



**UNIVERZITET SINGIDUNUM**  
**Departman za poslediplomske studije**

**Master akademski program**  
**Softversko i informaciono inženjerstvo**

**MASTER RAD**

**Primena mikroservisne arhitekture**  
**za razvoj aplikacije za rentiranje vozila**

**Mentor:**

**prof. dr Violeta Tomašević**

**Kandidat:**

**Miloš Ilić 440092/2023**

**Beograd, 2024.**

## Apstrakt

Proces razvoja većine današnjih softvera je vrlo složen. U njemu jednu od najvažnijih aktivnosti predstavlja izbor arhitekture sistema. Primijenjena arhitektura ima direktan uticaj na brzinu i efikasnost realizacije. Savremene softverske arhitekture pružaju projektantima široke mogućnosti pri izboru načina na koji će realizovati neko softversko rešenje. Zahvaljujući velikom broju arhitektura sa različitim osobinama, može se izabrati ona koja najbolje odgovara prirodi konkretne aplikacije koja se razvija.

U ovom radu je najpre analizirano nekoliko najznačajnijih softverskih arhitektura, a zatim je, jedna od najpopularnijih, mikroservisna arhitektura primijenjena za razvoj aplikacije za rentiranje vozila. Aplikacija je namenjena *rent-a-car* agencijama za online obavljanje poslova evidencije klijenata i vozila u okviru njihovih svakodnevnih aktivnosti. Dostupna je anonimnim korisnicima koji žele samo da se informišu o mogućnostima rentiranja, klijentima koji koriste usluge agencije i administratorima koji vode računa o celom sistemu. U realizaciji je primenjen klijent-server stil sa mikroservisima na serverskoj strani aplikacije. Zahvaljujući ovakvom izboru arhitekture, aplikacija je efikasno realizovana sa vrlo ograničenim resursima pomoću raspoloživih savremenih tehnologija i platformi. Osim toga, ova arhitektura je omogućila dobijanje fleksibilne i skalabile aplikacije koja pruža velike mogućnosti za dalje unapređenje.

**Ključne reči:** *softverska arhitektura, mikroservisi, RESTful, API, rent-a-car*

## Abstract

The design and development of many contemporary software applications is a quite complex task. One of the vital design decisions is the choice of the system architecture since the speed and efficiency of the implementation heavily depend on the chosen architecture. Modern software architectures offer to the designers vast possibilities for different implementation approaches. From this wide spectrum of architectures the designer opts for most convenient one for the application being developed.

This paper firstly analyzes several most prominent software architectures. Then, it applies one of the most popular - microservice architecture - in the development of the car renting application. This application is intended for car renting agencies for every day book keeping of clients and cars. It is available to the potential users to be informed of agency's services, to the clients who use these services, and to the administrators responsible for the entire system. In implementation, the client-server style with microservices on the server side is applied. Because of that, the application is efficiently implemented with quite restricted resources using modern technologies and platforms. In addition, this architecture allowed for obtaining a flexible and scalable application which is amenable for further upgrades and improvements.

**Keywords:** *software architecture, microservices, RESTful, API, rent-a-car*

## Sadržaj

<b>1. Uvod .....</b>	<b>5</b>
1.1. Metodologija rada .....	5
1.2. Struktura rada .....	6
<b>2. Softverske arhitekture .....</b>	<b>7</b>
2.1. Klijent-server arhitektura .....	7
2.2. Arhitektura sa ravnopravnim pristupom .....	8
2.3. Slojevita arhitektura .....	9
2.4. Arhitektura zasnovana na događajima .....	10
2.5. Servisno-orijentisana arhitektura .....	11
2.5.1. Definicije SOA .....	13
2.5.2. Principi SOA .....	13
2.5.3. Komponente SOA .....	14
2.5.4. SOAP servisi .....	15
<b>3. Mikroservisna arhitektura .....</b>	<b>17</b>
3.1. REST servisi .....	19
<b>4. Opis aplikacije .....</b>	<b>22</b>
4.1. Funkcionalni zahtevi .....	22
4.2. Korisnici i prava pristupa .....	24
<b>5. Realizacija aplikacije .....</b>	<b>27</b>
5.1. Arhitektura aplikacije .....	27
5.2. Razvojno okruženje, alati i tehnologije .....	30
5.2.1. Visual Studio Code Editor .....	30
5.2.2. Angular Framework .....	31
5.2.3. NodeJS .....	31
5.2.4. NestJS Framework .....	32
5.2.5. HeidiSQL .....	32

5.3. Implementacioni detalji .....	32
5.3.1. Implementacija mikroservisa .....	32
5.3.2. Korisnički interfejs .....	37
<b>6. Zaključak .....</b>	<b>42</b>

## **Literatura**

## 1. Uvod

Pod arhitekturom nekog računarskog sistema se podrazumeva struktura koja obuhvata različite softverske i hardverske elemente, i to kako spoljašnja, vidljiva svojstva tih elemenata, tako i veze koje postoje između njih. Prema [1], elementi računarske arhitekture su: arhitektura tehnologije (fizički dizajn), tehnološka infrastruktura (tehnološko okruženje koje obuhvata hardver i softver) i računarski program (aplikacija ili skup instrukcija za izvršavanje nekog zadatka).

Tehnološka arhitektura opisuje softverske i hardverske funkcionalnosti koje su potrebne za podršku, razvoj i podelu poslovanja, podataka i aplikacionih servisa. Tehnološka arhitektura uključuje IT infrastrukturu, računarske mreže, protokole komunikacije, tehnološke standarde i dr.

Tehnološka infrastruktura predstavlja platformu na kojoj se programi implementiraju. Iako su hardver i softver delovi tehnološke arhitekture, postoje njihovi delovi koji se uključuju u infrastrukturu. To su različite hardverske komponente, kao što su serveri, mrežna oprema, računarska oprema, i sl. ili softverske komponente, kao što su baze podataka, direktorijumi, API, operativni programi, itd.

Računarski program je uređeni niz instrukcija koji procesor u računaru može da izvrši. Programi se mogu prevoditi pomoću prevodioca (kompajlera), interpretirati ili u posebnim slučajevima direktno izvršavati na računaru. Programi se pišu na različitim programskim jezicima uz poštovanje određenih pravila. Program može da napiše programer, a postoje programi koji nastaju izvršavanjem drugih programa na računaru.

Ovaj rad se bavi savremenim softverskim arhitekturama sa teorijskog i praktičnog aspekta [2]. Teorijski, u njemu je predstavljeno više softverskih arhitektura kroz prikaz njihovih najvažnijih osobina. Istaknute su njihove prednosti i nedostaci, kao i mogućnosti primene. Jedna od njih, mikroservisna arhitektura je posebno detaljno analizirana, a zatim i praktično primenjena za razvoj konkretne aplikacije za rentiranje vozila.

Aplikacija je realizovana tako da mogu da joj pristupe tri vrste korisnika (gost, klijent i administrator) kako bi dobili potrebne informacije ili usluge u vezi sa mogućnošću iznajmljivanja različitih vrsta vozila. Implementirana je korišćenjem savremenih tehnologija, alata i razvojnih okruženja koji su, takođe, opisani u radu. Nakon realizacije aplikacije, razmatrane su mogućnosti njenog daljeg unapređenja.

### 1.1. Metodologija rada

Predmet istraživanja u ovom radu je upoznavanje sa naprednim softverskim arhitekturama, njihovo proučavanje i utvrđivanje mogućnosti njihove primene u realizaciji veb aplikacija. Posebna pažnja je posvećena mikroservisnoj arhitekturi, kao jednoj od danas najpopularnijih i najčešće korišćenih arhitektura. Analizirane su mogućnosti koje pruža ova arhitektura i njen uticaj na brži i efikasniji razvoj skalabilnih veb aplikacija.

Cilj sprovedenih istraživanja je sticanje znanja o softverskim arhitekturama, posebno o mikroservisnoj arhitekturi. Takođe, cilj je i primena mikroservisne arhitekture u okviru klijent-server stila projektovanja za razvoj veb aplikacije za rentiranje vozila. Kroz

ovu konkretnu primenu, pokušalo se da se ukaže na sve koristi koje *rent-a-car* agencija može da ima u svom svakodnevnom poslovanju od korišćenja ovakve aplikacije, a koje proističu iz primenjene arhitekture.

Osnovni metod istraživanja u ovom radu je prikupljanje i analiza literature koja se bavi softverskim arhitekturama, posebno mikroservisnom arhitekturom. Osim toga, bilo je potrebno prikupiti i proučiti literaturu koja se odnosi na različite alate, tehnologije i razvojna okruženja koji su neophodni za razvoja veb aplikacije, zatim instalirati potrebne softvere i na kraju ih primeniti u razvojnom okruženju za generisanje aplikacije.

## **1.2. Struktura rada**

U prvom poglavlju je najpre dat uvod u oblast kojom će se rad baviti, a zatim su opisani primenjena metodologija istraživanja i struktura rada po poglavljima.

Drugo poglavlje teorijski obrađuje nekoliko najpoznatijih i najčešće korišćenih savremenih softverskih arhitektura: klijent-server arhitekturu, arhitekturu sa ravnopravnim pristupom, slojevit arhitekturu, arhitekturom vođenu događajima i servisno-orijentisanu arhitekturu.

U trećem poglavlju detaljno je predstavljena mikroservisana arhitektura kroz njene osobine, prednosti i nedostatke. U okviru ove arhitekture, poseban naglasak je stavljen na opis REST servisa.

Mikroservisna arhitektura je primenjena za razvoj aplikacije za rentiranje vozila. Funkcionalni zahtevi koje ta aplikacija treba da ispuni, tema su četvrtog poglavlja u radu. Osim zahteva, u ovom poglavlju su definisane i klase korskornika koje aplikacija može da ima sa njihovim pravima pristupa kroz opis funkcionalnosti koje su im dostupne.

Detaljan opis realizacije aplikacije je dat u petom poglavlju. On obuhvata prikaz arhitekture sistema prema principima mikroservisne arhitekture, zatim opis korišćenih razvojnih okruženja, IT tehnologija i alata, kao i detalje implementacije mikroservisa i korisničkog interfejsa.

U šestom poglavlju je sumirano sve što je urađeno u radu, a zatim su date smernice za nastavak daljih istraživanja.

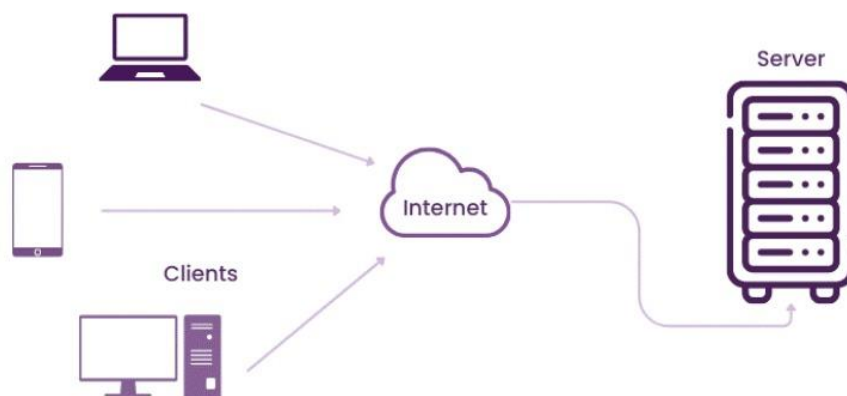
## 2. Softverske arhitekture

Softverska arhitektura predstavlja skup strukturnih elemenata koji svojim osobinama i međusobnim vezama čine neki softverski sistem. Izbor arhitekture softvera je veoma važan korak koga je neophodno učiniti pre početka realizacije samog softvera. Funkcionalnost, izrada i održavanje softvera zasnovani su na dobro odabranoj i isplaniranoj arhitekturi. Realizacija softverske arhitekture je tehnika planiranja i kreiranja rešenja za izradu funkcionalnog softvera. Ona služi da se uspostavi veza između tehničkih i poslovnih zahteva aplikacije, kao i za određivanje strukture aplikacije.

U nastavku je opisano nekoliko najznačajnijih savremenih softverskih arhitektura koje su našle široku primenu u brojnim softverskim rešenjima. Jedna od najnovijih je mikroservisna arhitektura koja je izdvojena i posebno predstavljena u narednom poglavlju zato što je primenjena za razvoj aplikacije kojom se ovaj rad bavi.

### 2.1. Klijent-server arhitektura

Klijent-server arhitektura je arhitekturni stil u kome su omogućeni komunikacija i razmena podataka između različitih delova aplikacije putem računarske mreže sa jednim ili više servera.



Slika 1. Prikaz klijent-server arhitekture (Izvor: [3])

Kao što se vidi na slici 1, klijent-server arhitektura predstavlja model arhitekture koji se sastoji od tri osnovne komponente: klijenta (korisnika usluga), servera (davaoca usluga) i mreže (komunikacionog posrednika).

Klijentska aplikacija zahteva usluge od servera. Te usluge mogu biti, na primer, preuzimanje i skladištenje podataka, ali i niz drugih funkcionalnosti. Klijent je aktivan učesnik u toj komunikaciji zato što on zahteva određeni resurs, i čeka na odgovor servera.

Serverska aplikacija obrađuje zahteve klijenata, šalje im odgovore i pruža određene usluge. Jedna od usluga koje pruža server je skladištenje podataka. Komunikacija između klijenata i servera uglavnom se odvija pomoću HTTP protokola, a za realizaciju određenih

procesa se koriste i FTP i SMTP protokoli. Server je pasivan učesnik u toj komunikaciji zato što čeka zahtev klijenata, izvršava ga i šalje odgovore.

Mreža predstavlja komunikacioni kanal pomoću koga su klijent i server povezani kako bi se obavio prenos podataka između njih. Olakšava razmenu zahteva i odgovora, takođe je odgovorna za brzinu prenosa i sigurnost prenetih podataka.

Klijent-server arhitektura može biti jednoslojna, dvoslojna, troslojna i višeslojna.

Jednoslojna arhitektura je samostalna aplikacija na jednoj platformi, gde su klijent, server i baza podataka na istoj mašini. Koristi se kod manjih aplikacija koje nemaju velike sistemske zahteve.

Dvoslojna arhitektura je jedna od osnovnih klijent-server arhitektura kod koje se obavlja direktna komunikacija između klijenta i servera bez međusloja. Laka je za implementaciju, ali je nedovoljno bezbedna i zahteva visok mrežni saobraćaj.

Troslojna arhitektura je složenija klijent-server arhitektura jer poseduje međusloj koji upravlja biznis logikom, i deluje kao most između klijenta i servera. Kod troslojne arhitekture podeljene su uloge klijenta i servera. Klijent se bavi korisničkim interfejsom, a server upravlja skladištenjem podataka. Ovaj tip arhitekture poboljšava performanse i bezbednost za razliku od dvoslojne arhitekture, ali je samim tim i daleko složenija i skuplja.

Višeslojna arhitektura je fleksibilnija klijent-server arhitektura koja ima više od tri nivoa. Svaki nivo se može nezavisno distribuirati i ažurirati na različitim mašinama. Ovaj tip arhitekture se koristi kod složenih sistema i zahteva više resursa.

Pored pomenutih klijent-server arhitektura, postoje još i neki njihovi podtipovi, kao što su: *peer to peer*, arhitektura tankog sloja, itd.

Prednosti korišćenja klijent-server arhitekture su:

- čuvanje, upravljanje podacima se vrši na jednom mestu,
- klijentu su potrebne samo klijentska aplikacija i mreža,
- mogućnost istovremene komunikacije sa velikim brojem klijenata istovremeno.

Nedostaci klijent-server arhitekture:

- kapacitet i dostupnost servera,
- zavisnost od kvaliteta mreže,
- mogući problem mogu biti i bezbednost i sinhronizacija.

