Razvoj web aplikacije temeljene na mikrouslugama za organiziranje poslovnih aktivnosti tvrtke

Stipanović, Zvonimir

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

Sveučilišni preddiplomski studij računarstva

RAZVOJ WEB APLIKACIJE TEMELJENE NA MIKROUSLUGAMA ZA ORGANIZIRANJE POSLOVNIH AKTIVNOSTI TVRTKE

Završni rad

Zvonimir Stipanović

Osijek, 2018.

Sadržaj

1. UVOD 1

2. ARHITEKTURE APLIKACIJA 2

2.1. Monolitne aplikacije 2

2.2. Uslugama orijentirana arhitektura 5

2.2.1. Modularne aplikacije 6

3. MIKROUSLUGE 8

3.1. Karakteristike mikrousluga 9

3.2. Prednosti i nedostaci mikrousluga 10

3.3. Usporedba mikrousluga s monolitnim aplikacijama 11

3.4. Komunikacija između mikrousluga 12

4. PROGRAMSKO RJEŠENJE WEB APLIKACIJE 13

4.1. Prikaz programskih komponenti 15

4.2. Primjer koda za mikrousluge 18

5. TESTIRANJE APLIKACIJE I KORIŠTENJE 24

5.1. Korištenje aplikacije 24

5.2. Testiranje aplikacije 26

6. ZAKLJUČAK 30

LITERATURA 31

SAŽETAK 32

ABSTRACT 33

ŽIVOTOPIS 34

PRILOZI 35

1. UVOD

Tema ovog završnog rada je izrada web aplikacije za poslovne aktivnosti tvrtke temeljenih na mikrouslugama. Aplikacija je u prvom redu namijenjena tvrtkama radi lakšeg organiziranje poslovnih aktivnosti. Ideja je da korisnik (djelatnik) može vidjeti listu zaposlenika, dostupnosti pojedinih prostorija i dostupnosti vozila, te da u svakom trenutku može dodati neku od atributa te ih izbrisati sa liste.

Cilj ovog rada je na jednostavan način prikazati rad i funkcionalnosti mikrousluga. Kako je aplikacija namijenjena da joj se može pristupiti u bilo kojem trenutku i s bilo kojeg uređaja, najbolje rješenje je bilo izrada web aplikacije rađena u .NET programskom okviru, a za samu realizaciju korisničkog sučelja korišten je programski jezik Javascript. Za kreiranje aplikacije korišten je Visual Studio Code.

U drugom poglavlju ovog završnog rada objašnjeno je koje arhitekture aplikacija postoje, kratko su opisane te arhitekture, te su opisani prednosti i nedostaci tih arhitektura. U trećem poglavlju je detaljno objašnjen rad mikrousluga, kako one funkcioniraju i koji su im prednosti, a koji nedostaci. U četvrtom poglavlju napravljena je realizacija web aplikacije mikrousluga u Visual Studio Codeu, objašnjeno je kako se došlo do programskog rješenja, te su opisani pojedini dijelovi koda. U petom poglavlju objašnjeno je korištenje web aplikacije, a sukladno sa korištenjem provedeno je i testiranje funkcionalnosti aplikacije.

Zadatak završnog rada

U radu je potrebno analizirati i opisati arhitekturu mikrousluga u odnosu na monolitne i uslugama usmjerene arhitekture, te razraditi načine podjele web aplikacije na funkcionalne mikrousluge i njihovo međudjelovanje. Programski je potrebno u okolini .NET ostvariti web aplikaciju za organiziranje poslovnih aktivnosti tvrtke uzimajući u obzir vrijeme, planirane aktivnosti i nadležnost zaposlenika. Te pokazatelje treba uzeti u obzir kod slaganja mikrousluga i povezati ih s podatkovnim modelom, a koristiti odgovarajući okvir usluga i uzorak programske arhitekture. Web aplikaciju potrebno je prikladno testirati i analizirati.

2. ARHITEKTURE APLIKACIJA

S ciljem što boljeg shvaćanja mikrousluga, bit će opisan rad i karakteristike monolitnih aplikacija, te usluge orijentiranje arhitekture (*engl. Service Oriented Architecture, SOA*).

2.1. Monolitne aplikacije

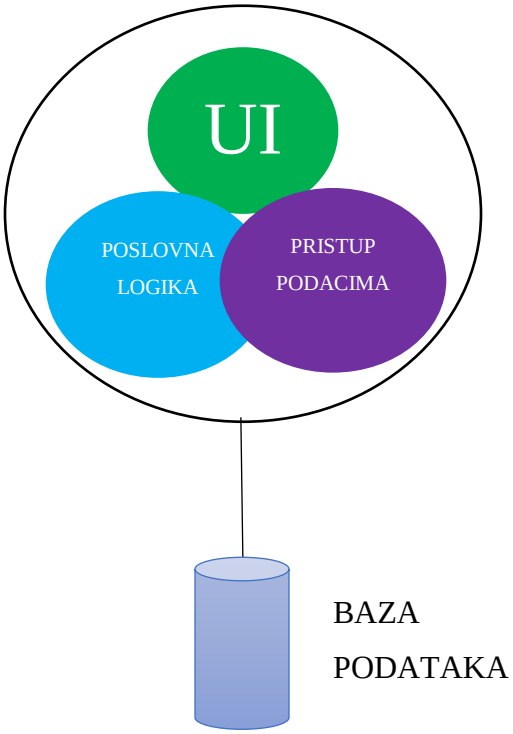
Monolitna arhitektura je tradicionalni model za dizajniranje aplikacija. Monolitno, u svom kontekstu, znači ukomponirano sve u jednom što znači da se sve funkcionalnosti nalaze na jednom mjestu. Monolitne aplikacije su dizajnirane da budu samostalne, te komponente programa su međusobno povezane i međusobno ovisne. Ukoliko jedan dio monolitne aplikacije ne radi, cijela aplikacija neće raditi. U takvoj čvrstoj povezanoj arhitekturi, sve komponente moraju biti prisutne kako bi se kod izvršio. Prilikom mijenjanja komponenti programa u ovakvim aplikacijama, cijela aplikacija mora biti promijenjena. To dodatno otežava rad u većim programskim timovima, jer prilikom izmjene koda drugi programer ima otežan pristup pri izmjeni tog koda [1].

Nekoliko je prednosti programiranja monolitnih aplikacija. Ovakve aplikacije su jednostavan način za izradu web aplikacija. Uz to, postoje puno tečajeva koji pomažu početnicima pri učenju programiranja ovakvih web aplikacija [2].

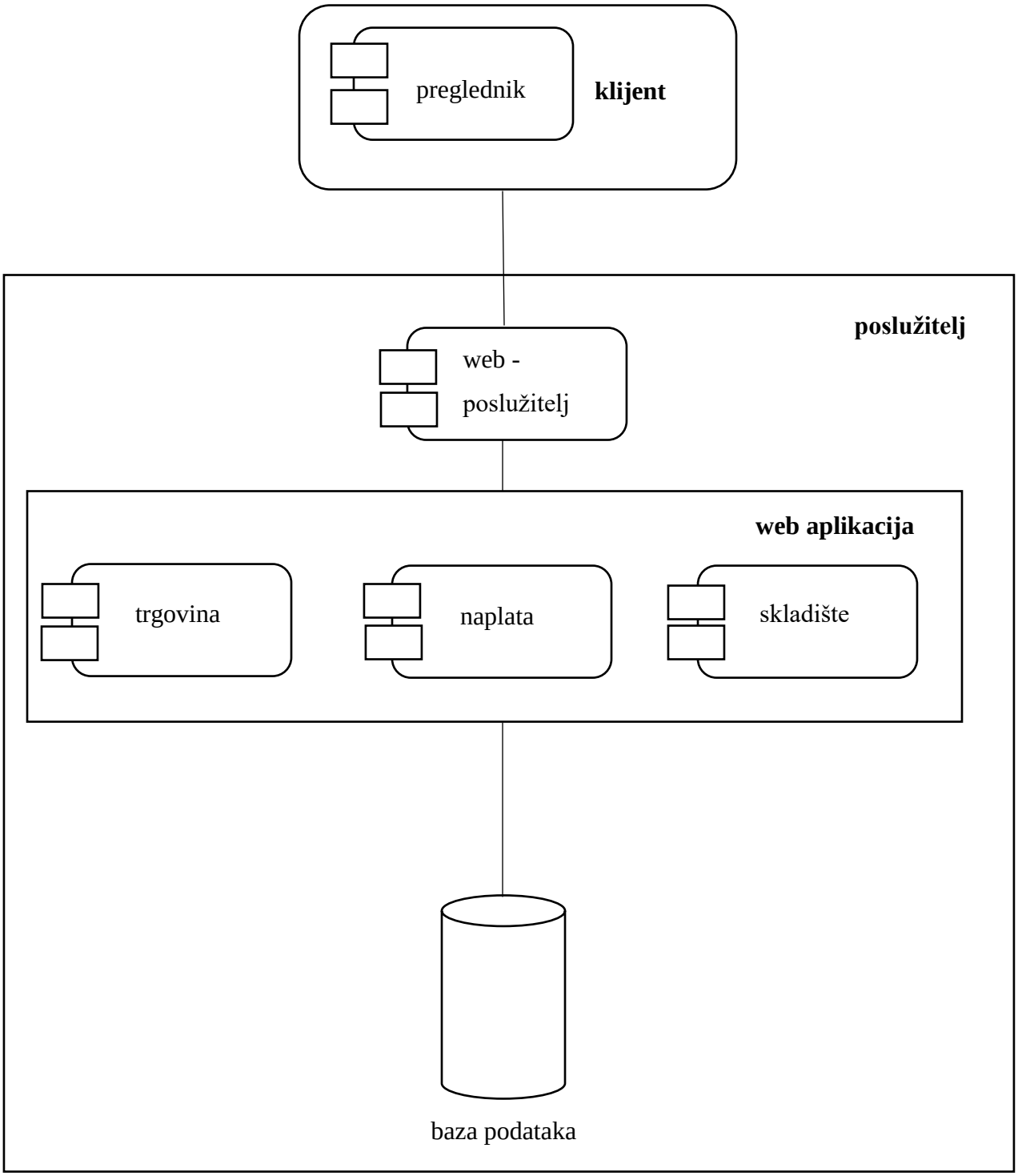
Prema [3], neke od prednosti monolitnih aplikacija u odnosnu na mikrousluge su:

1. Otklanjanje pogrešaka i testiranje – općenito govoreći, monolitne aplikacije su lakše za otklanjanje pogrešaka i za testiranje. Jednom kada se uvede više stotina novih varijabli, dolazi do mogućnosti da se u nekim stvarima pogriješi.
2. Performanse – puno je brži odziv ukoliko se neka stranica učitava sa jedne usluge, nego da poziva više API (*engl. Application programming interface*) koje poziva više mikrousluga.
3. Sigurnost i operacije – ako se rastavi aplikacija u mnogo male usluge, morat će se upravljati uslugama za usluge koje osiguravaju zaštitu. Međutim upravljanje sa uslugama „mnogima kao jedne“ doprinosi puno boljoj zaštiti, te puno manje testiranja sigurnosti.
4. Planiranje i dizajn – mikrousluge imaju puno veći trošak dizajna i mogu uključivati puno kompliciranije rasprave među timovima oko organiziranja posla.

