1. Uvod

Postoji veliki broj web i mobilnih aplikacija koje rešavaju problem rezervisanja avionskih karata ili *rent-a-car* usluga. Takve aplikacije su promenile tržište i način na koji se vrši rezervacija i prilagodile se novom stilu i brzini života. Drastično je olakšan proces potrage za željenim sadržajem i upoređivanjem cena i nivoa usluge. Ovakav način rezervacije omogućio je manjim kompanijama da se brže razvijaju jer mogu na lakši način da se probiju sa već postojećom konkurencijom.

U okviru ovog rada će biti dato rešenje na gore opisani problem. Pored samog nivoa kompleksnosti i realističnosti rešenja, osnovna razlika u odnosu na već postojeće aplikacije jeste da ovo rešenje nema mogućnost plaćanja putem interneta, čime se zapravo završava ceo proces odabira, rezervacije i kupovine karata ili iznajmljivanja *rent-a-car* vozila, dok su ostale funkcionalnosti odrađene verodostojno drugim rešenjima.

Sam zadatak je realizovan kao SPA (engl. *Single Page Application*) veb aplikacija sa serverskom stranom razvijenom na mikroservisnoj arhitekturi uz osvrt na razmenu poruka između samih servisa.

U okviru drugog poglavlja biće detaljnije opisan problem koji je rešavan dalje u radu. Treće je posvećeno tehnologijama koje su korišćene kako bi se implementiralo rešenje. Početno rešenje opisano je u četvrtom poglavlju, dok peto daje detaljan opis realizacije nedostataka početnog rešenja. Šestim poglavljem se sumira ceo rad i daje se predlog za dalje unapređenje.

2. Opis problema

U okviru ovog poglavlja će biti razmotreno pitanje komunikacije između komponenata, a u petom poglavlju će biti detaljno razrađeno jedno od mogućih rešenja, razlog zašto je odabran taj način, njegove prednosti i način za dalje unapređenje.

Kada se posmatra monolitna arhitektura, u okviru takvog sistema ne postoji potreba za razmenom poruka i podataka u klasičnom smislu pomoću nekog od odabranih protokola ili pomoću neke druge tehnologije. Razmena podataka se može realizovati na neki od naredna dva načina, instanciranjem same komponente čiji su nam podaci potrebni ili injekciom, tako što se referencira apstraktni objekat, a ne konkretna instanca objekta. Monolitni sistemi se izvršavaju na jednom serveru i izvršava ih jedan proces.

Jedan od ključnih problema na koji se nailazi kada je sistem potrebno prebaciti na mikroservisnu arhitekturu je razrešiti sve reference koje su kreirane korišćenjem drugih komponenata ili njihovom injekciom. Zbog prirode mikroservisne arhitekture koja nam omogućava da se servisi mogu izvršavati na različitim serverima, komunikacija se mora vršiti pomoću nekih protokola poput HTTP, AMPQ ili TCP. Poželjno je da se komunikacija između komponenata svede na minimum kako bi bile nezavisne jedne od drugih u što većoj meri. Razdvajanjem komponenata i smanjivanjem obima komunikacije se postiže veća nezavisnost samih servisa, što dovodi do boljeg rada same aplikacije i do prijatnijeg iskustva samih korisnika. Ovakav način razmišljanja se naziva „*smart endpoints and dumb pipes*" i njime se propagira ideja da servisi treba da budu što razdvojeniji, a logika u okviru mikroservisa što kohezivnija. Dva najčešće korišćena protokola su HTTP, za izvršavanje upita i jednostavna asinhrona razmena poruka između mikroservisa.

Način razmene poruka se može podeliti u četiri grupe u odnosu na dva kriterijuma. Prvi kriterijum je sinhronost protokola. Sinhroni protokoli vrše razmenu na principu slanja poruke i čekanja na njen odgovor. Klasičan primer sinhronog protokola jeste HTTP. Pri korišćenju ovakvih protokola nit koja izvršava određenu metodu je blokirana dok ne dobije odgovor. Mana je potencijalno usporavanje rada servera ili klijenta. Drugu grupu predstavljaju asinhroni prokoli koji nakon poslate poruke ne čekaju na njen odgovor, već nastavljaju sa svojim radom. Predstavnik ove grupe protokola jeste AMPQ protokol.

Drugi kriterijum za podelu protokola jeste broj primaoca poruka. Ukoliko poruka ima jednog primaoca, treba da je obradi tačno jedan proces. Primena šablona Komanda najbolje oslikava ovu situaciju i slučaj korišćenja. U slučaju da postoji više primaoca, poruku može da primi nula ili više procesa. Kada se koristi ovakav tip komunikacije, neophodno je obezbediti da onda ona bude asinhrona. Primer jeste mehanizam *publish/subscribe* koji koriste arhitekture bazirane na događajima.

U mikroservisnoj arhitekturi se najčešće koristi sinhrona komunikacija sa jednim primaocem kada je u pitanju razmena poruka sa prednjom stranom, dok se za komunikaciju među servisima koristi asinhron način.