## **2.2. Arhitektura sa ravnopravnim pristupom**

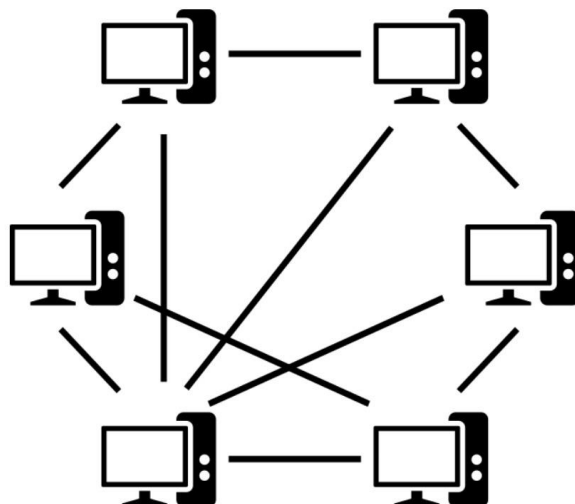
Arhitektura sa ravnopravnim pristupom (P2P) je arhitektura koju čine ravnopravni članovi koji komuniciraju preko mreže (slika 2). Tu komunikaciju obavljaju softveri koji mogu vršiti ulogu klijenta (zahtevaju potrebne resurse od ostalih članova) ili ulogu servera (pružaju zahtevane resurse ostalim članovima).



U ovoj arhitekturi svi članovi mogu direktno da komuniciraju jedni sa drugima, vrše razmenu fajlova i dele propusni opseg mreže, što je vrlo bitno kod prenosa podataka jer je on mnogo brži, a to je kod velikih datoteka veoma važno.

Snagu arhitekture sa ravnopravnim pristupom čini to što veći broj ravnopravnih članova može da učestvuje u komunikaciji. Svaki novi član poboljšava sposobnost mreže, povećava njen kapacitet. Ovo je zapravo i najveća razlika između klijent-server arhitekture i arhitekture sa ravnopravnim pristupom. U slučaju klijent-server arhitekture, sa dodavanjem klijenata, server se dodatno opterećivao, a samim tim, smanjivao se propusni opseg.

Pored velike prednosti koju poseduje arhitektura sa ravnopravnim pristupom, ona ima i nedostatke. Glavni nedostatak se odnosi na pitanje bezbednosti. Bezbednost može biti narušena od strane nekog člana koji učestvuje u komunikaciji, jer nema autoriteta među članovima, tj. svi su međusobno ravnopravni. Isto tako, neki član može da koristi resurse mreže, a da ništa ne ulaže u nju, što nepovoljno utiče na performanse same arhitekture.



Slika 2. Prikaz arhitekture sa ravnopravnim pristupom (Izvor: [4])

### 2.3. Slojevita arhitektura

Slojevita arhitektura je arhitektonski stil u kome se softver sastoji od više slojeva, pri čemu svaki sloj ima svoju funkcionalnu ulogu. Broj slojeva zavisi od veličine softvera. Obično se za realizaciju nekog softvera koristi četiri sloja (prezentacioni, sloj biznis logike, perzistentni sloj i sloj baze podataka), a ponekad se sloj biznis logike i perzistentni sloj objedinjuju [5].

Kao što je pomenuto, svaki od ovih slojeva ima svoju funkcionalnu ulogu ili zadatak, njihov raspored je definisan, njihova komunikacija se obavlja u smeru odozgo na dole (slika 3).



Slika 3. Prikaz slojevite arhitekture (Izvor: [5])

Prezentacioni sloj je najviši sloj softvera. U njemu se nalazi korisnički interfejs. U ovom sloju logika je najjednostavnija. Prezentacioni sloj korisnicima omogućava pozivanje potrebnih funkcija preko korisničkog interfejsa (UI - *User Interface*).

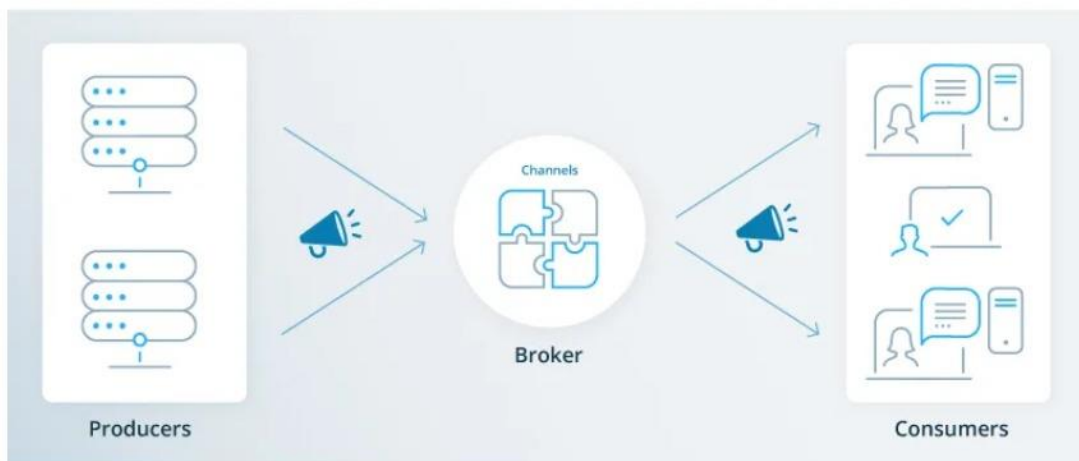
Sloj biznis logike je sloj u kome se nalazi logika za realizaciju aplikacije. Na ulaznom delu ovoga sloja nalazi se struktura podataka primljena od prezentacionog sloja, a na izlaznom delu struktura podataka primljena od baze podataka. Sloj biznis logike se često objedinjuje sa slojem perzistencije podataka. Sloj biznis logike je podeljen na više manjih delova, što omogućava lakše testiranje, upravljanje i održavanje. U sloju biznis logike može biti smešteno više korisničkih interfejsa što smanjuje mogućnost dupliranja nepotrebnog koda za realizaciju aplikacije.

Sloj baze podataka je sloj u kome se nalaze svi podaci potrebni za realizaciju aplikacije. U sloju baze podataka se nalaze strukturirani i nestrukturirani podaci. Takođe, tu su smešteni sistemi koji služe za manipulaciju nad podacima, bilo da je u pitanju relacionalna baza podataka (RDBMS) ili nerelacionalna baza podataka (*NoSql*).

Svaki sloj kod slojevite arhitekture je zatvoren, što znači da slojevi ne zavise jedan od drugog, pa samim tim nema potrebe da bilo koji sloj ima uvid u logiku sloja ispod ili iznad njega. Jedino je bitno da zahtev mora proći redom kroz svaki sloj.

## 2.4. Arhitektura zasnovana na događajima

Arhitektura zasnovana na događajima je arhitektonski stil koji omogućava da odvojene aplikacije asinhrono komuniciraju i obavljaju neki posao [6]. Ova arhitektura obezbeđuje protok informacija u realnom vremenu između aplikacija, mikroservisa i svih uređaja koji učestvuju u poslovanju.



Slika 4. Prikaz arhitekture zasnovane na događajima (Izvor: [6])

Kod arhitekture zasnovane na događajima, postupak se odvija tako što se informacije o događaju koji se desio šalju svim aplikacijama, sistemima i ljudima koji su uključeni u proces, kako bi reagovali u realnom vremenu.

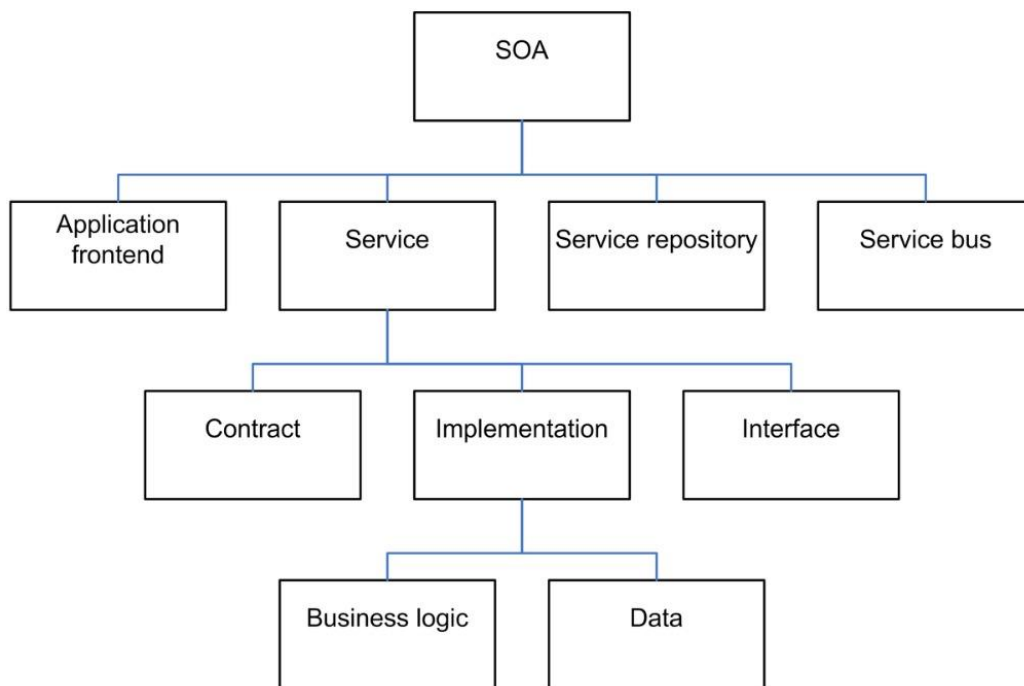
U arhitekturi zasnovanoj na događajima, postoji posrednik za određeni događaj, te nije potrebno znati poreklo informacija koje se u njemu koriste, jer ovu funkciju obavlja posrednik. Ovakav način komunikacije omogućava labavo povezivanje aplikacija, što olakšava nezavisan razvoj i primenu sistema kao i integraciju sa spoljnim sistemima.

Arhitektura zasnovana na događajima ima široku primenu u *online* bankarstvu, u igrama za više igrača, u strimovanju i dr.

## 2.5. Servisno-orijentisana arhitektura

Servisno-orijentisana arhitektura (SOA) zasnovana je na procesu maksimalne agilnosti, fleksibilnosti i rasprostranjenosti [7]. SOA koristi labavo povezane servise radi postizanja maksimalne poslovne fleksibilnosti na interoperabilan i tehnološki nezavisan način. SOA predstavlja arhitekturu koja pruža usluge drugim aplikacijama putem vidljivih i objavljenih interfejsa, pozivanjem usluga preko mreže. SOA pruža mogućnost da se sa svakog računara može pokrenuti veliki broj usluga, a svaki servis je izgrađen tako da može da razmenjuje informacije sa bilo kojim servisom u mreži, bez interakcije čoveka i bez izmena osnovnog programa.

SOA arhitektura, prikazana na slici 5, sastoji se od više komponenata koje međusobno komuniciraju. Servisi (*service*) su osnovne komponente SOA arhitekture. Oni mogu biti javni i privatni. Servis ugovora (*Contract*) zadužen je za odredbe, uslove i kvalitet pruženih usluga. Servis implementacije (*Implementation*) zadužen je za izradu biznis logike, vršenje određenih funkcija kao što je autentifikacija korisnika. Servis interfejs (*Interface*) zadužen je za komunikaciju usluga ili sistema. On jasno definiše način pozivanja usluga za obavljanje raznih aktivnosti i razmenu podataka unutar aplikacije.



Slika 5. Prikaz SOA arhitekture (Izvor: [7])

Registar servisa (*Service repository*) je repozitorijum u kome se nalaze dokumenta sa opisom usluga i načina pristupa uslugama. Servis za dopremanje usluga (*Service bus*) kreira, održava i obezbeđuje jednu ili više usluga koje su na raspolaganju korisnicima. Korisnik usluga (*Application frontend*) služi za komunikaciju sa servisom za dopremanje usluga, bilo da se radi o delu aplikacije ili o celom sistemu. Međusobna interakcija se mora obaviti uz poštovanje propisanih pravila u okviru ugovora.

Kao što je ranije već pomenuto, SOA arhitektura koristi nepovezane ili slabo povezane funkcionalnosti koje su samostalne. Unutar SOA, usluge koriste definisane protokole koji opisuju proces slanja poruka koristeći metapodatke opisa koji konkretno opisuju karakteristike usluga i podatke koji ih pokreću.

SOA zavisi od podataka i usluga koje su opisane metapodacima. Da bi SOA funkcionisala regularno, metapodaci treba da zadovolje sledeće kriterijume:

- metapodaci treba da budu u obliku softverskih sistema koji se mogu koristiti za dinamičko konfigurisanje, otkrivanje i spajanje definisanih usluga, a istovremeno treba voditi računa o ispravnosti i zaštiti podataka
- meta podaci treba da budu u odgovarajućem formatu da bi dizajneri sistema mogli bez poteškoća da ih koriste i manipulišu sa njima, da učinak bude što veći, a da iskoristivost resursa bude što manja.

SOA omogućava korisnicima da kombinuju veliki deo funkcionalnosti za formiranje *ad hoc* aplikacija koje se kreiraju od postojećih softverskih servisa. Prednost korišćenja većih delova funkcionalnosti je u tome što je potrebno manje interfejsa za implementiranje bilo kog dela funkcionalnosti, ali kod većih delova funkcionalnosti problem se može pojaviti kod ponovne upotrebe zbog brzine prenosa podataka. Da bi se

odabrao odgovarajući interfejs, potrebno je razmotriti troškove obrade i mogućnosti, a ne samo brzinu prenosa podataka.

SOA arhitektura omogućava korišćenje tehnologije po izboru, odnosno nije vezana za određene tehnološke standarde i protokole. Procesi u SOA arhitekturi se mogu realizovati primenom bilo kojeg standarda interoperabilnosti, kao što su SOAP (*Simple Object Access Protocol*), ORB (*Object Request Broker*), DCOM (*Distributed Component Object Model*).

