SVEUČILIŠTE U ZAGREBU
FILOZOFSKI FAKULTET
ODSJEK ZA INFORMACIJSKE I KOMUNIKACIJSKE ZNANOSTI

Vedran Deković

Mikroservisna arhitektura u razvoju kompleksnog IT sustava

Diplomski rad

Mentor: dr.sc. Vedran Juričić, doc.

Zagreb, ožujak 2018.

Sadržaj

1. Uvod

Informacijski sustav u današnje vrijeme ima vrlo značajnu ulogu gotovo svakog poslovnog sustava, a mnoge poslovne organizacije danas svoje cjelokupno poslovanje temelje na određenom informacijskom sustavu. Razvoj informacijskih i komunikacijskih tehnologija raste iz dana u dan te se svakim danom javljaju nove tehnologije i alati kako bi osigurali efektivnost i efikasnost u radu informacijskih sustava. U računalnom svijetu danas također se često javlja pojam računalstvo u oblaku (eng. cloud computing), a to je pojam koji se javlja kada se govori o nastanku i razvoju informacijskih sustava. Ovaj je pojam uveo značajne napretke u razvoju složenih i distribuiranih sustava koji zajedno rade u rješavanju zadataka i ostvarivanju krajnjih ciljeva pojedinaca ili organizacija.

Računalstvo u oblaku omogućuje pojedincima i organizacijama da djeluju globalno te da svoje poslovanje ostvaruju putem usluga. Velike tvrtke kao što su Amazon, Microsoft ili Google osim svojih primarnih djelatnosti, omogućuju svojim klijentima i iznajmljivanje računalne i mrežne infrastrukture te time značajno doprinose širenju i razvoju računalstva u oblaku. Tema ovoga rada biti će prikaz mikroservisne arhitekture u razvoju kompleksnog informacijskog sustava, a u sklopu rada biti će prikazan model web aplikacije koja će biti izrađena uporabom ove arhitekture. Tokom rada biti će objašnjen sam pojam mikroservisne arhitekture te će biti opisane i druge bitne softverske arhitekture koje su značajno utjecale na nastanak ove arhitekture. Web aplikacijom prikazanom kroz ovaj rad nastoji se prikazati način pomoću kojeg se model složenog sustava može razdijeliti na više manjih i neovisnih sustava te povećati efikasnost rada i održavanja sustava.

2. Web aplikacija

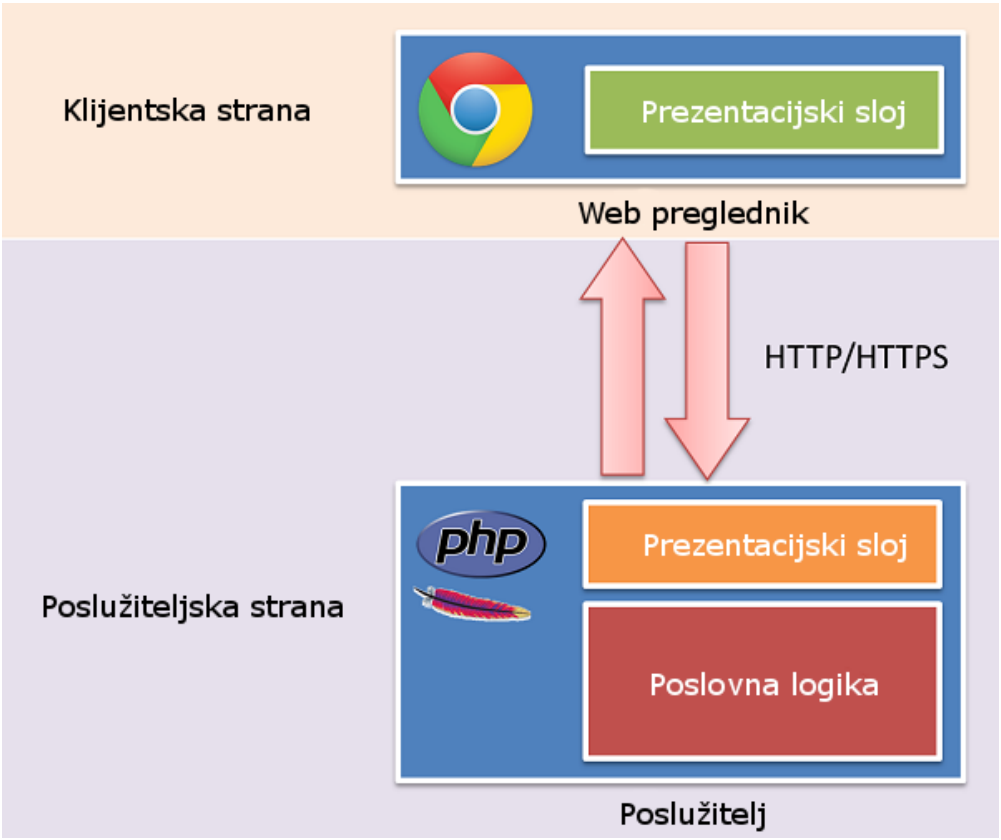
Aplikacija, bilo da je riječ o web aplikaciji, aplikaciji za stolna računala (eng. desktop application) ili aplikaciji za mobilni uređaj, ima svoju svrhu te svojim djelovanjem rješava specifične poslovne probleme u informacijskom sustavu koji podupire poslovni sustav. Aplikacija kao dio informacijskog sustava organizacije ima važnu ulogu u obradi i pohranjivanju poslovnih podataka na temelju kojih se donose poslovne odluke u organizaciji. Rast i razvoj web aplikacije prate i mnogi problemi vezani za: skalabilnost, performanse, održavanje i testiranje web aplikacije. Navedeni problemi samo su neki od problema koji mogu značajno utjecati na mnoge probleme poslovnog sustava, pa čak i na krajnji poslovni rezultat poslovnog sustava kojeg informacijski sustav podupire.

Mikroservisna arhitektura koja je u središtu pozornosti ovoga rada, pravilnom implementacijom može riješiti prethodno navedene probleme, no i ona u razvoj web aplikacije donosi neke druge probleme. Razvoj tehnologije omogućio je mnogim organizacijama koje se bave razvojem softverskih rješenja da svoje djelatnike organiziraju u timove koji se bave rješavanjem specifičnih poslovnih problema koji se aplikacijom nastoje riješiti te se tako softversko rješenje može brže i lakše isporučiti krajnjem korisniku.

2.1. Dijelovi web aplikacije

Web aplikacija najčešće se sastoji od mnogih dijelova koji rade zajedno kako bi ostvarili smisao i cilj same aplikacije. Većina današnjih web aplikacija se sastoji od dva dijela: klijentska strana (eng. frontend side) i poslužiteljska strana (eng. backend side) [29]. Web preglednik komunicira s poslužiteljem putem HTTP protokola (eng. Hypertext Transfer Protocol - HTTP) koji predstavlja mrežni protokol koji djeluje na tzv. aplikacijskom sloju mrežnog modela [11]. Slika 1. prikazuje shemu osnovnih dijelova web aplikacije te njihovu međusobnu komunikaciju. Poslužiteljska strana web aplikacije zaprima zahtjeve od klijentske strane web aplikacije tj. korisnika te vraća rezultate u obliku koji korisnikova aplikacija (npr. web preglednik) može interpretirati na odgovarajući način.

Web tehnologije rastu i razvijaju se svakodnevno te pojedini dijelovi web aplikacije dobivaju sve veću odgovornost u svome radu. Klijentska strana web aplikacije danas, uz web preglednike, obuhvaća i mobilne uređaje te tablete, a mnoge se web aplikacije strogo „prilagođavaju“ takvim vrstama uređaja. Iako je HTTP protokol vrlo jednostavan protokol za komunikaciju koji ne prati stanja, vrlo je bitno da uređaji koji sudjeluju u komunikaciji prate stanja odnosno održavaju komunikacijsku sesiju.



Slika 1. Dijelovi web aplikacije[1]

2.2. Skalabilnost, testiranje i održavanje web aplikacije

Ispravnost rada web aplikacije u interesu je svake poslovne organizacije koja provodi određeni poslovni proces. U procesu planiranja izrade bilo koje aplikacije, pa tako i web aplikacije, važno je uzeti u obzir mnoge faktore među kojima su: skalabilnost, testiranje i održavanje web aplikacije koja se implementira. Na poslužitelj web aplikaciju paralelno se povezuje veći broj klijenata s zahtjevima koje web aplikacija mora obraditi te ukoliko web aplikacija nije u mogućnosti obraditi sve pristigle zahtjeve dolazi do zagušenja sustava i vrlo često i do pada sustava što može dovesti do negativnih posljedica kao što je na primjer gubitak broja klijenata ili gubitak podataka. Ovoj temi se može posvetiti mnogo pozornosti te se treba obraditi mnogo povezanih područja, no kako to nije tema ovoga rada biti će samo ukratko opisana.

Kada se govori i izradi web aplikacija, govori se o dvije vrste skalabilnosti: vertikalna i horizontalna skalabilnost [27]. Vertikalna skalabilnost je vrsta skalabilnosti sustava kod koje se na postojeći sustav nadograđuju nove komponente ili zamjenjuju postojeće komponente s

[1] https://www.nczonline.net/images/wp-content/uploads/2013/10/nodejs1.png

novim i boljim računalnim komponentama. Horizontalna skalabilnost sustava je skalabilnost kod koje se u rad sustava dodaju nova računala tzv. čvorovi (eng. nodes). Obje vrste skalabilnosti služe kako bi se poboljšale performanse postojećeg sustava i omogućio korisnicima nesmetan rad s sustavom. Nadogradnja postojećeg sustava obuhvaća izradu novih komponenti te izmjenu i nadogradnju postojećih komponenti te stoga lako može doći do pogrešaka u radu aplikacije. Kako bi se izbjegle pogreške, važno je ispravno testirati aplikaciju prije puštanja u produkcijsku okolinu.

Testiranje aplikacije može se izvesti na mnogo načina, bilo da se radi o automatskom testiranju ili testiranju putem testnih korisnika sustava. Danas, mnogi se programeri odlučuju na pisanje automatskih testova koji su definirani prema unaprijed određenim slučajevima, a zadovoljavajući rezultati tih testova preduvjet su za puštanje novih mogućnosti aplikacije u produkcijsku okolinu. Automatsko testiranje omogućuje brzo testiranje aplikacije iz razloga što testiranje odrađuje računalo koje samo nakon testiranja vrati korisniku izvještaj o testiranju, no problem je što se mogu testirati samo određeni slučajevi, dok se kod testiranja s korisnicima mogu provesti i testovi koji nisu prethodno točno određeni.

Izradom novih komponenti u web aplikaciji, znatno se povećava broj linija programskog koda te se povećava i sveukupna složenost aplikacije. Iz prethodno navedenog razloga, potrebno je pažljivo odrediti temelje odnosno arhitekturu nad kojom će se graditi buduća aplikacija. Loše odabrana arhitektura može u budućem razvoju dovesti do mnogih problema, pa čak i do nemogućnosti nadogradnje aplikacije s novim komponentama, što će biti objašnjeno u idućim poglavljima. Prethodno navedenim faktorima koji utječu na životni vijek aplikacije (i odgovarajućim rješenjima), mogu se posvetiti cijela poglavlja, pa čak i knjige, no kako nisu tema ovoga rada, nije ih potrebno detaljnije opisivati.

2.3. Računalstvo u oblaku

Web aplikacija predstavlja skup programskog koda koji se izvršava s kako bi se postigli određeni ciljevi. Kako bi se programski kod mogao izvršavati, mora biti smješten unutar računalnog sustava tj. poslužitelja koji ima mogućnosti izvršavanja programskog koda. Okruženja računalstva u oblaku su dinamična, a resursi se prema potrebi dodjeljuju i otpuštaju iz virtualnih i dijeljenih bazena resursa (eng. resource pool) [14]. Ovakva dinamična okruženja omogućuju vrlo jednostavno i brzo nastajanje novih sustava čime se sprječava nastajanje zagušenje sustava aplikacije. Računalstvo u oblaku omogućuje korisniku ili organizaciji zakup računalnih resursa prema potrebi čime se mogu efektivno iskoristiti financijska sredstva osigurana za računalnu infrastrukturu.

3. Softverske arhitekture

Na području razvoja softverskih rješenja susrećemo se s dva vrlo bitna pojma a to su: softverska arhitektura i softverski dizajn [34]. Softverska arhitektura opisuje strukturu sustava odnosno komponente sustava i načine kako komponente toga sustava međusobno komuniciraju. Softverska arhitektura vrlo je bitna u ranom životnom ciklusu softvera iz razloga što se postavlja osnovna struktura sustava nad kojom se dalje izgrađuju dijelovi. Softverski dizajn odnosi se na proces definiranja softverskih metoda, funkcija, objekata i cjelokupne strukture i interakcije dijelova programskog koda što rezultira novim funkcionalnostima koje zadovoljavaju potrebe korisnika.

U ovome poglavlju i pripadajućim potpoglavljima, biti će opisane neke od danas najznačajnijih arhitektura koje se upotrebljavaju prilikom razvoja softverskih rješenja s naglaskom na web aplikacijama, a pritom je bitno za napomenuti da ne postoji „savršena“ arhitektura koje može poslužiti kao rješenje apsolutno svakog problema prilikom izrade aplikacije i to iz razloga što svaka softverska arhitektura ima svoje prednosti i nedostatke.

3.1. Ciljevi softverske arhitekture

Softverska arhitektura predstavlja poveznicu između poslovnih i tehničkih zahtjeva pomoću razumijevanja slučajeva uporabe. Kako bi se ostvarili odgovarajući ciljevi softverske arhitekture, potrebno je poštivati ključne principe softverske arhitekture. Softversku arhitekturu treba graditi na način da se kasnije u životnom vijeku aplikacije arhitektura može prilagođavati novim promjenama ili korisničkim zahtjevima. Softverska arhitektura nije ograničena na jedan arhitekturalni stil, već se može koristiti zajedno s drugim arhitekturama kako bi se izgradio cjelokupni sustav. Arhitekturalni stil ili arhitekturalni uzorak dizajna predstavlja obrazac, odnosno apstraktni okvir za oblikovanje sustava i njihove strukture. Arhitekturalni stilovi mogu se podijeliti u kategorije ovisno o arhitekturalnim problemima koje nastoje riješiti, a ta je podjela prikazana u Tablica 1.

Tablica 1. Podjela arhitekturalnih stilova prema kategorijama[2]

Kategorija	Arhitekturalni stilovi
Komunikacija	Servisno-orijentirana arhitektura, Message-bus arhitektura
Razvoj	Klijent-poslužitelj arhitektura, višeslojna arhitektura
Domena	Arhitekturalni stil orijentiran na domenu (eng. Domain Driven Design architecture)
Struktura	Objektno-orijentirana arhitektura, Arhitektura bazirana na komponentama

3.2. Monolitna arhitektura

Monolitna arhitektura ili jednoslojna arhitektura (eng. 1-tier architecture) [23] predstavlja danas softversku arhitekturu čije temelje sadrže mnoge današnje web aplikacije. Ova arhitektura prikladna je za mnoge manje i srednje velike web aplikacije. Monolitna arhitektura sadrži mnogo prednosti ali i nedostataka koje je vrlo bitno za napomenuti kako bi se kasnije moglo objasniti mikroservisnu arhitekturu te probleme koje ona rješava.

