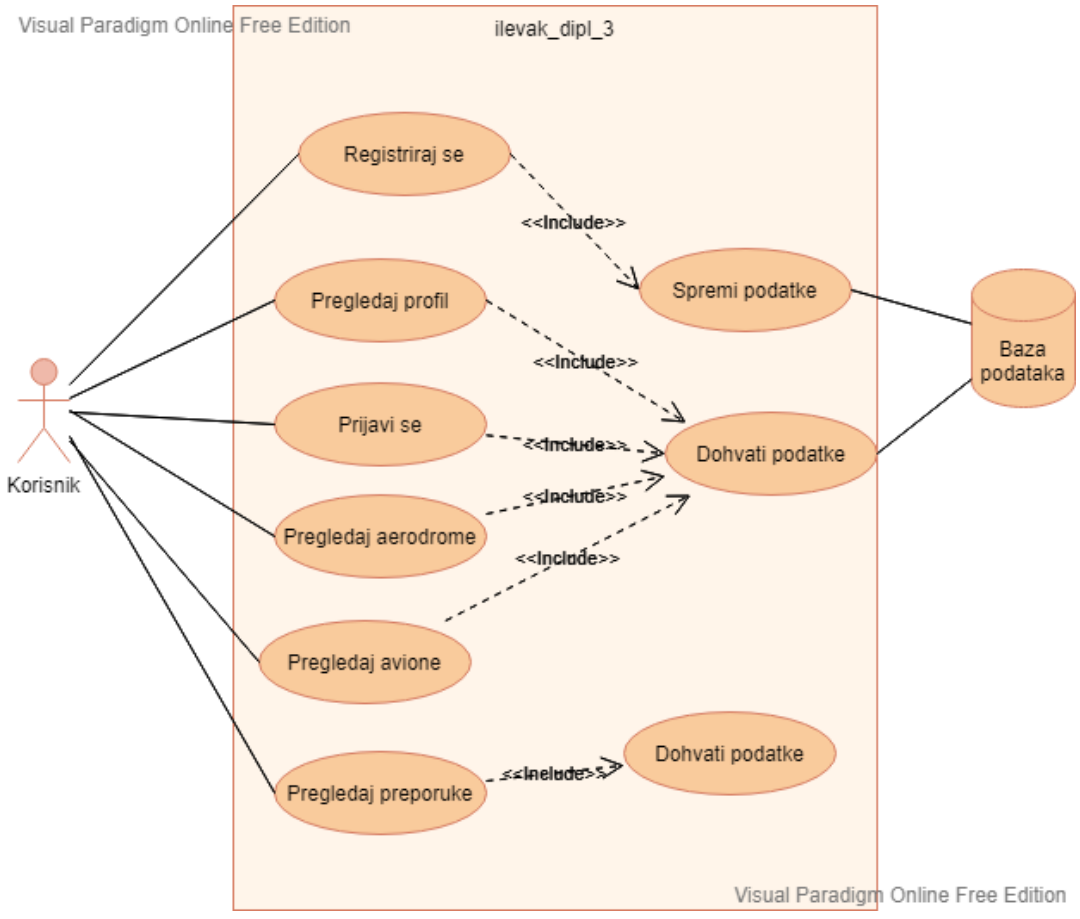
Unutar projekta za bazu podataka se koristi Java DB i Apache Derby. Java DB je distribucija Apache Derby baze podataka otvorenog koda, odnosno to je relacijski sustav za upravljanje bazom podataka baziran na programskom jeziku Java i jeziku upita SQL. S obzirom na to da Java DB uključuje Derby bez ikakvih modifikacija, oni imaju iste funkcionalnosti. [2]

7.1.4. OpenSky Network

OpenSky Network stranica je na kojoj se mogu pronaći svi podaci o letovima aviona, avionima, aerodromima i slično. Osim što se može pregledati podatke, OpenSky Network pruža REST servis za dohvaćanje podataka. Popis metoda i detalji podataka koji se dohvaćaju se nalaze unutar njihove dokumentacije. S obzirom na to da je ograničen pristup podacima koji se dohvaćaju, potrebno se prijaviti na stranici. Nadalje u tekstu bit će objašnjeno korištenje ovog REST servisa. [43]

7.2. Opis korisničke aplikacije

Aplikacija je namijenjena jednom tipu korisnika kojem pruža mogućnost praćenja podataka o avionima i aerodromima. Novi korisnici najprije trebaju registrirati račun. Jednom kada korisnik ima profil može se prijaviti unutar aplikacije i koristiti ostale funkcionalnosti. Nakon prijave korisnik unutar izbornika može izabrati pregled aviona koje prati, pregled aerodroma koje prati, pregled profila i pregled preporuka aviona. Ako izabere pregled aviona, prikazuju mu se podaci o njegovim praćenim avionima, te podaci o avionima koje korisnik ne prati. Korisnik može pregledati informacije o avionima, pregledati letove određenog aviona te zapratiti/otpratiti određeni avion. Ako pak korisnik izabere pregled aerodroma, prikazuju mu se aerodromi koje on prati i one koje može zapratiti. Korisnik može zapratiti/otpratiti aerodrom, pregledati polaske aviona s određenog aerodroma i slično. Osim pregleda korisničkih aviona, korisnik može pogledati preporuke sustava za avione, koje može na mjestu zapratiti. Zadnja opcija izbornika je pregled profila gdje korisnik ima uvid u detalje svojeg profila koje može uređivati. Opisana interakcija korisnika s aplikacijom je dana na sljedećoj slici na dijagramu pri čemu je prikazan rezimiran, nezgrapan dijagram slučajeva korištenja.

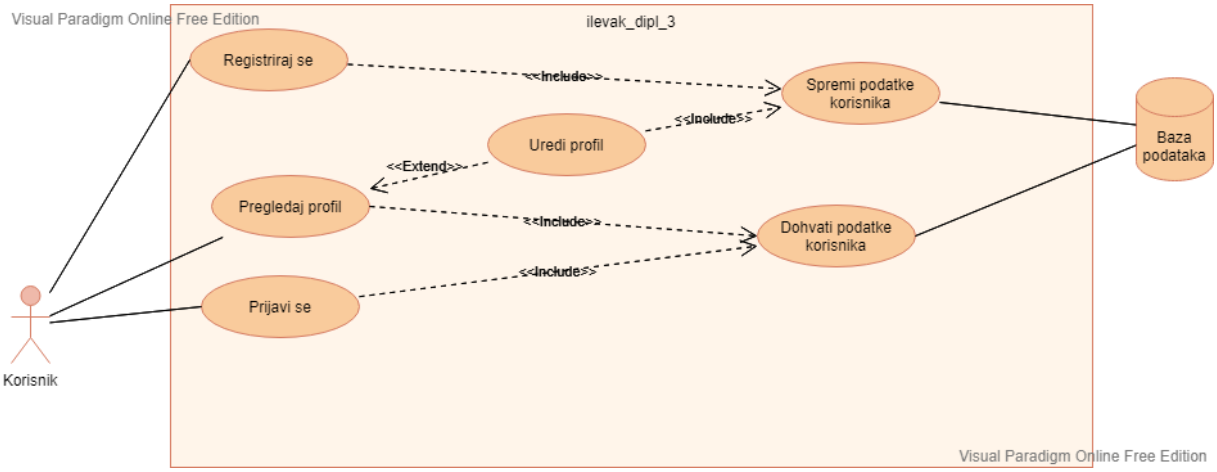


Slika 6: UML dijagram slučajeva korištenja (Izvor: vlastita izrada)

Kao što se može vidjeti iz slike, korisnički niz korištenja rezultira određenim akcijama na bazi podataka. Unutar dijagrama nije detaljno pojašnjen odnos korisničkog djelovanja i baze podataka, te korisničkog djelovanja i Kafka sustava poruka. Nadalje će biti prikazane detaljne interakcije sustava i sudionika.

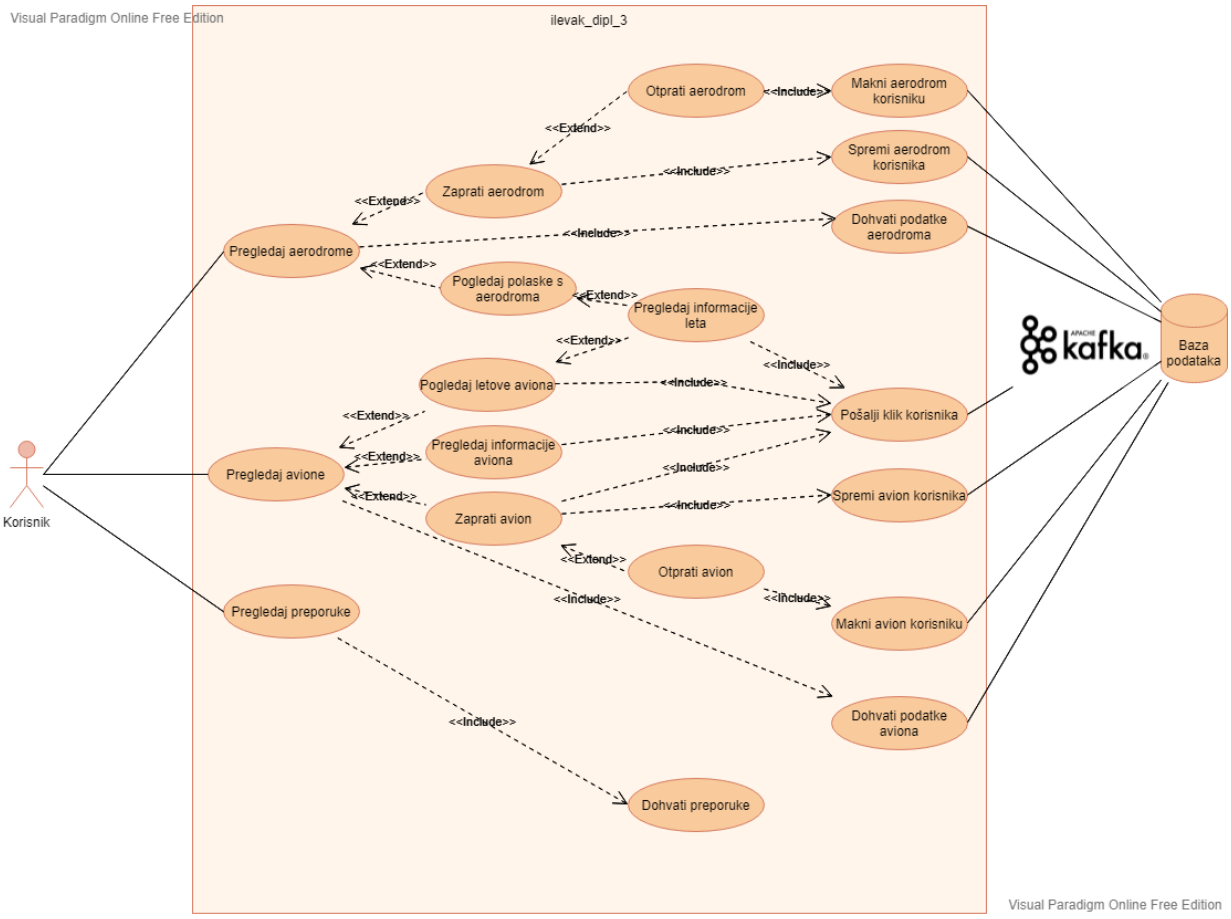
7.3. Opis slučajeva korištenja

Već su dijelom opisani sami slučajevi korištenja aplikacije, no korisnikove radnje ne rezultiraju samo čitanjem i upisivanjem u bazu podataka. Osim baze podataka i korisnika kao sudionika, pojavljuje se Kafka sustav poruka, čijim se proizvođačem šalju poruke na sustav poruka. S obzirom na veličinu i nezgrapnost, dijagram slučajeva korištenja je podijeljen u dva dijela pri čemu će jedan detaljnije prikazivati slučaj korištenja prijave, registracije i pregleda profila, dok će drugi detaljnije prikazivati slučajeve korištenja pregleda aerodroma, aviona i preporuka. Na sljedećoj slici može se vidjeti slučaj korištenja prijave, registracije i pregleda profila.