### 2.5.1. Definicije SOA

Svaka kompanija koja se bavi izradom softvera ima svoj pogled na SOA arhitekturu, a samim tim i svoju definiciju, pa tako IBM smatra da bi SOA trebalo da kombinuje poslovne logike, organizacije i informacione tehnologije. Definicija IBM-a je:

„Servisno orijentisana arhitektura je IT arhitektura prilagođena velikim preduzećima, a namenjena iskorišćenju IT resursa na zahev i prema potrebi. Ti resursi su reprezentirani kao poslovno orijentisani servisi koji se mogu koristiti i kombinovati kao dodatne vrednosti postojećem IT-u organizacije, ili kao podrška za poslovne aktivnosti. Osnovni strukturni element za SOA aplikacije je servis, za razliku od podsistema, sistema ili komponenti.“

Korporacija Gartner, Inc. ima svoju definiciju za SOA arhitekturu [8]:

„Servisno orijentisana arhitektura je softver dizajniran prema klijent/server modulu u kojem se aplikacija sastoji od softverskih servisa i korisnika tih servisa (klijenata). SOA se razlikuje od standardnog klijent/server modela u nastojanju na razdvajanju softverskih komponenti te u korišćenju različitih načina pristupa do tih servisa.“

Definicija iz [9] je:

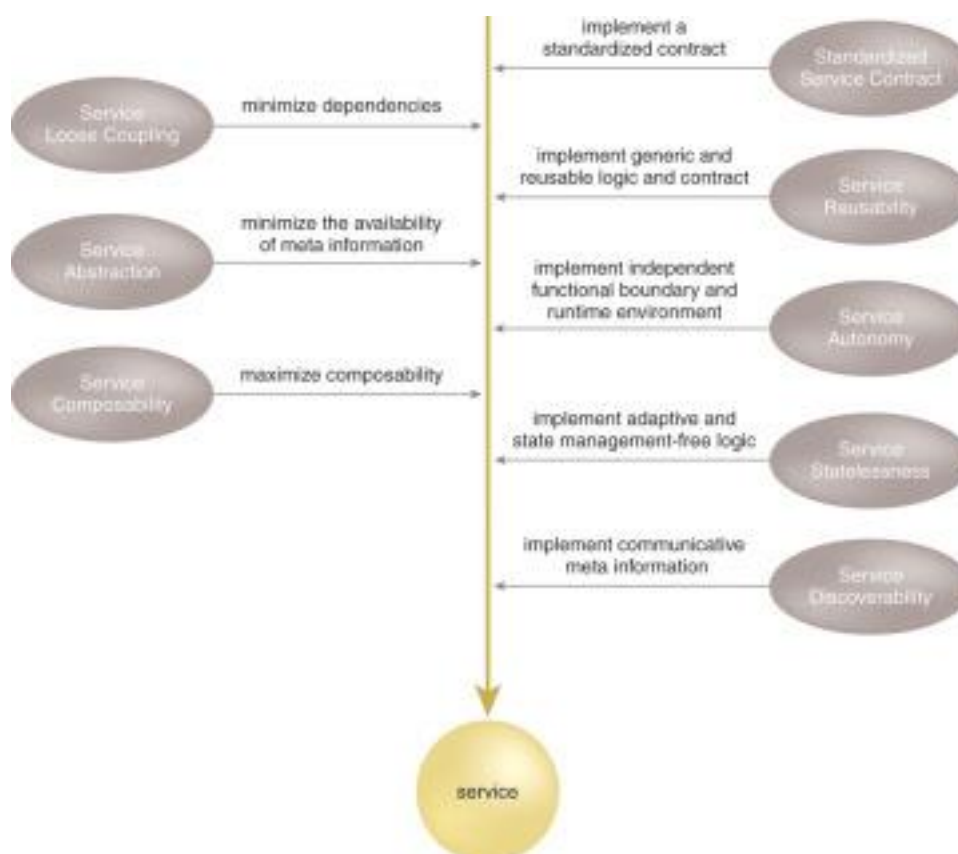
„Servisno orijentisana arhitektura je softverska arhitektura za izgradnju biznis aplikacija koje implementiraju biznis procese ili servise koristeći skup slabo povezanih *black-box* komponenti orkestriranih tako da pružaju dobro definisan servisni nivo.“

### 2.5.2. Principi SOA

SOA arhitektura je proces u kome se softverski sistem izrađuje iz delova koji čine mrežu poslovnih oblasti u okviru datog problema. Da bi na kraju sve korektno funkcionisalo, potrebno je unapred napraviti analizu sistema. Da bi se pomenuto postiglo, uvedeni su principi za realizaciju servisno-orijentisane arhitekture. Prema [10], osnovni principi SOA su (slika 6):

- standardizovani servisni ugovor (*Standardized Service Contract*)
- labava povezanost servisa (*Service Loose Coupling*)
- servisna apstrakcija (*Service Abstraction*)

- ponovna upotreba servisa (*Service Reusability*)
- autonomija servisa (*Service Autonomy*)
- servisi bez stanja (*Service Statelessness*)
- mogućnost pronalaženja servisa (*Service Discoverability*)
- mogućnost kompozicije servisa (*Service Composability*)



Slika 6. Prikaz SOA principa (Izvor. [10])

Upotreba navedenih principa obezbeđuje i olakšava izradu nezavisnih servisa i međusobnu komunikaciju pri izradi poslovnih sistema sa dužim vekom upotrebe.

### 2.5.3. Komponente SOA

U SOA arhitekturi, sve komponente poslovnog sistema međusobno komuniciraju i prosleđuju poruke i podatke, što povoljno utiče na brzinu i kvalitet izrade poslovnog sistema. Prema [11], najvažnije komponente SOA implementacije su:

- Servisna sabirnica kompanije
- SOA registar i repozitorij
- *Business proces orchestration manager*
- *Broker servisa*
- *SOA Service manager*

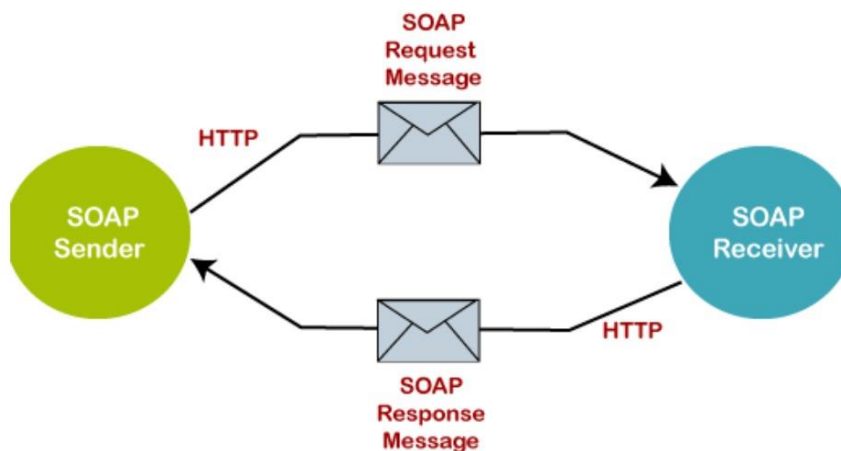
Svaka od navedenih komponenti ima svoju samostalnu ulogu ili je u relaciji sa drugim komponentama, sa istim ciljem, a to je što bolja organizacija strukture aplikacije.

#### 2.5.4. SOAP servisi

SOAP (*Simple Object Access Protocol*) je protokol koji se koristi za razmenu poruka i struktuiranih informacija pri implementaciji veb servisa u računarskim mrežama [12]. SOAP protokol koristi XML format za pristupanje veb servisima pomoću HTTP protokola. Najčešća primena SOAP protokola danas je razmena RPC poruka preko HTTP protokola, što je veoma značajno za klijentske aplikacije. Na taj način je omogućeno jednostavno povezivanje sa udaljenim servisima, pokretanje potrebnih metoda, što je veliko olakšanje pri povezivanju veb aplikacija.

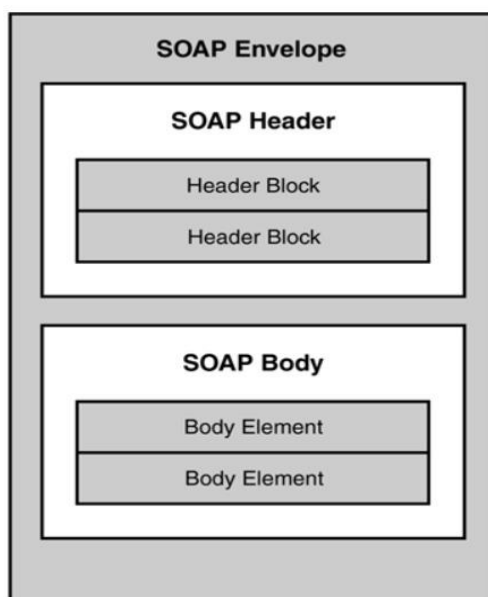
SOAP protokol omogućava međusobnu komunikaciju aplikacija napisanih na različitim programskim jezicima. Nezavisan je od korišene platforme i operativnog sistema. S obzirom da SOAP protokol funkcioniše iznad HTTP protokola, koji se koristi kod veb aplikacija, nije potrebna dodatna modifikacija da se veb servisi koji koriste SOAP protokol izvršavaju na veb-u.

SOAP komunikacija se odvija tako što pošiljalac (SOAP *sender*) šalje zahtev preko HTTP protokola prijemniku (SOAP *receiver*), gde se odvija obrada zahteva. Po završetku tog procesa, odgovor se šalje pošiljalcu (slika 7).



Slika 7. Komunikacija putem SOAP (Izvor: [13])

SOAP protokol sadrži u sebi XML dokument definisan kao SOAP poruka. To je dokument koji se šalje veb servisu i klijentskoj aplikaciji. SOAP poruka se sastoji od sledećih komponenti (slika 8): SOAP koverta (SOAP *Envelope*), SOAP zaglavlje (SOAP *Header*) i SOAP telo (SOAP *Body*).



Slika 8. Komponente SOAP poruke (Izvor: [14])

SOAP koverta je okvir u kome se nalaze zaglavlje i telo SOAP poruke. Ona služi za identifikaciju XML dokumenta, kao i za označavanje početka i kraja poruke.

SOAP zaglavlje sadrži detalje o poruci i sve ostale informacije koje su potrebne, poput podataka za autentifikaciju.

SOAP telo sadrži podatke koji se razmenjuju između veb servisa i aplikacija koje taj servis pozivaju.

Navedene komponente su osnovne komponente jednostavne SOAP poruke, a kao opcionu komponentu, poruka može da sadrži element greške što može da pomogne lakšem nalaženju greške, ukoliko se ona pojavi.



### 3. Mikroservisna arhitektura

Dizajnerski okvir servisa predstavlja vizuelnu strukturu koja poboljšava organizaciju informacija i praktičnost kod rešavanja problema [15]. Nudi timovima alate i tehnike za njihovo rešavanje.

Servisi obezbeđuju zajedničku platformu na kojoj mogu komunicirati različite aplikacije napisane u različitim programskim jezicima. Sposobnost servisa je da pruži sve funkcionalnosti koje su potrebne da bi se neometano obavila komunikacija između korisničkih aplikacija. Prvi servisi koji su se pojavili, a nisu vezani za arhitekturu su veb servisi. Oni za razmenu podataka koriste XML i JSON format.

Korisnici servisa mogu biti drugi servisi, veb servisi i drugi korisnici. Kad neki program pozove i komunicira sa servisom programa, tada i on postaje korisnik, ali privremeno dok koristi program za razmenu podataka.

Da bi aplikacija zadovoljila kriterijume korisnika, mora biti pažljivo dizajnirana. Sam proces razvoja olakšava odabir pravog arhitekturnog stila, koji ima veliki značaj za timove koji se bave razvojem, testiranjem i održavanjem aplikacije. Ranije se primarno koristila monolitna arhitektura pri razvoju softvera. Međutim, u poslednje vreme se češće koriste modernije, servisno-orijentisane arhitekture, kao što je mikroservisna.

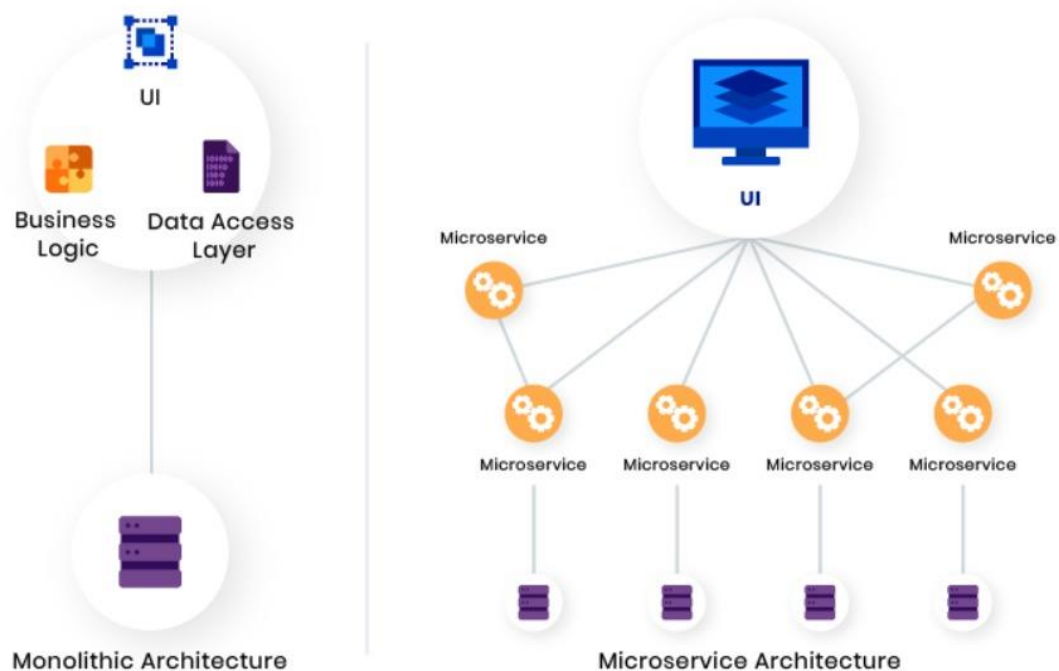
Monolitna arhitektura je zasnovana na objedinjavanju različitih komponenti softvera u jedan program na jednoj platformi, tako da se svim njegovim funkcijama upravlja sa jednog mesta. Ona sadrži bazu podataka (uglavnom jednu), korisnički interfejs na strani klijenta i aplikaciju na strani servera. Ova arhitektura se primenjuje za izradu manjih aplikacija. Pošto su komponente usko povezane, performanse su zadovoljavajuće. Ova arhitektura je pogodna za razvoj u manjim timovima.

Za razliku od monolitne arhitekture, mikroservisna arhitektura pristupa razvoju jedne aplikacije u vidu skupa malih usluga. Svaka od njih se pokreće u svom procesu i komunicira jednostavnim mehanizmima, često putem HTTP protokola. Postoji minimum centralizovanog upravljanja ovim uslugama, koje mogu biti napisane na različitim programskim jezicima i koristiti različite tehnologije za skladištenje podataka.

Mikroservisna arhitektura je zasnovana na nizu nezavisnih komponenti koje su direktno povezane sa API-jima. Da bi se zahtev klijenta ispunio, servisi moraju međusobno sarađivati. Komunikacija između servisa može biti sinhrona i asinhrona. Takođe, može biti direktna ili indirektna (komunikacija putem posrednika). Standardna komunikacija se obavlja pomoću REST-a, gRPC-a, itd.

Upotreba mikroservisne arhitekture povećava agilnost timova, što znači da se svaki deo aplikacije može realizovati nezavisno od ostalih komponenti. Razvoj se brže odvija jer se usluge, kao što je testiranje pojedinačnih delova, mogu nezavisno primeniti. Naime, nema potrebe da se čeka realizacija cele aplikacije, već je deo po deo spreman za upotrebu. Kod mikroservisa omogućeno je kreiranje više usluga u istom skupu, njihov broj nije ograničen i one se mogu dinamički pokrenuti. Da ne bi bilo propusta u primeni mikroservisne arhitekture, vrlo su bitni timska saradnja, posedovanje potrebnih veština,

pažljivo planiranje toka razvoja aplikacije, kao i oprez pri komunikaciji sa spoljašnjim uslugama.



Slika 9. Poređenje monolitne i mikroservisne arhitekture (Izvor: [15])

Monolitna arhitektura je orijentisana ka tehnološkim slojevima korisničkog interfejsa i baze podataka, dok je mikroservisna arhitektura okrenuta ka poslovnim potrebama i mogućnostima.

Servisno orijentisana arhitektura (SOA) kreirana je od labavo povezanih servisa koji međusobno komuniciraju preko ESB-a. Dve osnovne uloge SOA arhitekture su pružanje i korištenje usluga. Komponente SOA arhitekture se mogu koristiti u više aplikacija zbog svoje labave povezanosti.



Slika 10. Poređenje monolitne, SOA i mikroservisne arhitekture (Izvor: [15])

Monolitne aplikacije izgrađene su od međusobno zavisnih komponenti, njihova realizacija je sporija. SOA aplikacije su izgrađene od više labavo povezanih komponenti. Njihov razvoj je takođe sporiji, za razliku od mikroservisne aplikacije gde su mikroservisi veoma male komponente koje su labavo povezane, samostalne usluge koje pružaju konstantan i brži razvoj.

### 3.1. REST servisi

REST je skup arhitektonskih principa koji su prilagođeni za izradu lakih (*lightweight*) veb servisa i mobilnih aplikacija koje se lako održavaju (*maintanable*) i koje su skalabilne (*scalable*), tj. lako proširive. REST za komunikaciju koristi isključivo HTTP protokol, a za reprezentaciju podataka XML, JSON, YAML ili neki drugi format. Veb servisi čija je osnova izgrađena na REST principima se nazivaju RESTful servisi. Oni služe da vraćaju poruke u različitim formatima kao što su: JSON, XML, HTML. JSON se pokazao kao najbolji format za komunikaciju jer ga može pročitati bilo koji programski jezik [16].