Kada se kreće u izradu web aplikacije, programeri se iz mnogo razloga najčešće odlučuju na monolitnu arhitekturu, najčešće u kombinaciji s tzv. MVC (eng. Model View Controller - MVC) uzorkom dizajna. Monolitna arhitektura je odgovarajuća arhitektura za gotovo većinu web aplikaciju u početnoj fazi razvoja iz razloga zato što je složenost programske logike vrlo mala te se pojedini dijelovi mogu relativno jednostavno testirati, a i razvoj novih dijelova aplikacije je brz i jednostavan. MVC predstavlja tzv. arhitekturalni uzorak dizajna koji se u kombinaciji s gotovim skupom alata tj. okvirom (eng. framework) koristi kod izrade web aplikacije. Navedeni uzorak dizajna omogućava vrlo jednostavnu te dosta preglednu i organiziranu strukturu web aplikacije te time i vrlo brzi razvoj aplikacije u mladoj fazi. Iako ovaj uzorak dizajna svoju primjenu najčešće nalazi kod aplikacija na poslužiteljskoj strani, danas i mnoge stolne (eng. desktop), pa čak i mobilne aplikacije koriste ovaj uzorak dizajna prilikom implementacije svojih rješenja. U aplikacije koje se izrađuju ovom arhitekturom ubrajamo i izvršne stolne aplikacije (eng. desktop applications) koje izvršavaju operacije uporabom svojeg procesa koji nema zadaću komunicirati s ostalim sustavima ili aplikacijama.

[2] https://msdn.microsoft.com/en-us/library/ee658117.aspx

Razdvajanje poslovne logike, podatkovnog sloja, te sloja korisničkog sučelja ima vrlo važnu ulogu kod aplikacija čija složenost raste iz dana u dan. Rast i razvoj aplikacije te razvoj tima ljudi koji održavaju i nadograđuju aplikaciju može predstavljati vrlo ozbiljan problem iz razloga što je previše programskog koda na jednom mjestu te je vrlo teško pisati testove koji provjeravaju ispravnost funkcionalnosti web aplikacije. Problemi kompleksnosti web aplikacije često se znaju rješavati na način da se u softverskom dizajnu dodaju novi slojevi aplikacije te se time između ostaloga smanjuje „gustoća" programskog koda, a povećava se organiziranost i preglednost programskog koda. Problem koji se javlja kod monolitne arhitekture (a nastoji se riješiti pomoću servisno orijentiranih arhitektura, npr. mikroservisnom arhitekturom) je taj što je budući razvoj aplikacije sve teži ukoliko se u obzir uzme izrada novih dijelova te njihovo održavanje.

Tokom životnog ciklusa aplikacije djelatnici se mijenjaju te stoga novim zaposlenicima treba više vremena za upoznavanje s velikim brojem linija programskog koda na kompleksnijim dijelovima aplikacije. Problem kod monolitne arhitekture je i skaliranje sustava iz razloga što se uz aplikaciju, na sustavu nalaze i druge aplikacije (npr. baza podataka) koje su potrebne da bi aplikacija obavljala svoj posao. Svi procesi koji su na istom poslužitelju dijele iste računalne resurse što može usporiti rad cjelokupnog sustava, pa je potrebno ispravno izvršiti skaliranje aplikacije ili promijeniti softversku arhitekturu. U današnje vrijeme kada se sustavi i aplikacije sve više povezuju putem mreže, uporaba ovakve vrste arhitekture za izradu aplikacija gubi na značaju.

3.2.1. Višeslojne arhitekture

Dodavanjem novih slojeva na jednoslojnu arhitekturu odnosno granulacijom sustava na manje, logički povezane dijelove, izgrađuje se višeslojna arhitektura (eng. N-tier architecture). Više slojna arhitektura, kao i što sam naziv govori, sastoji se od više slojeva (najčešće tri ili četiri sloja u stvarnoj primjeni) koji su napravljeni na način da mogu djelovati na zasebnim računalnim sustavima. Kod opisa monolitne arhitekture, rečeno je da svi procesi monolitne arhitekture dijele iste računalne resurse što predstavlja problem ukoliko dođe do preopterećenja računalnih resursa te se stoga javljaju višeslojne arhitekture kod kojih svaki sloj ima svoj računalne resurse.

3.2.1.1. Klijent-poslužitelj arhitektura

Višeslojna arhitektura koja je sačinjena od dva sloja koji međusobno komuniciraju naziva se klijent-poslužitelj arhitektura (eng. Client-server architecture) [3], a predstavlja

dvoslojnu arhitekturu (eng. 2-tier architecture). Klijent-poslužitelj arhitektura je arhitektura u izradi aplikacija koja je nastala 80-ih godina prošlog stoljeća [4], a svoju „popularnost“ nalazi i danas u rješavanju problema na području izrade softverskih rješenja. Kod ovakve vrste arhitekture nailazimo na dva ili više odvojenih procesa (procesi klijenta i proces poslužitelja) koji putem mreže komuniciraju i razmjenjuju podatke. Jedna od prednosti ovakve vrste arhitekture je ta što klijentski uređaji mogu razmjenjivati podatke s poslužiteljem koji je najčešće zadužen za rješavanje složene poslovne logike te trajnom pohranom i obradom podataka.

Budući da poslužitelj predstavlja središnje mjesto za obradu i pohranu podataka može doći do problema u radu sustava ukoliko se na poslužitelj poveže veliki broj klijenata te ukoliko poslužitelj nema dovoljno snažne računalne resurse za paralelnu obradu svih klijentskih zahtjeva. Kod klijent-poslužitelj arhitekture, poslužitelj dijeli računalne resurse s bazom podataka koja najčešće trpi veliko opterećenje kada obrađuje upite korisnika stoga se javlja potreba za odvajanjem programske logike poslužitelja i baze podataka. Ovakvo razdvajanje programske logike dovodi do još jednog sloja te se time dolazi do troslojne arhitekture.

3.2.1.2. Troslojna arhitektura

Troslojna arhitektura predstavlja softversku arhitekturu koja se sastoji od tri sloja (klijentski, aplikacijski i podatkovni) koji međusobno djeluju. Slika 2. prikazuje shemu tj. slojeve troslojne arhitekture i njihovu međusobnu povezanost. Prezentacijski sloj komunicira s aplikacijskim slojem, odnosno s poslužiteljem koji obrađuje korisnikove zahtjeve te po potrebi komunicira s slojem baze podataka unutar kojeg se trajno pohranjuju i obrađuju poslovni podaci. Troslojna arhitektura je najčešća arhitektura koja se javlja kod višeslojnih arhitektura iz razloga što se gotovo svaka aplikacija sastoji od prezentacijskoj dijela, dijela baze podataka i dijela koji „povezuje“ dva prethodno navedena dijela.

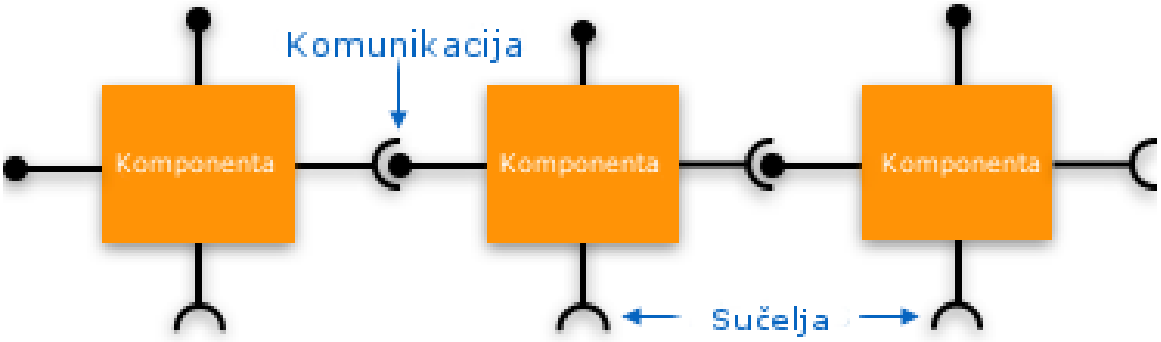
Danas se često javljaju i četveroslojne arhitekture koje u još jednom, dodatnom sloju nastoje riješiti ograničenja koja nudi prethodno opisana troslojna arhitektura. Kada kompleksnost aplikacije naraste na razinu da je sve teže raditi na razvoju i održavanju aplikacije, javi se potreba za granulacijom aplikacije na još manje dijelove tj. sustave koji prividno djeluju kao jedna cjelina te se time dolazi do servisno orijentiranih arhitektura koje će biti opisane u idućem poglavlju.

Slika 2. Shema troslojne arhitekture[3]

3.2.2. Komponentna arhitektura

Komponentna arhitektura (eng. component architecture) predstavlja softversku arhitekturu koja u središte promatranja stavlja komponentu. Slika 3. prikazuje shemu komponente arhitekture koja se sastoji od komponenata (predstavljene narančastim pravokutnicima). Komponente se međusobno povezuju putem dogovorenih sučelja za komunikaciju, a na komponente se na visokoj razini može promatrati kao crne kutije čija implementacija nije važna. Komponenta arhitektura ima mnogo pozitivnih strana, a jedna od važnijih je orijentiranost na ponovnu iskoristivost dijelova tj. komponenata. Komponente se mogu ponovo iskoristiti za dizajn neki drugih dijelova aplikacije što skraćuje vrijeme izrade aplikacije. Prednost ponovne iskoristivosti je i ta što se jednom napravljena i testirana komponenta može brže staviti u rad. Komponente kod ove arhitekture moraju biti dizajnirane na način da se mogu jednostavno zamijeniti s novim komponentama, a to se postiže putem tzv. komunikacijskih sučelja koja predstavljaju „ulazna vrata“ ili „izlazna vrata“ kod komponenta. Komunikacijska sučelja osiguravaju da će komponenta za odgovarajuće ulazne podatka vraćati odgovarajuće izlazne podatke s obzirom na poslovnu logiku koju komponenta sadrži.

[3] http://4.bp.blogspot.com/-uEdWYuTHUnU/VMSClSFgTcI/AAAAAAAAAPw/uV2YelL9luc/s1600/3-tier.PNG

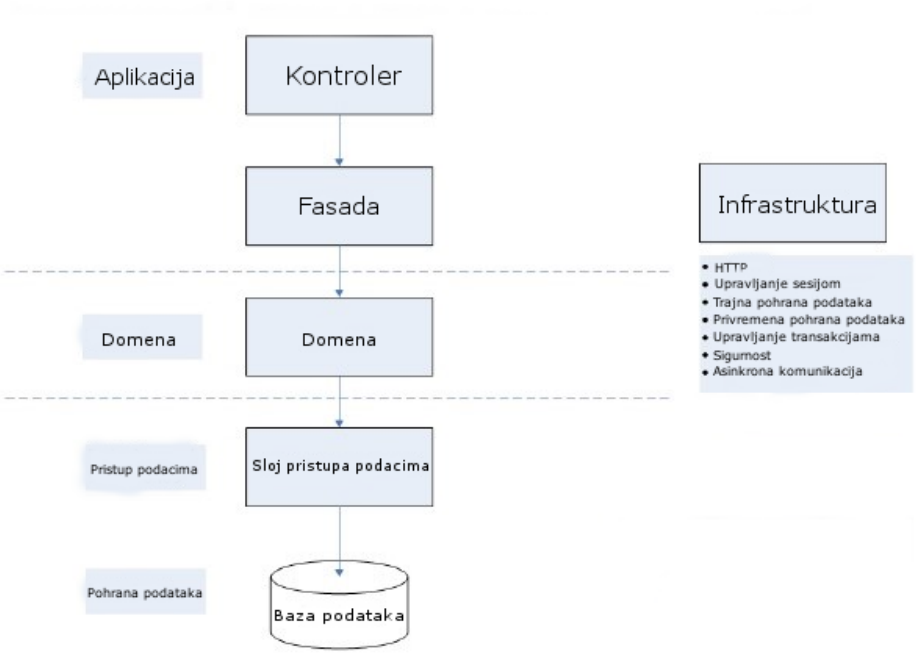


Slika 3. Shema komponentne arhitekture[4]

3.2.3. Arhitekturalni stil orijentiran na domenu

Arhitekturalni stil orijentiran na domenu (eng. Domain driven design) predstavlja arhitekturalni stil kod kojega domena sadrži cijelu poslovnu logiku te je središnji dio oko kojega se razvijaju svi ostali dijelovi tj. slojevi sustava. Kod ovog pristupa oblikovanja arhitekture aplikacije, funkcionalnosti kao što su: pohrana podataka, implementacija korisničkog sučelja i načini komunikacije između slojeva gube na važnosti te se nadograđuju kasnije na aplikaciju, nakon što se osnovna funkcionalnost tj. sloj domene izradi [12]. Slika 4. prikazuje shemu arhitekturalnog stila, gdje se može jasno vidjeti kako je sloj domene sloj između aplikacijske logike i podatkovnog sloja. Može se uočiti da je ovaj stil prilično sličan troslojnoj arhitekturi, ali u svome središtu ima sloj domene na koji se nadograđuju ostali dijelovi sustava. Sloj domene se najčešće izrađuje u suradnji s stručnjacima područja kako bi se ispravno implementirala poslovna logika aplikacije.

[4] https://www.tutorialspoint.com/software_engineering/images/components.png

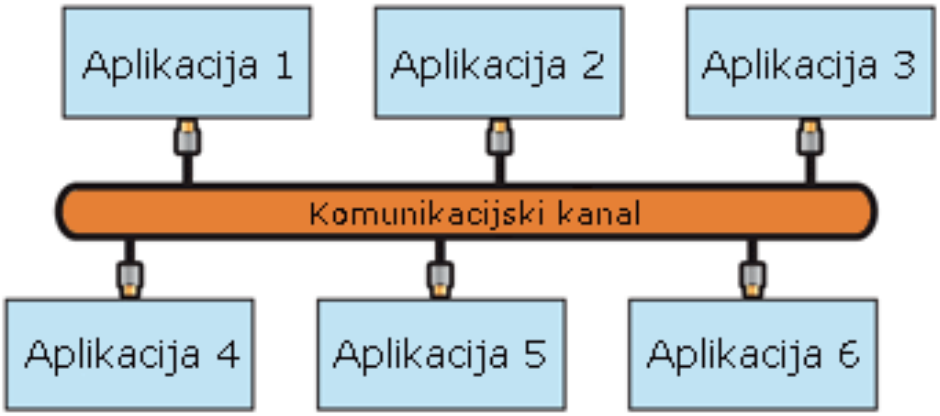


Slika 4. Shema arhitekturalnog stila orijentiranog na domenu[5]

3.2.4. Message-bus arhitekturalni stil

Message-bus arhitekturalni stil predstavlja arhitekturalni stil kod kojega se aplikacije priključuju na komunikacijski kanal koji „osluškuju“ kako bi zaprimile podatke za obradu. Održava povezanost između velikog broja rastućih čvorova koji se priključuju po potrebi te u svojoj interakciji s drugim sustavima ne trebaju poznavati detalje ostalih čvorova u komunikaciji. Osnovne karakteristike ove arhitekture su povećanje kompleksnosti sustava i povećanje decentralizacija sustava. Priključivanje novih čvorova nema utjecaja na već priključene čvore iz razloga što se čvorovi, kao komponente kod komponentne arhitekture, nadodaju na komunikacijski kanal koji je implementiran na tako da prihvaća nove čvorove. Čvorovi, tj. aplikacije mogu biti izrađene putem različitih tehnologija (kao što je to primjer kod komponentne arhitekture ili servisno-orijentiranih arhitektura), ali moraju poštivati standarde prilikom komunikacije s ostalim čvorovima preko komunikacijskog kanala na koji su priključeni. Slika 5. prikazuje shemu ove arhitekturi gdje se može uočiti komunikacijski kanal i aplikacije koje su na njega povezane.