Slika 7: UML dijagram slučajeva korištenja: prijava registracija i pregled profila (Izvor: vlastita izrada)

Pri slučaju registracije korisnika korisnički podaci se spremaju unutar baze podataka. Ako se pak korisnik prijavi, potrebno je provjeriti podatke i dohvatiti iste iz baze podataka. Ako se korisnik prijavi, može pregledati profil te ako želi može ga urediti. Uređivanje profila također je popraćeno spremanjem u bazu podataka. Sljedeći dijagram slučajeva korištenja opisuje pregled aerodroma, aviona i preporuka aviona.



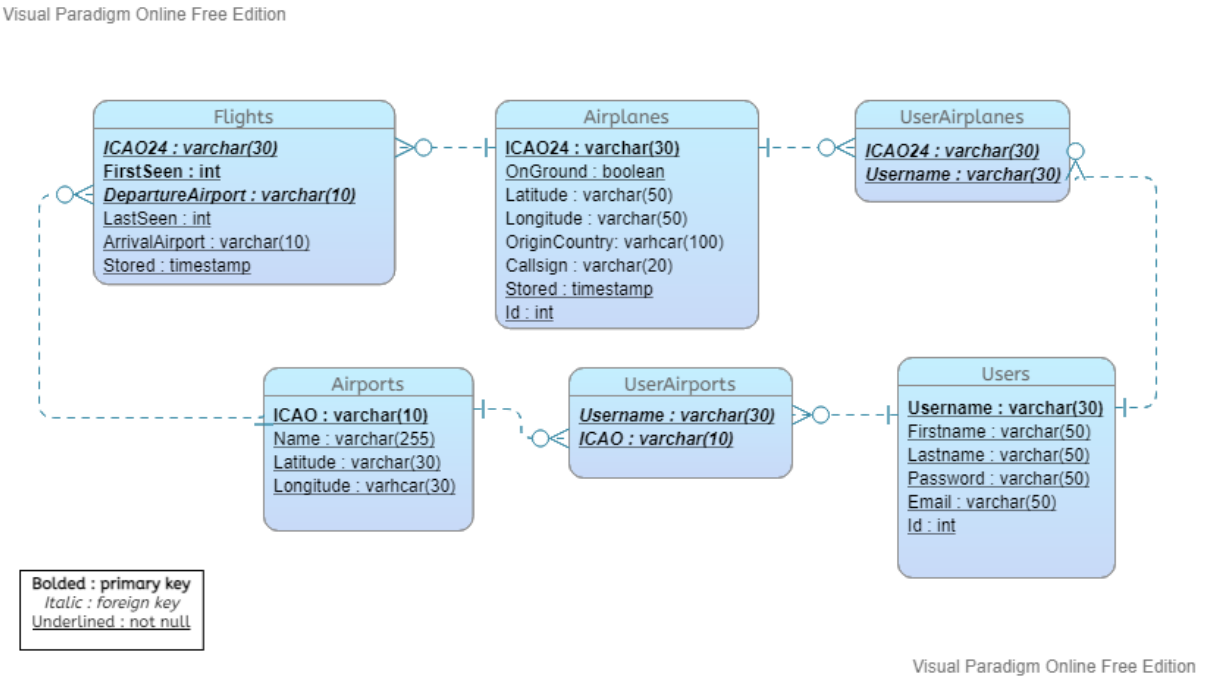
Slika 8: UML dijagram slučajeva korištenja: pregled aviona, aerodroma i preporuka (Izvor: vlastita izrada)

U svakom slučaju korištenja kod pregleda podataka podaci se trebaju najprije dohvatiti, bilo to iz baze podataka pri pregledu aerodroma i aviona ili iz datoteke pri pregledu preporuka aviona. S obzirom na to da se ti dohvaćeni podaci prosljeđuju dalje po atributima, nije potrebno ponovo dohvaćati podatke sve dok se ne dogodi neka promjena poput novog praćenja aviona i slično. Pri pregledu aerodroma korisnik može zapratiti određeni aerodrom. Ako korisnik zaprati aerodrom, može ga otpratiti. Osim praćenja aerodroma mogu se zapratiti i avioni što isto otvara mogućnost brisanja praćenja aviona. Oba slučaja korištenja rezultiraju sa spremanjem podataka prilikom novog praćenja ili brisanjem podataka prilikom isključivanja praćenja. Unutar prikaza aerodroma korisnik može pregledati polazne letove s određenog aerodroma, a pri prikazu aviona može pregledati detalje aviona i pregledati letove jednog aviona. Osim pregleda popisa letova, može vidjeti detalje samog leta. U slučaju pregleda informacija aviona, pregleda informacija leta i pregleda letova aviona šalje se podatak o korisniku i avionu na Kafka temu.

Ovi slučajevi korištenja prikazuju svu interakciju korisnika s aplikacijom i reakciju sustava na njegovo korištenje. U sljedećim odjeljcima bit će opisan sam rad aplikacijske pozadine.

7.4. Baza podataka

Baza podataka za aplikaciju se sastoji od šest tablica. U ovom odjeljku navedene su tablice i njihova povezanost. Zbog lakšeg shvaćanja baze podataka izrađen je ERA model putem online alata Visual Paradigm. Na sljedećoj slici može se vidjeti ERA model baze podataka na kojem se temelji rad aplikacije. Na slici su unutar tablica podebljani oni atributi koji se odnose na primarne ključeve, nakošeni oni atributi koji su vanjski ključevi te podcrtani oni atributi koji ne smiju biti jednaki nuli. S obzirom na korištenje JDBC i Java DB sustava za upravljanje bazom podataka, unutar naziva tablica i atributa u bazi podataka zanemaruje se veliko ili malo slovo. Prilikom izrade ERA modela su se koristila velika i mala slova za lakše čitanje i razumijevanje atributa i tablica.



Slika 9: ERA model baze podataka (Izvor: vlastita izrada)

Baza podataka se sastoji od sljedećih tablica:

Users – tablica koja opisuje korisničke podatke. Sastoji se od osnovnih opisnih atributa samog korisnika kao što su ime i prezime, korisničko ime, lozinka i email pri čemu je korisničko ime primarni ključ. Dodatno se sastoji od atributa *id* čije korištenje će biti objašnjeno kod izračunavanja preporuka korisnika pri čemu se atribut *id* koristi kao unikatni identifikator korisnika.

Airports – tablica koja se sastoji od podataka o aerodromima. Kao identifikator odnosno primarni ključ se koristi *ICAO* kod (krat. za eng. „**I**nternational **C**ivil **A**viation **O**rganization“ – međunarodna civilna avijacijska organizacija) od četiri slova koji se koristi za identifikaciju aerodroma. Tablica sadrži podatke o geografskoj širini i dužini te nazivu aerodroma.

Airplanes- tablica opisuje stanje jednog aviona u danom vremenu ako postoje podaci o njemu. Sastoji se od primarnog ključa *ICAO24* koji je 24 bitna adresa aviona. ICAO 24 je kritični element za rad s Mode S radarom, a zapisan je i u samom certifikatu registracije aviona. Unutar baze podataka *ICAO24* je prikazan u heksadecimalnom formatu. Tablica *Airplanes* se dodatno sastoji od atributa *id* čije korištenje također služi, kao i kod korisnika, pri izračunavanju preporuka, a koristi se kao unikatni identifikator aviona. Osim navedenog sastoji se od atributa koji opisuju je li avion na tlu, njegovu geografsku širinu i dužinu, zemlju porijekla aviona, njegovu oznaku i zadnje vrijeme spremanja podataka o avionu.

Flights - tablica se sastoji od podataka o letu određenog aviona. Kako su letovi prikupljani kao polasci za određen aerodrom u određeno vrijeme, identifikator je složeni primarni ključ koji se sastoji od *ICAO* polaznog aerodroma, *ICAO24* aviona i vremena polijetanja aviona *FirstSeen*. *ICAO* i *ICAO24* su ujedno i vanjski ključevi referencirani respektivno na tablice *Airports* i *Airplanes*. Osim navedenih atributa, sastoji se i od vremena slijetanja aviona na aerodrom, vremena zapisa unutar baze podataka i *ICAO* aerodroma slijetanja.

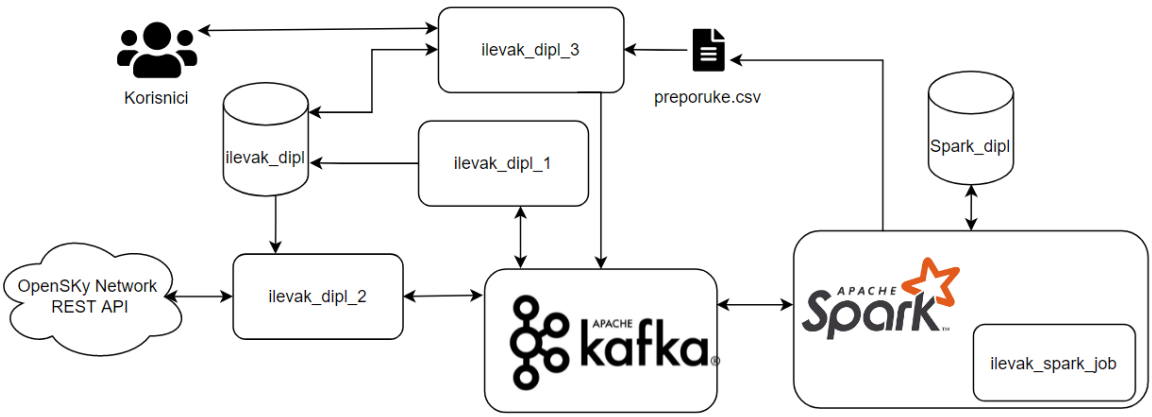
UserAirports - tablica koja opisuje korisničko praćenje aerodroma. Sastoji se od složenog primarnog ključa koji uključuje vanjski ključ *Username* referenciran na tablicu *Users* i *ICAO* vanjski ključ referenciran na tablicu *Airports*.

UserAirplanes - tablica opisuje korisničko praćenje aviona pa se stoga sastoji od složenog primarnog ključa koji uključuje vanjski ključ *Username* referenciran na tablicu *Users* i *ICAO24* vanjski ključ referenciran na tablicu *Airplanes*.

Osim navedene baze podataka, Spark posao koristi dodatnu bazu podataka u kojoj su spremljeni podaci o klikovima korisnika. Baza podataka se koristi kao spremište podataka za klikove i sastoji se od jedne tablice naziva *UserAirplanes*. Ova baza podataka nije dijeljena, odnosno koristi ju samo Spark posao. Umjesto baze podataka može se koristiti i Hadoop sustav spremanja podataka. Tablica sadrži složeni primarni ključ sastavljen od *UserId* odnosno identifikacije korisnika i *AirplaneId* odnosno identifikacija aviona. Osim primarnog ključa, tablica se sastoji od stupca *clicks* koji označava broj klikova korisnika za određeni avion.