REST je skraćenica od *Representational State Transfer* („prenos stanja reprezentacije“) i predstavlja skup ograničenja koja pri primeni na dizajn sistema realizuju arhitektonski stil. Prema [17], ograničenja REST servisa su:

- klijent – server  
Odnosi se na interakciju klijenta i servera koji imaju svoja nezavisna zaduženja. Ona se mogu menjati, dok se interfejs koji služi za komunikaciju ne menja.
- bez stanja  
Server ne pamti informacije koje dobija od klijenta, već je potrebno da klijent obezbedi sve potrebne informacije za taj poziv. Klijent čuva stanje sesije.
- podržava keširanje  
Odgovor na klijentski zahtev treba biti jasno definisan o tome da li podaci mogu da se keširaju ili ne. Ako odgovor bude potvrđan, podaci se mogu keširati i koristiti za odgovarajuće zahteve.
- uniformno dostupan  
Koristi se uniformni interfejs među komponentama koji obezbeđuje razdvajanje servisa implementacije od servisa usluga koje pruža. Da bi se ovo razdvajanje postiglo, postoje određena ograničenja (identifikacija resursa, manipulacija resursima, poruke koje same sebe opisuju i HTTP protokol za perzistenciju stanja aplikacije).
- slojevit  
Obezbeđuje mogućnost hijerarhijskog uređivanja slojeva. Komponente mogu komunicirati međusobno samo sa komponentama sloja sa kojim vrše interakciju.
- pristup kodu na zahtev (opciono)

Opciono ograničenje koje služi da se omogući proširenje klijentske funkcionalnosti, što olakšava implementaciju klijentske strane.

Pomenuta ograničenja ne utiču na odabir tehnologije koja će se koristiti za izradu aplikacije, već imaju zadatak da definišu prenos podataka između komponenti sistema.

Najvažnija apstrakcija u REST-u je resurs [18]. On predstavlja bilo koju informaciju koja se može imenovati. Na primer, resurs mogu da budu: slike, dokumenti, kolekcije drugih resursa, objekti, i sl.

REST servis funkcioniše tako što se podaci (resursi) koji se nalaze u njemu, prilikom slanja zahteva za određenim resursom od strane klijenta, pomoću određenog URL-a šalju serveru i od njega dobija odgovor.

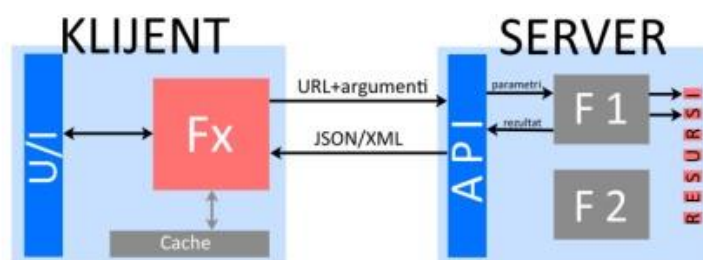
Prema [19], zahtev (URL kojem se pristupa) sadrži četiri komponente:

- *endpoint* koji ima ulogu URL-a sa korenom strukturom aplikacije
- metode koje se koriste GET, POST, PUT, PATCH, DELETE, tj. HTTP zahtevi
- zaglavlja koja sadrže sve potrebne informacije i obezbeđuju pristup određenim resursima kroz zahtev, kao što je autorizacija i slično
- telo zahteva koje šalje podatke ka serveru pomoću metoda (POST, PUT, PATCH, DELETE)

HTTP zahtevi omogućavaju rad sa bazama podataka:

- GET zahtev služi za čitanje zapisa i preuzimanje resursa (poput slika, dokumenata, itd.)
- POST zahtev služi za kreiranje novog zapisa
- PUT i PATCH zahtevi služe za ažuriranje zapisa, s tim što je PUT zahtevu potrebno proslediti sve informacije i on vrši ažuriranje kompletnog entiteta, a PATCH ima mogućnost da obavi ažuriranje samo dela entiteta
- DELETE zahtev služi za brisanje zapisa

Pomenuti HTTP zahtevi su RESTful metode koje se koriste kao servisi za izradu aplikacija u REST arhitekturi. Ključni elementi RESTful implementacije su: *endpoint*, HTTP metode, zaglavlje zahteva, telo zahteva, telo odgovora i statusni kodovi.



Slika 11. Prikaz načina komunikacije između korisnika i sistema (Izvor: [20])

Prikazani način komunikacije korisnika i sistema (slika 11), odnosi se na klijenta koji zahteva i server koji pruža odgovor. Korisnik (klijent) pomoću korisničkog interfejsa (UI) inicira Fx funkciju, koja kreira korisnički zahtev i šalje ga API funkciji na serveru. API predstavlja skup funkcija koje su na raspolaganju za upotrebu celom spoljnom svetu. API funkcije su mapirane na F1 i F2 funkcije, koje se nalaze unutar modula i služe za operacije nad resursima. Odgovor servera korisniku se vraća istim putem u vidu HTTP odgovora.

API (*Application Programming Interface*) predstavlja skup procedura i funkcija koje su na raspolaganju za pristup podacima, aplikacijama i sistemskim funkcijama na nekom sistemu. Kao odgovor vraća unapred definisani tip podataka. REST API je API koji se pridržava REST ograničenja.

REST nije protokol kao SOAP, već je koncept baziran na promeni stanja klijenta i sve akcije se vrše u trenutku promene tog stanja, sa mogućnošću vraćanja na neko od predhodnih stanja [21].

## 4. Opis aplikacije

U ovom radu je mikroservisna arhitektura primenjena za razvoj aplikacije koja se bavi iznajmljivanjem automobila. Aplikacija je namenjena *rent-a-car* agencijama i treba da im pomogne u obavljanju svakodnevnih poslova. U nastavku su detaljno opisani funkcionalni zahtevi koje aplikacija treba da ispuni, kao i prava pristupa pojedinim funkcionalnostima.

### 4.1. Funkcionalni zahtevi

Poslovanje *rent-a-car* agencije se može predstaviti skupom sledećih funkcionalnih zahteva koje aplikacija treba da implementira:

Zahtev 1: Osnovni podaci o vozilu

O svakom vozilu koje *rent-a-car* agencija poseduje treba da budu raspoloživi sledeći osnovni podaci:

- kategorija vozila (automobil, suv, putnički kombi)
- model vozila (po proizvođačima: Opel, Mazda, Fiat,...)
- fotografija vozila
- tip goriva koje vozilo koristi (benzin, dizel, gas)
- tip menjača u vozilu (manuelan, automatski)
- pređeni km (ukupan broj kilometara koje vozilo prešlo)
- registracija (vreme isticanja registracije vozila)

Zahtev 2: Dodatni podaci o vozilu

Ovo su podaci o vozilu koji su bitni pri njegovom iznajmljivanju:

- cena iznajmljivanja vozila/danu
- status vozila (da li je vozilo raspoloživo ili ne u datom trenutku)
- recenzije drugih korisnika koji su ranije iznajmljivali vozilo

Zahtev 3: Podaci o klijentu

Za svakog klijenta koji ima mogućnost iznajmljivanja vozila, treba da budu dostupni sledeći podaci:

- ima i prezime klijenta
- JMBG (identifikator klijenta)
- e-mail adresa klijenta
- telefon klijenta
- vreme važenja vozačke dozvole klijenta

Zahtev 4: Podaci o rentiranju vozila

Prilikom rentiranja vozila, klijent treba da unese sledeće podatke:

- lokacija sa koje želi da preuzme vozilo
- lokacija na koju želi da vrati vozilo

- vreme preuzimanja vozila
- vreme vraćanja vozila
- vreme važenja vozačke dozvole klijenta
- godine starosti klijenta

#### Zahtev 5: Otvaranje naloga

Prilikom otvaranja naloga (registracije na sistem), korisnik treba da unese sledeće podatke:

- ime i prezime, JMBG, e-mail adresu (Zahtev 3)
- lozinku

#### Zahtev 6: Prijava na sistem

Prilikom ulaska u aplikaciju, korisnik treba da unese:

- svoju e-mail adresu koja služi kao korisničko ime
- svoju lozinku

#### Zahtev 7: Pregled raspoloživih vozila

Korisniku treba omogućiti:

- prikaz liste svih trenutno raspoloživih vozila
- pristup osnovnim podacima o željenom vozilu iz liste (Zahtev 1)
- pristup dodatnim podacima o željenom vozilu iz liste (Zahtev 2)

#### Zahtev 8: Ažuriranje podataka o vozilu

Treba omogućiti ažuriranje podataka o voznom parku agencije putem:

- dodavanja novog vozila u ponudu
- isključivanja vozila iz ponude
- izmene podataka o postojećem vozilu (Zahtev 1 i Zahtev 2)

#### Zahtev 9: Filtriranje liste vozila

Treba omogućiti filtriranje liste vozila po:

- kategoriji vozila (automobil, suv, putnički kombi)
- tipu goriva koje vozilo koristi (benzin, dizel, gas)

#### Zahtev 10: Pregled raspoloživih klijenata

Korisniku treba omogućiti:

- prikaz liste svih klijenata
- pristup podacima o željenom klijentu (Zahtev 3) na osnovu njegovog JMBG

#### Zahtev 11: Ažuriranje podataka o klijentu

Treba omogućiti ažuriranje podataka o klijentima putem dodavanja novog klijenta.

Zahtev 12: Filtriranje liste klijenata

Treba omogućiti filtriranje liste klijenata po JMBG broju.

Zahtev 13: Rezervacija vozila

Korisnicima treba omogućiti:

- rentiranje vozila unosom podataka iz Zahteva 4
- unos recenzije za vozilo koje su koristili (Zahtev 2)

Zahtev 14: Ostvarivanje popusta

Redovnim korisnicima usluga agencije treba omogućiti popust u vidu bonusa koji se računa po sledećem pravilu: pri svakom trećem iznajmljivanju vozila, ostvaruje se popust od 5%.

#### 4.2. Korisnici i prava pristupa

Aplikaciji mogu da pristupe tri vrste korisnika: gost, klijent i administrator. Gost može samo da vidi pojedine podatke, bez mogućnosti bilo kakve interakcije sa aplikacijom. Klijent i administrator, osim pregleda podataka, mogu da ostvare i interakciju sa aplikacijom, ali samo u okviru funkcionalnosti koje su im dodeljene.

Gost ima pristup sledećim funkcionalnostima:

- može da pregleda listu raspoloživih vozila iz voznog parka agencije (Zahtev 7)
- može da pogleda osnovne i dodatne podatke o raspoloživim vozilima iz liste (Zahtev 1 i Zahtev 2)
- može da pregleda recenzije koje su raniji korisnici ostavili o vozilu (Zahtev 13)

Na slici 12 je dat dijagram slučaja korišćenja sa jedinom funkcionalnošću koja su dostupna gostu.



Slika 12. Slučaj korišćenja dostupan gostu

Klijent može da koristi sledeće funkcionalnosti aplikacije:

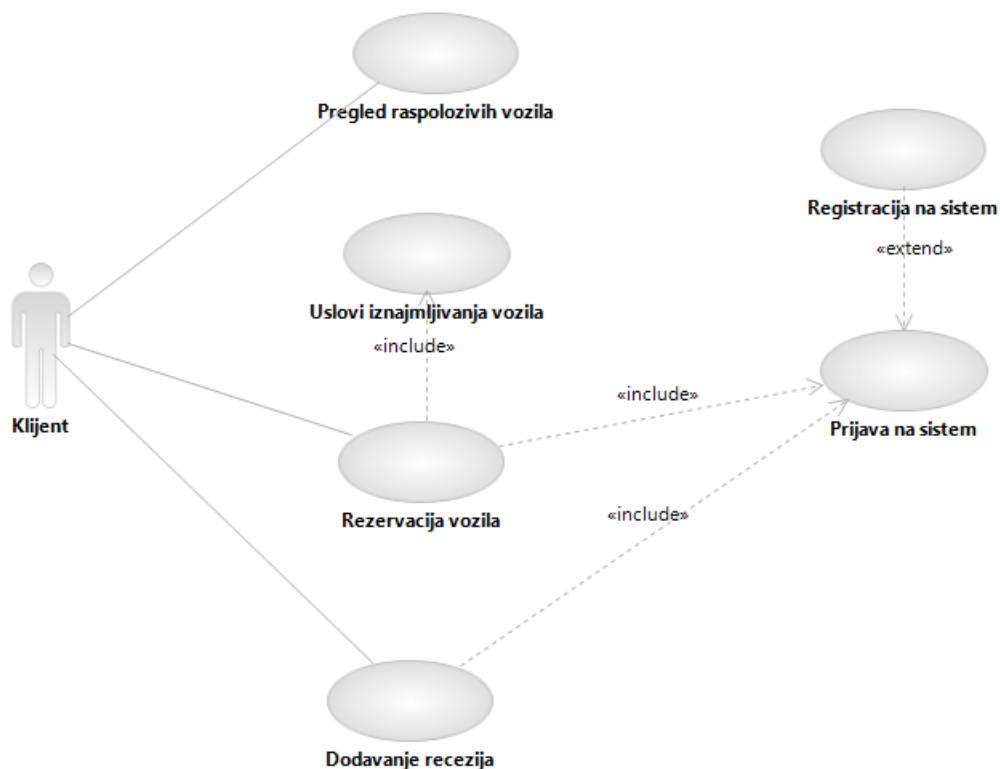
- da se uloguje u aplikaciju, ukoliko ima nalog (Zahtev 6); ako ga nema, može da se registruje i otvori nalog (Zahtev 7), pa onda da se uloguje
- da pregleda listu trenutno raspoloživih vozila (Zahtev 7)
- da rezerviše vozilo koje želi da iznajmi (Zahtev 13); da bi rezervacija bila uspešna potrebno je ispoštovati određena ograničenja (datum isteka vozačke



dozvole i godine starosti vozača); kada klijent pokrene rezervaciju, sistem proverava ova ograničenja; ukoliko su ona validna, klijent na e-mail dobija poruku o uspešnoj rezervaciji; u slučaju nevalidnih rezultata, sistem ne dozvoljava rezervaciju odabranog automobila

- da ostavi recenziju nakon vraćanja iznajmljenog vozila (Zahtev 13)

Slika 13 pokazuje koje su funkcionalnosti dostupne klijentu.

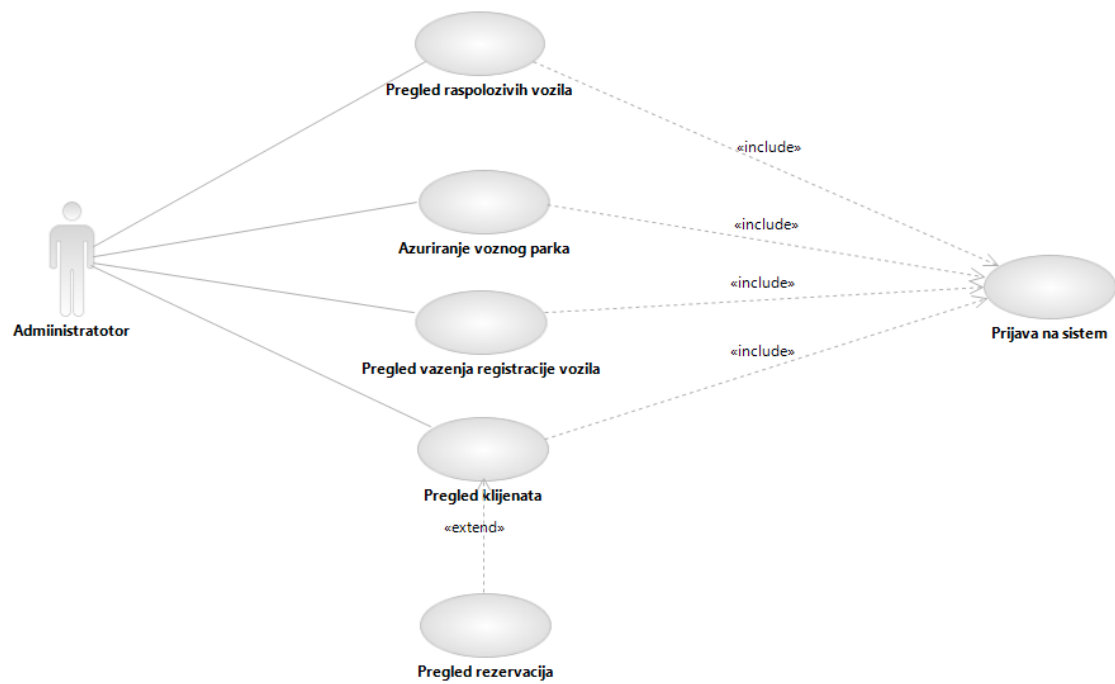


Slika 13. Slučajevi korišćenja dostupni klijentu

Administrator može da pristupi sledećim funkcionalnostima aplikacije:

- da se prijavi na sistem pomoću svog korisničkog imena i lozinke, koji su kreirani na nivou agencije
- da pregleda listu trenutno raspoloživih vozila (Zahtev 7)
- da ažurira podatke o vozilima, kako osnovne, tako i dodatne (Zahtev 8)
- da pregleda listu klijenata koji su registrovani na platformi (Zahtev 10)
- da za izabranog klijenta pogleda podatke o njegovim rezervacijama i povlasticama (bonusima) (Zahtev 14)

Na slici 14 su date funkcionalnosti dostupne administratoru.