S druge strane, dosta je nedostataka pri izradi monolitnih web aplikacija. Jednom kada aplikacija postane prevelika i poveća se tim programera sve je teže održavanje takvih aplikacija. U tom slučaju puno su efikasnije modularne aplikacije korištene na modelu mikrousluga. Na početku programiranja monolitnih aplikacija bira se tehnologija koja će se koristiti za izradu, te se tokom cijelog programiranja aplikacije stvara dugotrajna iteracija sa tom tehnologijom [2]. Nadalje, ako izabrani okvir (*engl. framework*) zastari, tu nastaje problem jer je postupak prebacivanja na noviji i bolji okvir je relativno kompliciran. Na Slici 2.1 opisan je model monolitne aplikacije. Na toj slici možemo vidjeti kako su korisničko sučelje (*engl. User interface*), poslovna logika (*engl. Business Logic*) i sloj pristupa podacima (*engl. Data Access* Layer) objedinjeni u jednu uslugu koja je povezana na jednu, istu bazu podataka. Na slici 2.2 možemo vidjeti jedan primjer monolitne arhitekture gdje se klijent preko poslužitelja spaja na web aplikaciju, te nadalje web poslužitelj prespaja na modele web aplikacije koje su spojene u jednu bazu podataka. U poglavlju 3 oblikovana je ova aplikacija u aplikaciju temeljenu na mikrouslugama.



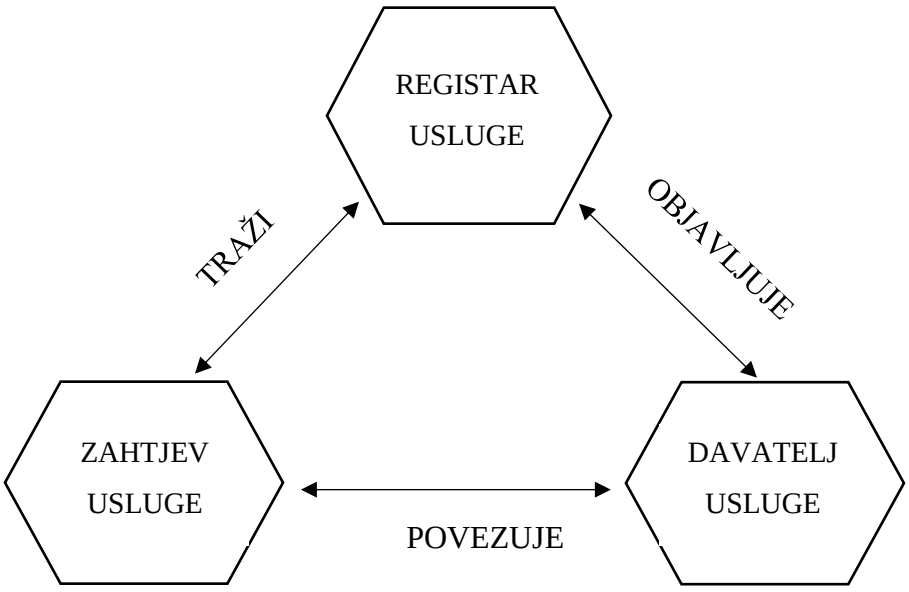
Slika 2.1 Model monolitne arhitekture



Slika 2.2 Primjer monolitne aplikacije

2.2. Uslugama orijentirana arhitektura

Uslugama orijentirana arhitektura (*engl. Service Oriented Architecture, SOA)* je arhitektura koja je namijenjena za razmjenu informacija između dva ili više računala, ili za razmjenu između računala i korisnika. Osnovna ideja je da određeni skup ili usluga pruža različite funkcionalnosti (API), a da druge usluge koriste te iste funkcionalnosti. Na slici 2.3 koja je dobivene iz [7] opisan je model uslugama orijentirane arhitekture. Davatelj usluga je odgovoran za kreiranje grafičkog opisa usluge, objavljivanje opisa usluge jednoj ili više registara usluga i za povezivanje web usluga od jednog ili više zahtjeva usluga. Zahtjev usluge je odgovoran za pronalaženje opisa usluge koji je objavljen u jednom ili više registara usluge, te je odgovoran za korištenje opisa usluga kako bi se vezao ili povezivao na web usluge koje pružaju davatelji usluga. Registar usluge je odgovoran za oglašavanje opisa web usluga koje su mu poslali davatelji usluga i za dopuštanje tražiteljima usluga da pretraže prikupljanje opisa usluga sadržanih u registru usluga. Također, uslugama orijentirana arhitektura sadrži tri operacije: traženje, objavljivanje i povezivanje. Operacija objavljivanja je prijavljivanje usluge ili oglašavanje usluge. Djeluje kao ugovor između registra usluge i davatelja usluge. Operacija traženja logička je dvojna operacija operacije objavljivanja. Operacija traženja je ugovor između podnositelja usluge i registra usluge. Operacija povezivanja je odnos između zahtjeva usluge i davatelja usluge.



Slika 2.3 Model uslugama orijentirane arhitekture

Uslugama orijentirana arhitektura je diskretna jedinica funkcionalnosti kojoj se može pristupiti na daljinu, te je zapravo osmišljena radi poboljšanja većih monolitnih aplikacija i pristup arhitekturi je modularan. Pojam modularnosti objašnjen je u sljedećem poglavlju. Osnovni pristup koji se koristi kod ovakve arhitekture je pristup korištenju zasebnih funkcionalnih usluga. Svaka usluga je zasebna i neovisne su jedna o drugoj. Svakoj usluzi moguće je pristupiti preko bilo koje druge usluge, te je to omogućilo bržu izradu web aplikacija jer programeri mogu raditi i vršiti operacije na pojedinoj usluzi, neovisno o drugim uslugama.

Prema [8] opisane su prednosti korištenja uslugama orijentirane arhitekture:

1. Bolji povrat ulaganja – podjelom usluga na više manjih usluga smanjuje se opterećivanje jedne usluge, te je samim time njezin životni vijek duži
2. Mobilnost koda – pretraživanje i dinamičko vezanje na uslugu klijentu nije važno gdje se usluga nalazi, stoga aplikacija je fleksibilna premještanja usluga na drugačije lokacije
3. Usmjerene uloge programera – aplikacija ima više slojeva gdje svaki sloj ima specifičnu ulogu za programera. Programeri koji su specijalizirani za određeno područje mogu se usredotočiti na programiranje posebno sloja
4. Više sigurnosti – kod sustava izgrađenih na monolitnoj arhitekturi, sigurnost se obrađivala na klijentskoj strani. Kod uslugama orijentirane arhitekture postoji mrežni sloj koje može koristiti više aplikacija, te svaka od tih aplikacija ima vlastite sigurnosne mehanizme.
5. Bolje testiranje – testovi se mogu pokrenuti na zasebnim uslugama neovisno o bilo kojoj aplikaciji koja koristi uslugu.
6. Podrška za više vrsta klijenata – budući da su slojevi podijeljeni na klijentske i uslužne slojeve, puno je lakše implementirati različite vrste klijenata.

2.2.1. Modularne aplikacije

Modularne aplikacije koriste slojevitu arhitekturu koja se bazira na raspodijeli uloga i odgovornosti pojedinog sloja. Modularni stil programiranja omogućava nam da vidimo sustav u manjim fizičkim modulima. Moduli imaju jasan poslovni kontekst i ograničeni su na razini fizičkog sloja.

Modularno programiranje je tehnika dizajniranja softvera koja razdvaja funkcionalnosti programa u nezavisne, izmjenjive module. Svaki od tih modula sadrži sve što je potrebno za izvršavanje jednog aspekta željene funkcionalnosti. Prilikom izrade softvera, jedan od najčešćih koraka je da se vodi računa da je aplikacija proširiva i modularna. Jedan primjer modularnih aplikacija su mikrousluge.