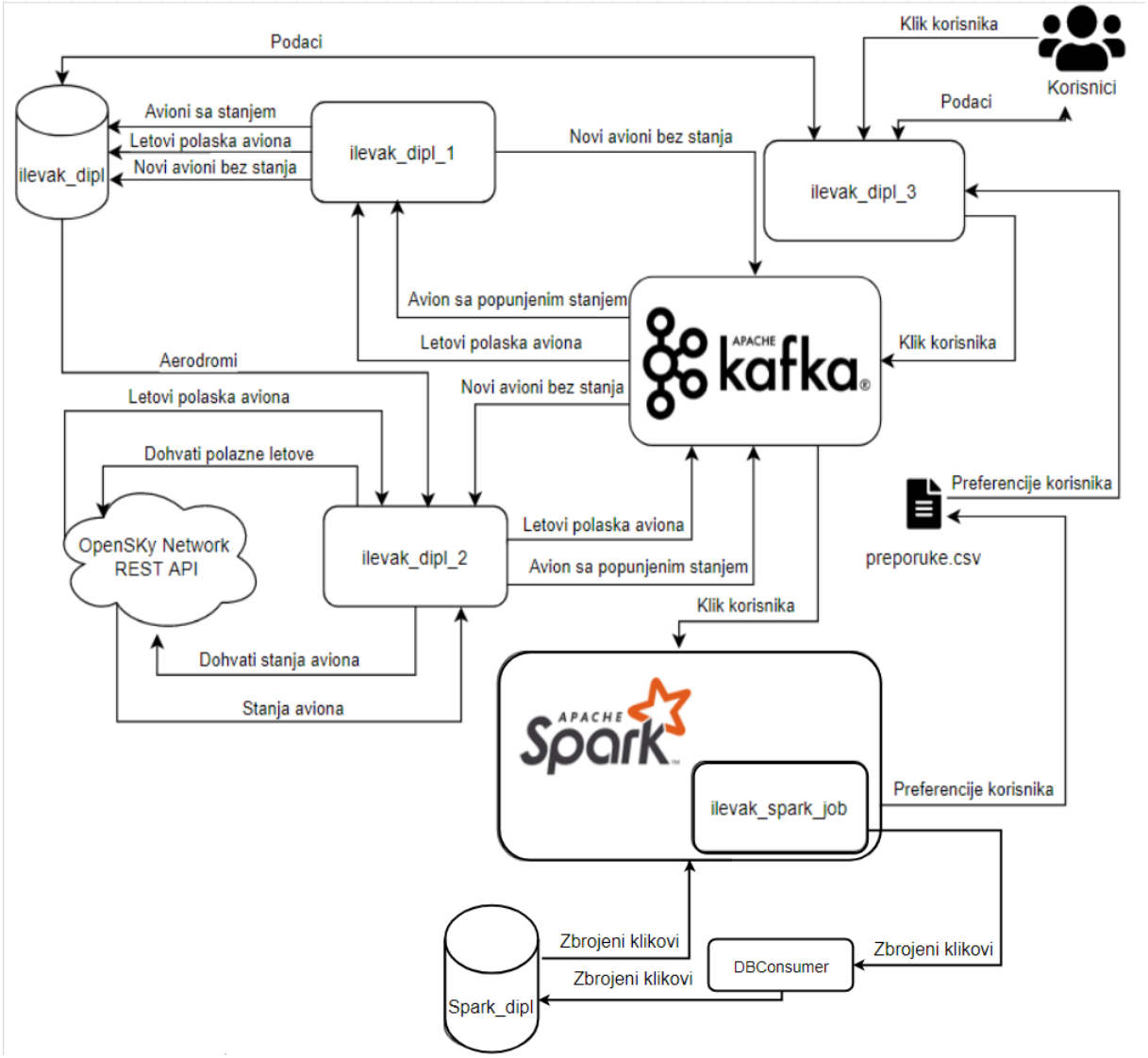
7.5. Arhitektura i rad projekta

Arhitektura aplikacije je temeljena na arhitekturi aplikacije na bazi strujanja podataka. Kao što je navedeno, arhitektura aplikacije na bazi strujanja podataka se sastoji od četiri komponente; posrednika poruka, ETL alata, analize podataka i spremišta podataka. Kao posrednik poruka izabran je Apache Kafka, za analizu podataka odabran je Apache Spark, kao spremište podataka odabrana je Java DB, a kao ETL alati su korišteni Kafka Streams i Apache Spark Streaming. Na idućoj slici nalazi se arhitektura projekta.



Slika 10: Arhitektura aplikacije (Izvor: vlastita izrada)

Projekt se sastoji od aplikacije „ilevak_dipl_2“ koja radi kao klijent za OpenSky Network REST servis i dohvaća podatke, aplikacije „ilevak_dipl_3“ koja pruža korisničko sučelje prema korisniku i aplikacije „ilevak_dipl_1“ koja sprema podatke o avionima i letovima unutar baze podataka. Također se sastoji od Kafka sustava poruka preko kojeg komunicira velik broj komponenti kao što je prikazano na slici, od programa koji se pokreće na Spark sustavu, baze podataka za Spark aplikaciju i baze podataka za glavni dio projekta. Zbog boljeg pojašnjenja samog rada projekta i komunikacije između komponenti u nastavku će biti opisan primjer komunikacije unutar projekta. Treba naglasiti da su komponente projekta neovisne jedna o drugoj te da se komunikacija odvija istovremeno i paralelno. Bez obzira na to najprije će se rad osvrnuti na komunikaciju komponenata kao o ovisnoj komunikaciji te definirati prividni redoslijed dijeljenja podataka i komunikacije radi boljeg razumijevanja međudjelovanja komponenti projekta. Na sljedećoj slici detaljnije je prikazana komunikacija i dijeljenje podataka između komponenata.



Slika 11: Shema aplikacije (Izvor: vlastita izrada)

Projekt se sastoji od tri web aplikacije pod nazivom „ilevak_dipl_1“, „ilevak_dipl_2“ i „ilevak_dipl_3“, no radi jednostavnosti nadalje u tekstu su nazvani redom kao prva aplikacija, druga aplikacija i treća aplikacija. Druga aplikacija u dretvi najprije dohvaća aerodrome iz baze podataka za koje je potrebno pronaći polazne letove. Nakon toga u pravilnim intervalima povlači podatke za svaki aerodrom o polascima aviona s REST servisa OpenSky Network za zadano razdoblje. Nakon što dobije podatke o letovima aviona šalje ih na Kafka temu. Prva aplikacija povlači podatke s Kafka teme o letovima aviona, transformira ih i filtrira te na samom kraju sprema u bazu podataka. Ako ne postoji avion za dani let unutar baze podataka, kreira se novi avion i šalju se podaci o novokreiranom avionu na Kafka temu. Druga aplikacija povlači podatke o novokreiranim avionima s Kafka teme te za dani avion povlači stanja preko REST servisa OpenSky Network, filtrira ih, uzima zadnje vremenski dostupan podatak, popunjava podatke o avionu i šalje na Kafka temu avion s popunjenim podacima. Prva aplikacija pri

primitku popunjenih podataka o avionu, sprema nove podatke unutar baze podataka. Stoga prva i druga aplikacija komuniciraju preko Kafka sustava poruka.

Treća aplikacija radi neovisno od prve dvije u smislu obrade podataka i dijeli s njima samo bazu podataka. Ona pruža korisničko sučelje i u interakciji s korisnicima, dohvaća podatke iz baze za pojedinog korisnika, ažurira podatke, kreira nove i slično. Na svaki klik korisnika povezanog s avionom; kao što je prikaz informacije o avionu, praćenje aviona ili prikaz letova aviona, treća aplikacija šalje podatak o avionu i korisniku na Kafka temu. Spark posao koji je postavljen na Spark sustavu čeka na poruku s Kafka teme. Ako dođe do poruke o nekom kliku, on izračunava broj novih klikova te šalje podatke na Kafka temu. Potrošač Kafka teme nadodaje izračunati broj na spremljeni broj klikova iz baze. Ako dođe do zadanog kontrolnog broja klikova korisnika, Spark posao izračunava moguće ocjene za korisnike i proizvode preko ALS algoritma te ih sprema u datoteku zadanog naziva ili putanje. Ako korisnik odabere „moje preporuke“, treća aplikacija čita podatke iz datoteke s mogućim ocjenama korisnika, filtrira ih i prikazuje korisniku one avione koje bi mogao sljedeće zapratiti.

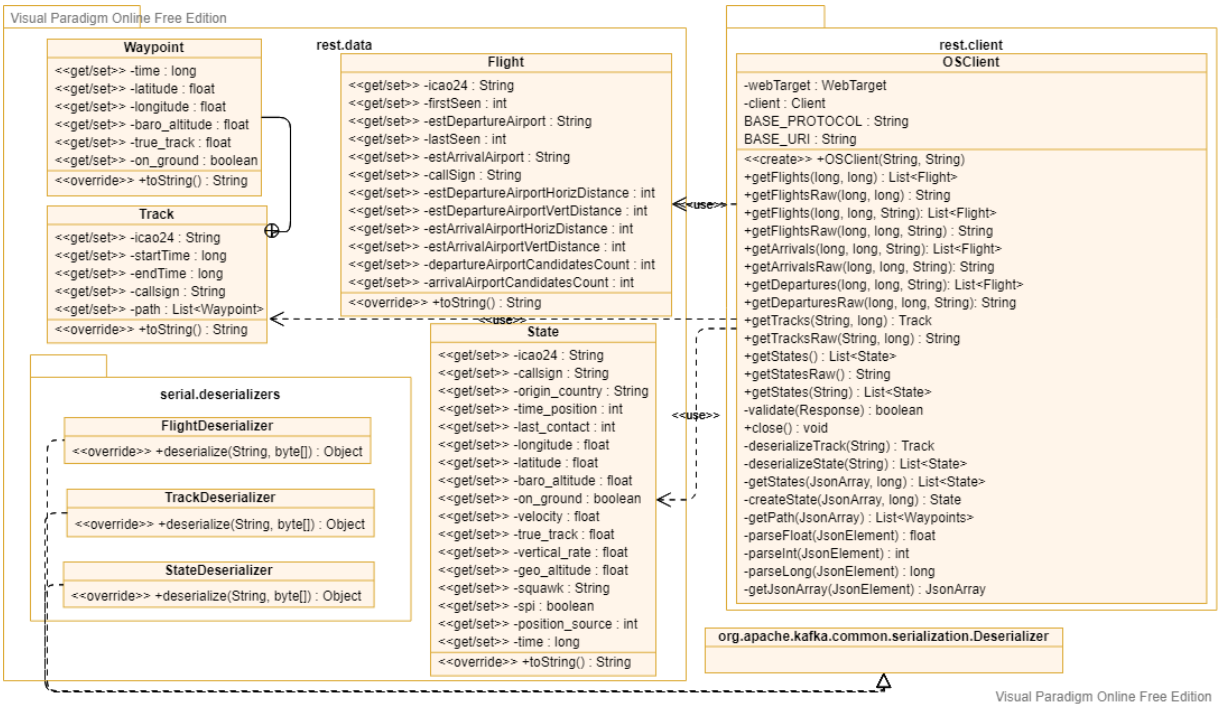
U nastavku bit će objašnjen sadržaj projekta i izrada dijelova projekta povezanih sa strujanjem podataka.

7.6. Izrada aplikacija

Pod pretpostavkom da je već kreirana baza podataka u ovom odjeljku objašnjeno je kreiranje projekta i dodatnih biblioteka. Najprije su proučene kreirane biblioteke koje služe za učitavanje konfiguracijskih datoteka, kreiranje Kafka proizvođača i potrošača, postavljanje Kafka konfiguracija sustava poruka i izradu REST klijenta za OpenSky Network REST servis. Nakon pojašnjenja biblioteka objašnjena je izrada samih aplikacija.

7.6.1. Biblioteke

Biblioteka REST klijenta za OpenSky Network naziva „ilevak_OSNetwork“ se sastoji od klasa koji odgovaraju dohvaćenim JSON podacima s OpenSky Network servera i od klase REST klijenta za povezivanje i dohvaćanje podataka. Podaci koji se dohvaćaju *GET* metodama s OpenSky Networka su dani u njihovoj dokumentaciji. Na sljedećoj slici može se vidjeti dijagram klasa biblioteke. Unutar dijagrama dodano je svojstvo *<<get/set>>* izvan konvencionalnog kreiranja UML dijagrama klasa, a koje označava postojanje *get* i *set* metoda za dane atribute. S druge strane, svojstvo *<<override>>* označava metodu koja je nadjačana. Konstruktori nisu prikazani, osim specijalnih konstruktora kao kod klase *OSClient*.



Slika 12: UML dijagram klasa "ilevak_OSNetwork" (Izvor: vlastita izrada)

Biblioteka se sastoji kao što se može primijetiti od opisnih klasa *Waypoint*, *Track*, *Flight* i *State*. Te klase su korištene pri dohvaćanju podataka i pretvaranju JSON formata podataka u objekte. Klasa *OSClient* sadrži osnovne metode za dohvaćanje podataka sa servisa OpenSky Network-a (kraće OS Network), a koji su letovi aviona, polazni letovi aerodroma, dolazni letovi aerodroma, stanja aviona i putanje aviona. Klasa omogućuje dohvaćanje podataka u obliku prikladnih objekata ili u obliku *String* objekta. Uz to klasa se sastoji i od deserijalizatora klasa podataka za Apache Kafka, kako bi se mogli slati podaci u obliku zadanog objekta čime nije potrebno ponovo transformirati ili prikupljati podatke.