[5] https://image.slidesharecdn.com/domaindrivendesigndevelopmentspringportfolio-090301170119-phpapp01/95/domain-driven-design-development-spring-portfolio-13-728.jpg?cb=1235926922



Slika 5. Message-bus arhitekturalni stil[6]

[6] https://i-msdn.sec.s-msft.com/dynimg/IC136906.gif

4. Servisno orijentirane arhitekture

Prethodno opisane arhitekture svoju primjenu uspješno nalaze kod mnogih najčešće manje kompleksnih aplikacija, no povećanjem kompleksnosti sustava i aplikacija javlja se potreba za uvođenjem novih arhitektura na području razvoja softvera. Servisno orijentirane arhitekture (eng. Service Oriented Architecture - SOA) u razvoju web aplikacija u svoj prvi plan stavljaju uslugu koja je „osnovna jedinica“ za daljnji razvoj aplikacije. Kada govorimo o ovoj vrsti arhitekture najčešće se odnosi na dislocirane tj. distribuirane sustave koji putem određenog, unaprijed definiranog sučelja komuniciraju s ostalim aplikacijama i sustavima.

4.1. Mrežne usluge

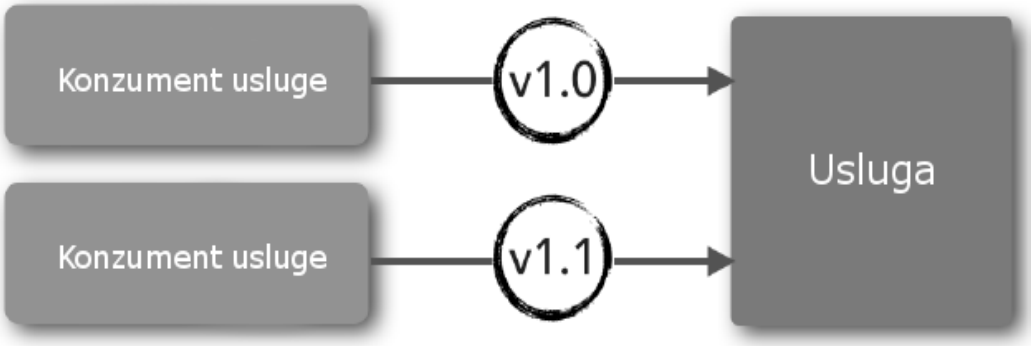
Mrežne usluge (eng. web services) predstavljaju skup otvorenih protokola i standarda koji se koriste za razmjenu podataka između sustava ili aplikacija [32]. Aplikacije pristupaju mrežnim uslugama putem prethodno spomenutog mrežnog protokola HTTP te razmjenjuju podatke preko standardiziranih formata za razmjenu podataka. Pojam mrežnih usluga vrlo je bitan pojam za ovaj rad iz razloga što se servisno orijentirane arhitekture (koje će biti opisane kasnije tokom rada) temelje na ovome principu rada.

Pojava mrežnih usluga omogućila je komunikaciju i razmjenu podataka između klijenta i poslužitelja bez održavanja stalne TCP veze između dva kraja. REST (eng. REST - Representational state transfer) [25] je arhitekturalni stil za izradu distribuiranih sustava u kojem su podaci i funkcionalnosti predstavljeni putem adresa, odnosno web poveznica kao i drugi resursi (slike, datoteke i sl.) na internetu. Ovakvom komunikacijom računala mogu vrlo jednostavno razmjenjivati podatke putem mreže te se stoga danas vrlo često koristi u razvoju aplikacija. Iako nije jedina, komunikacija putem mrežnih usluga je najčešća komunikacija između sustava u mikroservisnoj arhitekturi te joj treba posvetiti više pažnje.

4.1.1. Servisni ugovor

Servisni ugovor (eng. service contract) predstavlja dogovor između (najčešće distribuiranog) servisa i konzumenta toga servisa (klijenta) koji specificira dolazne i odlazne podatke zajedno s dogovorenim oblikom podataka (XML, JSON itd.) u kojem se podaci šalju ili zaprimaju. Servisni ugovor poput protokola definira pravila prema kojima će dva sudionika komunikacije komunicirati. Verzioniranje servisnih ugovora predstavlja izmjene u strukturi i sadržaju servisnog ugovora, a posljedica je proširenja mogućnosti servisa, a ono može biti homogeno i heterogeno. Slika 6. prikazuje shemu homogenog verzioniranja servisnih ugovora

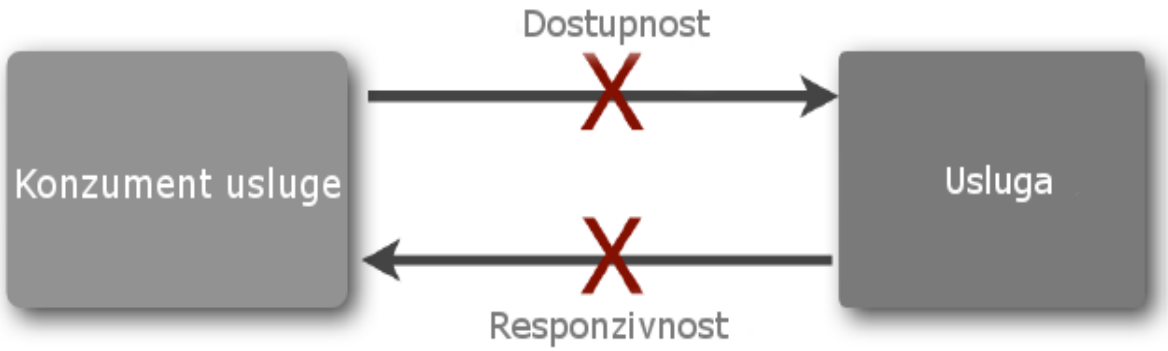
kod kojeg se inačica ugovora navodi u samom ugovoru, dok se kod heterogenog verzioniranja javljaju u potpunosti različiti ugovori (različiti po strukturi i sadržaju) te se inačica eksplicitno ne navodi u ugovoru.



Slika 6. Shema homogenog verzioniranja servisnih ugovora

4.1.2. Dostupnost servisa

Kada se govori o dostupnosti servisa valja napomenuti dva vrlo bitna pojma, a to su dostupnost i responzivnost. Dostupnost se odnosi na mogućnost uspostave komunikacije s servisom, a responzivnost se odnosi na mogućnost vraćanja odgovora od strane servisa (Slika 7.). Kod servisno-orijentiranih arhitektura praćenje dostupnosti sustava može se implementirati putem uzorka dizajna koji se naziva „Circuit breaker“. Ovaj uzorak dizajna ima ulogu u sustavu slično kao što i osigurač ima ulogu u strujnom krugu. Uloga ovoga uzorka dizajna je važna kod servisno-orijentiranih arhitektura iz razloga što ako u sustavu aplikacije jedan od sustava prestane s radom, tada se zamijeni s nekim drugim dostupnim sustavom te aplikacije nastavlja dalje s radom. Prestanak rada jednog od sustava moglo bi prouzročiti probleme u radu cjelokupnog sustava i problema što može dovesti do gubitka podataka ukoliko se sustav ne implementira ispravno.



Slika 7. Dostupnost i responzivnost servisa

4.1.3. Transakcije

Transakcije su prisutne u gotovo svim web aplikacijama, a svoju izuzetnu važnost imaju u poslovnim aplikacijama gdje se upravlja vrlo važnim i osjetljivim podacima gdje ne smije doći do pogreške tj. treba se očuvati konzistentnost podataka. Transakcija predstavlja skup operacija koje se izvršavaju kao jedna atomarna operacija gdje se podaci pohranjuju u bazu podataka tek nakon što su uspješno izvršene sve operacije od kojih je sačinjena transakcija. Kod aplikacija monolitne arhitekture, transakciju u bazi podataka je jednostavno izvršiti jer se cijela transakcija izvršava na jednom sustavu i jednoj bazi podataka. Kod aplikacija servisno-orijentirane arhitekture gdje se radi s distribuiranim sustavima, realizacija transakcije predstavlja prilično složen zadatak ukoliko su podaci pohranjeni na više različitih sustava. Uzorak dizajna koji rješava problem transakcija kod servisno-orijentiranih arhitektura naziva se Saga [21]. Objašnjenje rada ovoga uzorka može se objasniti kroz primjer klasične transakcije (u monolitnim arhitekturama) gdje kupac vrši narudžbu koja se sastoji od više stavaka te se nakon izvršene transakcije mijenja stanje blagajne. Svaki od sustava trebao bi bilježiti stanje vezano za narudžbu, pa bi tako svaki sustav nakon što obradi narudžbu trebao dodijeliti neku vrijednost ili zastavicu kako bi ostali sustavi mogli dalje obrađivati narudžbu u skladu s tom vrijednosti. Ukoliko neki od sustava prestane funkcionirati može se dogoditi da se transakcija nikada ne provede do kraja, pa čak može doći i do trajnog gubitka podataka.

4.2. Mikroservisna arhitektura

Iako je monolitna arhitektura danas vrlo popularna arhitektura izrade web aplikacija, ona sadrži i pojedine nedostatke koji mogu poslužiti kao prikladan uvod u opisu mikroservisne arhitekture. Rastom i razvojem web aplikacije te uključivanjem novih članova u tim ili timove razvoja programskog rješenja javljaju se mnogi problemi kao što su skaliranje i održavanje web aplikacije, što rezultira velikom kompleksnosti programskog koda te otežanim budućim održavanjem programskog koda i nadogradnjom aplikacije. U takvim slučajevima svoju prikladnu primjenu nalazi mikroservisna arhitektura čija je zadaća riješiti mnoge od prethodno navedenih problema monolitne arhitekture razvoja web aplikacije. Web aplikacija može se sastojati od mnogih sustava od kojih neki podupiru stvarne poslovne podsustave kao što su računovodstvo, prodaja, korisnička podrška i sl. Uloga mikroservisne arhitekture je da se web aplikacija „podijeli“ u manje dijelove tj. sustave koji su zaduženi za rješavanje specifičnih problema u aplikaciji.

Iako se vrlo često pojam servisno orijentiranih aplikacija poistovjećuje s mikroservisnim aplikacijama, valja napomenuti kako su to dva slična koncepta koji dijele mnoge zajedničke

karakteristike. Mikroservisna arhitektura je proizašla iz servisno orijentirane arhitekture, odnosno mikroservisna arhitektura je podskup servisno orijentiranih arhitektura (Slika 8.). Najvažnija zajednička karakteristika ovih arhitektura ja da svaka funkcija (servis) ima svoju odgovornost te svaka funkcija mora obavljati svoju zadaću potpuno neovisno o funkcijama ostalih sustava. Već ranije bila je opisana komponentna arhitektura koja se koristi kod izrade sustava aplikacije, i spomenuto je da je kod komponentne arhitekture u središtu promatranja neovisna i zamjenjiva komponenta. Kod servisno-orijentiranih arhitektura u središtu promatranja je sustav koji putem definiranih sučelja može komunicirati s okolinom te se mora moći zamijeniti s novim sustavom.

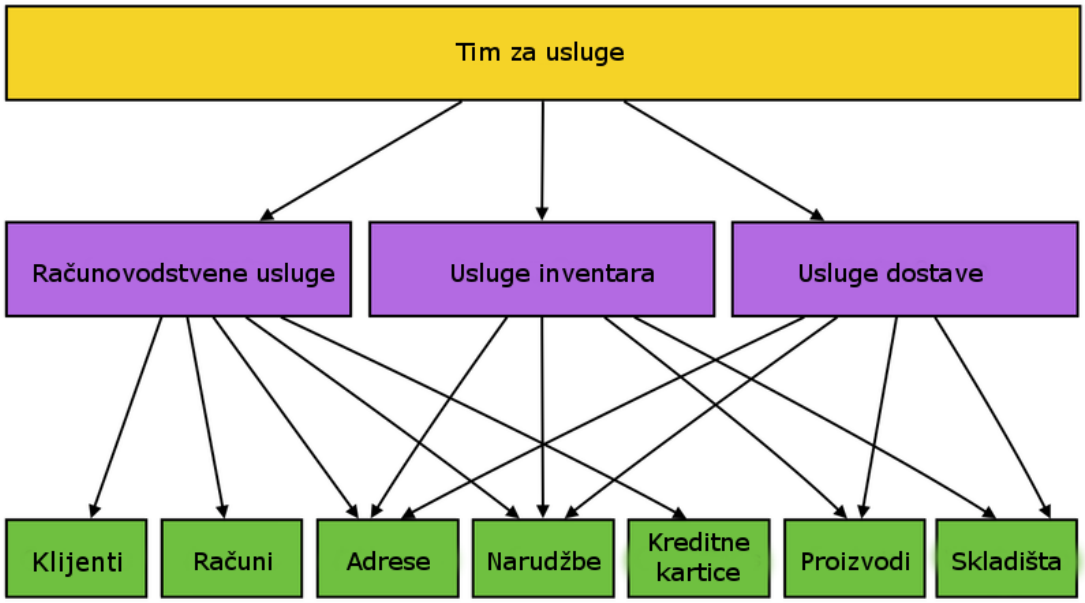
Slika 8. Odnos servisno orijentirane i mikroservisne arhitekture[7]

4.2.1. Dekompozicija sustava

Dekompozicija je proces kod kojeg se cjelina razlaže na manje dijelova i to je proces koji se vrlo često koristi u razvoju softverskog rješenja. Dekompozicijom monolitne softverske arhitekture dolazi se do višeslojnih arhitektura pa čak i do mikroservisne arhitekture. Unutar mikroservisne arhitekture postojeći se sustavi mogu dekomponirati na nove, manje sustave ovisno o potrebi. Slika 9. prikazuje primjer sheme dekompozicije sustava na manje sustave i to takozvanom „top-down“ metodom, gdje se kreće od najviše razine promatranja prema nižim razinama. Dekompozicija se provodi tako da je kohezivnost unutar sustava mikroservisne aplikacije što veća te da nema ovisnosti o nekom drugom, vanjskom sustavu. Nepisano je pravilo da ukoliko promjene na jednom sustavu stvaraju potrebu za promjenama na drugom

[7] https://dzone.com/articles/microservices-vs-soa-2

sustavu, potrebno pretvoriti ta dva sustav u jedan sustav. Prikazana slika predstavlja jedan od načina prikaza dekompozcije sustava na manje dijelove gdje se može uočiti da su podatkovni entiteti (na najnižoj razini promatranja) povezani s više različitih sustava odnosno servisa. Kada u implementaciji mikroservisne aplikacije dođe do ovakve situacije tada je potrebno sustave „grupirati“ u grupe tako da je povezanost unutar grupe što veća (kohezivnost), a povezanost s ostalim grupama što manja. Ukoliko bismo imali situaciju u sustavu kao što je prikazano na navedenoj slici, promjene na entitetu narudžba, zahtijevale bi promjene u programskoj logici svih sustava koji koriste entitet narudžbe.