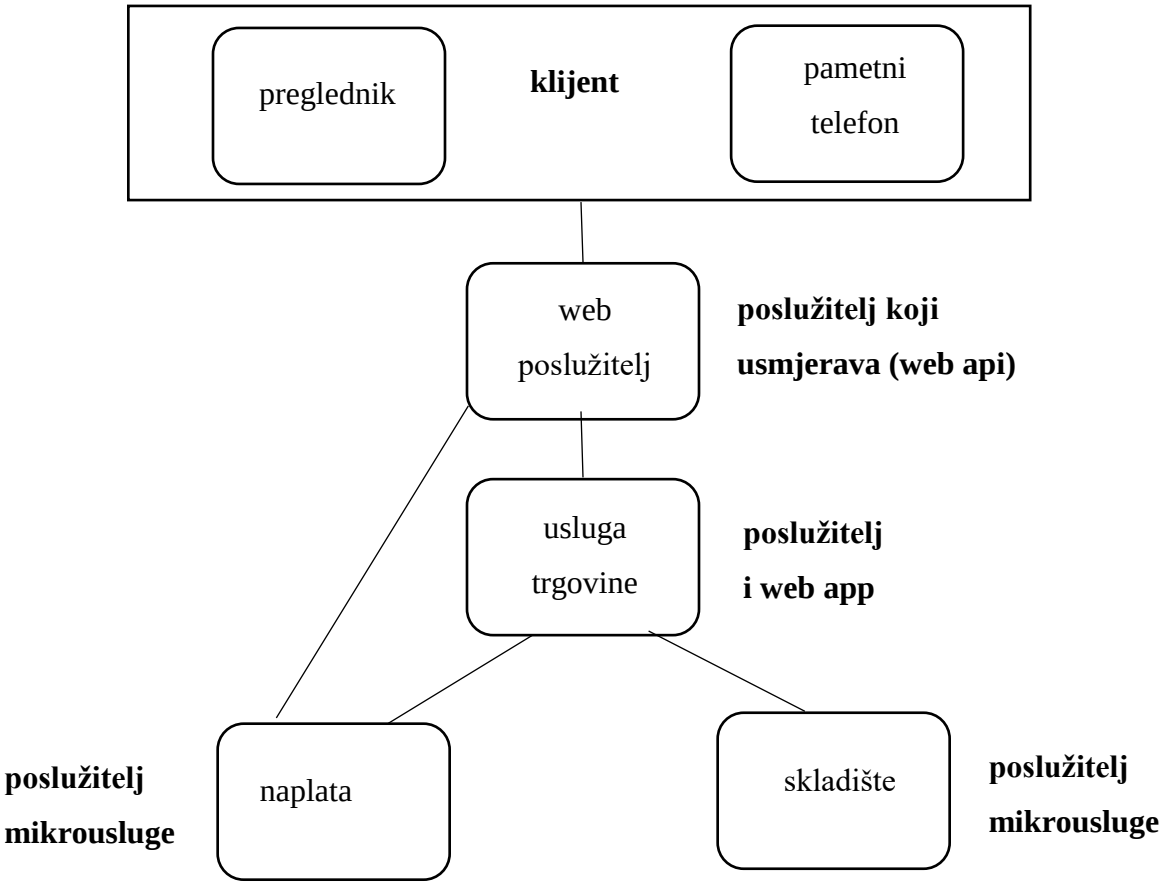
Neke od prednosti programiranja modularnih aplikacija su:

1. Manje koda se mora pisati
2. Za ponovnu upotrebu može se razviti jedna procedura, bez potrebe za ponovnim pisanjem koda više puta
3. Programi se mogu lakše dizajnirati jer se mali tim bavi samo manjim dijelom cijelog koda
4. Modularno programiranje omogućuje mnogim programerima da surađuju na istoj aplikaciji
5. Kod se pohranjuje na više datoteka
6. Kod je mal, jednostavan i lagan za razumijevanje
7. Pogreške se mogu lako identificirati, jer su smještene u nekom potprogramu ili u funkciji
8. Isti kod se može koristiti u više različitih aplikacija
9. Opseg varijabli se može lako kontrolirati

3. MIKROUSLUGE

Mikrousluge su primjer modularnih aplikacija koje strukturiraju aplikaciju u posebno odvojenu kolekciju labavo povezanih usluga koje su zajedno kao jedan zajednički sustav. Mikrousluge se uglavnom koriste u većim, složenijim aplikacijama na kojima rade veći tim programera kako bi oni mogli neometano i neovisno o drugima raditi svoj dio posla. Arhitektura mikrousluga nije savršena, ima nekih nedostataka, ali ima i dosta prednosti [4].

Kod kreiranja mikrousluga moramo imati korisničko sučelje koje će se spajati na poslužitelj koji će prespajati zadane upite na pojedinu uslugu. Kod mikrousluga svaka usluga je jedna zasebna aplikacija koja je neovisna jedna o drugoj. Potrebno je napraviti aplikaciju za korisničko sučelje, aplikaciju web poslužitelja, te više aplikacija za svaku uslugu sa svojim funkcionalnostima koja će biti povezana na svoju ili već neku postojeću bazu podataka. Teoretski je moguće napraviti neograničeni broj manjih usluga. Na slici 3.1 je prepravljena modularna aplikacija sa slike 2.2 u mikrouslugu.



Slika 3.1 Primjer web aplikacije temeljene na mikrouslugama

3.1. Karakteristike mikrousluga

Prema [5], šest glavnih karakteristika mikrousluga su:

1. Višestruke komponente – program (*engl. software*) izgrađen kao mikrousluga može biti razlomljen u više komponenti usluga. Svaka od tih usluga se može implementirati i onda ponovno premjestiti bez ugrožavanja rada aplikacije. Za potpunu funkcionalnost možda će se morati promijeniti jedna ili više različitih usluga umjesto da se mora preusmjeriti cijela aplikacija.
2. Pojednostavljena poslovna logika – za razliku od monolitnih aplikacija, gdje je svaki tim imao specifičan fokus na jedan dio aplikacije (kao primjerice korisničko sučelje ili baze podataka), u mikrouslugama jedan tim može napraviti cijelu uslugu. Odgovornost tog tima je stvaranje određenih proizvoda na temeljnu jedne ili više pojedinačnih usluga koje komuniciraju putem sabirnice poruka.
3. Jednostavno usmjeravanje – mikrousluge primaju zahtjeve, obrađuju ih i generiraju odgovor u skladu s tim. Moglo bi se reći da mikrousluge imaju pametne krajnje točke koje obrađuju informacije i primjenjuju logiku, te prenose informacije kroz cijevi.
4. Decentralizacija – mikrousluge koriste različite tehnologije i platforme, te samim time stara metoda centralizacije nije povoljna. Monolitne aplikacije koriste jednu logičku bazu podataka u različitim aplikacijama, dok kod mikrousluga svaka usluga upravlja jedinstvenom bazom podataka.
5. Otpornost na neuspjeh – budući da više jedinstvenih i raznovrsnih usluga međusobno komuniciraju, moguće je da jedna od usluga ne uspije. U takvim slučajevima, klijent bi trebao omogućiti funkcioniranje susjednih usluga, te se na način praćenja mikrousluga može spriječiti rizik od neuspjeha, ali je taj način puno kompliciraniji u usporedbi s monolitnim arhitekturama.
6. Raznovrsnost – arhitektura je pogodna za evolucijske sustave u kojima u potpunosti ne možete predvidjeti vrste uređaja koje bi jednog dana mogle pristupiti aplikaciji. Mnoge aplikacije počnu na temelju monolitne arhitekture, ali se zbog nepredviđenih razloga mogu obnoviti kao mikrousluge koje preko API-ja komuniciraju preko starije monolitne arhitekture.

3.2. Prednosti i nedostaci mikrousluga

Potreba za mikrouslugama ovisi o zahtjevima aplikacije, te ona nije savršena, postoje neke prednosti i nedostaci korištenja mikrousluga.

Prema [5], navedene su neke prednosti korištenja mikrousluga:

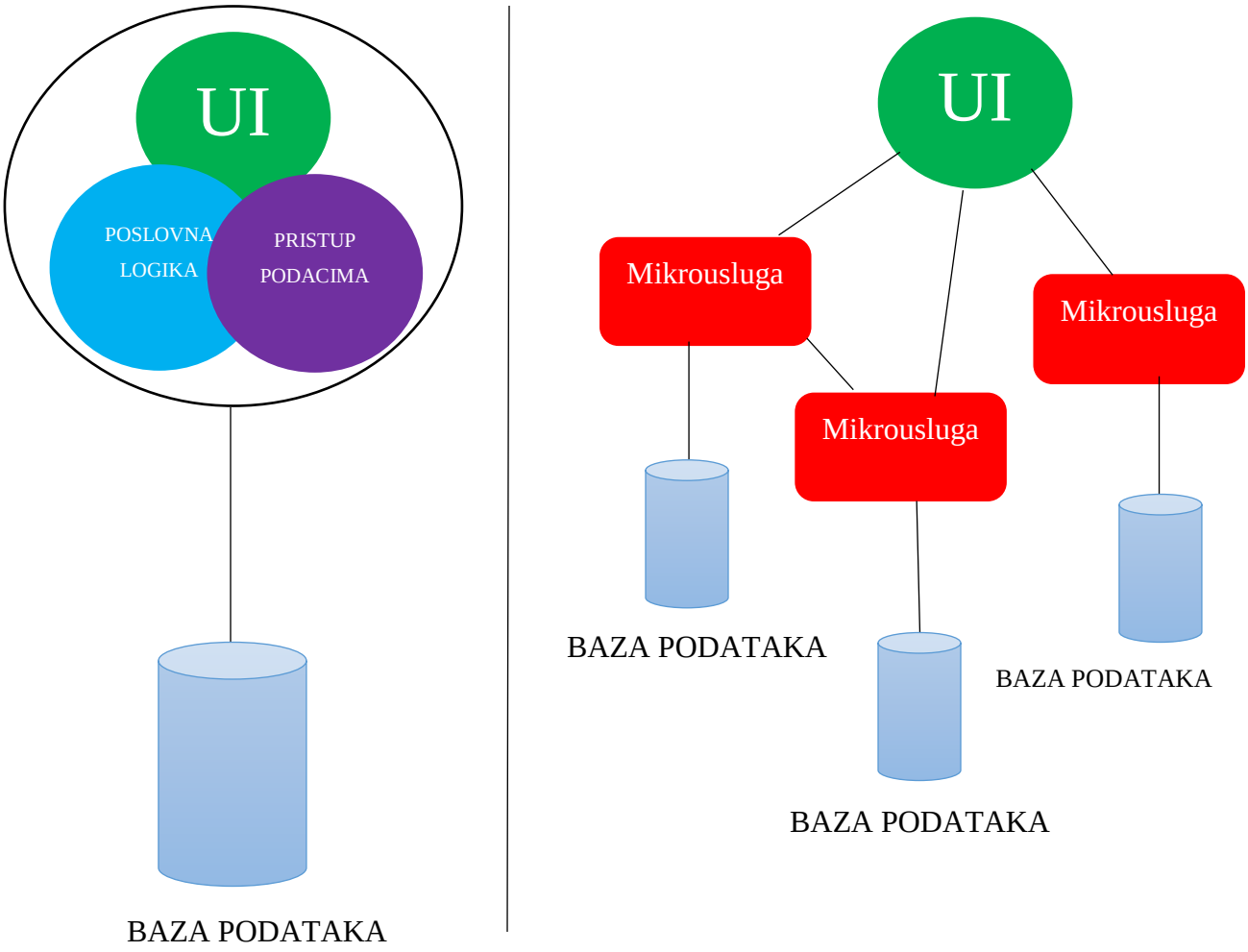
1. Mikrousluge daju razvojnom programeru slobodu da samostalno razvijaju i implementiraju uslugu, te ona može biti razvijena od strane relativno malog tima.
2. Kod različitih usluga može se pisati u različitim programskim jezicima što znači da aplikacija nije striktno vezana za jedan programski jezik.
3. Vrlo jednostavna integracija i automatsko implementiranje, što vrlo brzo može pomoći novom članu tima da brzo postane produktivan.
4. Programeri mogu koristiti najnovije tehnologije, te se kod organizira oko poslovnih mogućnosti.
5. Kada se treba promijeniti dio aplikacije, samo se povezana usluga može mijenjati i preusmjeriti, te nema potrebe za izmjenom i preusmjeravanjem cijele aplikacije.
6. Brzo rješavanje grešaka - ukoliko jedna usluga ne radi, druga usluga će nastaviti raditi neovisno o toj prethodnoj usluzi.