U fokusu je kreiranje klase *OSClient* s obzirom na njenu korisnost unutar projekta. Klasa *OSClient* sadržava konstruktor koji prima ulazne podatke korisničkog imena i lozinke kako bi se mogao kreirati URI za spajanje na OS Network. Naime za dohvaćanje stanja aviona preko REST servisa OS Network-a nije potrebna lozinka i korisničko ime, no za ostale vrijednosti je potrebna i oni se navode unutar URL-a kao što je dano u sljedećem primjeru:

*„https://{**korisničko_ime**}:{**lozinka**}@opensky-network.org/api/states/all“*

Stoga se pri pozivanju korisnika konačno kreira URL kao što se može vidjeti u sljedećem primjeru koda.

```
private final WebTarget webTarget;
private final Client client;
private static final String BASE_PROTOCOL = "https://";
private static final String BASE_URI = "opensky-network.org/api/";

public OSClient(String username, String password) {
    client = ClientBuilder.newClient();
    String url=BASE_PROTOCOL + username + ":" + password
        + "@" + BASE_URI;
    webTarget = client.target(url);
}
```

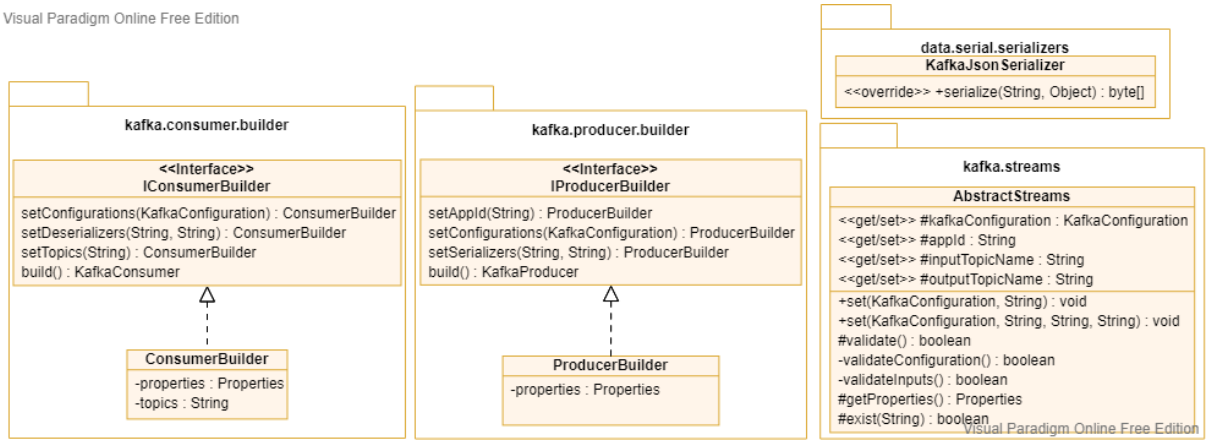
Također unutar dokumentacije su opisane sve *GET* metode za dohvaćanje podataka. Kako se metode razlikuju po parametrima prikazana je jedna metoda koja se poziva unutar samog projekta, a čiji je naziv *getDepartures()*. Dohvaćanje polaznih letova s aerodroma zahtijeva određivanje atributa upita *begin* i *end* kao početnog i krajnjeg vremena od i do kojeg će se dohvaćati podaci o letovima, te od atributa upita *airport* koji označava ICAO kod aerodroma. Uz navedene atribute potrebno je nadodati na originalni URL, put „*../flights/departure/..*“. Podaci koji su dohvaćeni su u JSON formatu pa se tako i sami podaci koriste, odnosno transformiraju u listu objekata.

```
public List<Flight> getDepartures(long begin, long end, String airport)
        throws ClientErrorException, ClientConnectionException {
    WebTarget resource = webTarget;
    resource = resource.path("flights").path("departure");
    resource = resource.queryParam("begin", begin)
        .queryParam("end", end)
        .queryParam("airport", airport);
    Response response =
        resource.request(MediaType.APPLICATION_JSON).get();
    String jsonString = response.readEntity((String.class));
    return validate(response) ?
        Arrays.asList(
            new Gson().fromJson(jsonString, Flight[].class)
        ) : null;
}
```

Unutar projekta se na više mjesta koriste postavke konfiguracijske datoteke te je zbog toga kreirana biblioteka „ilevak_configurationsDBKafka“ koja sadržava klase za čitanje konfiguracijske tekstualne ili XML datoteke. Kako se postavke unutar konfiguracijskih datoteka najčešće odnose na postavke Kafka potrošača, proizvođača i Kafka Streams ili na postavke baze podataka, kreirane su dvije direktne klase *KafkaConfiguration* i *DBConfiguration* koje sadržavaju metode dohvaćanja podataka postavka iz datoteke tipične za njih. Tako primjerice klasa *KafkaConfiguration* sadržava metode za dohvaćanje identifikacije sustava poruka, identifikacije grupe, najvećeg broja povlačenja podataka i sl.

Sljedeća zanimljiva biblioteka koja je kreirana u svrhu projekta je biblioteka „ilevak_kafka“ koja sadržava klase za kreiranje proizvođača i potrošača na temelju uzorka

Builder. Potrošač i proizvođač u ovom projektu kreirani su uvijek na isti način te je u svrhu smanjivanja dupliciranog koda kreiran primjeren uzorak izgradnje. Svaki od njih sadržava metode za postavljanje postavaka, metode za postavljanje deserijalizatora ili serijalizatora vrijednosti i ključeva, te ovisno o vrsti komponente postavljanje Kafka teme. Poslane postavke se iščitavaju iz datoteke, odnosno objekta *KafkaConfiguration*. Može se postaviti niz postavka potrošaču i proizvođaču, no s obzirom na to da Kafka definira zadane vrijednosti za većinu postavka, potrebno je postaviti one koje su nam od bitnog značaja unutar aplikacije. Jako je bitno da se definiraju postavke za deserijalizatore i serijalizatore vrijednosti i ključeva poruka, broj količine povlačenja poruka odjednom, identifikacija grupe, identifikacija servera te u slučaju potrošača Kafka teme. Vrlo se često mijenjaju i postavke omogućavanja automatskog izvršavanja i automatsko ponovno namještanje pomaka. Stoga se unutar samih klasa postavljaju navedene postavke. Osim klasa potrošača i proizvođača, kreirana je apstraktna klasa za kreiranje Kafka Streams koja definira zajedničke metode i varijable. Dijagram klasa ove biblioteke može se vidjeti na sljedećoj slici.

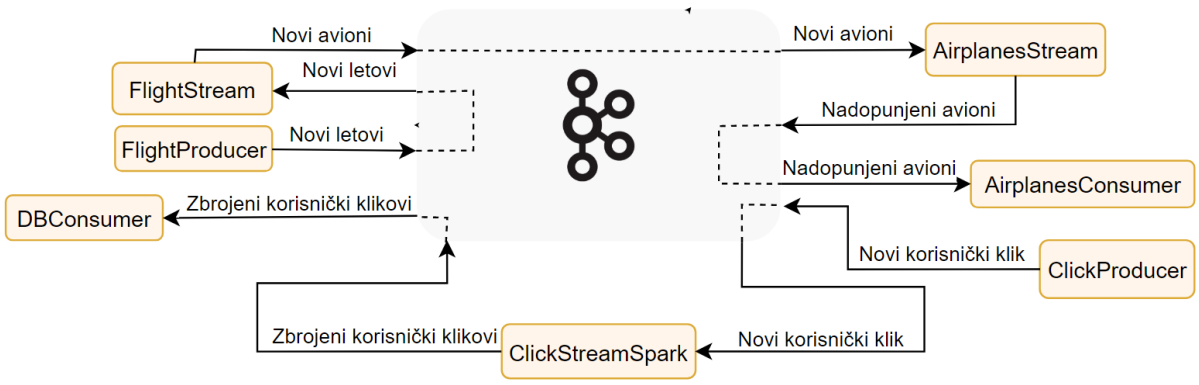


Slika 13: UML dijagram klasa biblioteke "ilevak_kafka" (Izvor: vlastita izrada)

Osim već navedenih klasa, unutar biblioteke nalazi se i serijalizator podataka za projekt. S obzirom na to da deserijalizatori podataka pretvaraju niz bajtova u određeni objekt potreban je jedan serijalizator koji pretvara objekte u niz bajtova. Pored toga mogu se primijetiti učahurene metode unutar klase *AbstractStreams*, a koje služe za provjeru vrijednosti i dohvaćanja postavljenih postavki Kafka Streams.

7.6.2. Postavke Kafka servera

Nakon što su instalirani Apache Kafka i Zookeper i postavljene potrebne konfiguracije za pokretanje, može se pokrenuti server Zookeper, a nakon toga i Apache Kafka. Ako se programi pokreću preko Windows sustava potrebno se pozicionirati na direktorij *„../bin/windows“* unutar Kafka direktorija i pozivati *„.bat“* datoteke. Nakon pokretanja servera moguće je otvoriti novi konzolni prozor i kreirati i definirati Kafka teme, no prije pokretanja aplikacije bitno je odrediti koje teme su potrebne i koje postavke im odrediti. Na temelju slike 11 može se prepoznati koliko tema je potrebno da bi kreirali aplikaciju, no sam način izrade i implementacije aplikacije diktira i broj tema. Stoga je najprije analizirana komunikacija komponenti putem Apache Kafka u napravljenom projektu. Na sljedećoj slici prikazan je prijenos podataka na temelju vrste komponenta unutar cijelog sustava kao što su proizvođač, potrošač, Kafka Streams i Apache Spark Streaming.



Slika 14: Prikaz komunikacije komponenti putem Apache Kafka (Izvor: vlastita izrada)

Unutar dijagrama prikazana je komunikacija komponenti pri čemu pune strelice prikazuju slanje ili zaprimanje poruka, isprekidane crte prikazuju poveznicu između točno određenih slanja i zaprimanja poruka, tekst nad crtama označava podatke koji se šalju, a u sredini sustava je Kafka sustav poruka. Može se primijetiti da postoji pet poveznica odnosno trebat će nam minimalno pet tema na Kafka sustavu poruka. S pretpostavkom da je pokrenut Kafka server i Zookeper server, kreiranje tema na Kafka sustavu poruka se izvršava sljedećom linijom:

```
kafka-topics.bat --create --topic {naziv_teme} --bootstrap-server
localhost:9092
```

Dodatno se sama tema može opisati kod kreiranja ili nakon kreiranja ažuriranjem kao što je prikazano u sljedećoj liniji:

```
kafka-topics.bat --bootstrap-server localhost:9092 --alter --topic
{naziv_teme} –partitions 4
```

Sljedeće teme su kreirane u svrhu projekta:

- *rawDataFlights,*
- *rawDataAirplanes,*
- *streamedDataAirplanes,*
- *rawDataClicks,*
- *DBDataClicks i*
- *streamedDataFlights*.

Svakoj od temi pridruženo je četiri particija. Kao što je objašnjeno ranije, particije su tijekom izvršavanja automatski pridružene Kafka potrošaču. Time nije potrebno ništa dodatno definirati, no pažnja se treba usmjeriti na čitanje podataka s Kafka sustava poruka. Kreiranjem tema obavljene su potrebne postavke za korištenje Kafka sustava poruka stoga je idući odjeljak posvećen izradi potrošača, proizvođača i Kafka Streams.

7.6.3. Izrada proizvođača, potrošača i Kafka Streams

Apache Kafka nudi biblioteke za rad s potrošačem, proizvođačem i Kafka Streams unutar Java aplikacije kao što su biblioteke „kafka-clients“ i „kafka-streams“. Korištenje metoda i objekata iz biblioteka je intuitivno, a kreirani potrošači i proizvođači imaju uglavnom isti način rada.