Slika 9. Shema dekompozicije sustava[8]

4.2.2. Neovisni heterogeni sustavi mikroservisne arhitekture

Sustavi mikroservisne arhitekture na najvišoj razini promatranja mogu se tretirati kao „crne kutije“ (eng. black boxes) jer za određene ulazne parametre vraćaju odgovarajuće izlazne podatke što je analogno funkcijama u programskim jezicima ili komponentama u komponentnoj arhitekturi koja je prethodno opisana. Svaki sustav razvija jedan ili više članova neovisno o razvoju drugih sustava koji sačinjavaju aplikaciju. Kako se radi o paralelnom razvoju sustava, sustavi se razvijaju na brži način te su nove mogućnosti i unaprjeđenja prije dostupni krajnjem korisniku na korištenje.

Ovakav razvoj aplikacije naziva se agilni razvoj, a uključuje mnoge danas popularne metode kao što su: Scrum, Kanban, Lean, XP i sl. [33] Navedene metode razvoja se koriste kada u razvoju aplikacije sudjeluje veći broj članova, najčešće raspoređenih u timove. Ovi

[8] http://i.imgur.com/fF2X6HY.png?1

sustavi su heterogeni iz razloga što nije bitno kojim su tehnologijama izrađeni, pa tako sustavi mogu biti izrađeni u raznim programskim jezicima te mogu koristiti razne vrste baza podataka, što će biti prikazano i u praktičnom dijelu ovoga rada. Jedino što je bitno kod rada ovih sustava je njihova međusobna komunikacija koja je dobro definirana putem dobro definiranih protokola.

Neovisni heterogeni sustavi mikroservisne arhitekture u razvoj aplikacije uvode velike probleme kao što su uvođenje velike kompleksnosti u razvoj i održavanje aplikacije, povećana potražnja za novim za računalnim stručnjacima, uvođenje novih servisa, nadzor rada i održavanje paralelnih sustava i sl. Mikroservisna arhitektura ne određuje alate koji se koriste prilikom izrade aplikacije te stoga aplikacija može biti izrađena putem različitih tehnologija. Danas najpopularnije vrste baza podataka su relacijske baze podataka koje su pouzdane te pomoću kojih se mogu vrlo jednostavno i efikasno riješiti mnogi problemi vezani za pohranu i upravljanje podacima u današnjim složenim informacijskim sustavima.

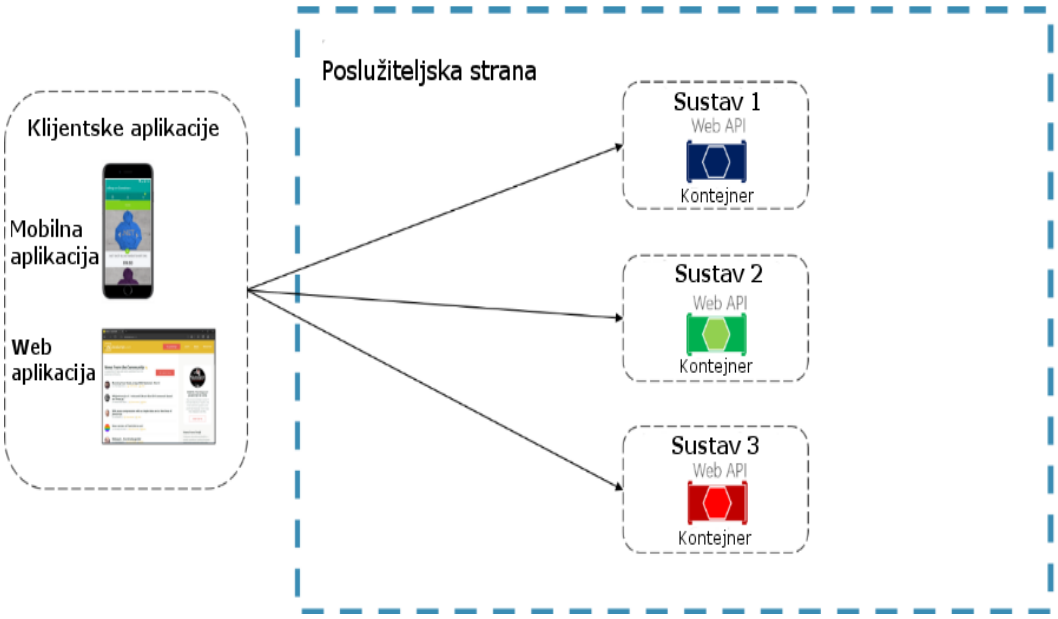
Relacijske baze podataka imaju i neke nedostatke kao što je rad s visoko povezanim podacima (ukoliko se koriste podaci koji su povezani s mnogim relacijama može doći zagušenja baze podataka) i rad s rekurzivnim problemima. Kod ovakvih problema softverski inženjeri se često okreću drugim, najčešće nerelacijskim bazama podataka, koje mogu ponuditi određena rješenja za ovakve vrste problema. U izradi aplikacije vezane za praktični dio ovoga rada korišteni su programski jezici: Javascript, Python, PHP i C# te su za pohranu podataka korištene baze podataka: SQLite (relacijska baza podataka) i Neo4j (nerelacijska baza podataka) kako bi se pokazala heterogenost korištenih tehnologija u sustavu.

4.2.3. Komunikacija između klijentske i poslužiteljske strane

Komunikacija između klijentske i poslužiteljske strane na području mikroservisne vrlo je složena tematika kada se implementiraju sustavi ove arhitekture. Kod monolitne arhitekture korisnički zahtjevi zaprimaju se u glavnom sustavu gdje se propagiraju do same aplikacije na tome sustavu, a aplikacija obrađuje te zahtjeve te vraća klijentskoj strani odgovarajuće odgovore. Kako se mikroservisna arhitektura sastoji od više sustava koji komuniciraju međusobno, ali i s klijentom, dolazi se do podjele na sustave mikroservisne arhitekture s izravnom ili neizravnom komunikacijom [5].

Slika 10. prikazuje izravnu komunikaciju između klijentske i poslužiteljske strane web aplikacije u mikroservisnoj arhitekturi. Kod izravne komunikacije svaki klijent aplikacije može

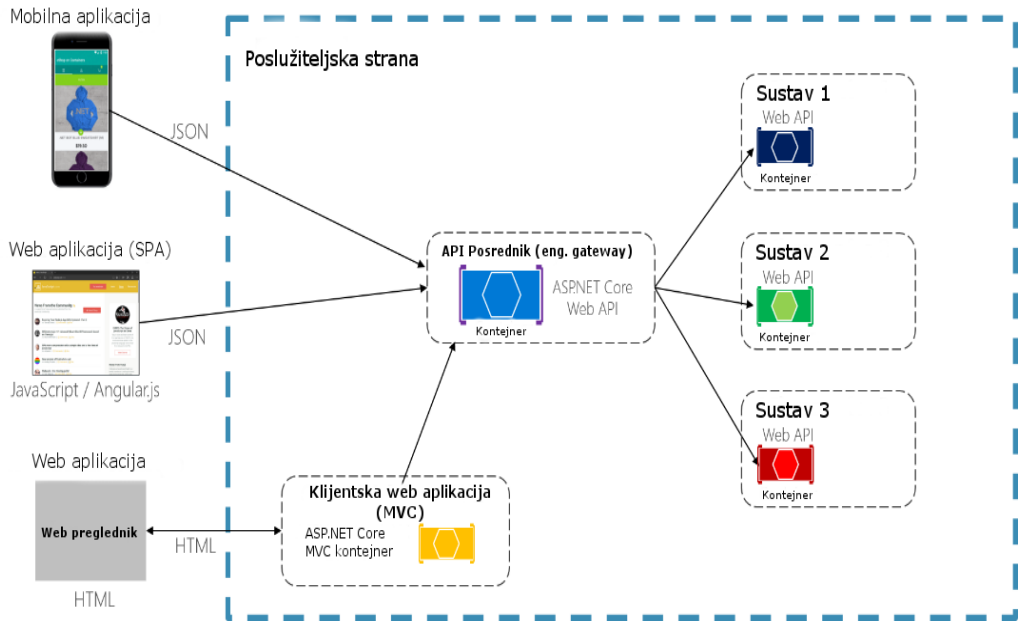
pristupiti bilo kojem sustavu, što znači da u komunikaciji nije potreban posrednik na poslužiteljskoj strani web aplikacije.



Slika 10. Izravna komunikacija u mikroservisnoj arhitekturi[9]

Slika 11. prikazuje neizravnu komunikaciju između klijentske i poslužiteljske strane kod web aplikacije izrađene uporabom mikroservisne arhitekture. Ovakav način komunikacije između sustava zahtijeva da sustav sadrži još jedan, tzv. „posrednički“ (eng. proxy) sustav koji zahtjeve korisnika prema određenim pravilima dijeli na ostale sustave arhitekture. Prednost ovakve vrste komunikacije je ta što nitko ne može izravno pristupiti ostalim sustavima (sigurnosni aspekt) aplikacije i posrednički sustav može obaviti potrebne operacije prije prosljeđivanja podataka ostalim sustavima ili prilikom vraćanja rezultata krajnjem korisniku. Ovakvom vrstom komunikacije može se lako postići tzv. efekt „uskog grla“ (eng. bottleneck effect) koji nastaje kada se posrednički sustav preopteretі s korisničkim zahtjevima te više nije u mogućnosti upravljati s novim korisničkim zahtjevima.

[9] https://docs.microsoft.com/en-us/dotnet/standard/microservices-architecture/architect-microservice-container-applications/media/image12.png



Slika 11. Neizravna komunikacija u mikroservisnoj arhitekturi[10]

4.2.4. Komunikacija između sustava mikroservisne arhitekture

Vrste komunikacija unutar aplikacija (između sustava unutar aplikacije) koje se baziraju na mikroservisnoj arhitekturi, mogu se promatrati prema dva smjera [5]. Prvi smjer može se proučavati kroz protokol koji govori da li je riječ o sinkronoj ili asinkronoj komunikaciji. Sinkrona komunikacija je komunikacija kod koje klijent pošalje zahtjev prema poslužitelju i čeka na odgovor kako bi dalje mogao ostvariti svoje zadatke. Sinkrona i asinkrona komunikacija u programskim jezicima odnose se na tijek izvršavanja programskog koda. Prethodno opisane vrste komunikacija (sinkrona i asinkrona) mogu se s razine programskog koda prenijeti i na razinu sustavu koje proučavamo u mikroservisnoj arhitekturi.

Asinkrona komunikacija je vrsta komunikacije kod koje klijent pošalje zahtjev prema poslužitelju te nastavlja svoj rad neovisno o odgovoru poslužitelja. Kada poslužitelj vrati klijentu odgovor, u sustavu se detektira taj događaj (eng. event) te se dalje u skladu s time izvršavaju operacije. Ove vrste komunikacija se u mikroservisnoj arhitekturi mogu realizirati na mnogo načina, a najčešći načini su putem: HTTP protokola, mrežnih utičnica (eng. web sockets) ili putem redova tj. AMQP protokola (eng. AMQP - Advanced Message Queuing Protocol). Sinkrona komunikacija upotrebljava se onda kada je tijek izvršavanja aplikacije definiran određenim operacijama koje se izvršavaju sekvencijalno (iduća operacija je ovisna o ishodu prethodne operacije). Iako je sinkrona vrsta komunikacije najčešća vrsta komunikacije

[10] https://docs.microsoft.com/en-us/dotnet/standard/microservices-architecture/architect-microservice-container-applications/media/image13.png

kada se govori o web aplikacijama, složene web aplikacije tzv. „Enterprise web applications“ najčešće koriste obje vrste komunikacija iz razloga što se najčešće na poslužitelju moraju izvršiti neke složene operacije kao što su: obrada videozapisa, slanje velikog broja poruka elektroničke pošte i sl. koje zahtijevaju mnogo vremena i računalnih resursa. Takve vrste složenih operacija su „kandidati“ za novi sustav iz razloga što se između ostaloga mogu lakše nadograditi tj. skalirati s računalnim resursima prema potrebi.

4.2.5. Kontinuirana implementacija i nadogradnja novih mogućnosti sustava

Aplikacija se tokom svoga životnog vijeka mijenja odnosno razvija, bilo da je riječ o popravku trenutnih pogrešaka na sustavu ili nadogradnji novih mogućnosti na postojećoj aplikaciji. Kada je riječ o stolnim ili mobilnim aplikacijama, tada se nova inačica podešava na način da se pokrene instalacijska izvršna datoteka koja najčešće potpuno zamijeni prethodno instaliranu aplikaciju. Kod takvih vrsta aplikacija korisnik nadogradnje aplikacije dobiva dosta rjeđe nego kod web aplikacija. Kada se govori o web aplikacijama implementacija novih mogućnosti je mnogo jednostavnija i češće se odvija iz razloga što se nove promjene odvijaju na programskom kodu koji se izvršava na poslužitelju ili se nova skripta s programskim kodom šalje klijentu gdje se izvršava u web pregledniku.

Kada se govori o kontinuiranom implementiranju novih mogućnosti i nadogradnji sustava kod ove arhitekture, posebnu pozornost valja posvetiti testiranju. Osobe koje sudjeluju u razvoju sustava ove arhitekture, moraju se pobrinuti da svaki sustav obavlja svoje zadaću neovisno o ispravnosti drugih sustava. Testiranje se najčešće vrši automatiziranim putem što znači da se za svaku funkcionalnost sustava pišu testovi koji se izvršavaju prije puštanja funkcionalnosti u produkciju. Razvoj svakog sustava je neovisan o razvoju drugih sustava te su nove mogućnosti znatno prije dostupne korisnicima nego što je to slučaj kod monolitne arhitekture.