Također, prema [5], navedeni su i nedostaci korištenja mikrousluga:

1. Zbog implementacije, testiranje može postati kompliciranо i dosadno.
2. Gomilanje broja usluga može dovesti do informacijskih prepreka, te samim time informacija sporije protječe. Nadalje, komunikacija između više timova mora biti brza i efikasna.
3. Arhitektura donosi dodatnu složenost, budući da programeri moraju ublažiti toleranciju grešaka, latenciju mreže i baviti se raznim formatima poruka.
4. Mora se uložiti dodatan trud u samu izgradnju mehanizma između usluga.
5. Ovakva arhitektura obično rezultira povećanom potrošnjom memorije.

3.3. Usporedba mikrousluga s monolitnim aplikacijama

Slika 3.2 prikazuje model monolitne aplikacije (lijevo) i model aplikacije temeljene na mikrouslugama (desno). Na lijevoj slici vidimo da je cijela aplikacija objedinjena u jednu uslugu, dok na desnoj slici vidimo kako je aplikacija podijeljena u više manjih usluga. Neke od tih usluga su logički povezane sa drugom uslugom, a ukoliko je potrebno, mikrousluga ima svoju bazu podataka te su sve mikrousluge spojene na jednu jedinstveno korisničko sučelje.



Slika 3.2 Usporedba monolitne aplikacije i mikrousluga

3.4. Komunikacija između mikrousluga

Jedan od glavnih aspekata pri programiranju mikrousluga je taj da treba voditi računa o međusobnoj komunikaciji između usluga. Kod monolitnih aplikacija, izvršavanje jednog procesa koji se poziva ostvaruje se pozivima na razini jezika. Kod MVC (*engl. Model-View-Controller*) dizajna programiranja, kreira se klasa modela koja povezuje relacijske baze podataka na objektni model. Nakon toga, mogu se praviti metode koje pomažu pri izvođenju standardnih operacija nad tablicama baze podataka kao što su – stvaranje (*engl. Create*), čitanje (*engl. Read*), ažuriranje (*engl. Update*) i brisanje (*engl. Delete*). Kod prebacivanja iz monolita na aplikaciju temeljenu na mikrouslugama, najveći je izazov promjena komunikacijskog mehanizma. U mikrouslugama je bitno podijeliti objektne modele u samostalno razvijene i razmještene usluge, koje također formiraju mrežu sa komunikacijskim vezama. Ta podjela nije tako očita i ne moraju se sve komponente podijeliti u zasebne mikrousluge [9].

Komunikacijski stilovi se mogu podijeliti u dvije skupine protokola:

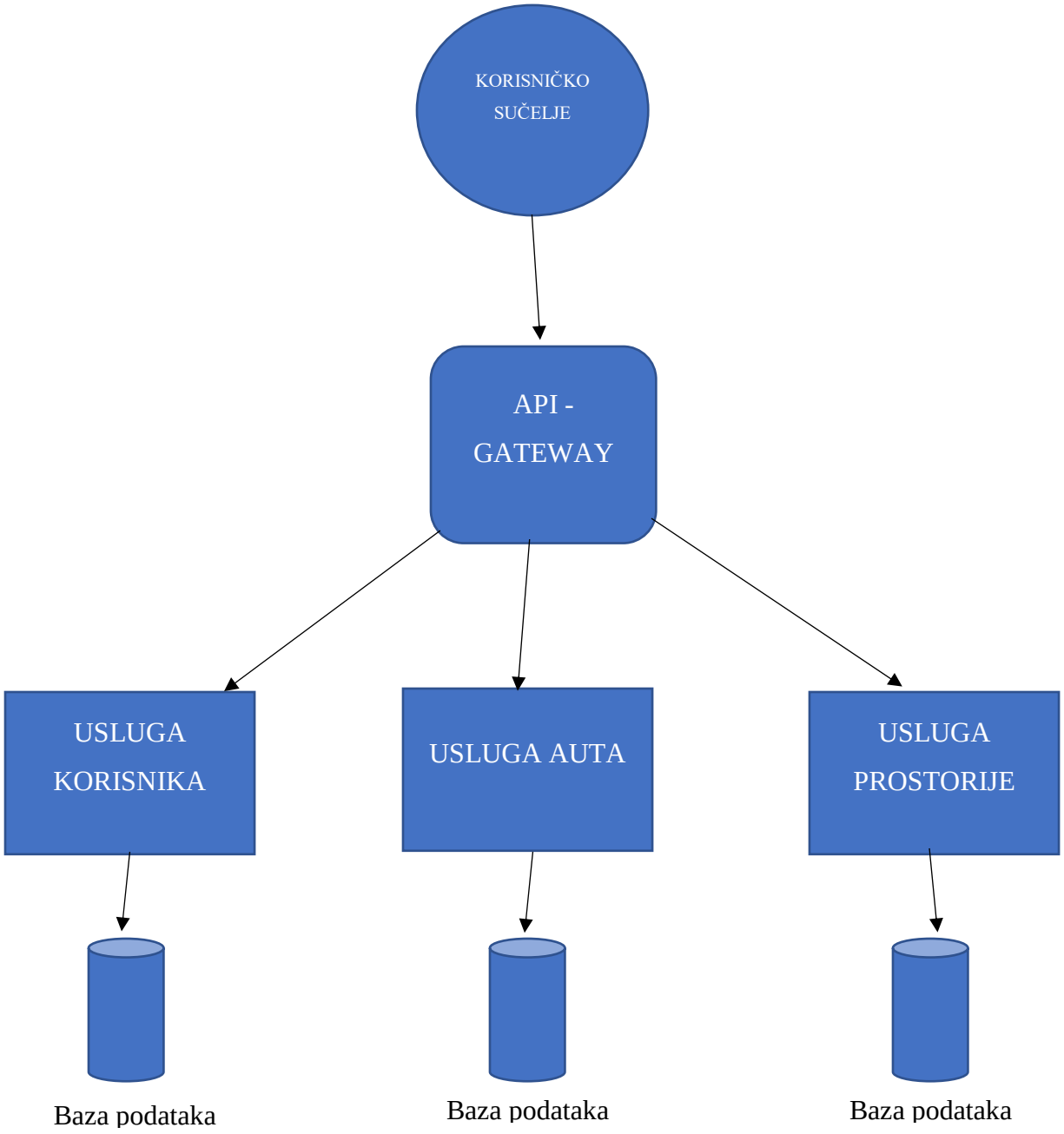
1. Sinkroni – za komunikaciju između web aplikacija, HTTP protokol predstavlja standard za komunikaciju. U ovoj komunikaciji klijent šalje zahtjev i čeka odgovor od usluge
2. Asinkroni – ovakva komunikacija se ostvaruje putem posrednika za razmjenu poruka, te uobičajeno ne čeka na odgovor i čeka potvrdu da je poruka primljena.

Najčešća metoda za pravljenje mikrousluga su REST (*engl. Representational State Transfer*), ali to nije jedina metoda. U kreiranju mojeg rješenja web aplikacije, napravljen je web poslužitelj koji na asinkroni način komunicira sa mikrouslugama.

4. PROGRAMSKO RJEŠENJE WEB APLIKACIJE

Web aplikacija za poslovne aktivnosti tvrtke temeljene na mikrouslugama je realizirana kao programsko rješenje u ASP.NET programskom jeziku. Sastavljena je od više programskih komponenti i funkcionalnih dijelova aplikacije. Aplikacija je sastavljena od korisničkog sučelja pisanog u JavaScript programskom jeziku koji se spaja na api-gateway koji nadalje šalje zahtjeve na jedan od tri napravljena manje usluge (korisnici, auti i prostorije). Svaka od usluga ima svoju posebnu bazu podataka (iako nije nužno da ima svoju bazu podataka) i pokreće se na posebnom poslužitelju na kojoj api-gateway prosljeđuje zahtjev. Usluge su pisane u ASP.NET programskom jeziku, iako su se međusobno mogle pisati i u različitim programskim jezicima.