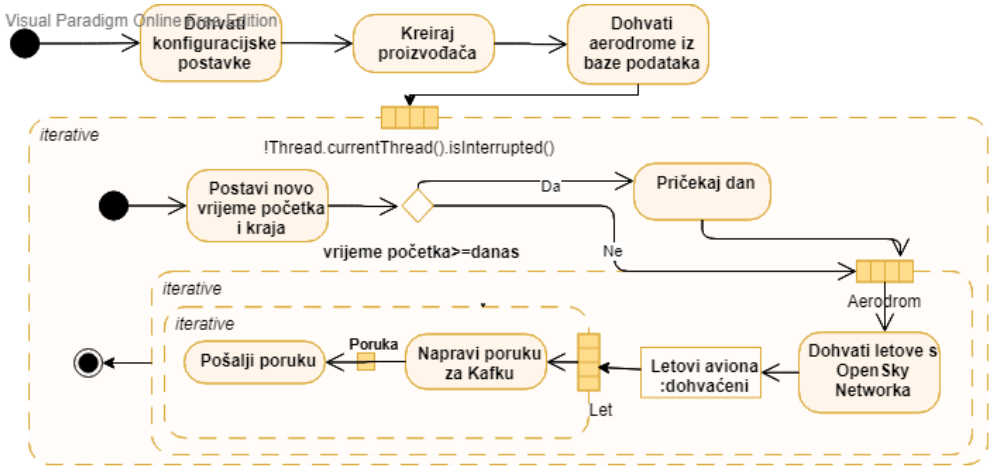
7.6.3.1. Proizvođač

Glavna uloga proizvođača unutar sustava poruka je proizvodnja poruka za sustav. Sama izrada i vrsta poruka se razlikuje od proizvođača do proizvođača, no slanje poruka je sadržajno slično svakom proizvođaču. Prilikom slanja poruka na Kafka sustav potrebno je navesti temu na koju se šalje određeni podatak. Podaci koji se šalju mogu se sastojati od ključa i vrijednosti ili se mogu sastojati samo od vrijednosti. Slanje podataka se odvija korištenjem izrađenog proizvođača i klase *ProducerRecord*. Objekti proizvođač i *ProducerRecord* moraju sadržavati iste tipove ključa i vrijednosti. U sljedećem kodu nalazi se prikaz metode za slanje poruka na Kafka sustav pod pretpostavkom da je kreiran proizvođač.

```
private void sendToKafka(List<Flight> flights) {
    flights.forEach(f -> {
        ProducerRecord<String, Flight> record
            = new ProducerRecord<>(
                "rawDataFlights",
                f.getIcao24(),
                f
            );
        producer.send(record);
    });
}
```

U navedenom kodu kreira se za svaki objekt u listi *ProducerRecord* pri čemu se šalje podatak na određenu temu *rawDataFlights* s ključem i vrijednosti. Pored metode za slanje svi ostali podaci su kreirani poželjnim putem.

Sada kada je objašnjeno slanje poruka na Kafka sustav može se proučiti izrada projekta. Unutar projekta nalaze se tri proizvođača. Na idućoj slici nalazi se UML dijagram aktivnosti koji predstavlja rad dretve *FlightProducer*.



Slika 15: UML dijagram aktivnosti dretve "FlightProducer" (Izvor: vlastita izrada)

FlightProducer zapravo je dretva koja unutar *start()* metode priprema podatke za pokretanje *run()* metode. Metoda *run()* sadržava petlju koja je aktivna dokle god je dretva nije prekinuta. Unutar svakog ciklusa postavljaju se nove vrijednosti vremena početka i kraja, a koje su nastale uvećavanjem starih vrijednosti za jedan dan. Ako su nove vrijednosti vremena jednake današnjem vremenu, ili su veće od momentalnog vremena, dretva spava jedan dan. Nakon toga dohvaćaju se podaci o letovima za sve aerodrome iz baze podataka u intervalu od postavljenih vremena početka i kraja. Za svaki dohvaćeni let kreira se poruka za Kafku i šalje se.

Postoji proizvođač koji reagira na događaj, odnosno pritisak korisnika na bilo koji gumb korisničkog sučelja koji je povezan s avionom. Pri tome se šalju podaci s korisničkom identifikacijom i identifikacijom aviona na temu. Slučajeve slanja poruke na Kafka temu na klik korisnika može se iščitati iz slike 8.

Zadnji proizvođač unutar projekta odnosi se na slanje novih aerodroma u bazi podataka. Pri zapisu letova aviona, ako se zapisuje let za avion koji ne postoji u bazi podataka, on će se zapisati s identifikacijom *ICAO24*, a ostale vrijednosti će ostati prazne. Zatim se proizvođačem šalje poruka na Kafku s *ICAO24* aviona, što signalizira da su podaci od tog aviona prazni te da ih treba popuniti.

7.6.3.2. Potrošač

Kod izrade potrošača treba posebnu pažnju obratiti na asinkrono ili sinkrono slanje pomaka, maksimalnu količinu podataka jednog povlačenja i na postojanje više particija. Naime struktura koda ovisi o samom asinkronom i sinkronom slanju pomaka. Pomak se ne treba nužno slati, no tada će se pri svakom novom kreiranju potrošača koji čita s određene teme svaki puta dohvaćati sve poruke poslane na Kafka temu ako nisu izbrisani od strane Kafka sustava poruka. Kod asinkronog i sinkronog slanja pomaka, potrebno je paziti kada se pomak šalje. Unutar projekta se nalazi jedan potrošač poruka. Taj potrošač prima popunjene podatke aviona te ih sprema u bazu podataka. Konfiguriranje podataka kao što je postavljanje teme potrošača, maksimalne količine podataka jednog povlačenja ili postavljanja automatskog slanja pomaka se određuje konfiguracijskom datotekom. Navedene postavke se pridružuju pri kreiranju potrošača putem uzroka Builder iz biblioteke „ilevak_kafka“.

U idućem kodu može se vidjeti povlačenje podataka s Kafka teme.

```
public void runConsumer() {
    List<ConsumerRecord<String, Airplanes>> partitionRecords;
    Airplanes airplane;
    ConsumerRecords<String, Airplanes> records =
        consumer.poll(Duration.ofMillis(10000));
    for (TopicPartition partition : records.partitions()) {
        partitionRecords = records.records(partition);
        for (ConsumerRecord<String, Airplanes> record : partitionRecords) {
            if (record.value() != null) {
                String icao24=record.value().getIcao24().trim();
                airplane = findAirplaneIfExists(icao24);
                if (airplane != null) {
                    editAirplane(airplane, record);
                } else {
                    createAirplane(record);
                }
            }
        }
        commitOffset(partitionRecords, partition);
    }
}
```

Povlačenje podataka se odvija korištenjem metode *poll()*. Metoda *poll()* prima parametar koji određuje vrijeme povlačenja podataka odnosno vrijeme u kojem će poruke ako stignu biti povučene. Svaka poruka koja stigne poslije 10 sekundi će se obraditi u sljedećoj iteraciji ili pri sljedećem pozivu metode. S obzirom na to da se teme mogu sastojati od particija, potrebno je dohvatiti particije te prolazeći kroz particije dohvaćati podatke. Na samom kraju prije svakog novog čitanja particije šalje se pomak. Slanje pomaka može se vidjeti u idućem dijelu koda.

```
private void commitOffset(
        List<ConsumerRecord<String, Airplanes>> partitionRecords,
        TopicPartition partition)
{
    long lastOffset = partitionRecords
        .get(partitionRecords.size() - 1)
        .offset();
    consumer
        .commitSync(
            Collections
                .singletonMap(
                    partition,
                    new OffsetAndMetadata(lastOffset + 1)
                )
            );
}
```

Kao što se može primijetiti unutar koda se od dohvaćenih podataka uzima onaj zadnji te se izračunava njegov pomak. Nakon toga se sinkronom metodom *commitSync()* pomak šalje na Kafka sustav.

Ovaj potrošač je kreiran u svrhu konzumiranja poruka s teme *streamedDataAirplanes*. Pri primitku poruke u obliku objekta klase *Airplanes*, obrađuje ih i sprema vrijednosti unutar baze podataka.

7.6.3.3. Kafka Streams

Kafka Streams je klijentska biblioteka za izradu aplikacije pri čemu su ulazni i izlazni podaci spremljeni u Kafka sustav [8]. Kreiranje Kafka Streams objekata je jednostavno, a samo korištenje lako. Pri definiranju Kafka Streams veliku je pozornost potrebno obratiti na serijalizatore i deserijalizatore ključeva i vrijednosti. U svrhu projekta definirana su dva slučaja korištenja Kafka Streams; *AirplanesStream* i *FlightStream*. Klasa *FlightStream* služi za dohvaćanje podataka o letovima s Kafka sustava poruka i spremanje u bazu podataka dok s druge strane *AirplanesStream* služi za dohvaćanje poruka o novokreiranim avionima unutar baze podataka, dohvaćanje stanja aviona s REST servisa OpenSky Networka i slanje popunjenih podataka aviona.

Kako je već navedeno, *FlightStream* prikuplja podatke iz Kafka teme, transformira ih u objekt za bazu podataka, te sprema podatke o letovima u bazu podataka. Isto se može vidjeti u sljedećem dijelu koda.

```
KStream<String, Flight> inputStream = builder
    .stream(inputTopicName, Consumed.with(Serdes.String(), flightSerde))
    .filter((k, v) -> validateFlight(v));
KStream<String, Flights> out = inputStream
    .mapValues(v -> transform(v))
    .filter((k, v) -> v != null);
out.foreach((key, value) -> {
    saveFlight(value);
});
```

AirplanesStream za svaki primljeni podatak aviona, povlači podatke o stanjima aviona s OpenSky Network-a. Od podataka koje dobije, povlači vremenski najkasniji te kreira objekt klase *Airplanes* s popunjenim varijablama. Nakon toga šalje dalje na Kafka temu objekt *Airplanes*. Ovo ponašanje može se vidjeti u idućem kodu.

```
KStream<String, String> inputStream = builder
    .stream(
        inputTopicName,
        Consumed.with(Serdes.String(), Serdes.String())
    );
KStream<String, Airplanes> out = inputStream
    .mapValues(value ->  findValues(value));
out.to(outputTopicName, Produced.with(Serdes.String(), airplanesSerde));
```

U sljedećem poglavlju bit će opisana izrada Spark posla.

7.6.4. Izrada Spark posla

Na slici 10 vidi se povezanost Spark posla sa sustavom. Unutar projekta je korišten Spark u dva slučaja. U suštini prvi zadatak napravljenog Spark posla je primanje tokova podataka iz Kafka sustava poruka te ažuriranje i spremanje podataka u bazu podataka. Za razliku od već napravljenih Kafka Streams, Spark Streaming koristi drugačiju strukturu te koristi RDD strukturu podataka. Kako bi shvatili rad napravljenog Spark Streaming, bitno je naglasiti da Kafka proizvođač pri kliku korisnika na gumb povezan s avionom šalje poruke na Kafka sustav u sljedećem obliku „*{Id_korisnika},{id_aviona}*“.

Spark Streaming po primitku poruke s Kafka sustava poruka, povlači podatke iz baze podataka, te strukturira podatke tako da se mogu zbrojiti kasnije. Najprije je potrebno postaviti postavke za spajanje na Kafka sustav; *SparkSession* i *ConsumerStrategy*. Nakon toga može se kreirati *JavaInputStream* kojim se opisuje ponašanje nad porukama sustava. Kreirani Spark Streaming po primitku poruka prebrojava zapise istih vrijednosti te ih šalje na Kafka temu kako bi se mogli spremiti u bazu podataka.

Najprije se podaci pristigli s Kafka sustava prebrojavaju po istom ključu.

```
JavaPairRDD<String, Long> list = t.mapToPair(
    new PairFunction<ConsumerRecord<String, String>, String, Long>() {
        @Override
        public Tuple2<String, Long> call(
          ConsumerRecord<String, String> t
          ) throws Exception {
            return new Tuple2<>(t.value(), (long) 1);
        }
        })
        .reduceByKey((Long t1, Long t2) -> t1 + t2);
```

Nakon toga se podaci pripremaju za slanje na Kafka temu. Ovo ponašanje je prikazano u idućem kodu.

```
calculatedData = list.map((Tuple2<String, Long> t1) -> {
    String data[] = t1._1.split(",");
    return data[0].trim() + ";" + data[1].trim() + ";" + t1._2;
});
```

Na samom kraju potrebno je poslati podatke na Kafka temu koja sprema podatke u bazu podataka, što je ostvareno putem idućeg programskog koda.

```
calculatedData.foreach(new VoidFunction<String>() {
    @Override
    public void call(String t) throws Exception {
        Properties properties = new Properties();
        properties.put(ProducerConfig.BOOTSTRAP_SERVERS_CONFIG,
"localhost:9092");
        properties.put(ProducerConfig.KEY_SERIALIZER_CLASS_CONFIG,
StringSerializer.class.getName());
        properties.put(ProducerConfig.VALUE_SERIALIZER_CLASS_CONFIG,
StringSerializer.class.getName());
        Producer<String, String> producer = new
KafkaProducer<>(properties);
        ProducerRecord<String, String> record = new
ProducerRecord<>(topicProducer, t);
        producer.send(record);
        producer.close();
    }
});
```

Ako dođe određeni broj poruka na Spark Streaming, okida se metoda za izračunavanje preporuka koja je objašnjena u sljedećem poglavlju.