4.2.6. Verzioniranje programskog koda aplikacije

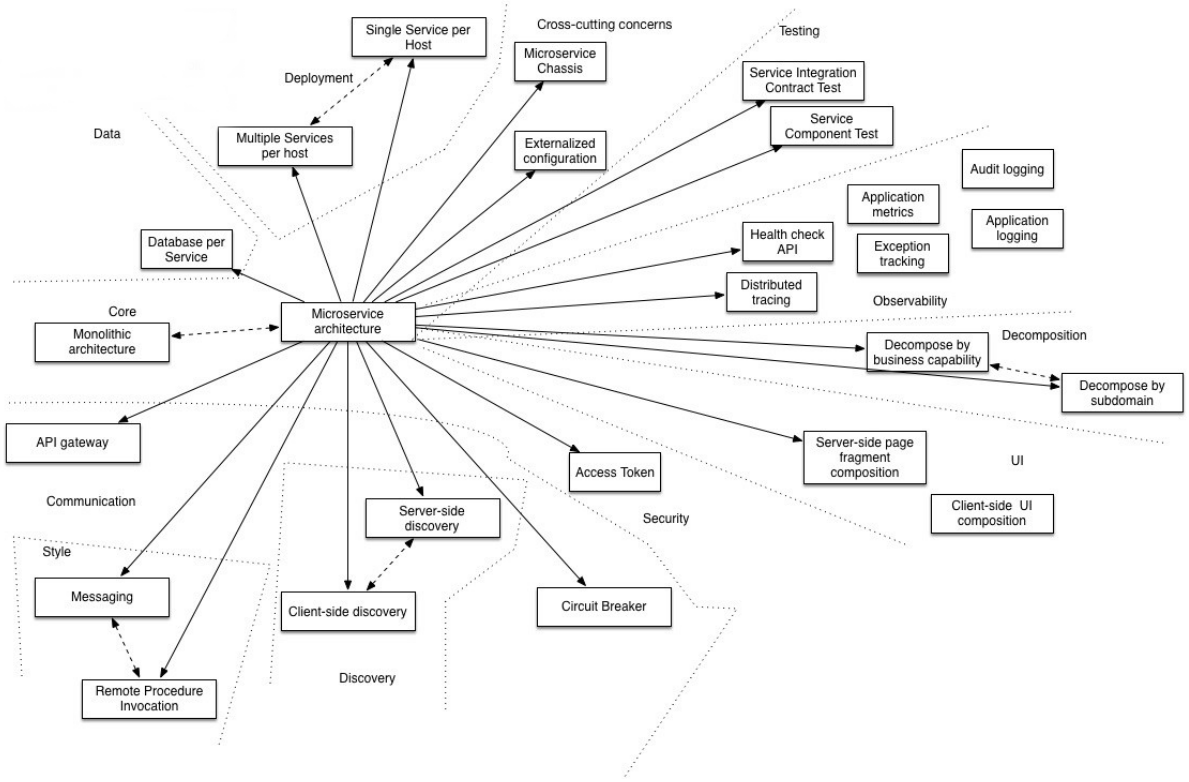
Verzioniranje je vrlo važan dio procesa izrade svakog programskog rješenja, naročito kada se govori o razvoju programskog rješenja u grupi od dvije ili više osoba. Proces verzioniranja ima dvostruku ulogu, jedna uloga odnosi se pohranu i trajno očuvanje programskog koda, dok se druga uloga odnosi dijeljenje i povezivanje programskog koda u timskom okruženje prilikom izrade dijelova programskog koda.

U klasičnim arhitekturama razvoja web aplikacija kao što je to slučaj kod monolitne arhitekture najčešće se izrađuje samo jedan repozitorij za verzioniranje unutar kojeg se nalazi

cjelokupni programski kod nad kojim radi cjelokupni tim osoba koje izrađuju programskog rješenje. Kada se govori o verzioniranju u mikroservisnoj arhitekturi, svaki podsustav bi trebao biti vezan za točno jedan repozitorij na kojem točno određena grupa osoba radi, kako bi razvoj svakog sustava bio što pregledniji i jednostavniji te kako bi se svaki sustav mogao što prije testirati i dati krajnjim korisnicima na konkretnu uporabu. Razvoj aplikacija u današnje vrijeme bio bi nezamisliv bez uporabe sustava za verzioniranje, a danas dva najpopularnija sustava za verzioniranje programskog koda su Bitbucket i Github.

4.2.7. Uzorci dizajna mikroservisne arhitekture

Kao što je već prethodno objašnjeno, uzorak dizajna predstavlja obrazac kojim se nastoji riješiti određeni problem na neki „preporučeni“ način. Slika 12. prikazuje podjelu uzoraka dizajna mikroservisne arhitekture prema definiranim kriterijima kao što su: sigurnost, komunikacija, testiranje, dekompozicija dijelova sustava i sl. Iako se na slici može uočiti veliki broj različitih skupina uzoraka dizajna, valja napomenuti kako je ovo relativno mlada arhitektura te se s novim alatima razvijaju i novi uzorci dizajna koji najčešće proizlaze iz stvarnih problema s kojima se današnje velike organizacije susreću.



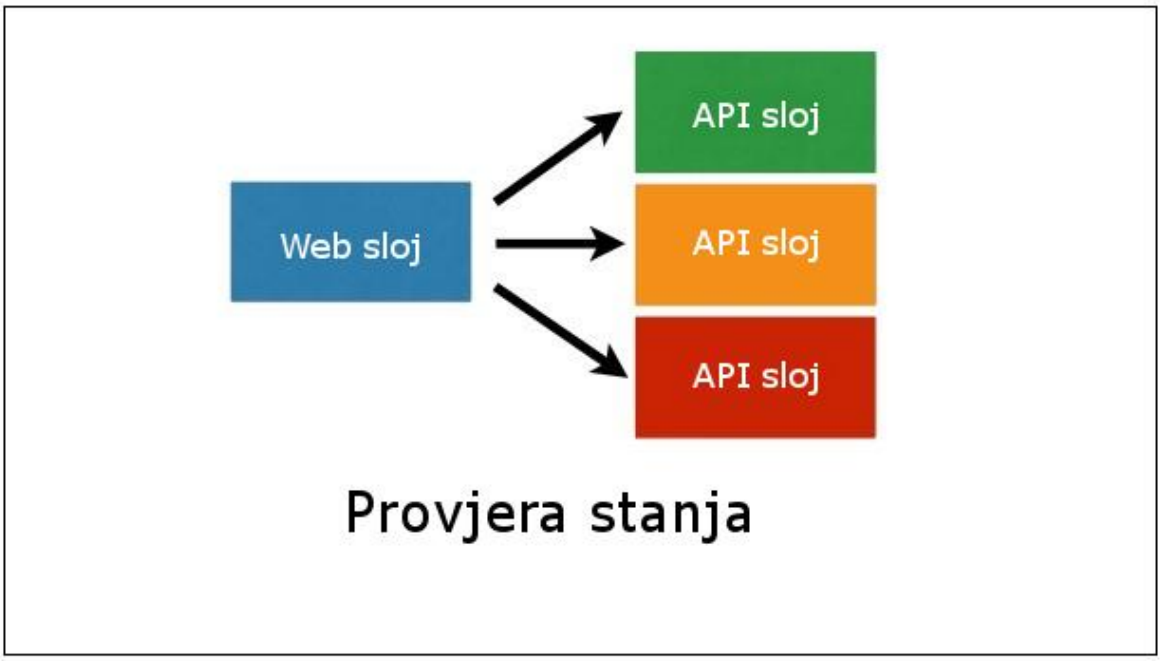
Slika 12. Uzorci dizajna mikroservisne arhitekture[11]

[11] http://microservices.io/i/PatternsRelatedToMicroservices.jpg

Pa tako na primjer, uzorci dizajna za sigurnost odnose se na uzorke koji osiguravaju sigurnost i zaštitu podataka korisnika te autentifikaciju i autorizaciju korisnika u sustavu. Kao primjer ove vrste uzoraka dizajna može se opisati „API access token“ koji predstavlja uzorak kojim se osigurava praćenje korisnika aplikacije tijekom njegovog rada u sustavu. Prilikom prijave u sustav korisniku se prosljeđuje jedinstveni žeton koji se sastoji od niza nasumično poredanih znakova koji predstavljaju korisnikovu aktivnost u sustavu aplikacije. Svaki od sustava mikroservisne aplikacije na temelju žetona može doći do korisnikovih podataka te može obaviti proces autentifikacije i autorizacije korisnika. Ovakav način autentifikacije korisnika prikazan je i u dijelu ovoga rada koji prikazuje aplikaciju izrađenu uporabom ove mikroservisne arhitekture. Broj znakova koji određuju žeton određuje i sigurnost žetona, što je veći broj znakova sadržan u žetonu to je veća sigurnost odnosno „neprobojnost“ korisnikovog žetona.

Uzorci dizajna za korisničko sučelje služe za rješavanje problema na relaciji sučelja aplikacije i poslužiteljskoj strani aplikacije. Osim računala, web aplikacija može zaprimati zahtjeve od raznih uređaja kao što su mobitel, tablet, pa čak i televizija u današnje vrijeme te se ponekad javlja potreba da se svaki od tih zaprimljenih zahtjeva proslijedi drugačijem servisu ovisno o odgovoru koji treba biti proslijeđen krajnjem uređaju. Također, pomoću ovih uzoraka se planira složenost klijentske strane aplikacija, odnosno procjenjuje se kolika će biti složenost obrade podataka na klijentskoj strani aplikacije te se procjenjuje koji će se sve podaci slati klijentskoj strani.

U skupinu uzoraka dizajna za osmotrivost ubrajamo uzorke za zapisivanje aktivnosti aplikacije, uzorke za praćenje stanja dostupnosti servisa, uzorke za praćenje rada sustava itd. Kao primjer uzorka dizajna ovoga sustava može se uzeti tzv. „Health check API“ uzorak (Slika 13.), čija je uloga praćenje stanja sustava aplikacije odnosno praćenje njihove propusnosti. Već je ranije bio spomenut problem tzv. „uskog grla“ gdje na jedan poslužitelj dolazi previše zahtjeva te nije u mogućnosti obraditi sve pristigle zahtjeve. Ovakav uzorak dizajna također ima i zadaću obavijestiti osobu koja nadgleda rad sustava da pojedini sustav nije u funkciji te zapisivati aktivnosti rada sustava mikroservisne aplikacije. Ovi uzroci dizajna u mikroservisnoj arhitekturi neophodni su kada aplikacija sadrži veliki broj sustava koje jedna ili više osoba nisu u mogućnosti nadgledati te kada je potrebno upravljati automatskim putem s jednog centralnog mjesta.

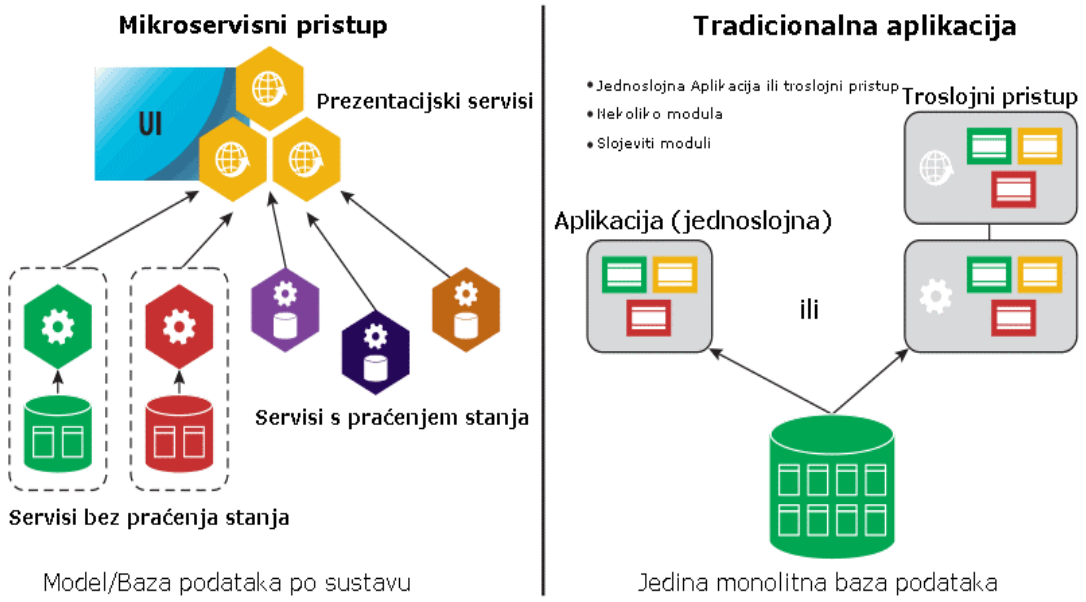


Slika 13. Shema Health check API-a[12]

4.2.8. Usporedba monolitne i mikroservisne arhitekture

Nakon prethodno opisanih karakteristika mikroservisne arhitekture, a i općenito servisne arhitekture, za kraj ovoga poglavlja mogu se istaknuti razlike između danas popularne monolitne arhitekture i mikroservisne arhitekture. Slika 14. prikazuje usporedbu shema aplikacija monolitnih i mikroservisnih arhitektura pri izradi web aplikacija.

[12] https://image.slidesharecdn.com/untitled-150309081332-conversion-gate01/95/consul-serviceoriented-at-scale-20-638.jpg?cb=1425888990



Slika 14. Usporedba mikroservisne arhitekture i monolitne arhitekture[13]

Kao što se može vidjeti s priložene slike, lijeva strana prikazuje shemu web aplikacije koja je izrađena uporabom mikroservisnog pristupa tj. mikroservisne arhitekture, dok desna strana slike prikazuje shemu tradicionalne aplikacije izrađene putem monolitne arhitekture. Prema priloženoj shemi može se uočiti kako kod monolitne arhitekture cijeli sustav komunicira s jednom bazom podataka dok se kod mikroservisnog pristupa može uočiti kako svaki sustav tj. servis ima svoju bazu podataka. Također, s priložene slike može se uočiti kako su kod mikroservisnog pristupa sustavi neovisni (ne postoji direktna povezanost u ovom slučaju), dok se kod monolitne arhitekture može uočiti jedan sustav (jednoslojna aplikacija) ili aplikacija od više slojeva kod koje su dijelovi čvrsto povezani.

Monolitna aplikacija sadrži funkcionalnosti vezane za domenu (poslovnu logiku) koje su smještene na jednom mjestu, a te su funkcionalnosti podijeljene kroz funkcijske slojeve kao što su prezentacijski sloj, poslovni sloj i podatkovni sloj. Može se reći da je mikroservisna arhitektura „proizvod“ dekompozcije monolitne arhitekture na manje, neovisne sustave koji djeluju kao jedna cjelina. Ono što je zajedničko za sve servisno-orijentirane arhitekture je to da su distribuirane arhitekture, što znači da im se pristupa putem nekog vanjskog sustava. Prilikom implementacije sustava mikroservisne arhitekture potrebno je osigurati da svi sustavi rade ukoliko jedan od sustava prestane s radom, što predstavlja vrlo težak zadatak za osobe koje implementiraju ovu arhitekturu.