Slika 14. Slučajevi korišćenja dostupni administratoru

## 5. Realizacija aplikacije

U ovom poglavlju je opisana realizacija aplikacije koja ispunjava prethodno navedene funkcionalne zahteve bitne za svakodnevno poslovanje *rent-a-car* agencije. Postupak realizacije je predstavljen kroz arhitekturu aplikacije, korišćeno razvojno okruženje, primenjene alate i tehnologije, kao i kroz detalje implementacije.

### 5.1. Arhitektura aplikacije

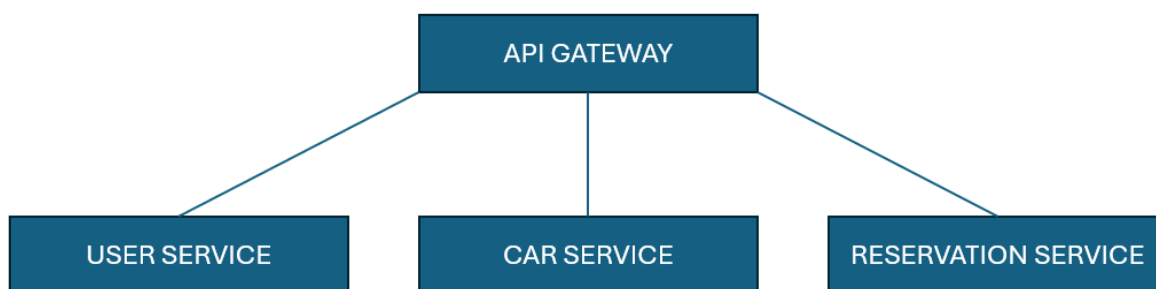
Arhitekturni stil aplikacije koja je realizovana je klijent - server. Klijentski deo je na raspolaganju korisnicima aplikacije, a serverski deo realizuje poslovnu logiku i omogućuje interakciju sa klijentskim delom aplikacije. Za potrebe kreiranja serverskog dela aplikacije korišćena je mikroservisna arhitektura. Primena osnovnih principa mikroservisne arhitekture omogućila je razlaganje serverskog dela aplikacije na tri mikroservisne usluge. U cilju postizanja komunikacije sa spoljnim svetom, kreiran je mrežni prolaz koji, takođe, služi i za povezivanje mikroservisnih usluga.

Na slici 15 je prikazana mikroservisna struktura sistema. Kao što se vidi, sistem se sastoji od sledećih mikroservisa:

1. USER SERVICE, koji sadrži sve komponente neophodne za korisničke naloge i ostale usluge u vezi sa korisnicima
2. CAR SERVICE, koji obuhvata sve potrebne usluge vezane za vozila koja se rentiranju
3. RESERVATION SERVICE, koji sadrži sve potrebne usluge vezane za rezervacije vozila

Navedene mikroservise povezuje API GATEWAY koji je realizovan kao klasična monolitna aplikacija. On ne sadrži bazu podataka, već samo kontrolere koji omogućavaju komunikaciju mikroservisa sa spoljnim svetom i servise koji omogućavaju komunikaciju sa mikroservisima. Komunikacija API GATEWAY-a i mikroservisa postiže se pomoću TCP protokola.

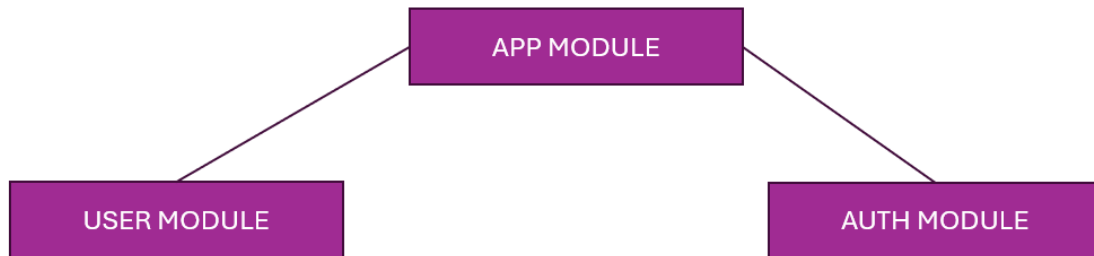
Mikroservisi su realizovani kao izolovane aplikacije, svaka poseduje svoju bazu podataka sa kojom radi.



Slika 15. Struktura sistema

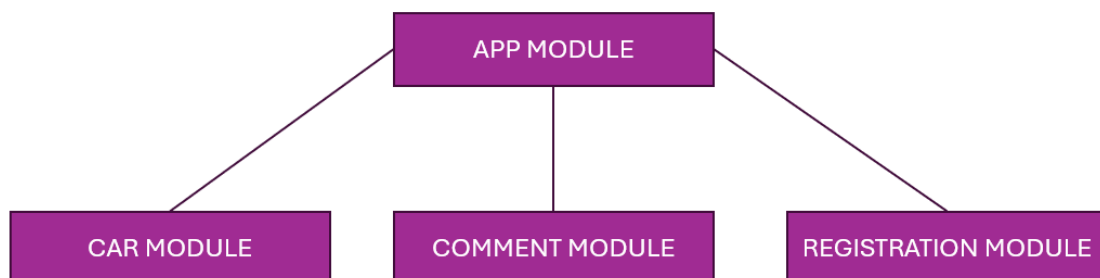
Kao što je prikazano na slici 16, mikroservis USER SERVICE se sastoji od tri

modula. APP MODULE je koreni modul u kome se nalazi sve što je potrebno za konekciju sa bazom podataka. USER MODULE sadrži funkcionalnosti koje se odnose na pretragu korisnika. AUTH MODULE sadrži funkcionalnosti potrebne za autentifikaciju korisnika (klijenta i administratora).



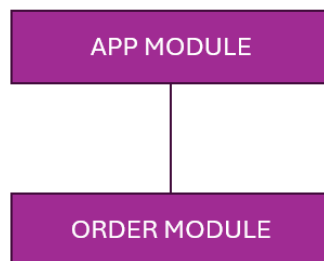
Slika 16. Struktura USER SERVICE mikroservisa

Struktura CAR SERVICE mikroservisa (slika 17) obuhvata APP MODULE koji ima sličnu ulogu kao kod USER SERVICE mikroservisa. Tu su još i CAR MODULE, koji sadrži funkcionalnosti za pretragu, unos i ažuriranje podataka o vozilima, COMMENT MODULE, u kome se nalaze funkcije za unos komentara od strane korisnika i REGISTRATION MODULE koji sadrži funkcije za unos i prikaz podataka o registraciji vozila.



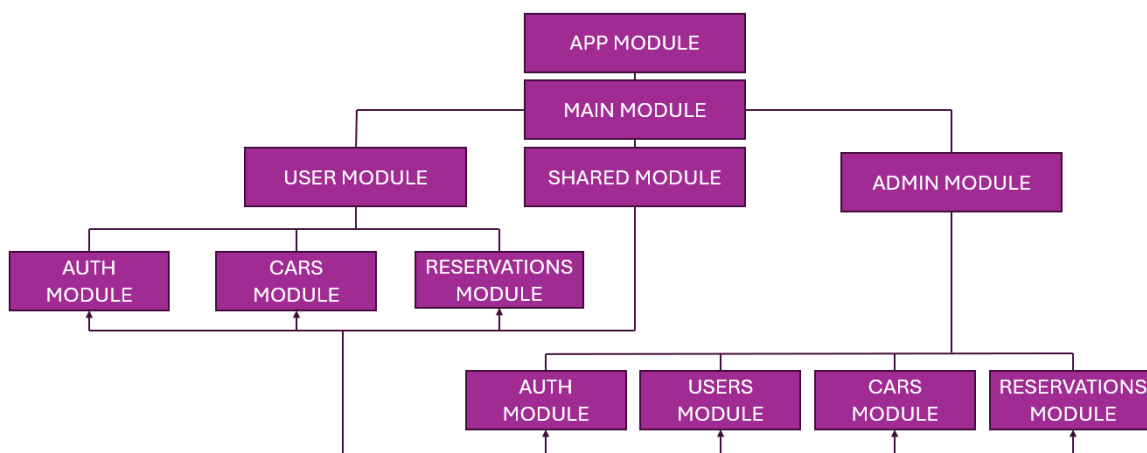
Slika 17. Struktura CAR SERVICE mikroservisa

Na slici 18, prikazana je struktura RESERVATION SERVICE mikroservisa. U okviru njega se nalaze APP MODULE (ima sličnu funkciju kao i kod druga dva mikroservisa) i ORDER MODULE, koji sadrži funkcije za pretragu unetih rezervacija i slanje e-mail poruka o potvrdi rezervacije korisniku i administratoru.



Slika 18. Struktura RESEVATION SERVICE mikroservisa

Struktura API GATEWAY aplikacije koja služi kao mrežni prolaz između mikroservisa i spoljnog sveta je data na slici 19. U njoj postoje tri glavna modula USER MODULE, SHARED MODULE i ADMIN MODULE, kao i MAIN MODULE.



Slika 19. Struktura API GATEWAY aplikacije (mrežni prolaz)

MAIN MODULE služi da enkapsulira glavne module. Unutar MAIN MODULA importovani su USER MODULE, SHARED MODULE i ADMIN MODULE.

USER MODULE obuhvata module koji u sebi imaju kontrolere i servise koji obezbeđuju pristup uslugama iz mikroservisa koje su potrebne korisniku i gostu. Unutar USER MODULA importovani su sledeći moduli: AUTH MODULE pruža uslugu za prijavu i registraciju korisnika (klijent i gost), CARS MODULE pruža usluge za informacije o vozilima, RESERVATIONS MODULE omogućava rezervaciju određenog vozila, ova usluga dostupna je klijentu.

SHARED MODULE poseduje konfiguraciju koja obezbeđuje komunikaciju sa mikro uslugama.

ADMIN MODULE sadrži module koji u sebi imaju kontrolere i servise koji obezbeđuju pristup uslugama iz mikroservisa koje su potrebne administratoru. Unutar ADMIN MODULE importovani su sledeći moduli: AUTH MODULE pruža uslugu za prijavu administratora. USERS MODULE pruža uslugu pristupu informacija o klijentu (pregled liste klijenata, pretraga po JMBG). CARS MODULE pruža uslugu manipulisanja

podacima o vozilima (ažuriranje i unos novog vozila). *RESERVATIONS MODULE* pruža uslugu pregleda informacija o rezervacijama određenog klijenta (lokacija preuzimanja i vraćanja vozila, vreme preuzimanja i vraćanja vozila, marka vozila i važenje vozačke dozvole).

Primena mikroservisne arhitekture u razvoju je omogućila lakšu organizaciju i realizaciju aplikacije, kao i postizanje boljih performansi sistema. Da bi mikroservisna arhitektura bila implementirana u ovoj aplikaciji, korišćene su tehnologije, alati i razvojno okruženje opisani u nastavku.

## **5.2. Razvojno okruženje, alati i tehnologije**

Kao što je ranije rečeno, aplikaciju čine serverski i klijentski deo. Za realizaciju oba dela je korišćeno razvojno okruženje *Visual Studio Code Editor*. Kod je pisan na *TypeScript* programskom jeziku.

U serverskom delu aplikacije je korišćena relacionalna *MySQL* baza podataka u *NestJS* okviru.

Klijentski deo aplikacije izrađen je od dve odvojene aplikacije: jedna je namenjena administratoru, a druga klijentu i gostu. Za izradu ovih aplikacija korišćen je *TypeScript* programski jezik u *Angular* okviru. Za potrebe stilizovanja korišćena je *Bootstrap* biblioteka, takođe iz *Angular* okvira.

*TypeScript* [22] je programski jezik koji pruža dodatnu funkcionalnost *JavaScript* jeziku. On je statički tipiziran jezik (striktно vodi računa o tipovima podataka) *JavaScript*-a, koji se nadovezuje na njegovu postojeću funkcionalnost i sintaksu. Da bi se program na *TypeScript* pokrenuo u veb pregledačima i okruženju kao što je *NodeJS*, mora biti transformisan u *JavaScript*.

*MySQL* [23] je sistem za upravljanje relacionim bazama podataka otvorenog koda. Relacionalna baza podataka organizuje podatke u jednu ili više tabela. Podaci mogu biti povezani jedni sa drugima, što olakšava strukturiranje podataka i manipulaciju nad njima. *SQL* je programski jezik koji se koristi za kreiranje, modifikovanje i izdvajanje podataka iz relacione baze podataka, kao i za kontrolu pristupa bazi podataka.

*Bootstrap* [24] je *CSS* okvir otvorenog koda koji omogućuje brz razvoj *front-end* dela aplikacije. Može da sadrži šablone dizajna zasnovane na *JavaScript*-u, što omogućava korišćenje komponenti interfejsa kao što su dugmad, navigacija, obrasci, itd..

Glavni alati i okruženja koji su korišćeni u razvoju, opisani su u nastavku.

### **5.2.1. Visual Studio Code Editor**

*Visual Studio Code Editor* je lagan, ali moćan uređivač izvornog koda koji je raspoloživ za rad razvojnim timovima (programerima) [25]. Dostupan je na vebu za različite operativne sisteme (*Windows*, *Linux*, *MacOs*). Ima ugrađenu podršku za *JavaScript*, *TypeScript* i *NodeJS* i ekstenziju za mnoge programske jezike (*C++*, *C#*, *Java*, *Python*...). U slučaju da podrška nije dostupna, postoji mogućnost za dodavanje osnovne

podrške za određeni programski jezik.

*Visual Studio Code Editor* poseduje *IntelliSense* dovršavanje koda za promenljive, metode i uvezene module, brzu navigaciju i refaktorisanje koda. Takođe, ima ugrađenu podršku za verzionisanje koda (*Git*).

*Visual Studio Code Editor* je realizovan korišćenjem datoteke *Electron Shell*-a, *NodeJS*-a, *TypeScript*-a i jezičkog serverskog protokola. Ažuriranje se vrši na mesečnom nivou, mada se mnoga proširenja ažuriraju po potrebi. Isporučuje se pod standardnom licencom za *Microsoft* proizvod. Uprkos komercijalnoj licenci, on je besplatan.

### **5.2.2. Angular Framework**

*Angular* je veb okvir koji omogućava kreiranje brze i pouzdane dinamičke veb aplikacije [26]. U *Angular*-u se mogu koristiti kako standardni *HTML* i *CSS*, tako i proširenje *HTML*-a.