7.6.4.1. ALS algoritam

Metoda alternirajućih najmanjih kvadrata (eng. *Alternating Least Squares - ALS*) je metoda za rješavanje problema faktorizacije matrice. To je algoritam koji faktorizira matricu

M na dva faktora; matrice K i P tako da je $R \approx K^T P$, odnosno na dva faktora pri čemu se množenjem transponirane matrice Ks matricom P dobivaju predviđene vrijednosti matrice R [3]. Kako se metoda alternirajućih najmanjih kvadrata koristi u okviru preporuka, nazivat će se matrica K kao matrica korisnika, i matrica P kao matrica proizvoda pa je prema tome matrica R, matrica ocjena.

Neka je k_i i-ti stupac matrice korisnika K, p_j j-ti stupac matrice proizvoda P , $r_{i,j}$ član matrice R i λ regulacijski faktor . Matricu R može se faktorizirati na K i P tako da vrijedi $R \approx K^T P$ pomoću formule [3]:

$$\min_{K,P} \sum_{\{i,j|r_{i,j}\neq 0\}} (r_{i,j} - k_i^T p_j)^2 + \lambda \left(\sum_i n_{k_i} \|k_i\|^2 + \sum_j n_{p_j} \|p_j\|^2 \right)$$

, pri čemu je n_{k_i} broj proizvoda koje je korisnik i ocijenio, a n_{p_j} broj koji opisuje koliko je puta proizvod j ocjenjen.

Ovaj problem je NP-problem, odnosno problem za koje je moguće definirati nedeterminističke Turingove strojeve koji ih rješavaju u polinomnom vremenu [50]. Takav stroj bi trebao isprobavati sve mogućnosti pa bi broj koraka narastao barem eksponencijalno [50]. Ovakav pristup bi rezultirao sporim iteracija koje troše puno resursa. No ako se postavi set varijabli K koje se tretiraju kao konstante, dobiva se konveksna funkciju od P i obratno, pa je problem reduciran na linearnu regresiju [39]. Zbog toga ALS algoritam fiksira matricu P i optimizira matricu K i obratno sve do dobivanja konvergencije. Konvergencija je približavanje niza matematičkih objekata, najčešće brojeva, određenoj vrijednosti. Kaže se da niz a_n konvergira k A ako ima konačnu graničnu vrijednost A. U našem slučaju konvergencija bi se odnosila na kompletnu ispunjenost matrice R s prijašnjim nepoznatim vrijednostima.

Neka je λ regulacijski faktor, K matrica korisnika i P matrica proizvoda, tada je k_i i-ti stupac matrice korisnika K, p_j j-ti stupac matrice proizvoda P i $r_{i,j}$ član matrice R tipa $i \times j$. Tako sljedeći koraci definiraju ALS algoritam za potpunu matricu R:

Tablica 1. Koraci ALS algoritma (Izvor: vlastita izrada, prema: [39])

Inicijaliziraj $\boldsymbol{K, P}$
Ponavljaj do konvergencije:
$\forall \boldsymbol{k_i} \in K \mid i > 0$ radi **:** $$\boldsymbol{k_i} = \left(\sum_{\boldsymbol{r_{ij}} \in \boldsymbol{r_{i*}}} \boldsymbol{p_j p_j^T} + \boldsymbol{\lambda I_k}\right)^{-1} \sum_{\boldsymbol{r_{ij}} \in \boldsymbol{r_{i*}}} \boldsymbol{r_{ij} p_j}$$
$\forall \boldsymbol{p_j} \in P \mid j > 0$ radi: $$\boldsymbol{p_j} = \left(\sum_{\boldsymbol{r_{ij}} \in \boldsymbol{r_{*j}}} \boldsymbol{k_i k_i^T} + \boldsymbol{\lambda I_k}\right)^{-1} \sum_{\boldsymbol{r_{ij}} \in \boldsymbol{r_{*j}}} \boldsymbol{r_{ij} k_i}$$

Korištenje ALS algoritma uz pomoć dane biblioteke Apache Spark je intuitivno i lako. Potrebno je jedino odrediti skup objekata *Rating* koji se sastoje od korisničke identifikacije tipa *Integer*, identifikacije proizvoda tipa *Integer* i ocjene tipa *Double*. Nakon toga pozivom statičke metode *train()* klase *ALS*, treba proslijediti *RDD* napravljenih objekata *Rating*, broj latentnih faktora, broj iteracija i regulacijski faktor. Regulacijski faktori se odnose na faktore poput spola, godina osobe i slično. Unutar Apache Spark dokumentacije navodi se da je dovoljno maksimalno 20 iteracija kako bi model izračunao preporuke [14]. Nakon toga se poziva nad modelom metoda *predict()* koja prima popis proizvoda i korisnika kojim treba predvidjeti vrijednosti. Navedeno se može vidjeti u sljedećem kodu.

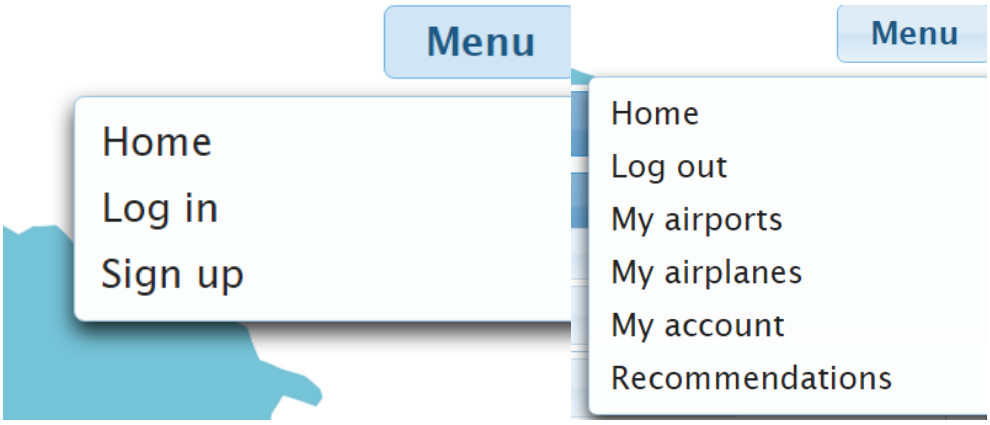
```
int rank = 10;
int numIterations = 20;
double lambda = 0.01;
MatrixFactorizationModel model = ALS
    .train(JavaRDD.toRDD(ratings), rank, numIterations, lambda);
JavaRDD<Integer> users = ratings.map((Rating t1) -> t1.user());
JavaRDD<Integer> products = ratings.map((Rating t1) -> t1.product());
JavaRDD<Tuple2<Object, Object>> userProducts
     = users
        .cartesian(products)
        .distinct()
        .map(
            (Tuple2<Integer, Integer> t1) -> new Tuple2<>(t1._1, t1._2)
        );
JavaRDD<Rating> userRatings
    = model
        .predict(userProducts.rdd())
        .toJavaRDD()
        .sortBy(
            Rating::rating, false, userProducts.partitions().size()
        );
```

Može se primijetiti da se u kodu nakon izračunavanja modela faktorizacije matrica kreiraju dva *RDD*-a *users* i *products*. U svakom od njih nalaze se zasebno sve vrijednosti identifikacije korisnika ili proizvoda. Nakon toga kreira se *RDD* koji je kartezijev produkt ta dva *RDD*-a kako bi mogli dobiti očekivanja ocjena za svaku kombinaciju korisnika i proizvoda. Na kraju se putem modela izračunavanju predviđene vrijednosti ocjena metodom *predict()*.

7.7. Primjeri korištenja korisničke aplikacije

U ovom poglavlju bit će navedeni neki primjeri korištenja same aplikacije. Aplikacija se sastoji od sedam pogleda i u nastavku će biti objašnjen svaki od njih. Najprije kada se pokrene aplikacija otvara se početna stranica putem koje se dolazi do ostalih stranica. Korisnik unutar

izbornika može doći do ostalih stranica. Pri pokretanju aplikacije moguće je pristupiti jedino registraciji, prijavi i početnoj stranici. Neprijavljenom korisniku se ne prikazuju drugi izbori unutar izbornika.

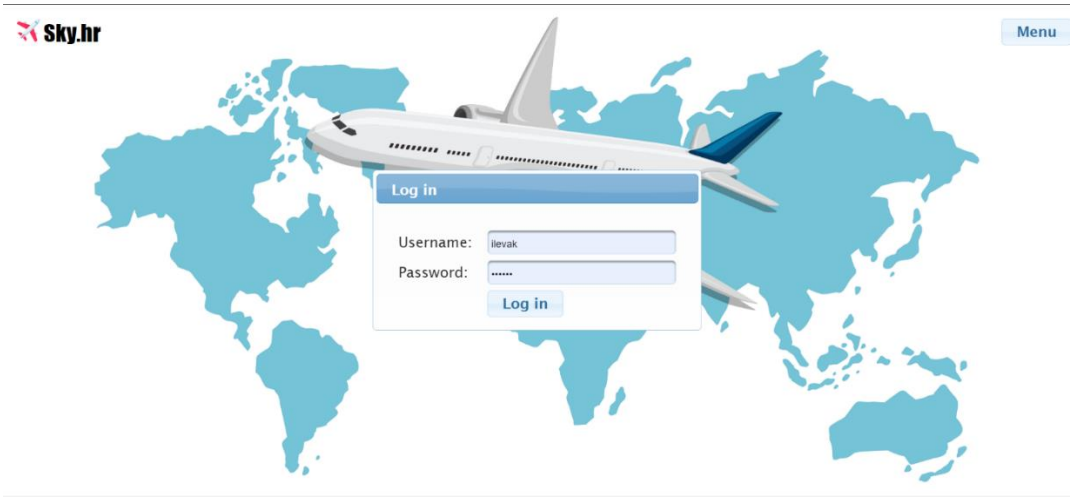


Slika 16: Prikaz izbornika neprijavljenog korisnika (lijevo) i prijavljenog korisnika(desno) (Izvor: vlastita izrada)

Upisom druge adrese, preusmjerit će ga se na samu prijavu. Ako korisnik nije registriran može se registrirati (slika 17), ako je registriran može se prijaviti (slika 18), a ako je prijavljen može se odjaviti.

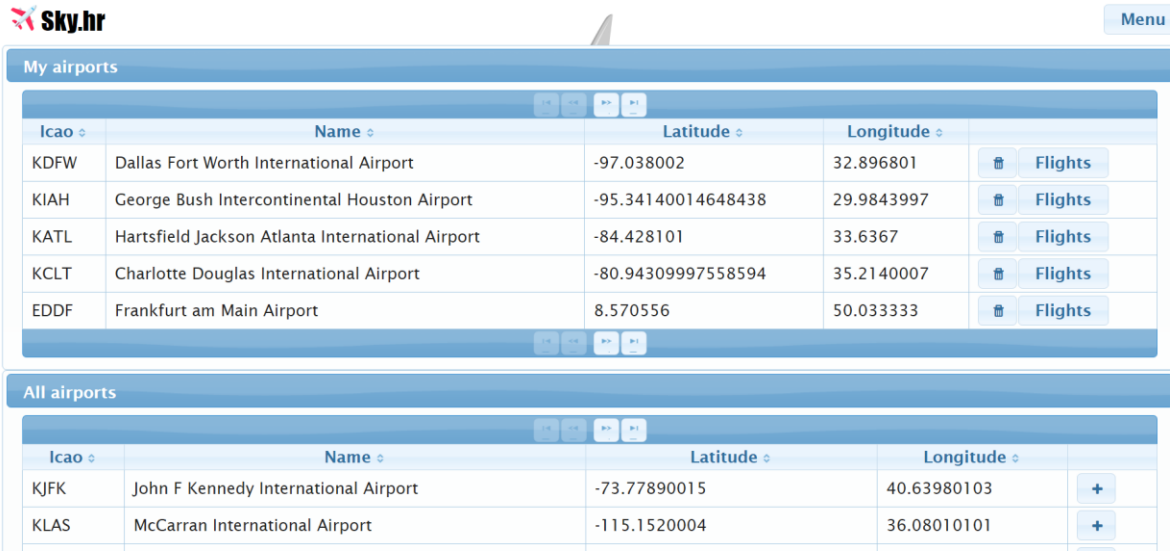
Slika 17: Stranica za registraciju (Izvor: vlastita izrada)

Uspješnom registracijom korisnika otvara se prikaz za prijavu korisnika (slika 18).