[13] https://i-msdn.sec.s-msft.com/dynimg/IC838040.png

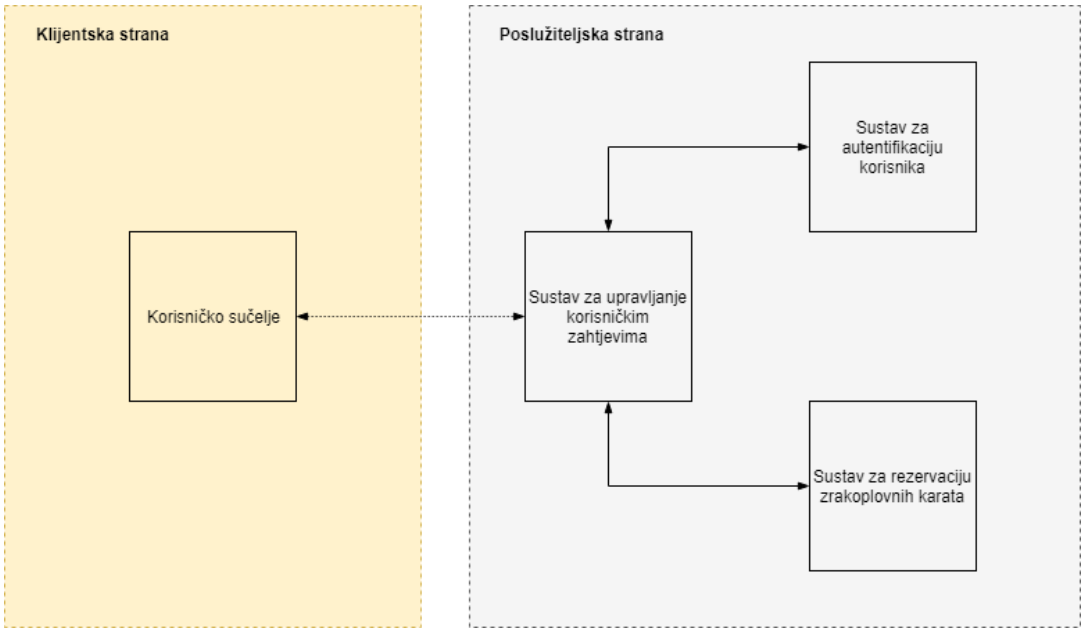
5. Praktični dio rada

5.1. Opis aplikacije

U ovome poglavlju biti će prikazana aplikacija koja je sastavljena uporabom prethodno opisane mikroservisne arhitekture. Slika 15. prikazuje opći tj. „grubi“ opis sustava čija je implementacija realizirana kroz ovaj rad putem mnogih, danas popularnih tehnologija za izradu web aplikacija. Web aplikacija prikazana u ovome radu može se kao i mnoge druge web aplikacije podijeliti na dva dijela: korisnički i poslužiteljski dio. Svaki pravokutnik s Slika 15. predstavlja jedan sustav mikroservisne arhitekture kojem je cilj rješavanje složenih problema u određenoj domeni. Navedena aplikacija predstavlja samo model, odnosno pojednostavljenu inačicu aplikacije ove tematike, što znači da su u stvarnosti web aplikacije ove tematike daleko složenije te sustavi takvih web aplikacija zahtijevaju daleko veću pozornost prilikom analize i izrade.

Lijeva strana slike prikazuje sustav korisničkog sučelja koji predstavlja najbitniju komponentu aplikacije za krajnjeg korisnika koji se služi web aplikacijom. Sama kompleksnost poslužiteljske strane u potpunosti je „skrivena“ od krajnjeg korisnika, što znači da korisnik uopće ne zna za procese i aktivnosti koje se izvršavaju na poslužiteljskoj strani web aplikacije. Prethodno navedeni sustavi biti će pobliže opisani tokom rada te će njegovi dijelovi biti opisani i programskim isječcima putem kojih su izrađeni.

Desna strana slike prikazuje poslužiteljsku stranu web aplikacije, a ova strana aplikacije opisana je putem tri pravokutnika koji se nalaze unutar jednog većeg pravokutnika. Veći pravokutnik predstavlja poslužiteljsku stranu sustava, dok ostali pravokutnici predstavljaju podsustave poslužiteljske strane aplikacije. Linije između pravokutnika prikazuju dvosmjernu komunikaciju između sustava te se još može vidjeti i povezanost sustava za upravljanje korisničkim zahtjevima s korisničkim sučeljem odnosno klijentskom stranom aplikacije. Kao što su već prethodno opisani načini komunikacije između klijenta i poslužitelja u mikroservisnoj arhitekturi (poglavlje 4.2.3.), u praktičnom djelu ovoga rada implementirana je komunikacija između klijenta i poslužitelja na način da se komunikacija odvija putem posrednika.



Slika 15. Opći prikaz sustava aplikacije

5.2. Opis alata korištenih u izradi aplikacije

Kao što je već prethodno navedeno, web aplikacije čija se tematika veže za planiranja putovanja i zrakoplovnih karata daleko su složenije od web aplikacije prikazane u ovome radu te se njihova kompleksnost njihovih dijelova često rješava na više različitih načina. Mikroservisna arhitektura ne određuje alate koji se koriste prilikom izrade aplikacije te stoga aplikacija može biti izrađena putem različitih tehnologija. Danas najpopularnije vrste baza podataka su relacijske baze podataka koje su pouzdane te pomoću kojih se mogu vrlo jednostavno i efikasno riješiti mnogi problemi vezani za pohranu i upravljanje podacima u današnjim informacijskim sustavima.

Relacijske baze podataka imaju i neke nedostatke kao što je rad s visoko povezanim podacima (ukoliko se koriste podaci koji su povezani s mnogim relacijama može doći zagušenja baze podataka) i rad s rekurzivnim podacima. Kod ovakvih problema osobe koje razvijaju aplikaciju često se okreću drugim, najčešće nerelacijskim bazama podataka, koje mogu ponuditi određena rješenja za ovakve vrste problema. U idućim potpoglavljima biti će opisani specifični programski jezici i baze podataka koje su bile korištene prilikom realizacije programskog rješenja vezanog za ovaj rad.

5.2.1. Programski jezik PHP i okvir CakePHP

Programski jezik PHP je objektno-orijentirani skriptni jezik koji je nastao 1994.godine od strane Rasmusa Lerdorfa, a namijenjen je izradi dinamičkih web aplikacija. Ovaj je skriptni jezik otvorenog koda i u potpunosti je besplatan te se može postaviti na gotove sve, danas najpopularnije poslužitelja kao što su Apache, Nginx i sl. Danas se u izradi web aplikacija često koristi već spomenuti uzorak dizajna MVC, pa i okvir (eng. framework) koji se koristi uz ovaj programski jezik u ovome radu koristi taj uzorak dizajna. Ovaj okvir za izradu web aplikacija u potpunosti je otvorenog koda te je namijenjen brzom i jednostavnom razvoju aplikacija. Trenutna stabilna inačica ovoga skriptnog jezika, u vrijeme pisanja ovoga rada je inačica 7.2. Osim okvira „CakePHP“, u ovome programskom jeziku, danas popularni okviri koji koriste MVC uzorak dizajna su još i: „Symfony“, „Laravel“, „FuelPHP“, „Phalcon“ itd.

5.2.2. Programski jezik Python i alata Flask

Programski jezik Python je objektno-orijentirani programski jezik opće namijene koji se koristi za razvoj raznih programskih rješenja, pa između ostaloga i za razvoj web aplikacija (uz odgovarajuće dodatke). Python je programski jezik visoke razine koji je nastao 1990.g, a autor ovoga programskog jezika je Guido Van Rossum. Programski jezik Python osim objektno-orijentirane paradigme, podržava i proceduralnu te imperativnu paradigmu, a jezik je u potpunosti besplatan te je otvorenog koda (eng. open-source) [9]. Flask predstavlja modul kojim se u ovome programskom jeziku mogu izrađivati web aplikacije. Ovaj modul omogućava rad s mnogim relacijskim i nerelacijskim bazama podataka te omogućava rad s svim standardnim modulima programskog jezika Python. Jedan od nedostatak programskog jezika Python je taj što je to kao i PHP skriptni jezik što znači da se linije programskog koda izvršavaju jedna po jedna te se uz to provjerava valjanost programskog kod što znatno utječe na brzinu izvođenja programa u odnosu na kompajlerske jezike.

5.2.3. Programski jezik C# i .NET okvir

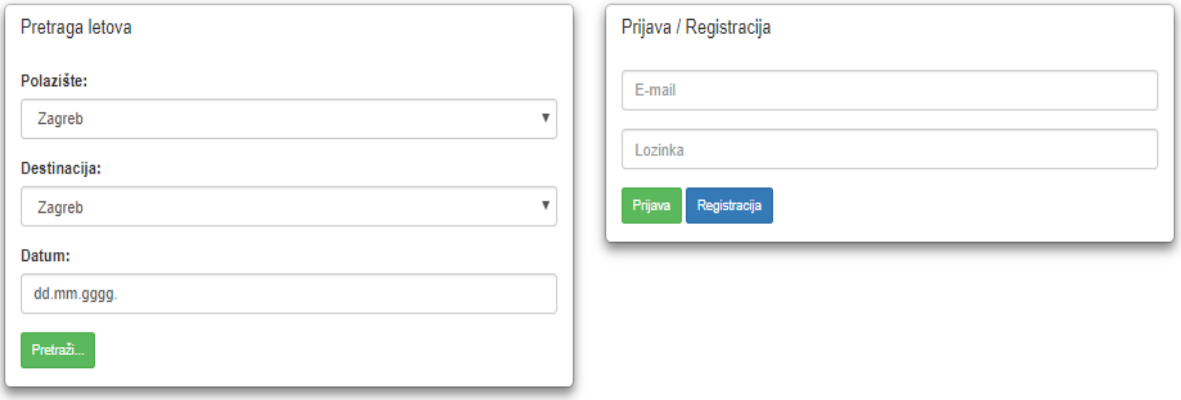
Programski jezik C# je objektno-orijentirani programski jezik za razvoj aplikacija za Windows operativni sustav. Projekt „.NET core 2.0“ predstavlja besplatni okvir otvorenog koda koji se može koristiti na svim danas najpopularnijim operacijskim sustavima (Windows, Linux i Mac) [2]. U odnosu na prethodno opisane programske jezike (Python i PHP), C# je programski jezik koji u svojoj „pozadini“ sadrži veliki broj gotovih alata i mogućnosti koje su uključene u sam .NET okvir što predstavlja vrlo snažnu podlogu za razvoj aplikacija. Web aplikacije koje

se izrađuju putem ovoga okvira i ovoga programskog jezika mogu se smjestiti na web poslužitelje IIS, Nginx, Apache i sl.

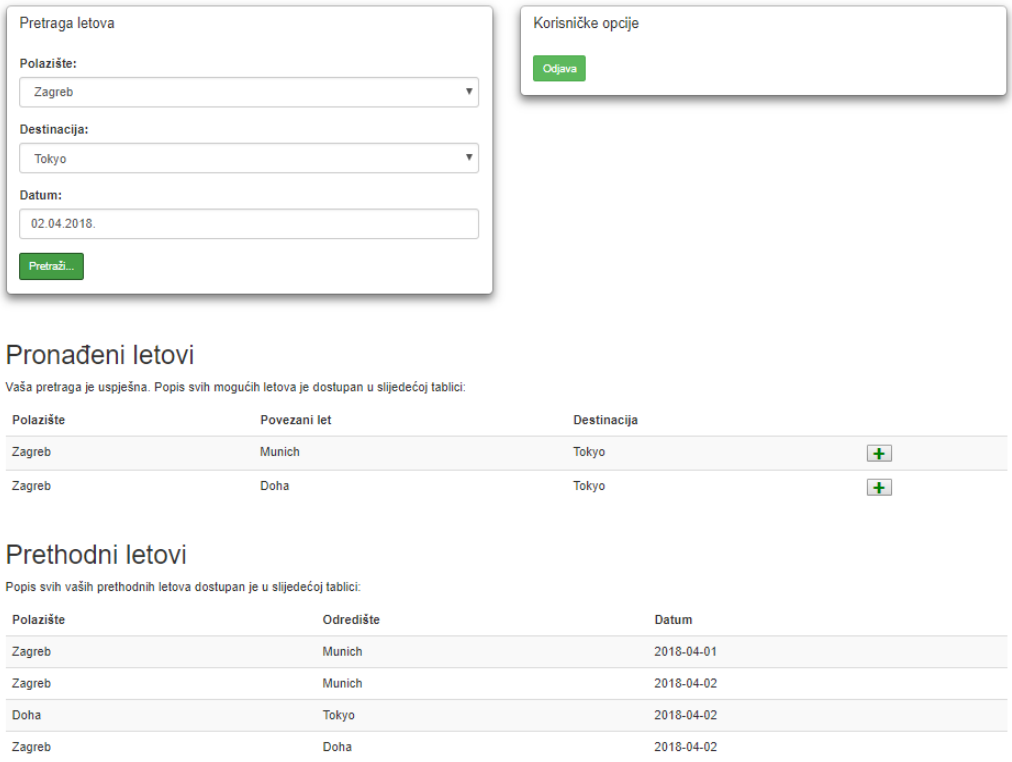
5.3. Korisničko sučelje aplikacije

Kao što je već navedeno, korisničko sučelje ove web aplikacije za krajnjeg korisnika predstavlja najvažniji dio aplikacije putem kojeg korisnik ima kontakt s cjelokupnim sustavom odnosno web aplikacijom. Korisničko sučelje koje se veže za ovaj rad prikazano je na Slika 16., a može se podijeliti u dva djela: dio za prijavu korisnika u sustav i dio za pretragu letova i rezervaciju zrakoplovnih karata.

Razlika između prijavljenog i neprijavljenog korisnika u ovoj aplikaciji je ta što prijavljeni korisnik nakon pretrage letova ima mogućnost rezervacije leta i svih povezanih letova. Prijavljeni korisnik također može vidjeti sve letove koje je prethodno rezervirao. Korisničko sučelje aplikacije komunicira s poslužiteljskom stranom putem posrednika, tj. s sustavom za upravljanje korisničkim zahtjevima koje će biti opisano u idućem poglavlju. Slika 16. prikazuje dio korisničkog sučelja koje se odnosi na neprijavljenog korisnika, dok Slika 17. prikazuje dio korisničkog sučelja koje se odnosi na prijavljenog korisnika koji je prethodno napravio pretragu letova. Korisnik koji je prijavljen u sustav, nakon pretrage letova može napraviti i rezervaciju leta klikom na gumb u tablici s pronađenim letovima (Slika 17.).



Slika 16. Korisničko sučelje aplikacije - neprijavljeni korisnik



Slika 17. Korisničko sučelje aplikacije – prijavljeni korisnik i pretraga letova

Korisničko sučelje aplikacije sadrži implementiranu funkcionalnost praćenja i održavanja sesije s sustavom aplikacije, a implementirano je na način da se žeton (niz nasumično poredanih znakova dodijeljen klijentskoj aplikaciji) pohranjuje u podatkovni prostor web preglednika koji korisnik koristi. Korisnik koji nije prijavljen u sustav aplikacije nema žeton za praćenje sesije te prema tome nema dostupne sve mogućnosti ove aplikacije.

Kada se neprijavljeni korisnik prijavi u sustav aplikacije, dodijeli mu se žeton za praćenje sesije te žeton vrijedi 40 minuta. Nakon što istekne navedeno vrijeme sesije, klijent aplikacije se mora ponovo prijaviti u sustav aplikacije s svojim korisničkim podacima. Prilikom svakog zahtjeva upućenog sustavu za upravljanje korisničkim zahtjevima, provjerava se da li web preglednik ima zapisan žeton o praćenju sesije, te ukoliko je zapisan, uključuje se u zahtjev kao dodatni parametar. Sustav za upravljanje korisničkim zahtjevima kod pojedinih akcija na početku pokreće operaciju provjere autentifikacije korisnika tako što komunicira s sustavom za autentifikaciju korisnika. Sustav za upravljanje korisničkim zahtjevima prosljeđuje žeton za praćenje sesije sustavu za autentifikaciju korisnika koji na temelju žetona vraća identifikacijski broj korisnika u sustavu.