Na slici 4.1 prikazan je model realizacije ove aplikacije. Prvo je bilo nužno napraviti funkcionalne mikrousluge korisnika, auta i prostorija. Ovakve usluge sadrže modele u kojima su opisana svojstva samih usluga, te se sastoje od kontrolera u kojima su REST metode koje omogućuju usmjeravanje na pojedinu uslugu. Krajnja točka aplikacije je povezivanje usluge sa bazom podataka napisane u SQLiteu. Povezivanje je omogućeno preko Entity Frameworka, te baze podataka nisu međusobno povezane i svaka baza podataka sadrži jednu tablicu. Nadalje, potrebno je bilo realizirati poslužitelj – api-gateway koji će prikupljati zahtjeve sa korisničkog sučelja te ih preusmjeravati na određenu uslugu. Na kraju je bilo potrebno realizirati korisničko sučelje koje je pisano u JavaScript programskom jeziku.

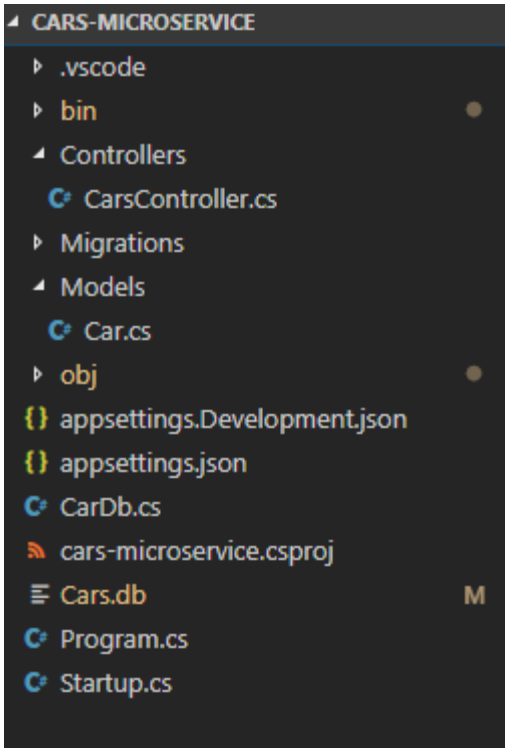


Slika 4.1 Model realizacije aplikacije

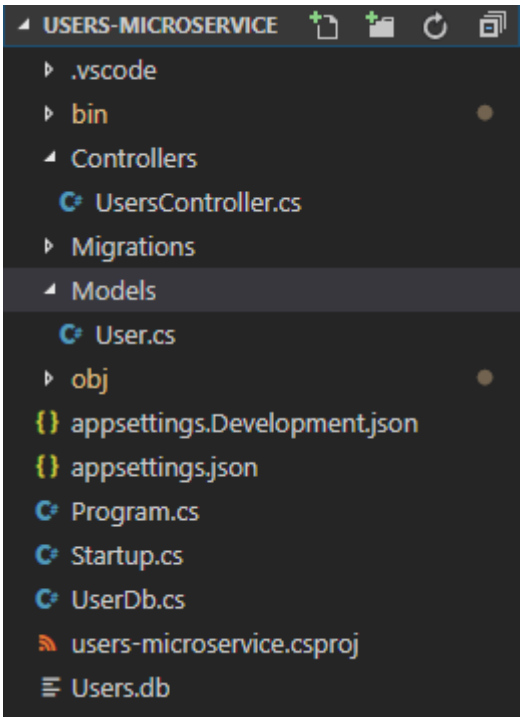
4.1. Prikaz programskih komponenti

Sama aplikacija sastoji se od pet manjih projekata. Projekt za korisničko sučelje koji je pisan u JavaScript programskom jeziku, api-gateway pisan u ASP.NET programskom jeziku koji preusmjerava zahtjeve korisničkog sučelja na određenu uslugu, te tri manje usluge koje su također pisane u ASP.NET programskom jeziku od kojih svaki ima svoju bazu podataka pisanu u SQLite. Povezivanje usluga sa bazom podataka omogućeno je preko *Entity Frameworka.* Svaka od usluga ima svoj model u kojemu su opisana sva svojstva pojedine usluge, te od kontrolera koji obrađuje određene zahtjeve na pojedinu uslugu.

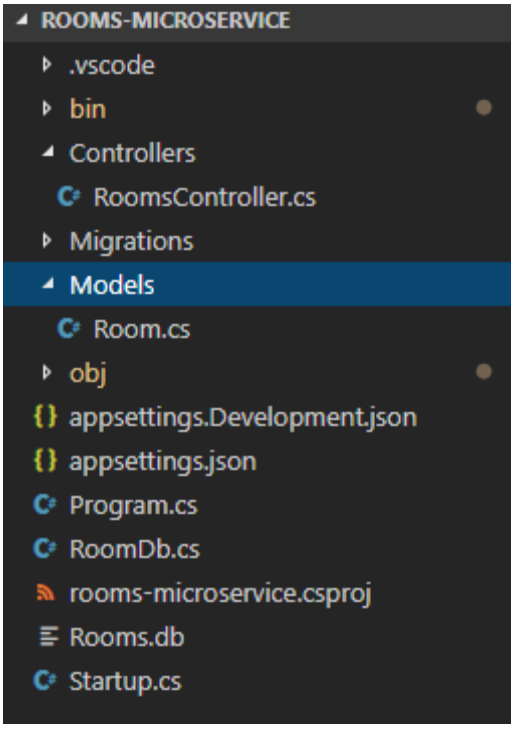
Na slikama 4.2, 4.3 i 4.4 dane su komponente sve tri usluge auta, korisnika i prostorija. Kontroleri ovih usluga pisane su u klasama .cs datoteka (CarsController.cs), te također i modeli ovih usluga realizirani kao klase pisane u .cs datotekama (Car.cs). Ove klase bit će objašnjenje u sljedećem poglavlju.



Slika 4.2 Komponente auto usluge

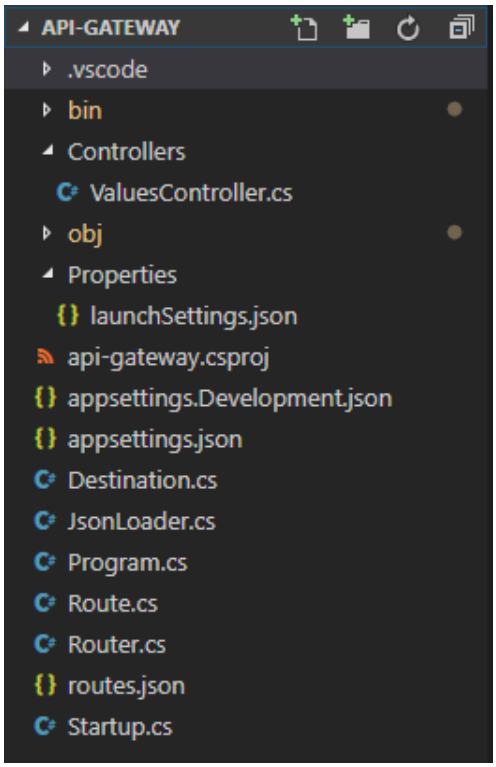


Slika 4.3 Komponente korisnika usluge

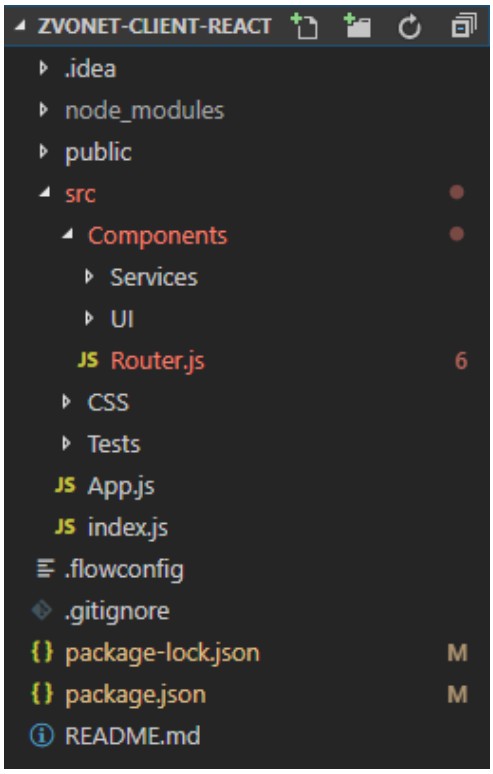


Slika 4.4 Komponente prostorija usluge

Na slici 4.5 je popis komponenti aplikacije poslužitelja – api-gatewaya koji preusmjerava sve dolazne upite na mikrousluge. U ovoj aplikaciji najbitnije su 3 datoteke – Route.cs koja predstavlja model u kojemu su svojstva ove aplikacije opisane, Router.cs u kojemu definirani zahtjevi i postavke poslužitelja, te od datoteke routes.json u kojoj su opisane odredišne adrese za preusmjeravanje na mikrouslugu. Ove klase su detaljnije opisane u sljedećem poglavlju. Na slici 4.6. je popis komponenti korisničkog sučelja.