U *Angular*-u su dostupne komponente koje su korisne i lake za upotrebu. *Angular*-ov *data binding* i *dependency injection* omogućavaju redukovanje koda. *Angular* se koristi za realizaciju klijentskog dela aplikacije. Da bi se aplikacija upotpunila, potrebno je dodati serverski deo aplikacije.

Aplikacija realizovana u *Angular*-u se sastoji od modula koji su osnovni gradivni delovi. Moduli su logičke celine koje obavljaju određenu funkcionalnost. Svaki modul se sastoji od komponente, a komponenta kontroliše deo ekrana koji se naziva pogled i obezbeđuje deo funkcionalnosti za aplikaciju. Komponenta sadrži klasu, metapodatak koji je zadužen za opis i proširenje klase, kao i šablon koji služi za definisanje *HTML* pogleda.

### **5.2.3. NodeJS**

*NodeJS* [27] je višeplatformsko *JavaScript* okruženje otvorenog koda koje može da funkcioniše na različitim operativnim sistemima (*Windows*, *Linux*, *MacOs*...). *NodeJS* omogućava pisanje alata komandne linije i skriptovanje na strani servera pomoću *JavaScript*-a. *NodeJS* omogućava da se serverski i klijentski deo aplikacije realizuju korišćenjem jednog programskog jezika.

*NodeJS* radi u jednom procesu, bez kreiranja nove niti za svaki zahtev. Ima arhitekturu vođenu događajima sposobnu za asinhroni INPUT/OUTPUT. Cilj ove arhitekture je da omogući propusnost i skalabilnost u veb aplikacijama sa većim brojem INPUT/OUTPUT operacija. Pri izvršavanju operacije, kao što je čitanje sa mreže, pristup bazi podataka ili sistemu datoteke, neće blokirati nit i trošiti procesorski ciklus na čekanju, već će izvršiti operaciju kada se odgovor vrati. Ovakav način funkcionisanja omogućuje upravljanje velikim brojem istovremenih zahteva sa jednim serverom bez uvođenja upravljanja paralelnim nitima jer bi to uzrokovalo greške.

*Node JS* u ovoj aplikaciji ima ulogu platforme za realizaciju klijentskog i serverskog dela aplikacije, kao i za instalaciju paketa koji su potrebni da bi aplikacije funkcionisale. Za serverski deo aplikacije neki od paketa su *npm* i *mysql bcrypt*, a za klijentski deo aplikacije *npm* i *bootstrap bootstrap-icons*.

#### 5.2.4. NestJS Framework

*NestJS* [28] je okvir (*framework*) za realizaciju efikasnih i skalabilnih *NodeJS* aplikacija na strani servera, koristeći *JavaScript* ili *TypeScript*. *NestJS* obezbeđuje pouzdanu platformu za realizaciju veb aplikacija zahvaljujući asinhronom načinu izvršavanja pojedinih zahteva vođenim događajima koji potiču od *NodeJS*-a. *NestJS* se koristi za izradu monolitnih i mikroservisnih aplikacija, za kreiranje *REST API*, *MVC* aplikacija, itd.

*NestJS* aplikacije uglavnom su pisane u *TypeScript*-u, pa je otkrivanje grešaka u fazi kompajliranja način da se kod zaštiti kao i da se izbegnu greške kod mikroservisa koji imaju isti način odgovora.

*NestJS* velikim delom zavisi od *Angular*-a. Deli datoteke u više modula što olakšava organizaciju baze koda, kao i fokusiranje na određenu funkcionalnost.

*NestJS* u ovoj aplikaciji je primenjen za izradu mikroservisa, kao i za regulisanje bolje organizacije baze koda.

#### 5.2.5. HeidiSQL

*HeidiSQL* je alat za upravljanje relacionim bazama podataka [29]. Otvorenog je koda i dostupan za rad pod *Windows* operativnim sistemom. Posедуje funkcije za upravljanje bazama podataka, od kreiranja, pa sve do izvoza podataka, u vidu *dump* ili *csv* datoteke. Sadrži integrisanu pomoć za *SQL* jezik. Ima mogućnost povezivanja na više lokalnih ili udaljenih servera baze podataka i može da se koristi sa parametrima u komandnoj liniji.

Baze podataka koje podržava *HeidiSQL* su *MariaDB*, *MySQL*, *PostgreSQL*, *SQL Server* i *SQLite*. *HeidiSQL* je višejezična alatka koja je dostupna na više jezika. Dostupan je kao softver otvorenog koda dugi niz godina, redovno se održava i unapređuje.

### 5.3. Implementacioni detalji

Pre implementacije servisa, bilo je potrebno osmisliti kako će sistem funkcionisati, kako ravnomerno rasporediti funkcionalnosti i kako organizovati klijentski deo da bi upotreba aplikacije bila jednostavna i na pravi način dostupna svim korisnicima.

#### 5.3.1. Implementacija mikroservisa

Postupak kreiranja jednog mikroservisa (*USER SERVICE*) je prikazan na 20. Aplikacija u *Nest Framework*-u se kreira pomoću komande `nest new <app-name>`. Prilikom pokretanja ove komande, kreira se novi projekat pod nazivom koji odgovara `app-name`. Da bi ta aplikacija postala mikroservis, potrebno je dodati zavisnost `@nestjs/mikroservices` pomoću *node packet management*-a komandom (`npm i @nestjs/mikroservices`) iz *NestFactory*-a pozvati njenu metodu `createMicroservice()` i proslediti joj dva argumenta. Prvi argument je koreni modul aplikacije (`AppModule`), a drugi argument je opcija mikroservisa, kao što je transportni sloj koji je korišćen i dodatne opcije kao što su naziv hosta, na koji je potrebno hostovati aplikaciju i broj porta na koji se



osluškuje ista.

```
import { NestFactory } from '@nestjs/core';
import { AppModule } from './app.module';
import { MicroserviceOptions, Transport } from '@nestjs/microservices';

async function bootstrap() {
  const app = await NestFactory.createMicroservice<MicroserviceOptions>(AppModule, {
    transport: Transport.TCP,
    options: {
      host: 'localhost',
      port: 3001
    }
  });
  await app.listen();
}
bootstrap();
```

Slika 20. Definisanje mikroservisa

Način konfigurisanja komunikacije sa mikroservisima se može videti na slici 21. Da bi komunikacija među mikroservisima bila moguća, upotrebljen je TCP protokol.

```
shared.module.ts U X
back-end > api-gateway > src > main > shared > TS shared.module.ts > SharedModule

7  @Module({
8    imports: [
9      ClientsModule.registerAsync([
10     {
11       inject: [ConfigService],
12       name: USER_SERVICE,
13       useFactory: async (configService: ConfigService) => ({
14         transport: Transport.TCP,
15         options: {
16           host: configService.get<string>('USER_SERVICE_HOST'),
17           port: configService.get<number>('USER_SERVICE_PORT'),
18         }
19       }),
20     },
21     {
22       inject: [ConfigService],
23       name: CAR_SERVICE,
24       useFactory: async (configService: ConfigService) => ({
25         transport: Transport.TCP,
26         options: {
27           host: configService.get<string>('CAR_SERVICE_HOST'),
28           port: configService.get<number>('CAR_SERVICE_PORT'),
29         }
30       }),
31     },
32     {
33       inject: [ConfigService],
34       name: RESERVATION_SERVICE,
35       useFactory: async (configService: ConfigService) => ({
36         transport: Transport.TCP,
37         options: {
38           host: configService.get<string>('RESERVATION_SERVICE_HOST'),
39           port: configService.get<number>('RESERVATION_SERVICE_PORT'),
40         }
41       }),
42     }
43   ]),
44 }
```

Slika 21. Konfigurisanje komunikacije sa mikroservisima

Kreiran je novi modul pomoću komande `nest g mo <name-module>`. U konkretnom slučaju, `name-module` je zamenjen sa `shared`. U njega je potrebno importovati `ClientModule` i koristiti njegovu asinhronu metodu `registerAsync()`. Ova metoda omogućava upotrebu `configService` pošto se u `.env` fajlu nalaze hostovi i portovi mikroservisa. Takođe je potrebno eksportovati `ClientModule` da bi bio vidljiv ostalim modulima koji vrše interakciju sa određenim mikroservisom.

Pozivanje određene usluge mikroservisa se vrši pomoću koda na slici 22. Konkretno, u ovom delu koda prikazano je pozivanje usluge iz mikroservisa `CAR SERVICE`.

Kreiran je novi servis pomoću komande `nest g s <name-service>`. U konkretnom slučaju `name-service` zamenjen je sa `cars`. Upotrebom *dependency injection* tehnike, kreirana je instanca koja mikrousluži korišćenjem klase `ClientProxy` koja se nalazi u `@nestjs/microservice` paketu. Ona omogućava slanje zahteva mikrousluži pomoću metode `send()` i vraća odgovor.



```
TS cars.service.ts U X
back-end > api-gateway > src > main > user > cars > TS cars.service.ts > ...
1  import { Inject, Injectable } from '@nestjs/common';
2  import { ClientProxy } from '@nestjs/microservices';
3  import { CAR_SERVICE } from 'src/config/services';
4
5  @Injectable()
6  export class CarsService {
7    constructor(@Inject(CAR_SERVICE) private _carService: ClientProxy) {}
8    availableCars = () => this._carService.send({cmd: 'cars_is_available'}, '')
9    singleCar      = (id: number) => this._carService.send({cmd: 'single_car_with_comment'}, id)
10   addComments    = (data: any) => this._carService.send({cmd: 'add_comment'}, data)
11   search         = (data: any) => this._carService.send({cmd: 'search'}, data)
12 }
13
```

Slika 22. Pozivanje usluga iz mikroservisa

Na slici 23 je prikazan način implementacije kontrolera unutar određenog mikroservisa (`RESERVATION SERVICE`). Kreiran je novi kontroler pomoću komande `nest g co <name-controller>`. U ovom slučaju `name-controller` zamenjen je sa `order`. Instanca servisa koja omogućava komunikaciju sa bazom podataka kreirana je upotrebom *dependency injection*-a. Jedina razlika između klasične aplikacije i mikroservisa je to što se umesto standardnih HTTP metoda koriste `MessagePattern` iz `@nestjs/microservices`.

```

TS order.controller.ts U X
back-end > reservation-service > src > order > controllers > TS order.controller.ts > ...
1 import { Controller } from '@nestjs/common';
2 import { Order } from '../entities/order.entity';
3 import { ApiResponse } from 'src/misc/api.response';
4 import { OrderService } from '../services/order.service';
5 import { MessagePattern, Payload } from '@nestjs/microservices';
6
7 @Controller('order')
8 export class OrderController {
9   constructor(private readonly _orderService: OrderService) {}
10
11   @MessagePattern({cmd: 'add_reservation'})
12   create(@Payload() data): Promise<Order|ApiResponse> {
13     const dateValidLicense = ((new Date(data.data.dateExpireLicense).getTime() - new Date(Date.now()).getTime()) / (1000 * 60 * 60)) > 15;
14     if (dateValidLicense < 15) {return Promise.resolve(new ApiResponse("error", -121, "License expire"))}
15     return this._orderService.save(data)
16   }
17
18   @MessagePattern({cmd: 'all'})
19   all(): Promise<Order[]> {
20     return this._orderService.findAll()
21   }
22
23   @MessagePattern({cmd: 'user_all_reservation'})
24   allByUser(id: number): Promise<Order[]> {
25     return this._orderService.findByUser(id)
26   }
27
28   @MessagePattern({cmd: 'search'})
29   search(@Payload() data) {
30     return this._orderService.search(data)
31   }
32 }
33
34

```

Slika 23. Implementacija kontrolera unutar mikroservisa

Komunikacija aplikacije sa spoljnim svetom se ostvaruje na način prikazan na slici 24. Konkretno, ovaj deo koda prikazuje pozivanje funkcija sa slike 22. Korisnik usluge aplikacije može da zatraži određenu funkcionalnost, kao što je dostupnost vozila, pretraga po karakteristikama, prikaz podataka o određenom vozilu i dr.

```

TS cars.controller.ts U X
back-end > api-gateway > src > main > user > cars > TS cars.controller.ts > ...
1 import { Body, Controller, Get, Param, Post, Query } from '@nestjs/common';
2 import { CarsService } from './cars.service';
3
4 @Controller('')
5 export class CarsController {
6   constructor(private readonly carService: CarsService) {}
7   @Get('')
8   availableCar() {
9     return this.carService.availableCars()
10   }
11   @Get('/:id/detail')
12   singleCar(@Param() id: number) {
13     return this.carService.singleCar(id)
14   }
15   @Get('v')
16   search(@Query() data: any) {
17     return this.carService.search(data)
18   }
19   @Post('comment')
20   saveComment(@Body() data: any) {
21     return this.carService.addComments(data)
22   }
23 }
24

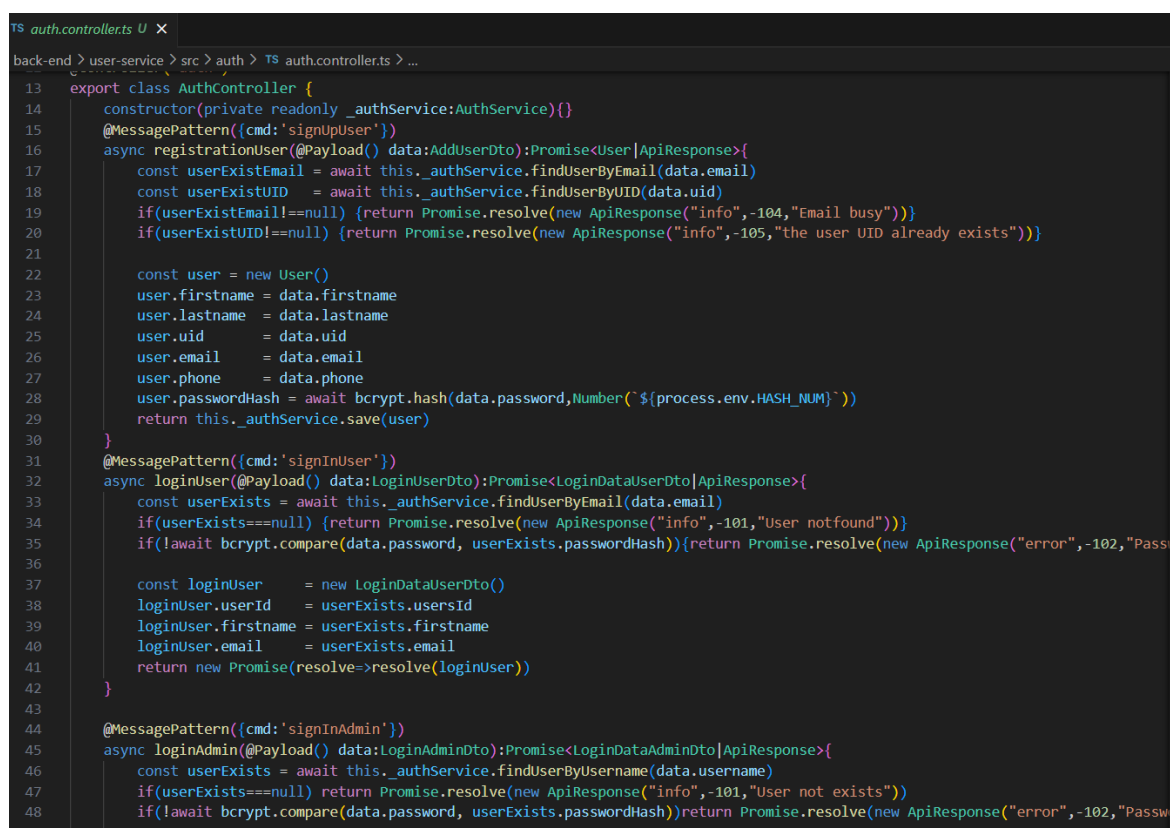
```

Slika 24. Komunikacija aplikacije sa spoljnim svetom

Kreiran je novi kontroler pomoću komande `nest g co <name-controller>`. U ovom slučaju, `name-controller` zamenjen je sa `cars`. Instanca servisa sa slike 22, kreirana je korišćenjem tehnike *dependency injection*-a. Mapiranjem na određenu rutu, pozivaju se metode sa slike 22, izvršava zahtev i vraća odgovor.