5.4. Sustav za upravljanje korisničkim zahtjevima

Uloga sustava za upravljanje korisničkim zahtjevima je da na poziv određene metode (funkcije) proslijedi zahtjev drugim sustavima aplikacije s obzirom na cilj. Glavna komunikacijska zamisao ove aplikacije je ta da sustavi aplikacije komuniciraju međusobno bez korisnikovog direktnog pristupa (osim u slučaju ovoga sustava). Sustav za upravljanje korisničkim zahtjevima je zapravo posrednik (eng. proxy) koji ciljano komunicira s sustavima aplikacije na korisnikov zahtjev putem grafičkog sučelja aplikacije. Kao što je već prethodno opisano, ovakav način komunikacije između klijentskog uređaja i web aplikacije ima svoje prednosti i nedostatke. Ovaj sustav pisan je u programskom jeziku PHP (eng. PHP – Hypertext Preprocessor), uporabom dodatnog alata, odnosno MVC okvira CakePHP inačice 3.

Tablica 2. sadrži sve javne URL adrese pomoću kojih se ostvaruje funkcionalnost ovoga sustava. Ukoliko se pokuša pristupiti nekoj adresi koja nije navedena u priloženoj tablici, zahtjev će biti odbijen. Od navedenih URL adresa, jedino resursi tj. adrese: „/bookFlight“,“/logout“ i „/bookedFlights“ zahtijevaju da je korisnik prethodno prijavljen u sustavu web aplikacije.

Tablica 2. Popis URL-ova za pristup sustavu za upravljanje korisničkim zahtjevima

URL	Poziv metode
/login	Poziva se metoda „login“ u sustavu za upravljanje korisnicima.
/register	Poziva se metoda „register“ u sustavu za upravljanje korisnicima.
/authenticate	Poziva se metoda „authenticate“ u sustavu za upravljanje korisnicima.
/logout	Poziva se metoda „logout“ u sustavu za upravljanje korisnicima.
/getCities	Poziva se metoda „GetCities“ u sustavu za rezervaciju zrakoplovnih karata.

/getFlights	Poziva se metoda „GetFlights“ u sustavu za rezervaciju zrakoplovnih karata.
/bookFlight	Poziva se metoda „GetFlights“ u sustavu za rezervaciju zrakoplovnih karata.
/bookedFlights	Poziva se metoda „BookedFlights“ u sustavu za rezervaciju zrakoplovnih karata.

5.5. Sustav za autentifikaciju korisnika

Sustav za autentifikaciju odnosno prijavu korisnika u sustav je relativno jednostavan sustav praktičnog dijela ovoga rada. Uloga ovoga sustava je da prema dobivenim podacima autentificira tj. prijavi korisnika u sustav web aplikacije. Autentifikacija je proces čija je zadaća provjeriti identitet korisnika, dok je autorizacija proces koji ispituje da li korisnik ima pravo pristupa za resurs kojem želi pristupiti. U aplikaciji ovoga rada postoje samo dvije uloge (prijavljeni i neprijavljeni korisnik) te stoga prijavljeni korisnik ima sve moguće ovlasti te se ne treba autorizirati njegova uloga prije pristupa svakom pojedinom resursu aplikacije. U stvarnoj primjeni, ovakav sustav bi osim autentifikacije korisnika imao i mogućnosti slanja obavijesti korisnicima, izmjene korisničkih podataka, izrade raznih statistika na temelju korisnikovog rada u aplikacija i sl.

Navedeni sustav služi kao servis čijim javnim metodama mogu pristupiti svi sustavi unutar aplikacije. Kao što je već i ranije navedeno, prednost mikroservisne arhitekture je ta što sustavi mogu obavljati svoje zadaće neovisno o drugim sustavima, pa tako i ovaj sustav može obavljati svoju zadaću (autentifikaciju) korisnika neovisno o drugim sustavima. Ovaj sustav pisan je u programskom jeziku Python, a za pohranu podataka u pozadini ima relacijsku bazu podataka SQLite. Tablica 3. prikazuje popis svih javnih URL-ova te metode koje se pozivaju prilikom poziva u ovome sustavu.

Tablica 3. Popis URL-ova za pristup sustavu za autentifikaciju korisnika

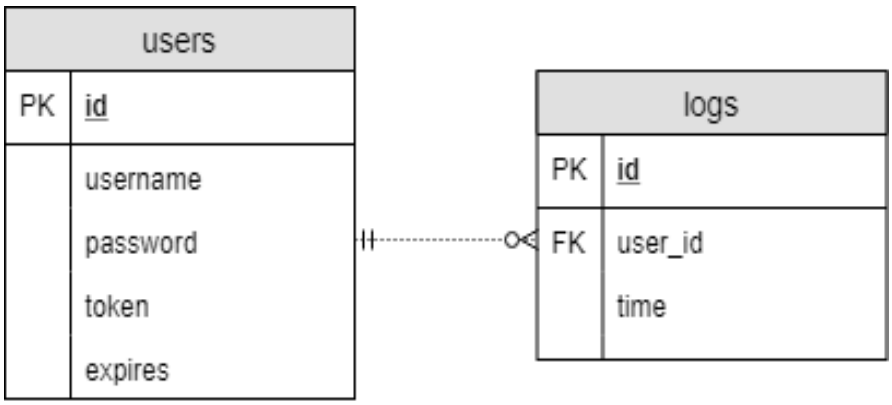
URL	Opis	Poziv metode i parametri
/login	Prijava korisnika u sustav na temelju korisničkog imena i lozinke (autentifikacija)	Poziva se metoda „login“ koja na ulazu prima tzv. JSON oblik podataka s autentifikacijskim podacima korisnika. Kao odgovor vraća JSON oblik podataka s odgovarajućom porukom, statusom operacije i žetonom za praćenje sesije.
/register	Registracija novog korisnika u sustav. Operacija obuhvaća i validaciju podataka novog korisnika i provjeru da li korisnik već postoji u sustavu.	Poziva se metoda „register“ koja na ulazu prima tzv. JSON oblik podataka koji sadrži podatke za izradu novog korisnika. Na izlazu vraća status operacije registracije.
/authenticate	Metoda koju pozivaju svi sustavi prije pristupa zaštićenim resursima (npr. rezervacija letova).	Poziva se metoda „authenticate“ koja na ulazu prima tzv. JSON oblik s žetonom sesije, a kao izlaz se vraća JSON oblik podataka koji sadrži rezultat operacije autentifikacije.
/logout	Na temelju žetona sesije brišu se podaci korisnikove sesije u sustavu.	Poziva se metoda „logout“ koja na ulazu prima tzv. JSON oblik podataka s žetonom sesije, a kao izlaz se vraća JSON oblik podataka koji sadrži rezultat operacije odjave korisnika.

Treba napomenuti da bi se u stvarnim uvjetima rada, kod ovakve vrste sustava trebalo osigurati da je veza dobro zaštićena što se može postići uporabom tzv. SSL (eng. SSL - Secure socket layer) certifikata. Ovakva vrsta sustava u svojoj komunikaciji s klijentom razmjenjuje

vrlo osjetljive podatke kao što su korisničko ime i lozinka, a mogu biti i podaci kreditnih kartica ili neki drugi osobni podaci korisnika. Kao što je već ranije napomenuto, sustavi u ovome radu komuniciraju preko posrednika (sustava za upravljanje korisničkim zahtjevima) te unutar poslužiteljske strane nije potrebna jaka zaštita jer vanjski sustavi ne mogu pristupiti izravno sustavima poslužiteljske strane web aplikacije.

Ovaj je sustav usko povezan s korisnikom i svim poslovima vezanim za korisnika, no postavlja se pitanja što se od podataka koji se vežu za pojedinog korisnika nalazi u drugim sustavima? Prema glavnim svojstvima koje opisuju mikroservisnu arhitekturu, svaki sustav bi trebao imati svoj trajni izvor podataka gdje pohranjuje i obrađuje podatke koji su vezani za sustav. U implementaciji ove aplikacije, sustav za rezervaciju zrakoplovnih karata prilikom pohrane rezervacije očekuje i identifikacijski broj korisnika kako bi mogao napraviti rezervaciju letu. Identifikacijski broj korisnika nalazi se u sustavu za autentifikaciju korisnika te nije izravno dostupan sustavu za rezervaciju zrakoplovnih karata.

Pošto se komunikacija između sustava ove aplikacije vrši putem posrednika, posrednik je zadužen da na temelju identifikacijskog žetona (eng. token) dohvati identifikacijski broj korisnika te da ga prilikom pohrane rezervacije proslijedi sustavu za rezervaciju zrakoplovnih karata. Sustav za organizaciju rezervaciju zrakoplovnih karata pohranjuje rezervaciju, ali pritom ne provjerava da li korisnik zaista postoji već pohranjuje korisnika na temelju „povjerenja“. Naravno, u ovakvim slučajevima se pretpostavlja da je sustav uključen u skrivenu tj. privatnu mrežu kojoj može pristupiti isključivo posrednik. Podatkovni model ovoga sustava prikazan je na Slika 18.

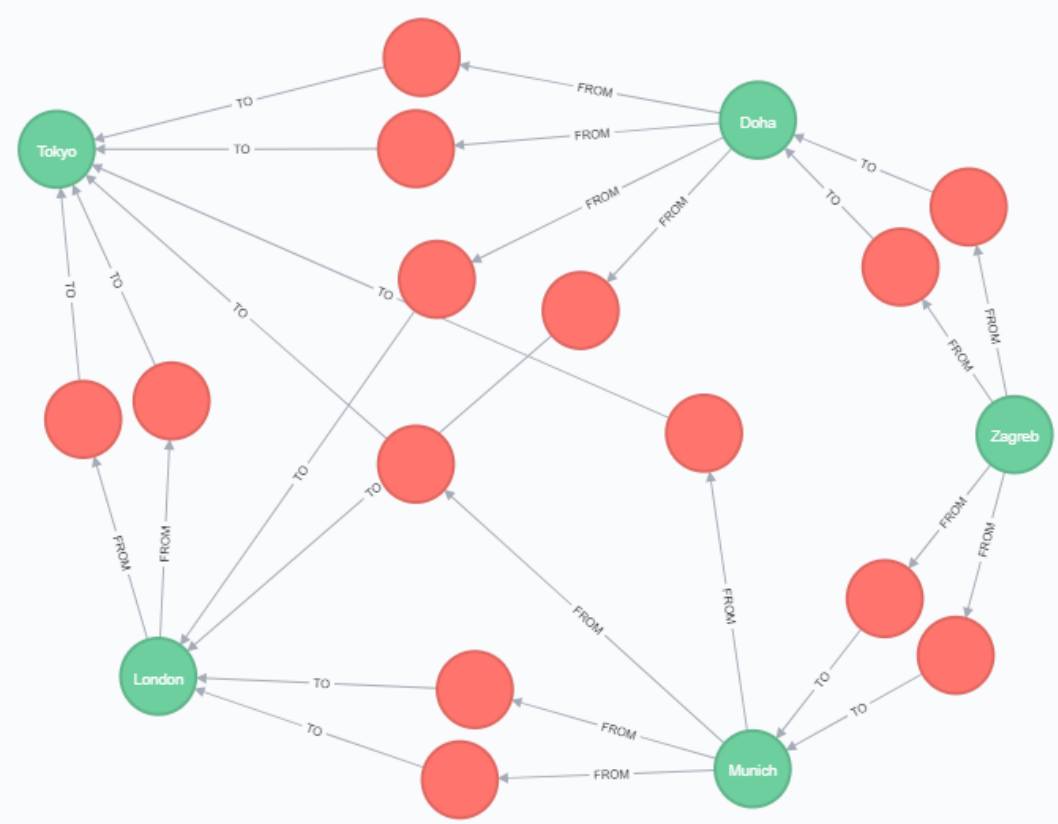


Slika 18. Podatkovni model sustava za autentifikaciju korisnika

5.6. Sustav za rezervaciju zrakoplovnih karata

Sustava za rezervaciju zrakoplovnih karata predstavlja najsloženiji dio praktičnog dijela ovog rada. Ovaj sustav pisan je u programskom jeziku C# te kao bazu podataka u pozadini sadrži nerelacijsku bazu podataka Neo4j. Baza podataka Neo4j predstavlja bazu čiji se rad temelji na grani matematike koja se naziva teorija grafova te svojim radom može brzo i efikasno riješiti probleme koje nerelacijske baze podataka ne mogu riješiti ili ne mogu riješiti u zadovoljavajućem vremenskom periodu.

Podatkovni model koji se koristi u ovome sustavu za rezervaciju zrakoplovnih karata prikazan je na Slika 19. Slika prikazuje zelene i crvene krugove koji predstavljaju čvorove grafa odnosno entitete ove baze podataka. Zeleni čvorovi predstavljaju gradove, odnosno zračne luke (radi smanjenja kompleksnosti pretpostavljeno je da jedan grad predstavlja jednu zračnu luku), a crveni čvorovi predstavljaju letove. Prema priloženom podatkovnom modelu jedna zračna luka može imati više dolaznih i odlaznih letova, a da li je let dolazni ili odlazni govore sive linije koje imaju oznake „TO“ (dolazni let) i „FROM“ (odlazni let). Čvorovi prikazani na slici unutar sebe mogu sadržavati mnoge atribute koji ih opisuju kao entitete, a svi ti atributi su u tzv. „ključ-vrijednost“ obliku na koji se često može naići u modernim programskim jezicima.