Slika 4.5 Komponente api-gatewaya



Slika 4.6 Komponente korisničkog sučelja

4.2. Primjer koda za mikrousluge

Kao što je u prethodnom poglavlju objašnjeno, potrebno je napraviti modele i kontrolere za svaki od pojedine usluge. Najprije su napravljene klase koje predstavljaju model za svaku uslugu koji su napisane u User.cs, Room.cs i Car.cs datotekama (slika 4.7). U tim modelima su predefinirana svojstva pojedine usluge. Za svaki model imam ključ po kojim prepoznajemo pojedini model. Nadalje, u svakoj od usluga je napravljena klasa koja predstavlja kontroler svake usluge i one su zaslužne za sve upite na bazi podataka. . Na slici 4.10 dan je primjer za kontroler usluge korisnika. Kontroler sadrži REST metode *Get*, *Post*, *Put* i *Delete* koje vrše operacije nad zadanom uslugom. Na kraju je bilo potrebno povezati svaku uslugu sa svojom bazom podataka. Povezivanje je omogućeno preko *Entity Frameworka* koji povezuje svaku uslugu sa SQLite bazom podataka.

```
public class Car{
[Key]
public int carId {get; set;}
public string manufacturer {get; set;}
public string model {get; set;}
public float motorType {get; set;}
public int kilometers {get; set;}
}
```

```
public class Room{
[Key]
public int roomId {get; set;}
public string name {get; set;}
public int seats {get; set;}
}
```

```
public class User{
[Key]
public int OIB {get; set;}
public string name {get; set;}
public string lastname {get; set;}
public int age {get; set;}
public string city {get; set;}
[DataType(DataType.EmailAddress)]
public string email { get; set; }
}
```

Slika 4.7 Modeli svih usluga

Na slici 4.8 je opisana Get metoda kontrolera usluge, imamo dvije mogućnosti za get metodu. Prvi slučaj je kada nemamo id korisnika, te ona prikazuje sve korisnike iz baze podataka. Na nju pristupamo preko http adrese lokalnog poslužitelja /controller, u ovom slučaju će to biti localhost:3000/users. Na slici možemo vidjeti alociranje nove memorije baze podataka koja prosljeđuje sve korisnike iz baze podataka u listu. U drugom primjeru možemo vidjeti traženje korisnika po njegovom ID-u (pr. localhost:3000/users/1). U ovom slučaju uspoređuje dani ID sa OIB-om korisnika iz baze podataka te ga prosljeđuje na zadani zahtjev.

```
[Route("[controller]")]
    public class UsersController : Controller
    {
        [HttpGet]
        public IEnumerable<User> Get()
        {
            using (UserDb db = new UserDb())
            {return db.Users.ToList();}
        }
        [HttpGet("{id}")]
        public User Get(int id)
        {
            using (UserDb db = new UserDb())
            {return db.Users.First(t => t.OIB == id);}
        }
}
```

Slika 4.8 Primjer Get metode

Na slici 4.9 je opisana Delete metoda usluge korisnika. Na ovom primjeru se vidi kako metoda Delete prima vrijednosti id, te ju uspoređuje u bazi podataka sa OIB-om, te ju na kraju briše sa liste iz baze podataka.

```
[HttpDelete]
        public void Delete(int id)
        {
            using (UserDb db = new UserDb())
            {
                if (db.Users.Where(t => t.OIB == id).Count() > 0) // Check if
element exists
                    db.Users.Remove(db.Users.First(t => t.OIB == id));
                db.SaveChanges();
            }
        }
```

Slika 4.9 Primjer Delete metode

```
[Route("[controller]")]
    public class UsersController : Controller
    {
        [HttpGet]
        public IEnumerable<User> Get()
        {
            using (UserDb db = new UserDb())
            {return db.Users.ToList();}
        }
        [HttpGet("{id}")]
        public User Get(int id)
        {
            using (UserDb db = new UserDb())
            {return db.Users.First(t => t.OIB == id);}
        }
        [HttpPost]
        public void Post([FromBody]JObject value)
        {
            User posted = value.ToObject<User>();
            using (UserDb db = new UserDb())
            {
                db.Users.Add(posted);
                db.SaveChanges();
            }
        }
        [HttpPut("{id}")]
        public void Put(int id, [FromBody]JObject value)
        {
            User posted = value.ToObject<User>();
            posted.OIB = id;
            using (UserDb db = new UserDb())
            {
                db.Users.Update(posted);
                db.SaveChanges();
            }
        }
        [HttpDelete("{id}")]
        public void Delete(int id)
        {
            using (UserDb db = new UserDb())
            {
                if (db.Users.Where(t => t.OIB == id).Count() > 0)
                    db.Users.Remove(db.Users.First(t => t.OIB == id));
                db.SaveChanges();
            }
        }
    }
```

Slika 4.10 Kontroler usluge korisnika

Nakon što smo realizirali sve potrebne usluge, potrebno je realizirati poslužitelj api-gateway koji će preusmjeravati sve potrebne zahtjeve sa korisničkog sučelja prema zadanim uslugama. Svaki od uslugama je podignut na jedan lokalni server na koje api-gateway preusmjerava sve potrebne zahtjeve. Na slici 4.11 opisana je klasa Router.cs iz api-gatewaya koja opisuje kako obrađuje zahtjev.

```
  public async Task<HttpResponseMessage> RouteRequest(HttpRequest request)

        {
            string path = request.Path.ToString();
            string basePath = '/' + path.Split('/')[1];

            Destination destination;
            try
            {
             destination = Routes.First(r =>
r.Endpoint.Equals(basePath)).Destination;
            }
            catch
            {
                return ConstructErrorMessage("The path could not be found.");
            }

            if (destination.RequiresAuthentication)
            {
                string token = request.Headers["token"];
             request.Query.Append(new KeyValuePair<string, StringValues>("token",
      new           StringValues(token)));
                HttpResponseMessage authResponse = await
AuthenticationService.SendRequest(request);
                if (!authResponse.IsSuccessStatusCode) return
ConstructErrorMessage("Authentication failed.");
            }

            return await destination.SendRequest(request);
        }
```

Slika 4.11 Obrada zahtjeva api-gatewaya

Na slici 4.12 je datoteka routes.json u kojoj je opisano na koji poslužitelj šalje zahtjev api-gateway, te je opisano na kojim lokalnim poslužiteljima su podignute usluge. Kao što možemo vidjeti iz slike, usluga korisnika podignuta je na lokalnom poslužitelju 6000, usluga auta podignuta je na 7000, a usluga prostorije podignuta je na 8000.

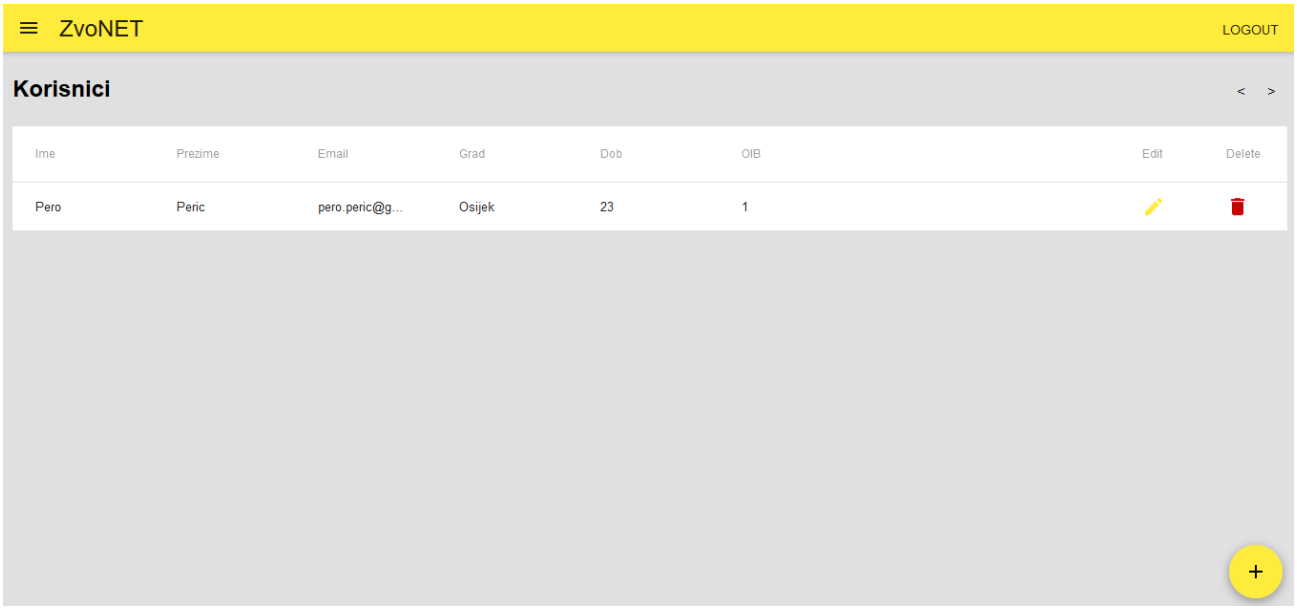
```
"routes": [

    {
      "endpoint": "/users",
      "destination": {
        "uri": "http://localhost:6000/",
        "requiresAuthentication": "false"
      }
    },
    {
      "endpoint": "/cars",
      "destination": {
        "uri": "http://localhost:7000/",
        "requiresAuthentication": "false"
      }
    },
    {
      "endpoint": "/rooms",
      "destination": {
        "uri": "http://localhost:8000",
        "requiresAuthentication": "false"
      }
    }
  ]
```

Slika 4.12 Datoteka routes.json

5. TESTIRANJE APLIKACIJE I KORIŠTENJE

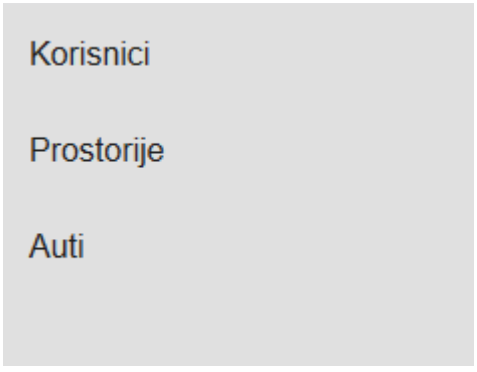
5.1. Korištenje aplikacije

Prilikom otvaranja aplikacije, korisniku se otvara početna stranica svih dostupnih korisnika u bazi (Slika 5.1). Zbog velikog obujma aplikacije, objašnjeno je korištenje usluge auta, a analogno se koriste i druge usluge.