U mikroservisu `USER SERVICE` se nalazi kod (slika 25) koji omogućava prijavljivanje novog klijenta na sistem, kao i autentifikaciju klijenta i administratora.

Pomoću komande `nest g co <name-controller>` kreiran je kontroler. U ovom slučaju `name-controller` zamenjen je sa `auth`. Korišćenjem tehnike *dependency injection*, kreirana je instance `auth` servisa.



```

13 export class AuthController {
14   constructor(private readonly _authService: AuthService) {}
15   @MessagePattern({cmd: 'signUpUser'})
16   async registrationUser(@Payload() data: AddUserDto): Promise<User | ApiResponse> {
17     const userExistEmail = await this._authService.findUserByEmail(data.email)
18     const userExistUID = await this._authService.findUserByUID(data.uid)
19     if (userExistEmail !== null) {return Promise.resolve(new ApiResponse("info", -104, "Email busy"))}
20     if (userExistUID !== null) {return Promise.resolve(new ApiResponse("info", -105, "the user UID already exists"))}
21
22     const user = new User()
23     user.firstname = data.firstname
24     user.lastname = data.lastname
25     user.uid = data.uid
26     user.email = data.email
27     user.phone = data.phone
28     user.passwordHash = await bcrypt.hash(data.password, Number(`${process.env.HASH_NUM}`))
29     return this._authService.save(user)
30   }
31   @MessagePattern({cmd: 'signInUser'})
32   async loginUser(@Payload() data: LoginUserDto): Promise<LoginDataUserDto | ApiResponse> {
33     const userExists = await this._authService.findUserByEmail(data.email)
34     if (userExists === null) {return Promise.resolve(new ApiResponse("info", -101, "User not found"))}
35     if (!await bcrypt.compare(data.password, userExists.passwordHash)) {return Promise.resolve(new ApiResponse("error", -102, "Passw
36
37     const loginUser = new LoginDataUserDto()
38     loginUser.userId = userExists.userId
39     loginUser.firstname = userExists.firstname
40     loginUser.email = userExists.email
41     return new Promise(resolve => resolve(loginUser))
42   }
43
44   @MessagePattern({cmd: 'signInAdmin'})
45   async loginAdmin(@Payload() data: LoginAdminDto): Promise<LoginDataAdminDto | ApiResponse> {
46     const userExists = await this._authService.findUserByUsername(data.username)
47     if (userExists === null) return Promise.resolve(new ApiResponse("info", -101, "User not exists"))
48     if (!await bcrypt.compare(data.password, userExists.passwordHash)) return Promise.resolve(new ApiResponse("error", -102, "Passw
49

```

Slika 25. Autentifikacija klijenata i administratora

Pristup podacima u bazi podataka kako bi se obavio posao koji zahteva administrator (pregled liste klijenata, pregled podataka određenog klijenta po JMBG-u) prikazan je na slici 26.

Komandom `nest g s <name-service>` kreiran je novi servis, pri čemu je `name-service` zamenjen sa `user`. Da bi se sprovela komunikacija sa bazom podataka potrebno je instalirati sledeće zavisnosti pomoću *node packet management*-a (*npm*): `mysql`, `typeorm`, `@nestjs/typeorm`. Upotrebom tehnike *dependency injection*, kreirana je instance repozitorijuma `user` entiteta koja omogućava elegantnije pisanje upita. Primena ove instance zamenila je klasično pisanje upita pomoću standardnog *query*-a.

```

TS user.service.ts U X
back-end > user-service > src > user > TS user.service.ts > ...
1 import { Injectable } from '@nestjs/common';
2 import { InjectRepository } from '@nestjs/typeorm';
3 import { User } from './user.entity';
4 import { Like, Repository } from 'typeorm';
5
6 @Injectable()
7 export class UserService {
8   constructor(@InjectRepository(User) private readonly _userRepo:Repository<User>){}
9   findAllUsers = ():Promise<User[]>=>this._userRepo.find()
10  findUserByUID = (uid:string):Promise<User[]|null>=>this._userRepo.findBy({uid:Like(`${uid}%`)})
11 }

```

Slika 26. Pristup podacima o klijentu u bazi

Kreiranje šablona i funkcije da bi klijent i administrator dobili potvrde o rezervaciji vozila prikazano je na slici 27.

Kreiran je novi servis pomoću komande `nest g s <name-service>`. Name-service preimenovan je u `ordermailer`. Upotrebom tehnike *dependency injection*, kreirana je instanca mail servisa. Da bi se omogućilo slanje e-pošte, potrebno je dodati zavisnosti pomoću komande `npm i @nestjs-modules/mailer`.

```

$ order-mailer.service.ts U X
back-end > reservation-service > src > order > services > TS order-mailer.service.ts > ...
1 import { MailerService } from '@nestjs-modules/mailer';
2 import { Injectable } from '@nestjs/common';
3
4 @Injectable()
5 export class OrderMailerService {
6   constructor(private readonly _mailerService:MailerService){}
7   sendOrderEmail(data){
8     this._mailerService.sendMail({
9       to:data.data.email,
10      bcc:`${process.env.MAIL_NOTIFICATION}`,
11      subject:"Podaci o rezervaciji",
12      encoding:"UTF-8",
13      html:this.makeHtml(data)
14    }).then(()=>console.log("Uspesno poslato"))
15     .catch(err=>console.log("Problem pri slanju"))
16   }
17   private makeHtml(data){
18     let sum = ((new Date(data.data.endDate).getTime()-new Date(data.data.startDate).getTime())/(1000 * 60 * 60 * 24))*data.car.
19     return `<p>Zahvaljujemo se na uspesnoj rezervaciji.</p>
20     <p>Detalji Vase rezervacije:<br/><br/>
21     Lokacija i datum preuzimanja vozila: ${data.data.locationStart}, `+
22     data.data.startDate.split('T')[0]+' '+data.data.startDate.split('T')[1]+
23     `<br/>Lokacija i datum vracanja vozila: ${data.data.locationEnd}, `+
24     data.data.endDate.split('T')[0]+' '+data.data.endDate.split('T')[1]+
25     `<br/>
26     </p><br/>
27     <b>Ukupni troskovi ${sum.toFixed(2)} EUR</b>`
28   }
29 }
30 }

```

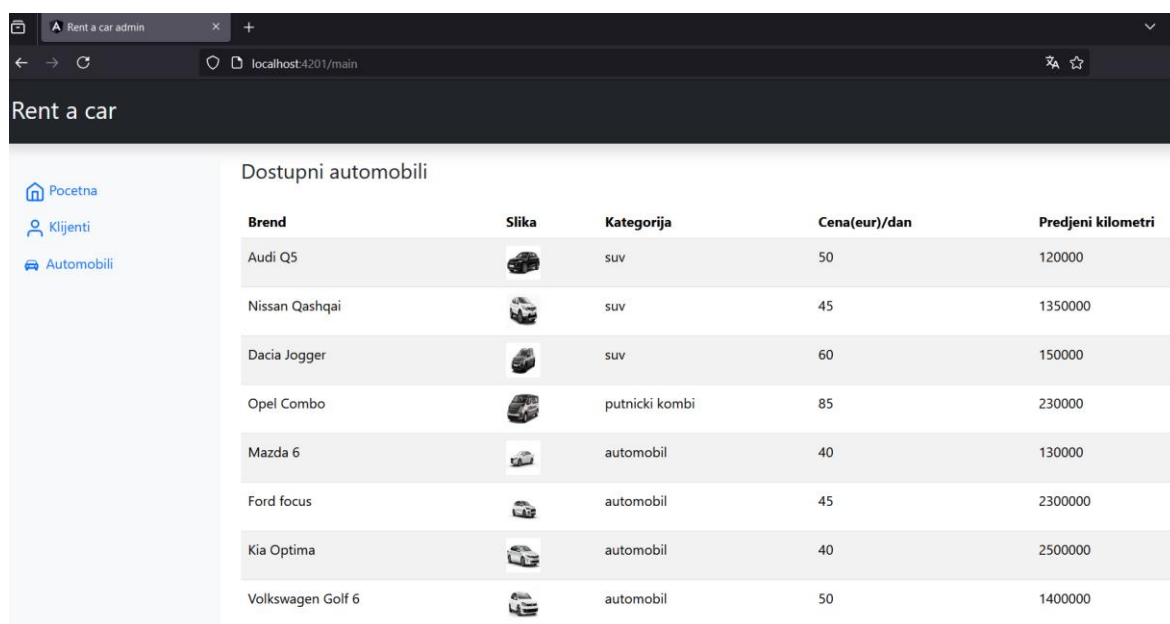
Slika 27. Slanje elektronske pošte

### 5.3.2. Korisnički interfejs









Da bi se stekao utisak o razvijenoj aplikaciji, u nastavku je prikazano nekoliko snimaka ekrana njenog korisničkog interfejsa (klijentski deo aplikacije).

Na slici 28 prikazan je izgled ekrana koji se pojavi nakon što se administrator prijavi na sistem. Kao što se vidi, administratoru su raspoloživi podaci o vozilima (osnovi i dodatni), a ima mogućnost i da izabere opciju *Klijenti* kako bi pogledao podatke o

korisnicima sistema.

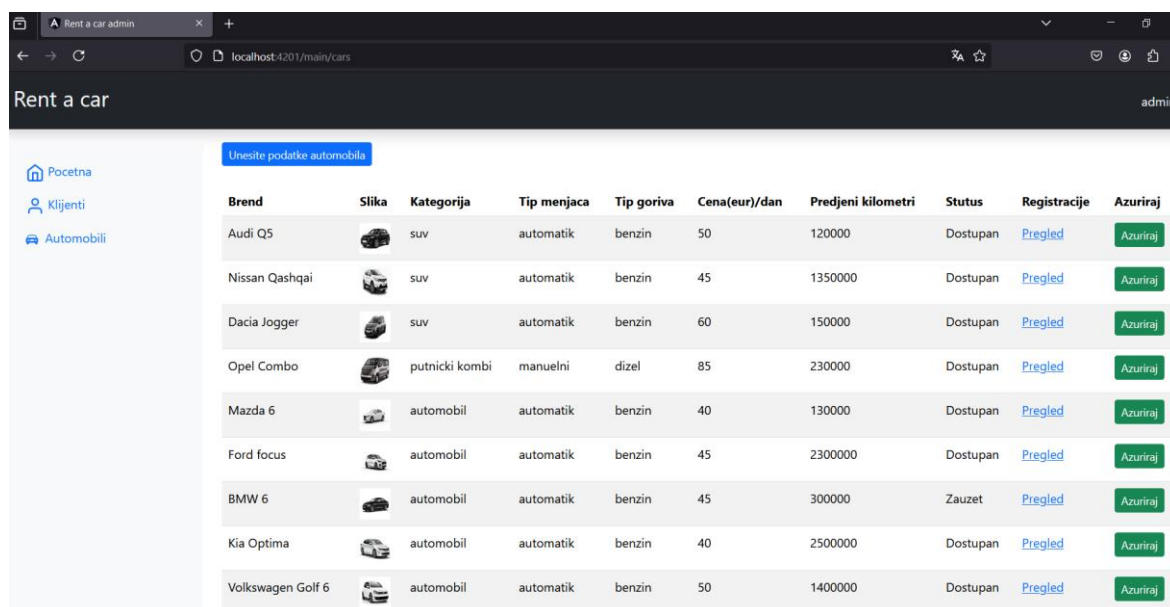


The screenshot shows a web application titled 'Rent a car admin' with a sidebar menu containing 'Pocetna', 'Klijenti', and 'Automobili'. The main content area is titled 'Dostupni automobili' and displays a table of available cars.










Brend	Slika	Kategorija	Cena(eur)/dan	Predjeni kilometri
Audi Q5		suv	50	120000
Nissan Qashqai		suv	45	1350000
Dacia Jogger		suv	60	150000
Opel Combo		putnicki kombi	85	230000
Mazda 6		automobil	40	130000
Ford focus		automobil	45	2300000
Kia Optima		automobil	40	2500000
Volkswagen Golf 6		automobil	50	1400000

Slika 28. Početni ekran za administratora

Na slici 29 je prikaz ekrana koji omogućava administratoru da pregleda i ažurira sve podatke o postojećem voznom parku agencije za iznajmljivanje vozila. Takođe, administrator može i da dodaje nova vozila u vozni park, ili da rashoduje postojeća (dugmad desno).



The screenshot shows the same web application, but the main content area is titled 'Unesite podatke automobila' and displays a more detailed table of cars. The table includes columns for 'Tip menjaca', 'Tip goriva', 'Status', 'Registracije', and 'Azuriraj'.

Brend	Slika	Kategorija	Tip menjaca	Tip goriva	Cena(eur)/dan	Predjeni kilometri	Status	Registracije	Azuriraj
Audi Q5		suv	automatik	benzin	50	120000	Dostupan	<a href="#">Pregled</a>	<a href="#">Azuriraj</a>
Nissan Qashqai		suv	automatik	benzin	45	1350000	Dostupan	<a href="#">Pregled</a>	<a href="#">Azuriraj</a>
Dacia Jogger		suv	automatik	benzin	60	150000	Dostupan	<a href="#">Pregled</a>	<a href="#">Azuriraj</a>
Opel Combo		putnicki kombi	manuelni	dizel	85	230000	Dostupan	<a href="#">Pregled</a>	<a href="#">Azuriraj</a>
Mazda 6		automobil	automatik	benzin	40	130000	Dostupan	<a href="#">Pregled</a>	<a href="#">Azuriraj</a>
Ford focus		automobil	automatik	benzin	45	2300000	Dostupan	<a href="#">Pregled</a>	<a href="#">Azuriraj</a>
BMW 6		automobil	automatik	benzin	45	300000	Zauzet	<a href="#">Pregled</a>	<a href="#">Azuriraj</a>
Kia Optima		automobil	automatik	benzin	40	2500000	Dostupan	<a href="#">Pregled</a>	<a href="#">Azuriraj</a>
Volkswagen Golf 6		automobil	automatik	benzin	50	1400000	Dostupan	<a href="#">Pregled</a>	<a href="#">Azuriraj</a>

Slika 29. Tabela sa detaljnim podacima o vozilima

Evidencija o registraciji vozila se može videti na slici 30. Administrator ima mogućnost da proveri postojeće ili unese nove podatke o registraciji vozila.

Pocetak vazenja registracije	Prestanak vazenja registracije
06.11.2024.	06.11.2025.

Slika 30. Provera važenja registracije određenog vozila

Podaci o klijentima *rent-a-car* agencije su prikazani na slici 31. Administrator može da pretražuje klijente po JMBG-u, kao i da pogleda sve dotadašnje rezervacije određenog klijenta.

Ime	Prezime	JMBG	Email	Telefon	Rezervacije
Milos	Ilic	7665180999467	ilicmilosmiki@gmail.com	0645678990	<a href="#">Pregled</a>

Slika 31. Pregled podataka o klijentima

Slika 32 prikazuje podatke o svim dosadašnjim rezervacijama određenog klijenta i ostvarenim bonusima koji se ostvaruju nakon određenog broja korišćenih usluga (broj rezervacija). Administrator ima uvid u broj rezervacija određenog klijenta i na osnovu tog podatka odobrava ostvarivanje bonusa (na svaku treću rezervaciju klijent ostvaruje popust od 5%). Tu su i podaci o važenju vozačke dozvole.