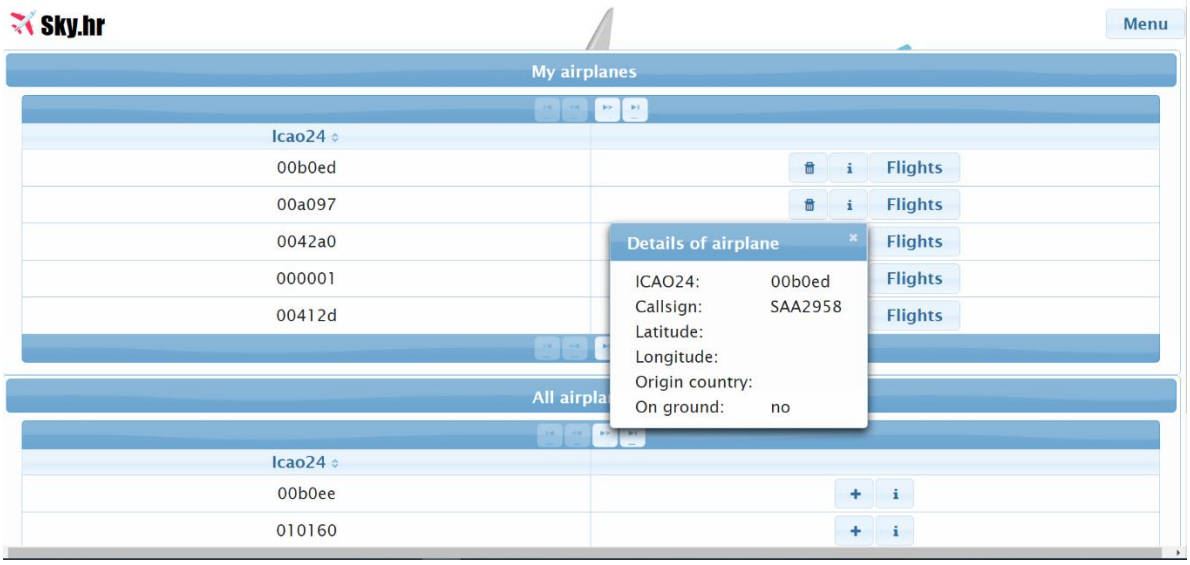
Slika 18: Stranica za prijavu (Izvor: vlastita izrada)

Prijavom korisnika otvaraju se mogućnosti pregleda vlastitih aerodroma, pregleda profila, pregleda vlastitih aviona i pregleda preporuka. Korisnik ako želi može pogledati vlastite aerodrome. Pri pogledu korisničkih aerodroma prikazuju se aerodromi koje korisnik prati i aerodromi koje korisnik ne prati. Ako želi može zapratiti aerodrom, otpratiti aerodrom, vidjeti polazne letove aerodroma i detalje jednog leta (slika 19).



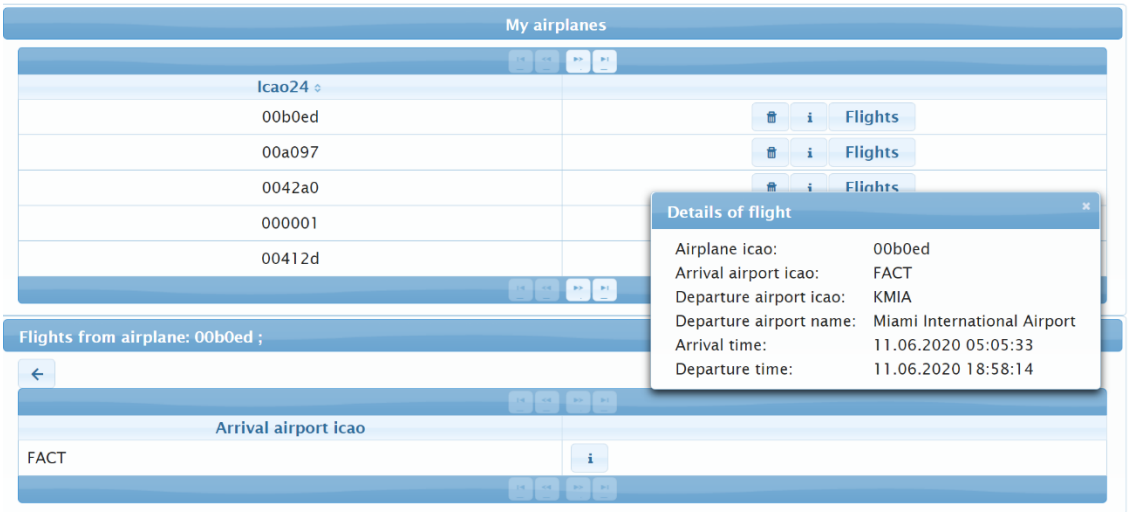
Slika 19: Prikaz stranice za pregled korisničkih aerodroma (Izvor: vlastita izrada)

S druge strane, pogled korisničkih aviona sadržava popis aviona koje korisnik prati i popis aviona koje korisnik nije još zapratio. Slično kao i kod pogleda korisničkih aerodroma, korisnik može otpratiti ili zapratiti određeni avion, pregledati letove aviona, pregledati detalje leta, ali i pregledati detalje samih aviona (slika 20).



Slika 20: Prikaz stranice za pregled aviona korisnika (Izvor: vlastita izrada)

Može se vidjeti iz slike 20 da su u gornjem dijelu ekrana prikazani avioni koje korisnik prati, a u donjem dijelu ekrana prikazani su svi ostali avioni. Korisnik može pregledati letove određenog aviona samo ako ga prati. Kada korisnik pritisne na gumb za pregled letova, prikaz svih aviona koje korisnik ne prati zamjenjuje se prikazom letova odabranog aviona.



Slika 21: Prikaz letova aviona i njegovih detalja (Izvor: vlastita izrada)

Osim pregleda aviona i aerodroma, korisnik može pregledati svoj profil i urediti ga. Na stranici za pregled preporuka može vidjeti deset aviona koji su mu preporučeni i koje ne prati. Na stranici preporuke preporučene avione korisnik može zapratiti ili vidjeti informacije o njima ako to želi (slika 22).

Slika 22: Prikaz preporuka korisniku (Izvor: vlastita izrada)

Prikazani avioni su učitani preko dostupne datoteke u kojoj se nalaze podaci o mogućim ocjenama korisnika za određene avione. Čitaju se samo podaci koji se odnose na prijavljenog korisnika. Nakon čitanja, sortiraju se silazno prema ocjeni te se brišu oni podaci o avionima koje korisnik već prati. Nakon toga prikazuje se prvih deset aviona koji imaju najveću izračunatu ocjenu korisnika. Broj preporuka može biti manji od deset.

8. Zaključak

Integracija sustava se može obaviti na više načina. Dijelimo integracije prema sloju integracije i prema sloju u kojem se implementiraju u troslojnoj arhitekturu. Prema slojevima integracije poznate su integracija procesa, integracija portala i agregacija entiteta. S druge strane, prema troslojnoj arhitekturi poznajemo podatkovnu, funkcionalnu i prezentacijsku integraciju. Iako svaka od ovih integracija ima nedostataka i prednosti, funkcionalna integracija i integracija procesa su najefikasnije i najkontroliranije integracije. Kao dio funkcionalne integracije javlja se integracija orijentirana servisima i integracija međuprogramom orijentiranog porukama. Web servisi omogućuju labavu povezanost sustava i interoperabilnost, a cilj korištenja web servisa je komunikacija porukama između aplikacija. Sustav poruka pak omogućuje upravljanje porukama, komunikaciju, labavu povezanost i posredništvo. Web servisi i sustavi poruka su kreirani sa željom odvajanja komponenti na način da rade neovisno jedan od drugog, da su otporni na greške, da mogu udaljeno komunicirati i da pružaju interoperabilnost. Upravo i sami reaktivni sustavi nalažu da je sustav reaktivan ukoliko je elastičan, otporan na greške, vođen porukama, održiv, proširiv i responzivan. Time možemo primijetiti da se najbolje prakse danas odnose na mogućnost integracije sustava na način da su neovisni, otporni na greške, vođeni porukama, proširivi, elastični i interoperabilni. Osim što sustavi trebaju biti povezani na navedeni način, bitno je da i sami mogu primiti i obraditi velike količine podataka. To se može ostvariti strujanjem podataka. Strujanje podataka je danas važno zbog velike količine podataka koja se proizvodi i analizira. Strujanje podataka je tehnologija prijenosa podataka preko interneta koja omogućava da se kasnije podaci obrađuju kao neprekidan niz. Time omogućuju proizvodnju i obradu beskonačnih kontinuiranih podataka.

Iako samo strujanje podataka nosi neku težinu apstrakcije, današnja tehnologija i biblioteke poput Apache Spark i Apache Kafka omogućuju jednostavno kreiranje web aplikacija na bazi strujanja podataka. Pažnju treba posvetiti svim postavkama Kafka teme, Kafka proizvođača, potrošača i Kafka Streams. Zbog toga se postavlja pitanje koliko je kvalitetno izvedeno rješenje. Apache Kafka otvara niz slučaja korištenja komponenti, slanje velike količine podataka bez ometanog rada komponenti i integraciju različitih sustava. Apache Spark zbog dostupnosti raznih modula omogućuje korištenje tih modula naizmjenično i bez zadrške što otvara bezbroj mogućnosti kao što je dohvaćanje podataka iz Kafka teme i upotreba analize nad tim podacima. Osim toga Apache Spark približava i pojednostavljuje korištenje kompleksnih algoritama strojnog učenja.

Izrađena je aplikacija na bazi strujanja podataka, a integrirana je s ostalim aplikacijama putem Apache Kafka posrednika poruka. Izradom ovog rada naučila sam kako slati i zaprimiti

velike količine podataka u kratkom roku i momentalno ih obrađivati u realnom vremenu. Tokom izrade rada došlo je do problema sa korištenjem Apache Spark biblioteka uz Apache Tomcat ili Glassfish server jer postoje razlike u verziji jezika Scala i Guava biblioteke pa je na kraju kreiran Spark posao fizički odvojen na Spark serveru. Kako danas strujanje podataka koristi većina web aplikacija, rad na ovu temu mi je pomogao u razumijevanju kako te aplikacije rade i kako kreirati samu aplikaciju na bazi strujanja podataka.

Popis literature

[1] Amazon Web services: „*What is streaming dana?*“, Amazon Web Services, Inc. Ili suradnici, Dostupno 05.06.2021 na: https://aws.amazon.com/streaming-data/

[2] Apache Derby : „*Documentation*“, The Apache Software Foundation, 2004 Dostupno 05.06.2021 na: https://docs.oracle.com/javadb/10.10.1.2/getstart/index.html

[3] Apache Flink : „*Alternating Least Squares*“. The Apache Software Foundation . Dostupno 29.05.2020 na: https://ci.apache.org/projects/flink/flink-docs-release-1.2/dev/libs/ml/als.html

[4] Apache Kafka : „*Apache Kafka*“, The Apache Software Foundation, Dostupno 05.06.2021 na: https://kafka.apache.org/

[5] Apache Kafka : „*Documentation : Kafka Connect*“, The Apache Software Foundation, Dostupno 05.06.2021 na: https://kafka.apache.org/documentation/#connect

[6] Apache Kafka : „*Documentation: Kafka 2.8 Documentation*“, The Apache Software Foundation, Dostupno 05.06.2021 na: https://kafka.apache.org/documentation/

[7] Apache Kafka : „*Introduction*“, The Apache Software Foundation, Dostupno 05.06.2021 na: https://kafka.apache.org/intro

[8] Apache Kafka : „*Kafka Streams*“, The Apache Software Foundation, Dostupno 05.06.2021 na: https://kafka.apache.org/documentation/streams/

[9] Apache Kafka : „*Powered by*“, The Apache Software Foundation, Dostupno 05.06.2021 na: https://kafka.apache.org/powered-by

[10] Apache Spark : „*Apache Spark*“. The Apache Software Foundation . Dostupno 04.05.2020 na: https://spark.apache.org/

[11] Apache Spark : „*JavaDoc dokumentacija Apache Spark*“. The Apache Software Foundation . Dostupno 04.05.2020 na: https://spark.apache.org/docs/latest/api/java/index.html?org/apache/spark/sql/Dataset.html

[12] Apache Spark : „*Spark Overview*“. The Apache Software Foundation . Dostupno 04.05.2020 na: https://spark.apache.org/docs/latest/

[13] Apache Spark : „*GraphX Programming Guide*“, The Apache Software Foundation, Dostupno 05.06.2021 na: https://spark.apache.org/docs/latest/graphx-programming-guide.html

[14] Apache Spark : „*Machine Learning Library (MLlib) Guide*“, The Apache Software Foundation, Dostupno 05.06.2021 na: https://spark.apache.org/docs/latest/ml-guide.html

[15] Apache Spark : „*Spark Streaming Programming Guide*“, The Apache Software Foundation, Dostupno 05.06.2021 na: https://spark.apache.org/docs/latest/streaming-programming-guide.html

[16] Apache Spark: „*RDD Programming Guide*“ , The Apache Software Foundation, Dostupno 05.06.2021 na: https://spark.apache.org/docs/latest/rdd-programming-guide.html

[17] Baldoni, Roberto; Contenti, Mariangela; Virgillito, Antonino: „The evolution of publish/subscribe communication systems“ Unutar: „*Future directions in distributed computing*“, Springer Verlag 2003.