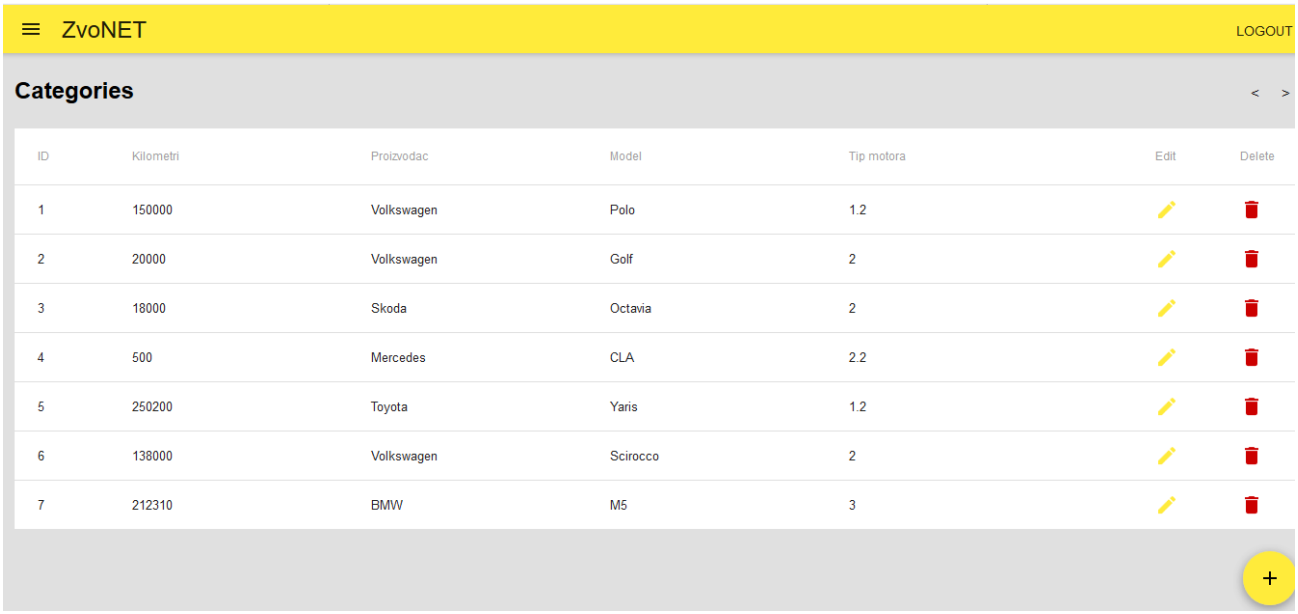
Slika 5.1. Početna stranica aplikacije

Klikom na izbornik u gornjem lijevom kutu otvara nam se izbornik sa svim uslugama (Slika 5.2).



Slika 5.2 Izbornik aplikacije

Klikom na „Auti“ dobivamo uslugu auta sa njegovom bazom podataka (Slika 5.3). Na toj stranici možemo vidjeti sve dostupne aute, te njegove informacije kao što su ime proizvođača, model auta, tip motora te kilometre koje je prešao.

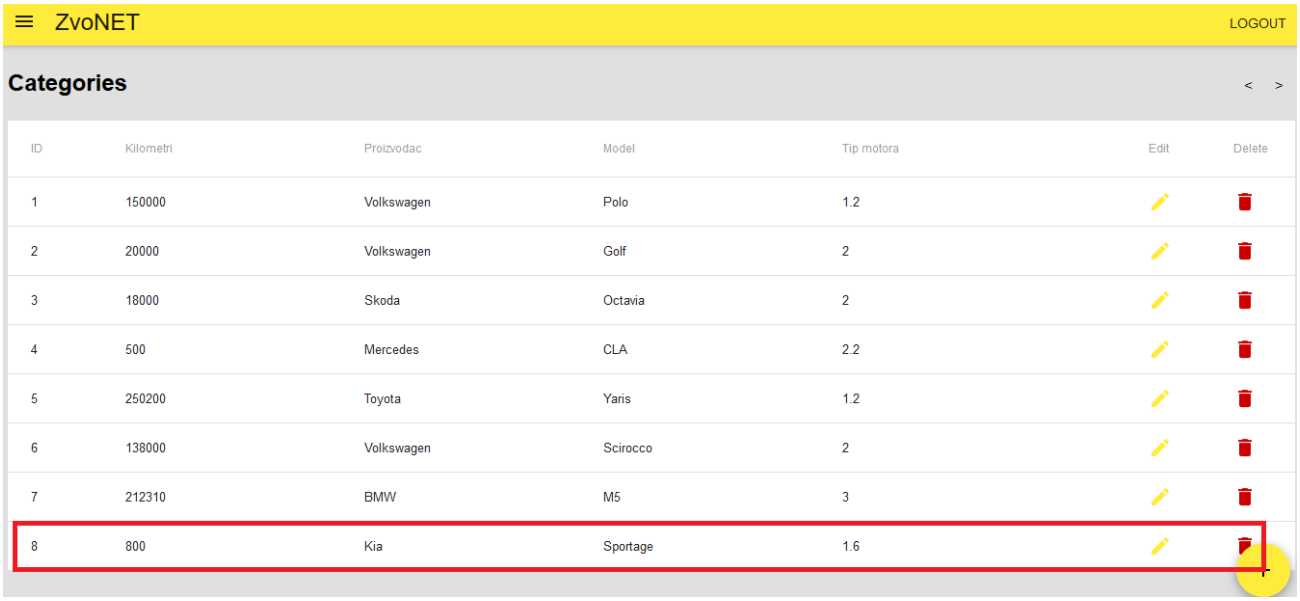


Slika 5.3. Prikaz dostupnih auta

Klikom na gumb plus u donjem desnom uglu možemo dodavati neki novi auto u uslugu auta (Slika 5.4)

Slika 5.4 Kreiranje novog auta

Klikom na „Submit“ naš auto je dodan u uslugu auta kojeg možemo vidjeti na slici 5.5. Također imamo opcije brisanja auta i uređivanja podataka auta.

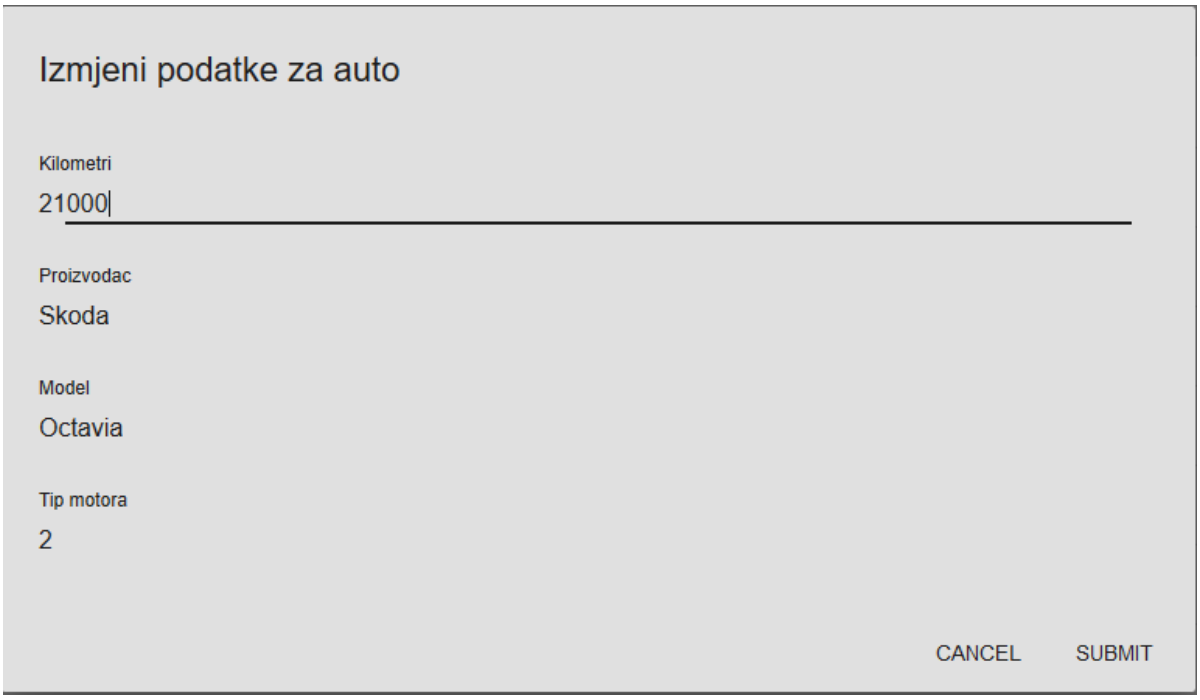


Slika 5.5 Prikaz dodanog auta

5.2. Testiranje aplikacije

Testiranje aplikacije podrazumijeva testno korištenje aplikacije u svrhu ispitivanja osnovnih funkcionalnosti aplikacije. U ovom slučaju je to izmjena podataka vozila, korisnika ili prostorije, te njegovo brisanje iz baze podataka.

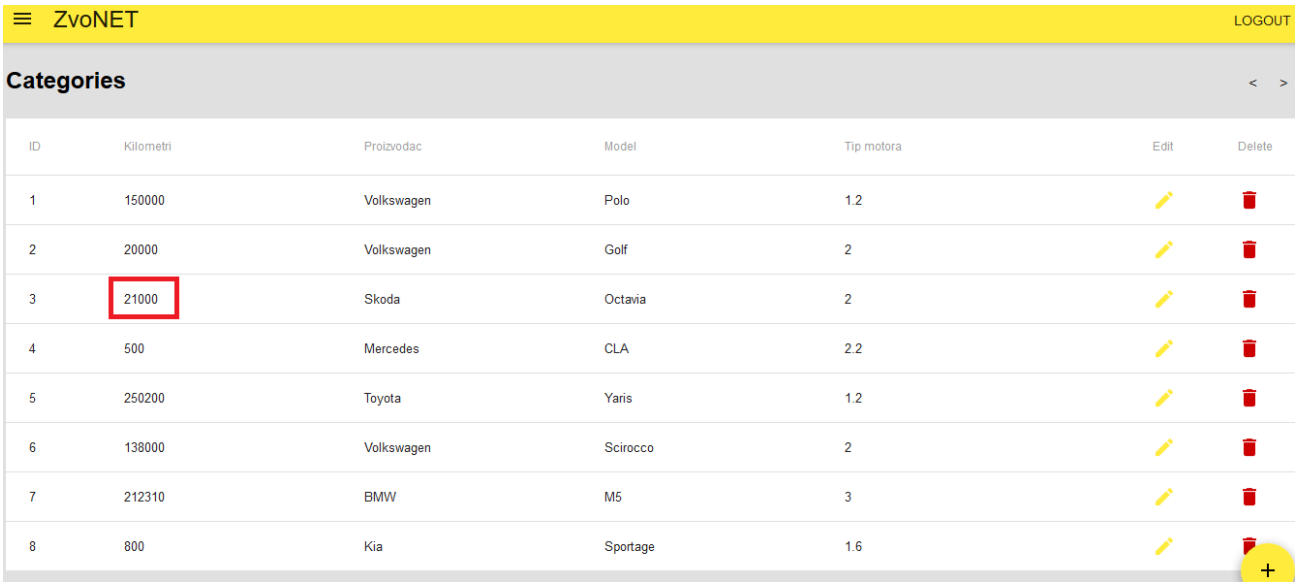
Klikom na izbornička traku, korisnik ponovno može birati između vozila, korisnika i prostorija (pr. Slika 5.2). Klikom na kategoriju vozila otvara nam se usluga svih dostupnih vozila. Na slici 5.6 je prikaz za uređivanja auta koji ima ID vrijednost 3.