Slika 19. Podatkovni model sustava za rezervaciju zrakoplovnih karata

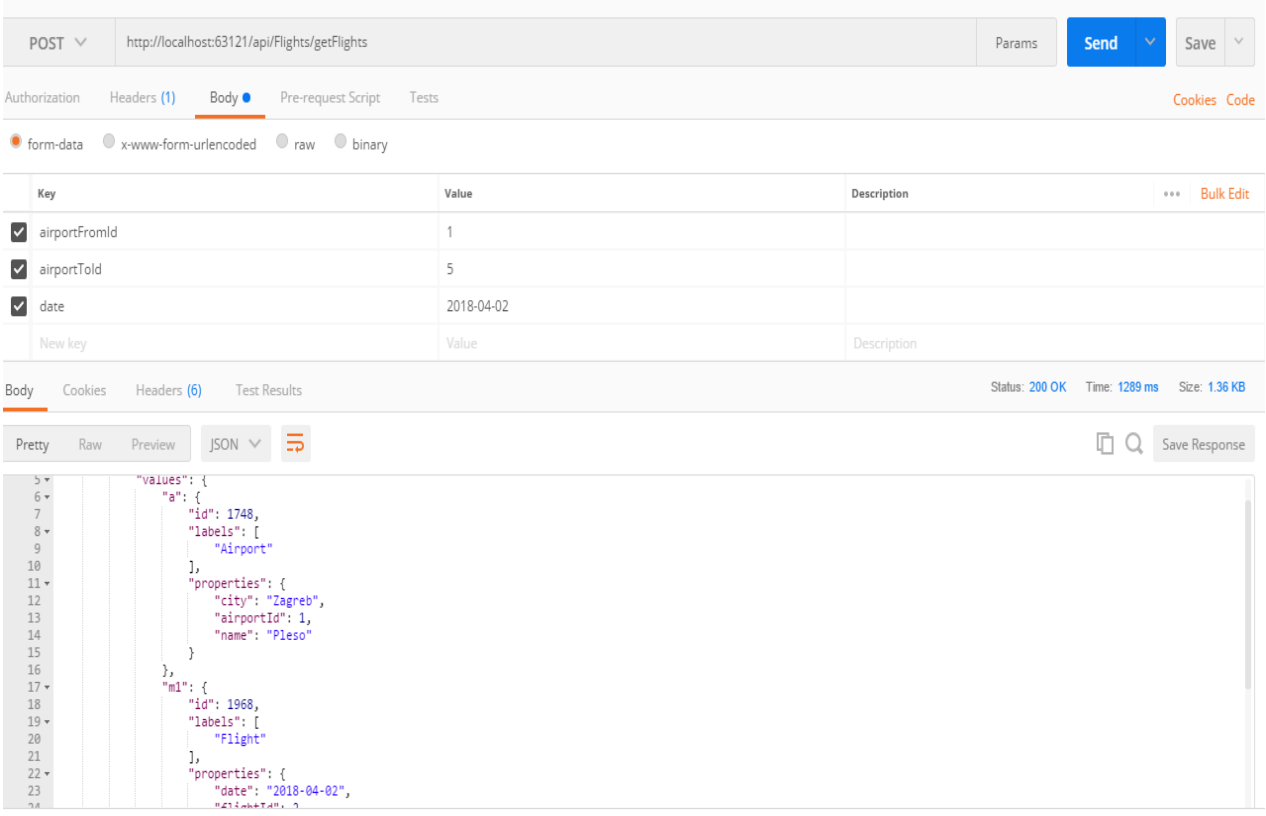
Ovaj se sustav sastoji od tzv. „kontrolera„ (eng. controller) koji se može naći i u popularnom MVC uzorku dizajna te metode koje se pozivaju prilikom poziva iz sustava za upravljanje korisničkim zahtjevima, komuniciraju s nerelacijskom bazom podataka te vraćaju rezultat u tzv. JSON (eng. JSON – JavaScript Object Notation) obliku. Metodama priloženog kontrolera ne može se pristupiti direktno iz korisničkog sučelja, već im se isključivo može pristupiti putem posrednika tj. sustava za upravljanje korisničkim zahtjevima koji je prethodno opisan. Tablica 4. sadrži popis svih javnih URL metoda putem kojih je moguće pristupiti funkcionalnostima sustava za rezervaciju zrakoplovnih karata.

Tablica 4. Popis URL-ova za pristup sustavu za rezervaciju zrakoplovnih karata

URL	Opis	Poziv metode i parametri
/api/Flights/getAirports	Dohvat svih dostupnih zračnih luka/gradova u sustavu. Prema bazi podataka ovoga sustava to se odnosi na sve čvorove koji su tipa „Airport“.	Poziva se metoda „GetAirports“ u kontroleru „Flights“. Kao opcionalni ulazni parametar metoda može primiti dio teksta prema kojem se mogu filtrirati rezultati.
/api/Flights/getFlights	Metoda vraća sve moguće kombinacije letova uključujući i povezane letove između polazne i odredišne zračne luke.	Poziva se metoda „GetFlights“ u kontroleru „Flights“. Ulazni podaci ove metode su odlazna zračna luka, dolazna zračna luka i datum leta. Navedeni parametri su obvezni prilikom poziva.
/api/Flights/bookFlight	Metoda koju služi za rezervaciju zrakoplovnih karata na putu od polazne do odredišne zračne luke.	Poziva se metoda „BookFlight“ u kontroleru „Flights“. Obvezni ulazni parametar je korisnički žeton kojim se dohvaća identifikator korisnika, te popis svih letova.

/api/Flights/bookedFlights	Metoda koja služi za dohvaćanje popisa svih letova koje je korisnik prethodno rezervirao.	Poziva se metoda „BookedFlights“ u kontroleru „Flights“. Obvezni ulazni parametar je korisnički žeton kojim se dohvaća identifikator korisnika.

Slika 20. prikazuje primjer poziva metode za dohvat popisa letova u ovome sustavu iz programa „Postman“. Kao što je spomenuto u Tablica 4. metoda za dohvat letova na ulazu zahtijeva točno tri parametra: identifikacijsku oznaku polazišne zračne luke, identifikacijsku oznaku odredišne zračne luke i planirani datum polaska.



Slika 20. Poziv metoda za dohvat letova i svih povezanih letova

6. Zaključak

Cilj ovoga rada bio je prikaz mikroservisne arhitekture u razvoju web aplikacija te se može zaključiti da u odnosu na mnoge druge prethodno opisane arhitekture, ova arhitektura ima mnoge prednosti, ali i nedostatke. Mikroservisna arhitektura je relativno „mlada“ arhitektura izrade web aplikacije koja se koristi u izradi kompleksnih i distribuiranih web aplikacija kod poduzeća gdje na razvoju aplikacije, odnosno točno određenog dijela aplikacije sudjeluje veći broj djelatnika. Ovakva vrste arhitekture najčešće se upotrebljava i kod velikih i kompleksnih sustava gdje se svakodnevno rade dorade na raznim sustavima koji su obuhvaćeni opsegom aplikacije. Kroz ovaj rad prikazane su mnoge prednosti, ali i nedostaci mikroservisne arhitekture te se ne može reći da je ova arhitektura idealna arhitektura za baš svaku web aplikaciju. Razvoj interneta i mrežnih tehnologija dosta unaprjeđuje rad s mikroservisnom arhitekturom, a najviše tome doprinosi računalstvo u oblaku (eng. cloud computing). Računalstvo u oblaku korisniku omogućava jednostavno korištenje računalnih i mrežnih resursa čime se mogu vrlo jednostavno i „brzo“ graditi novi sustavi koji sudjeluju u radu web aplikacije.

Mikroservisna arhitektura prilikom razvoja aplikacija obuhvaća i mnoge nedostatke na koje se svakako mora obratiti pozornost prilikom rada s ovakvom vrstom arhitekture. Kako se kompleksnost aplikacije dijeli na mnoge sustave koje razvijaju najčešće različiti timovi programera. Mikroservisna arhitektura danas rješava većinu problema kao što su skalabilnost, performanse, održavanje sustava i sl., kod kompleksnih distribuiranih sustava tj. web aplikacija, no vrlo je teško primijeniti ovu arhitekturu na web aplikaciju male kompleksnosti čiji se rast i razvoj može ostvariti putem „tradicionalne“ monolitne arhitekture. Iz praktičnog djela ovoga rada može se dati zaključak kako je trebalo relativno puno vremena i računalnih resursa da se i ovako mala aplikacija realizira uporabom ove arhitekture (više pokrenutih aplikacija za obradu programskog koda, više pokrenutih sustava za upravljanje s bazom podataka i sl.).

U svijetu web tehnologija svakodnevno se javljaju novi alati koji omogućavaju jednostavnije i brže rješavanje raznih problema u razvoju web aplikacija, no problem je što se te nove tehnologije ne mogu lako ili se uopće ne mogu implementirati na postojećim sustavima koji su najčešće izrađeni „tradicionalnom“ monolitnom arhitekturom. Uvođenje novih tehnologija može se ostvariti tako da se uvedu novi sustavi koji su implementirani putem novih tehnologija, a da su podešeni tako da drugi sustavi mogu komunicirati s njima. U konkretnom primjeru iz ovoga rada može se navesti sustav za rezervaciju zrakoplovnih karata koji rješava problem koji je po svojoj prirodi djelomično rekurzivni problem. Problem kod mikroservisne

arhitekture može predstavljati i mrežna infrastruktura iz razloga što su sustavi u odvojeni te komuniciraju putem mreže. Iz prethodno navedenog razloga, organizacije vrlo često zapošljavaju zaposlenike čija je uloga upravljanje i nadgledanje mrežne infrastrukture i poslužitelja. Sustavi mikroservisne arhitekture su neovisni i izolirani te se stoga mogu zasebno nadograđivati, razvijati i testirati, što nažalost nije slučaj kod aplikacija monolitne arhitekture. Mikroservisna arhitektura je relativno „mlada“ arhitektura u izradi web aplikacija koja se razvija paralelno s razvojem informacijsko-komunikacijskih tehnologija te se stoga može procijeniti da će se sve češće upotrebljavati u razvoju složenih web aplikacija.

Sažetak

Ovaj rad sadrži teorijsku i praktičnu obradu teme iz područja razvoja programskih rješenja za računalne sustave. Mikroservisna arhitektura je softverska arhitektura koja se danas koristi u razvoju složenih web aplikacija. Kroz rad su opisane i druge arhitekture koje se koriste u razvoju web aplikacija te su njihove prednosti i mane iskorištene kao podloga za opis mikroservisne arhitekture tj. problema koje ona nastoji riješiti.

Ključne riječi: Web aplikacija**,** softverska arhitektura**,** mikroservisna arhitektura

Abstract

This document presents theoretical and practical analysis of software development field involved with computer systems. Microservice architecture today presents software architecture pattern that is made for developing complex web applications. Through this document some other software architectures were described as a background of presenting microservice architecture i.e problems that were supposed to be solved using this architecture.

Keywords: Web application, software architecture, microservice architecture

Literatura

[1] A method of selecting appropriate software architecture styles: Quality Attributes and Analytic Hierarchy Process. Dostupno: https://gupea.ub.gu.se/bitstream/2077/30045/1/gupea_2077_30045_1.pdf. [Pokušaj pristupa 14.01.2018.].

[2] ASP.NET Core. Dostupno: https://docs.microsoft.com/en-us/aspnet/core/. [Pokušaj pristupa 11.03.2018].

[3] Client-Server Architecture. Dostupno: http://tutorials.jenkov.com/software-architecture/client-server-architecture.html. [Pokušaj pristupa 21.11.2017.].

[4] Client-Server Architecture. Dostupno: https://www.utdallas.edu/~chung/SA/2client.pdf. [Pokušaj pristupa 07.02.2018.].

[5] Communication in a microservice architecture. Dostupno: https://docs.microsoft.com/en-us/dotnet/standard/microservices-architecture/architect-microservice-container-applications/communication-in-microservice-architecture. [Pokušaj pristupa 26.12.2017.].

[6] Defining Software Architecture. Dostupno: https://www.sei.cmu.edu/architecture/. [Pokušaj pristupa 28.01.2018.].

[7] From Monolithic Three-tiers Architectures to SOA Vs Microservices. Dostupno: https://thetechsolo.wordpress.com/2015/07/05/from-monolith-three-tiers-architectures-to-soa-vs-microservices/. [Pokušaj pristupa 17.12.2017.].

[8] History of PHP. Dostupno: http://php.net/manual/en/history.php.php. [Pokušaj pristupa 07.03.2018.].

[9] History of Python. Dostupno: https://ir.lib.vntu.edu.ua/bitstream/handle/123456789/10471/461.pdf?sequence=3. [Pokušaj pristupa 11.03.2018].

[10] History Of The Client Server Architecture. Dostupno: https://www.ukessays.com/essays/information-technology/history-of-the-client-server-architecture-information-technology-essay.php. [Pokušaj pristupa 27.01.2018.].

[11] Hypertext Transfer Protocol. Dostupno: https://www.w3.org/Protocols/rfc2616/rfc2616.html. [Pokušaj pristupa 27.01.2018.].

[12] M. Eisele, Modern Java EE Design Patterns, O'Reilly, 2015.

[13] M. Richards, Microservices vs. Service-Oriented Architecture.

[14] Microservices best practices for Java. Dostupno: https://www.redbooks.ibm.com/redbooks/pdfs/sg248357.pdf. [Pokušaj pristupa 01.02. 2018.].

[15] Microservices from theory to practice. Dostupno: https://www.redbooks.ibm.com/redbooks/pdfs/sg248275.pdf. [Pokušaj pristupa 19.11. 2017.].

[16] Microservices vs. SOA. Dostupno: https://dzone.com/articles/microservices-vs-soa-2. [Pokušaj pristupa 17.12.2017.].

[17] Monolithic Application Design. Dostupno: http://www.c-sharpcorner.com/article/monolithic-application-design/. [Pokušaj pristupa 19.11. 2017.].

[18] Monolithic architecture. Dostupno: http://whatis.techtarget.com/definition/monolithic-architecture. [Pokušaj pristupa 17.12 2017.].

[19] Monolithic vs. Microservices Architecture. Dostupno: https://articles.microservices.com/monolithic-vs-microservices-architecture-5c4848858f59. [Pokušaj pristupa 27.11.2017.].

[20] N. S., Building Microservices, O'Reilly Media, 2015.

[21] Saga. Dostupno: http://microservices.io/patterns/data/saga.html. [Pokušaj pristupa 12. 03.2018].

[22] Service Oriented Architecture (SOA). Dostupno: http://tutorials.jenkov.com/soa/soa.html. [Pokušaj pristupa 17.12.2017.].

[23] Single Process Architecture. Dostupno: http://tutorials.jenkov.com/software-architecture/single-process-architecture.html. [Pokušaj pristupa 17.12.2017.].

[24] Three-tier Application Model. Dostupno: https://msdn.microsoft.com/en-us/library/aa480455.aspx. [Pokušaj pristupa 03.01.2018.].

[25] Understanding REST. Dostupno: https://spring.io/understanding/REST. [Pokušaj pristupa 05.02.2018.].

[26] Understanding Service-Oriented Arhitecture. Dostupno: https://msdn.microsoft.com/en-us/library/aa480021.aspx. [Pokušaj pristupa 19.11. 2017.].

[27] Vertical And Horizontal Scaling. Dostupno: http://social.dnsmadeeasy.com/blog/vertical-and-horizontal-scaling/. [Pokušaj pristupa 03.02.2018.].

[28] Web App Architectures. Dostupno: http://www.cs.toronto.edu/~mashiyat/csc309/Lectures/Web%20App%20AArchitecture.pdf. [Pokušaj pristupa 19.11.2017.].

[29] Web Application Architecture from 10,000 Feet. Dostupno: https://spin.atomicobject.com/2015/04/06/web-app-client-side-server-side/. [Pokušaj pristupa 03.01.2018.].

[30] Web Services Architectures. Dostupno: https://dzone.com/articles/web-services-architecture. [Pokušaj pristupa 28.01.2018.].

[31] What Are RESTful Web Services? Dostupno: https://docs.oracle.com/javaee/6/tutorial/doc/gijqy.html. [Pokušaj pristupa 08.02. 2018.].

[32] What are web services? Dostupno: https://www.tutorialspoint.com/webservices/what_are_web_services.htm. [Pokušaj pristupa 08.01.2017.].

[33] What is Agile Methodology? Dostupno: https://www.versionone.com/agile-101/agile-methodologies/. [Pokušaj pristupa 04.02.2018.].

[34] What is Software Architecture? Dostupno: https://msdn.microsoft.com/en-us/library/ee658098.aspx. [Pokušaj pristupa 27.01.2018.].

Popis tablica

Popis slika