[18] Bonér, Jonas; Farley, Dave; Kuhn, Roland; Thompson, Martin: „*The Reactive Manifest“* 2014. Dostupno 02.04.2021 na: https://www.reactivemanifesto.org/

[19] Booth, David; Champion, Michael; Haas, Hugo; McCabe, Francis; NewComer, Eric; Ferris, Chris; Orchard, David: „*Web Services Architecture“*. W3C (MIT, ERCIM, Keio) 2004. Dostupno 02.04.2021. na: https://www.w3.org/TR/ws-arch/wsa.pdf

[20] Bray, Tim; Paoli, Jim; Sperberg-McQueen, C. M.; Maler, Eva; Yergeau, Francois: „*Extensible Markup Language (XML) 1.0“ Peto izdanje, “*. W3C (MIT, ERCIM, Keio) 2008. Dostupno 02.04.2012. na: https://www.w3.org/TR/xml/

[21] Buschmann, Frank; Henney, Kevlin; Schmidt, Douglas C.: „*Pattern-Oriented Software Architecture, A Pattern Language for Distributed Computing.*“, 4. izdanje John Wiley & Sons 2007.

[22] Chinnici, Roberto ; Moreau , Jean-Jacques; Ryman , Arthur ; Weerawarana, Sanjiva „*Web Services Description Language (WSDL) Version 2.0*“, W3C (MIT, ERCIM, Keio), 2007. Dostupno 02.04.2021. na**:** https://www.w3.org/TR/wsdl.html

[23] Escoffier, Clement: *Building Reactive Microservices in Java: Asynchronous and Event-Based Application Design*. O'Reilly Media, 2017.

[24] Europska komisija „*New European Interoperability Framework Promoting seamless services and data flows for European public administrations*„ Publications Office of the European Union, 2017. Dostupno 02.04.2021 na: https://ec.europa.eu/isa2/sites/isa/files/eif_brochure_final.pdf

[25] Fu, Yupeng; Chen, Mingmin.: „*Disaster Recovery for Multi-Region Kafka at Uber*“ Uber Technologies, 2020. Dostupno 06.06.2020 na: https://eng.uber.com/kafka/

[26] Gantz, John; Reinsel, David; Rydning, John: **„***Data Age 2025: The Evolution of Data to Life-Critical“,* IDC, 2017**.**

[27] Gerić, Sandro; Vrček Neven: „*Elektroničko i mobilno poslovanje: Servisno orijentirana arhitektura“, Fakultet organizacije i informatike; Sveučilište u Zagrebu* 2020.

[28] Graham, Steve; Daniels Glen; Davis, Doug; Nakamura, Yuichi; Simeonov, Simon; Brittenham, Peter; Fremantle, Pauk; Konig Dieter; Zentner, Claudia: „*Building Web*

services with Java: making sense of XML, SOAP, WSDL, and UDDI". Drugo izdanje, SAMS publishing 2004.

[29] Halili, Festim; Ramadani, Erenis: „*Web services: a comparison of soap and rest services". Modern Applied Science*, 2018

[30] Hewitt, Carl; Bishop, Peter; Steiger, Richard: „A universal modular actor formalism for artificial intelligence." Unutar: „*Proceedings of the 3rd international joint conference on Artificial intelligence"*. 235-245 1973.

[31] Hohpe, Gregor; Woolf, Bobby: „*Enterprise integration patterns: Designing, building, and deploying messaging solutions"*. Addison-Wesley Professional, 2004. HOHPE

[32] IBM *„IBM MQ: Introduction to message queuing"* IBM 2020. Dostupno 02.04.2021. na: https://www.ibm.com/docs/en/ibm-mq/8.0?topic=overview-introduction-message-queuing

[33] Jansen, Grace; Gollmar, Peter: „*Reactive systems Explained*", O' Reilly Media, Inc. 2020.

[34] Kalin, Martin. „*Java Web Services: Up and Running: A Quick, Practical, and Thorough Introduction*" O'Reilly Media, Inc., 2013.

[35] Kaye, Doug: „*Loosely coupled: the missing pieces of Web services"*. RDS Strategies LLC, 2003.

[36] Kent, William: „*Data and Reality: A Timeless Perspective on Perceiving and Managing Information"*. Technics publications, 2012.

[37] Keeton, Chrissy: "*Poor Website Performance Undermines Customers' Purchase Intent and Brand Impression*" SingleStone 2014. Dostupno 02.04.2021 na: https://docplayer.net/9957467-Poor-website-performance-undermines-customers-purchase-intent-and-brand-impression.html

[38] Levy, Eran „*4 Key Components of a Streaming Data Architecture*" Upsolver, 2021. Dostupno 06.06.2020 na: https://www.upsolver.com/blog/streaming-data-architecture-key-components

[39] Li, Haoming; He, Bangzheng; Lubin, Michael; Perez, Yonathan: „*Distributed Algorithms and Optimization: Matrix Completion via Alternating Least Square(ALS)*" Stanford, 2015; Dostupno 29.05.2020. na: http://stanford.edu/~rezab/classes/cme323/S15/notes/lec14.pdf

[40] McFadin, Patrick: „*The SMACK stack*", O'Reilly, 2017. Dostupno 05.04.2021 na: https://www.oreilly.com/radar/the-smack-stack/

[41] Mitra, Nilo; Lafon, Yves : „*SOAP Version 1.2 Part 0: Primer (Second Edition)***",** W3C (MIT, ERCIM, Keio), 2007. Dostupno 02.04.2021. na : https://www.w3.org/TR/2007/REC-soap12-part0-20070427/#L1165

[42] Narkhede, Neha; Shapira, Gwen; Palino, Todd : „*Kafka: The definitive Guide*“ O'Reilly, 2017.

[43] OpenSky Network: „*The OpenSky Network API documentation*“, The OpenSky Network, 2017. Dostupno 05.06.2021 na: https://opensky-network.org/apidoc/

[44] Oracle: „*Programming WebLogic Web Services: Inteoperability“*, BEA Systems 2004. Dostupno 02.04.2021. na: https://docs.oracle.com/cd/E13222_01/wls/docs81/webserv/interop.html

[45] Oxford „*Oxford Learner's Dictionaries*“ Oxford University Press, 2021. Dostupno 02.04.2021 na: https://www.oxfordlearnersdictionaries.com/

[46] Ponge, Julien: „*Vert.x in action: Asynchronous and Reactive Java*“ Manning Publications, 2020. Dostupno 04.04.2021. na https://www.manning.com/books/vertx-in-action

[47] Primefaces : „Primefaces“ PrimeTek, 2021. Dostupno 04.06. na: https://www.primefaces.org/

[48] Roestenburg, Raymond; Bakker, Rob; Williams, Rob: „*Akka in action“*. Manning, 2017.

[49] Ruh, William; Maginnis, Francis X.; Brown, William J.*: „Enterprise Application Integration“.* A Wiley Tech Brief. Wiley, 2001.

[50] Šego, Vedran: „*P=NP?*“, *Matematičko fizički list*, vol.70, br. IZVANREDNI-J, str. 26-35, 2009. Dostupno 06.06.2020 na: https://hrcak.srce.hr/243630

[51] SOA software „*SOA Infrastructure Reference Model*“ SOA Software, Inc. 2002.

[52] Teale, Philip; Etz, Christopher; Kiel, Michael; Zeitz, Carsten: “*Data Patterns.” .NET Architecture Center*, 2003.

[53] Trowbridge, David; Mancini, Dave; Quick, Dave; Hohpe, Gregor; Newkirk, James; Lavigne, David: „*Enterprise Solution Patterns Using Microsoft“ .NET*. Microsoft Press, 2003

[54] TrowBridge, David; Roxburgh, Ulrich Hohpe, Gregor, Manolescu, Dragos, & Nadhan, E. G. „Integration Patterns“. Microsoft Corporation 2004. Dostupno 02.04.2012. na: http://download.microsoft.com/download/a/c/f/acf079ca-670e-4942-8a53-e587a0959d75/intpatt.pdf

[55] Visual Paradigm: „Visual Paradigm Online“, Visual Paradigm, 2021. Dostupno 05.06.2020. na: https://online.visual-paradigm.com/#

[56] Windley, Phillip J.: *„Service Oriented Architectures“* National IT- and Telecom Agency, Denmark, 2004.

[57] Wu, Steven i sur.: „*Evolution of the Netflix Data Pipeline*“ Medium, 2016. Dostupno 06.06.2020 na: https://netflixtechblog.com/evolution-of-the-netflix-data-pipeline-da246ca36905

[58] Wu, Steven i sur.: „*Kafka Inside Keystone Pipeline*“ Medium, 2016. Dostupno 06.06.2020 na: https://netflixtechblog.com/kafka-inside-keystone-pipeline-dd5aeabaf6bb

[59] Yee Andre, Apte Atul „*Integrating Your e-Business Enterprise“* Sams, 2001.

Popis slika

Slika 1: Arhitektura web servisa (Izvor: vlastita izrada, prema: [27]) 18
Slika 2: SOA model (Izvor: vlastita izrada) 21
Slika 3: Struktura SOAP poruke (Izvor: vlastita izrada, prema: [28]) 23
Slika 4: Dijagram procesa razmjene poruka (Izvor: vlastita izrada, prema: [31]) 26
Slika 5: Odnos principa reaktivnih sustava (Izvor: vlastita izrada, prema: [18]) 35
Slika 6: UML dijagram slučajeva korištenja (Izvor: vlastita izrada) 50
Slika 7: UML dijagram slučajeva korištenja: prijava registracija i pregled profila (Izvor: vlastita izrada) 51
Slika 8: UML dijagram slučajeva korištenja: pregled aviona, aerodroma i preporuka (Izvor: vlastita izrada) 52
Slika 9: ERA model baze podataka (Izvor: vlastita izrada) 53
Slika 10: Arhitektura aplikacije (Izvor: vlastita izrada) 55
Slika 11: Shema aplikacije (Izvor: vlastita izrada) 56
Slika 12: UML dijagram klasa "ilevak_OSNetwork" (Izvor: vlastita izrada) 58
Slika 13: UML dijagram klasa biblioteke "ilevak_kafka" (Izvor: vlastita izrada) 60
Slika 14: Prikaz komunikacije komponenti putem Apache Kafka (Izvor: vlastita izrada) 61
Slika 15: UML dijagram aktivnosti dretve "FlightProducer" (Izvor: vlastita izrada) 63
Slika 16: Prikaz izbornika neprijavljenog korisnika (lijevo) i prijavljenog korisnika(desno) (Izvor: vlastita izrada) 70
Slika 17: Stranica za registraciju (Izvor: vlastita izrada) 70
Slika 18: Stranica za prijavu (Izvor: vlastita izrada) 71
Slika 19: Prikaz stranice za pregled korisničkih aerodroma (Izvor: vlastita izrada) 71
Slika 20: Prikaz stranice za pregled aviona korisnika (Izvor: vlastita izrada) 72
Slika 21: Prikaz letova aviona i njegovih detalja (Izvor: vlastita izrada) 72
Slika 22: Prikaz preporuka korisniku (Izvor: vlastita izrada) 73

Popis tablica

Tablica 1. Koraci ALS algoritma (Izvor: vlastita izrada, prema: [39]) 68