Slika 5.6 Prikaz izmjene auta

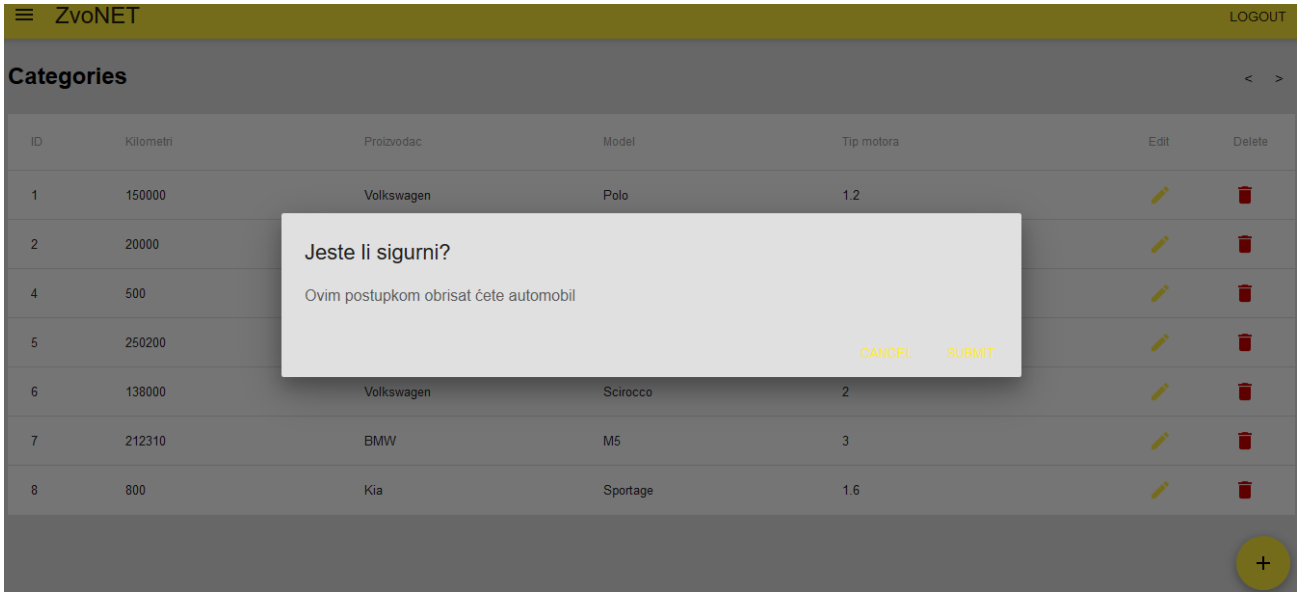
Ukoliko ne unesemo neku od vrijednosti, izbacit će nam iznimku „*This field is mandatory!*“ što možemo vidjeti na slici 5.7. i moramo unijeti traženu vrijednost. Izmijenit ću podatke auta koji ima ID vrijednost 3 te ću mu prepraviti vrijednost kilometara iz 18000 u 21000 što možemo vidjeti na slici 5.8.

Slika 5.7 Prikaz iznimke ako ne unesemo neku od vrijednosti



Slika 5.8 Prikaz izmijenjene vrijednosti auta

Konačno, možemo svaki auto izbrisati iz naše baze podataka. Klikom na taj gumb, traži nas potvrdu za brisanje auta, te ukoliko stisnemo „Submit“, naš auto će biti izbrisan iz baze podataka (Slika 5.9)



Slika 5.9 Brisanje auta iz baze podataka

6. ZAKLJUČAK

U radu se moglo vidjeti korištenje mikrousluga preko ostvarene web aplikacije za poslovne aktivnosti tvrtke. Aplikacija je načinjena u programskom okruženju Visual Studio Code, u kojem su programirane usluge u .NET programskom okviru, te su preko *Entity Frameworka* povezani sa svojom bazom podataka. Korisniku je trebalo omogućiti uvid u dostupnost zaposlenika, dostupnost automobila te dostupnost prostorija. Potrebno je bilo objasniti na ovom jednostavnom primjeru rad mikrousluga te ga realizirati.

Iz ove aplikacije može se vidjeti da je korištenje mikrousluga korisno, ali i relativno zahtjevno. Korištenje mikrousluga je vrlo poželjno raditi na većim aplikacijama, gdje programeri mogu raditi na pojedinoj usluzi bez ometanja drugih programera koji obrađuju drugu uslugu. Također, zbog puno bolje prilagodljivosti novim okolinama i zbog mogućnosti programiranja u više programskih jezika, korištenje mikrousluga je vrlo poželjno. Ukoliko jedna usluga ne radi, cijela aplikacija neće prestati raditi, nego će aplikacija neometano nastaviti raditi bez te pojedine usluge. Zbog sporijeg odziva na usluge, pri manjim aplikacijama je poželjnije raditi monolitne aplikacije.

LITERATURA

[1] M. Rouse, Monolithic Architecture, dostupno na: https://whatis.techtarget.com/definition/monolithic-architecture, datum pristupanja 25.06.2018.

[2] C. Richardson, Pattern: Monolithic Architecture, dostupno na: http://microservices.io/patterns/monolithic.html, datum pristupanja 25.06.2018.

[3] S. Schuller, Why monolithic apps are often better than microservices, dostupno na: https://gigaom.com/2015/11/06/why-monolithic-apps-are-often-better-than-microservices/, datum pristupanja: 25.06.2018.

[4] C. Richardson, What are microservices?, dostupno na: http://microservices.io/, datum pristupanja: 27.06.2018.

[5] Smartbear, What is Microservices Architecture? https://smartbear.com/learn/api-design/what-are-microservices/, datum pristupanja: 27.06.2018.

[6] M. Awasth, Microservice vs Monolithic Architecture, dostupno na: https://nodexperts.com/blog/microservice-vs-monolithic/, datum pristupanja: 27.06.2018.

[7] S. Graham, Building Web Services with Java, dostupno na: http://books.gigatux.nl/mirror/buildingwebserviceswithjava/0672321815_ch01lev1sec5.html/, datum pristupanja: 03.09.2018.

[8] M. Stevens, The Benefits of a Service-Oriented Architecture, dostupno na: https://www.developer.com/services/article.php/1041191/The-Benefits-of-a-Service-Oriented-Architecture.html/, datum pristupanja: 03.09.2018.

[9] P. Minkowski, Communicating Between Microservices, dostupno na: https://dzone.com/articles/communicating-between-microservices/, datum pristupanja: 03.09.2018.

SAŽETAK

Zadatak ovog završnog rada bio je izrada web aplikacije temeljene na mikrouslugma za aktivnosti tvrtke. Korisniku je trebalo omogućiti uvid u dostupnost zaposlenika, dostupnost automobila, te u dostupnost prostorija. Svaki ovaj model zaposlenika, automobila i prostorija predstavljao je jednu zasebnu mikrouslugu. Aplikacija omogućuje jednostavno dodavanje, uređivanje i brisanje zaposlenika, automobila i prostorija. U teorijskom dijelu rada sadržan je opis arhitektura web aplikacija, te opis arhitekture koja se koristi, opis postupka razvoja aplikacije, model aplikacije, opisano je programsko rješenje web aplikacije koje se sastoji od primjera koda za pojedino dijelove, te prikladno testiranje aplikacije. U realizaciji same aplikacije potrebno je spomenuti korištenje programskog okvira .NET, arhitekture mikrousluga, te okoline Visual Studio Code.

Ključne riječi: Arhitekture aplikacija, Mikrousluge, .NET okvir, Visual Studio Code

ABSTRACT

The task of this final paper was to make a web application based on microservices used for company activities. User has insight into employee availability, car access, and rooms availability. Each model of employee, cars and rooms is presenting separate microservice. Application allows user to add, edit and delete employees, cars and rooms. The theoretical part describes the web application architecture, architecture that is used in the application, application model and software solution. Software solution consist of code examples for individual parts and an appropriate application testing. In realization of the application, there is a need to mention using of .NET framework, the microservice architecture and the Visual Studio Code editor.

Key words: Microservices, .NET framework, Software architecture, Visual Studio Code

ŽIVOTOPIS

Zvonimir Stipanović rođen je 06.03.1996. godine u Đakovu, Hrvatska. Stanuje u Đakovu na adresi Andrije Hebranga 81. 2002. godine započinje osnovnoškolsko obrazovanje u OŠ Josipa Antuna Ćolnića gdje se iskazao na brojnim županijskim natjecanjima iz matematike i informatike i sudjelovao je na Državnom natjecanju 2010. godine u smotri radova na Infokupu. Nakon završene osnovne škole, 2010. godine upisuje Gimnaziju Antuna Gustava Matoša u Đakovu. Nakon Gimnazije, 2014. upisuje Preddiplomski studij računarstva na Fakultetu elektrotehnike, računarstva i informacijskih tehnologija u Osijeku. Tijekom studiranja stječe određeno znanje iz objektnog programiranja (C#, C++) te određeno znanje engleskog jezika.

PRILOZI (na CD-u)

Prilog 1: Završni rad u docx formatu

Prilog 2: Završni rad u pdf formatu

Prilog 3: Projekt web aplikacije