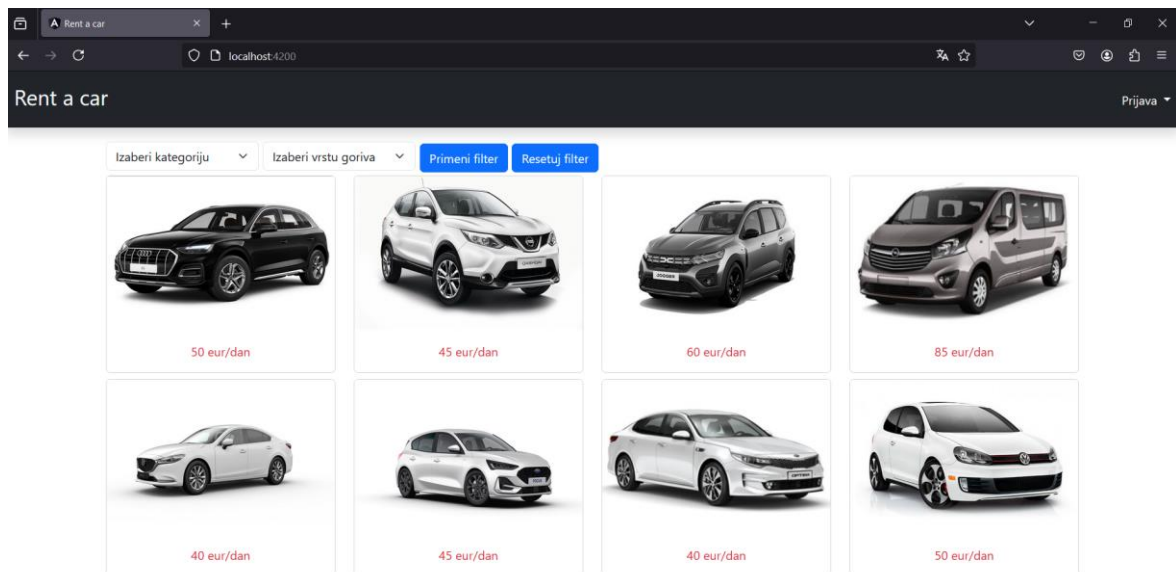
Lokacija preuzimanja	Lokacija vracanja	Automobil	Vreme preuzimanja	Vreme vracanja	Istek vazenja vozačke dozvole
Beograd, Aerodrom	Beograd, Aerodrom	Opel Combo	08.11.2024. 12:00	10.11.2024. 12:00	06.01.2025. 0:00

Slika 32. Prikaz rezervacija određenog klijenta

Početni ekrana aplikacije koji je dostupan klijentima i anonimnim posetiocima, tj.

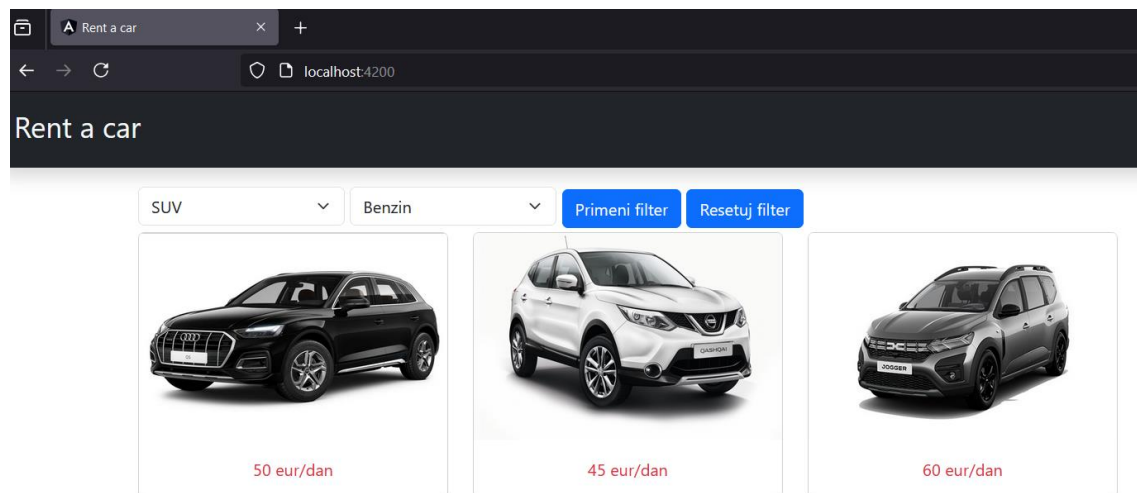


gostima prikazan je na slici 33. Oni mogu da pogledaju koja su vozila trenutno dostupna, kao i da pretraže dostupne podatke o njima.



Slika 33. Početni ekran za klijenta i gosta

Pretraga vozila sa mogućnošću selektovanja određenih kriterijuma pretraživanja (po tipu, kao i vrsti goriva) je data na slici 34. Pristup ovim mogućnostima imaju klijent i gost.



Slika 34. Pretraga vozila


Detaljan opis selektovanog vozila je dat na slici 35. Prijavljeni korisnik (klijent) ima mogućnost rezervacije vozila i unos komentara na osnovu ličnog iskustva o poslovanju agencije i udobnosti tog vozila. Gosti mogu samo da pogledaju konkretne karakteristike vezane za određeno vozilo i pročitaju komentare ranijih korisnika.



Rent a car

localhost:4200/4/detail

ilicmilosmiki@e



## Opis

Model: Opel Combo  
Kategorija: putnicki kombi  
Vrsta menjaca: manualni  
Vrsta goriva: dizel  
Cena: 85 eur/dan  
[Vratite se](#)

Unesite lokaciju preuzimanja	Unesite lokaciju vraćanja
Datum početka iznajmljivanja	Datum završetka iznajmljivanja
dd . mm . yyyy . --:--	dd . mm . yyyy . --:--
Datum isteka Vase vozačke dozvole	Godine starosti vozača
dd . mm . yyyy . --:--	

Recenzije: 0

Unesite komentar

Rezervisi

Slika 35. Detaljni opis vozila

## 6. Zaključak

Iskustvo u izradi aplikacije za rentiranje vozila je pokazalo da je mikroservisna arhitektura pogodna za brz i efikasan razvoj softvera. S obzirom da su aplikacije sa mikroservisnom arhitekturom podeljene na mikroservise, svaki mikroservis je moguće razvijati nezavisno u odabranom programskom jeziku, što je značajna pogodnost. Takođe, mogu se koristiti i različite baze podataka, jer svaki mikroservis ima posebnu bazu.

Mikroservisna arhitektura je pogodna za realizaciju ne samo malih, već i velikih sistema upravo zbog delova sistema koji su zasebni, pa ih je samim tim mnogo lakše realizovati. Kada je reč o isporuci ovako razvijenog softvera, moguće je isporučiti sistem kao celinu ili delimično, u delovima. Mikroservisi se mogu hostovati na različitim domenima, a da bi svi delovi sistema bili dostupni, mora da postoji veza između njih (mrežni prolaz).

Razvoj softvera zasnovan na mikroservisnoj arhitekturi omogućava brže i lakše detektovanje grešaka, kao i njihovo ispravljanje, zato što mikroservisi nisu previše obimni. Ukoliko je potrebno korigovati grešku u nekom mikroservisu, to neće uticati na rad ostalih mikroservisa u sistemu.

U ovom radu je mikroservisna arhitektura iskorišćena za razvoj savremene veb aplikacije za rentiranje vozila koja je dostupna na bilo kom uređaju. Pre same realizacije aplikacije, bilo je potrebno najpre osmisliti kako organizovati mikroservisnu strukturu sistema koja ispunjava zahtevane funkcionalnosti, a zatim i kako ostvariti neometanu komunikaciju između mikroservisa koji treba da funkcionišu kao celina. Jedna od većih nedoumica je bila kako adekvatno rasporediti funkcionalnosti po mikroservisima. Po rešavanju tog problema, dalji tok razvoja softvera je bio prijatan.

Kao kod svakog softvera, i u razvijenoj aplikaciji za rentiranje vozila postoji mogućnost daljeg proširenja i nadogradnje sistema novim funkcionalnostima. Neke od funkcionalnosti koje bi se u budućnosti mogle dodati su: mogućnost da klijent dobije uvid u svoje rezervacije, mogućnost da klijent modifikuje svoje lične podatke po potrebi, mogućnost da klijent izvrši transakciju plaćanja usluga elektronski, usavršavanje sistema kako bi se omogućio uvid u raspoloživost vozila u određenom vremenskom periodu. Ovo su samo neke od funkcionalnosti koje bi doprinele primeni aplikacije u realnom okruženju. Zahvaljujući primenjenoj arhitekturi, navedne modifikacije bi se mogle relativno jednostavno sprovesti uključivanjem i realizacijom novih mikroservisa i dopunom postojećih.

## Literatura

- [1] Grđan, Z. (2021) *Primjena SOA uzoraka u razvoju web aplikacija*, diplomski rad.  
<https://repozitorij.foi.unizg.hr/islandora/object/foi%3A6177/datastream/PDF/view>  
(dostupno: 01/10/2024)
- [2] Bass, L., Clements, P., Kazman, R. (2003) *Software Architecture in Practice (2nd edition)*, Addison-Wesley.  
[https://edisciplinas.usp.br/pluginfile.php/5922722/mod\\_resource/content/1/2013%20-%20Book%20-%20Bass%20%20Kazman-Software%20Architecture%20in%20Practice%20%281%29.pdf](https://edisciplinas.usp.br/pluginfile.php/5922722/mod_resource/content/1/2013%20-%20Book%20-%20Bass%20%20Kazman-Software%20Architecture%20in%20Practice%20%281%29.pdf)  
(dostupno: 01/10/2024)
- [3] Roberts, S. (2023) *What is Client-Server Architecture? Explained in detail*, The Knowledge Academy. <https://www.theknowledgeacademy.com/blog/client-server-architecture/> (dostupno: 06/10/2024)
- [4] Jain, N. (2023) *Peer-to-peer Architecture: A Deep Dive into the Future of Networking*. <https://medium.com/@nikhiljain1203/peer-to-peer-architecture-a-deep-dive-into-the-future-of-networking-d0f07945dca5> (dostupno: 06/10/2024)
- [5] Rubin, D. (2021) *The Three Layered Architecture*.  
<https://medium.com/@deanrubin/the-three-layered-architecture-fe30cb0e4a6>  
(dostupno: 16/10/2024)
- [6] Seetharamugn (2023) *Event-Driven Architecture*.  
<https://medium.com/@seetharamugn/the-complete-guide-to-event-driven-architecture-b25226594227> (dostupno: 17/10/2024)
- [7] Basic Knowledge (2024) *Service-oriented architecture*.  
<https://www.basicknowledge101.com/pdf/km/Service-oriented%20architecture.pdf>  
(dostupno: 03/10/2024)
- [8] Gartner, Inc. (2024) *Service-oriented Architecture (SOA)*.  
<https://www.gartner.com/en/information-technology/glossary/service-oriented-architecture-soa> (dostupno: 05/10/2024)
- [9] Jensen, C.T. (2013) *SOA Design Principles for Dummies*, John Wiley & Sons, Inc., Hoboken, New Jersey. [https://www.professores.uff.br/screspo/wp-content/uploads/sites/127/2017/09/1SOAforDummies\\_Final.pdf](https://www.professores.uff.br/screspo/wp-content/uploads/sites/127/2017/09/1SOAforDummies_Final.pdf)  
(dostupno: 05/10/2024)
- [10] Erl, T. (2017) *Service-Oriented Architecture Analysis and Design for Services and Microservice, 2nd Edition*.  
<https://www.informit.com/articles/article.aspx?p=2755721> (dostupno: 14/10/2024)
- [11] Ademović, S. (2015) *Servisno orijentisana arhitektura*, diplomski rad.  
<https://dokumen.tips/documents/soa-diplomski-rad-ademovic-saudin.html?page=1>  
(dostupno: 01/10/2024)

- [12] Hurwitz, J., Bloor, R., Kaufman, M., Halper, F. (2009) *Service Oriented Architecture (SOA) For Dummies Next Generation SOA: A Concise Introduction to Service Technology & Service-Oriented*, Wiley Publishing, Inc.
- [13] Hettiarachchi, D. (2024) *SOAP - Simple Object Access Protocol*.  
<https://www.linkedin.com/pulse/soap-simple-object-access-protocol-danuka-hettiarachchi-cxxoc> (dostupno: 10/10/2024)
- [14] [https://litux.nl/mirror/beaweblogic8.1/0672324873\\_ch30lev1sec4.html](https://litux.nl/mirror/beaweblogic8.1/0672324873_ch30lev1sec4.html)  
(dostupno: 10/10/2024)
- [15] Anastasia D., Dmytro H. (2019) *Best Architecture for an MVP: Monolith, SOA, Microservices or Serverless*. <https://rubygarage.org/blog/monolith-soa-microservices-serverless> (dostupno: 23/10/2024)
- [16] Živković, M., Bačanin Džakula, N., Tuba, E. (2022) *Programski jezici*, Univerzitet Singidunum, <https://singipedia.singidunum.ac.rs/izdanje/43024-programski-jezici>  
(dostupno: 20/10/2024)
- [17] Zečević, B. (2015) *REST servisi u Javi i njihova primena u aplikaciji telekomunikacionog operatera*, master rad.  
[https://www.racunarstvo.matf.bg.ac.rs/MasterRadovi/2013\\_04\\_15\\_Bojana\\_Zececic/ad.pdf](https://www.racunarstvo.matf.bg.ac.rs/MasterRadovi/2013_04_15_Bojana_Zececic/ad.pdf) (dostupno: 24/10/2024)
- [18] Fielding, R.T. (2000) *Representational State Transfer (REST)*.  
[https://ics.uci.edu/~fielding/pubs/dissertation/rest\\_arch\\_style.htm#sec\\_5\\_2\\_1\\_1](https://ics.uci.edu/~fielding/pubs/dissertation/rest_arch_style.htm#sec_5_2_1_1)  
(dostupno: 24/10/2024)
- [19] Mihajlović, N. *Kako da koristite REST API - vodič za početnike*  
<https://www.mcloud.rs/blog/kako-da-koristite-rest-api-vodic-za-pocetnike/>  
(dostupno: 24/10/2024)
- [20] Kilibarda, D., Vujošević D. (2017) *Sistem za upravljanje kladom korišćenjem REST principa*. <http://joc.raf.edu.rs/images/vol9/sistem-za-upravljanje-kladom-koriscenjem-REST-principa.pdf> (dostupno: 24/10/2024)
- [21] Živanović, D. *Web programiranje*, skripte. <https://www.webprogramiranje.org/web-servisi-osnove/> (dostupno: 24/10/2024)
- [22] Agrawal, H., Fateh, D. (2022) *What is Type Script and why should you use it*.  
<https://www.contentful.com/blog/what-is-typescript-and-why-should-you-use-it/>  
(dostupno: 04/11/2024)
- [23] *MySQL Tutorial*, <https://www.w3schools.com/MySQL/default.asp>  
(dostupno: 04/11/2024)
- [24] *Bootstrap Tutorial*, <https://www.w3schools.com/bootstrap/> (dostupno: 04/11/2024)
- [25] Heller, M. (2022) *What is Visual Studio Code? Microsoft's extensible code editor*.  
<https://www.infoworld.com/article/2335960/what-is-visual-studio-code-microsofts-extensible-code-editor.html> (dostupno: 03/11/2024)
- [26] Beogradska akademija poslovnih i umetničkih strukovnih studija, *Uvod u Angular*.  
<https://www.bpa.edu.rs/FileDownload?filename=6940f387-985d-49c8-be1b-577584657454.pdf&originalName=IT06.pdf> (dostupno: 03/11/2024)

- [27] Lanz, R. *Introduction to Node.js*. <https://nodejs.org/en/learn/getting-started/introduction-to-nodejs> (dostupno: 02/11/2024)
- [28] Dua, A. (2024) *What is Nest.js*. <https://www.turing.com/blog/what-is-nest-js-why-use-it> (dostupno: 02/11/2024)
- [29] *HeidiSQL - OpenSource Database Management Tool*, Martining & Associates. <https://www.methodsandtools.com/tools/heidisql.php> (dostupno: 03/11/2024)