3. Opis korišćenih tehnologija

Prilikom izrade datog projekta korišćeno je više tehnologija. U nastavku poglavlja će biti obrađene najznačajnije uz navođenje njihovih osnovih odlika i prednosti, kao obrazloženja razloga za njihovo korišćenje.

3.1 Angular

Angular predstavlja razvojno okruženje (engl. *framework*) namenjeno razvoju klijentskih veb aplika. Razvijen je od strane Google-a i pisan je u programskom jeziku TypeScript. Poslednja verzija u trenutku pisanja ovog rada je 10.0.4.

Angular je baziran na komponentama, koje zajedno čine stablo. Svaka komponenta se sastoji iz tri fajla, to su HTML (engl. *Hypertext Modeling Language*) dokument, CSS (engl. *Cascading Style Sheet*) dokument i TypeScript dokument. Kako više komponenata čini jednu celinu, njih grupišemo u module. Svaka aplikacija mora imati najmanje jedan moduo koji koristi korensku komponentu. Iako se Angular koristi za kreiranje SPA aplikacija, pomoću rutiranja, korisnici stiču uticaj kao da prelaze sa jedne strane na drugu. Pored komponenata, postoje i servisi, čuvari (engl. *guards*) i cevi (engl. *pipes*). Servisi služe kako bismo izmestili deo logike van komponenata. Servise u Angularu nije potrebno instancirati u okviru komponenata, već se oni koriste injekcijom. *Gurad*-ovi se mogu koristi za sprečavanje neautorizovanog pristupa određenim putanjama, za upozoravanje klijenata da neka akcija nije izvršena do kraja i da će svi podaci biti izgubljeni, za učitavanje podataka pre nego što se prikaže određena komponenta ili da proveri da li podaci mogu da se pribave za određenu komponentu. *Pipe*-ovi se najčešće koriste kako bi se vršila dinamička filtracija.

3.1.1 TypeScript

TypeScript je besplatan jezik, razvijan kao otvoren kod (engl. *open source*) od strane Microsoft-a. Predstavlja nadskup JavaScript-a, baziran je na ECMAScript6, stvarima planiranim za ECMAScript7 i sa dodacima statičkih tipova.

TypeScript se pomoću specijalnog programa, tzv. *transpiler*-a prevodi u čist JavaScipt programski kod koji se lako izvršava u bilo kom internet pretraživaču. Objektno je orijentisan i moguće je definisati tipove promenljivih, kako bi one bile poznate pre prevođenja koda. Neke od drugih stvari po kojima se razlikuje u odnosu na JavaScript su klase, lambda funkcije i moduli.

3.1.2 HTML

HTML predstavlja opisni jezik koji je namenjen za definisanje veb stranica i trenutna aktuelna verzija je HTML 5. U HTML-u se objekti definišu pomoću tagova. Tagovi mogu biti prosti, ako je dovoljan samo jedan tag da se opiše element ili mogu biti složeni, tada je potrebno koristiti otvarajući i zatvarajući tag. Svi tagovi imaju posebne attribute koji po potrebi mogu biti definisani.

HTML dokument ima predefinisanu strukturu, ceo dokument je obuhvaćen tagovima <html></html> u njemu se nalazi zaglavlje, definisano <head></head> tagovima, u okviru kog su definisani meta podaci, naziv, i sl. Telo se takođe nalazi između html tagova i obuhvaćeno je <body></body> tagovima. Telo HTML dokumenta sadrži elemente stranice.

3.1.3 CSS

CSS je jezik koji se koristi da opiše izgled komponenata na veb stranici. Uveden je kako bi rasteretio HTML stranicu i omogućio lakši način da se postigne konzistentnost na celoj veb stranici.

3.2 Bootstrap

Bootrstap je besplatan CSS *framework* namenjen za razvijanje responzivnih veb sadržaja. Pored definisanja CSS-a šablona, takođe definiše i JavaScript šablone za forme, dugmad i sl.

3.3 Angular Material

Angular Material je biblioteka koja definiše UI komponente koje implementiraju Material Design. Material Design je dizajnerski jezik razvijen od strane Google-a i forsira korišćenje rešetke za raspored elemenata na stranici i koristi motiv kartica, koji se može videti na velikom broju Google-ovih aplikacija i veb sajtova.

3.4 .NET Core

.NET Core je besplatan *open-source framework* za Windows, Linux i macOS i predstavlja *cross-platform* naslednik .NET Framework-a. Najvećim delom je razvijen od strane Microsofta. Prva verzija je puštena u rad 2016. godine, a poslednja aktuelna verzija je 3.1. .NET Core je baziran na C# programskom jeziku. Prednosti .Net Core su mogućnosti lakog pravljenja skalabilnih rešenja koristeći mikroservise, kontejnere i druge poznate tehnike kreiranja mikroservisne arhitekture. Srž jezika je preuzeta u potpunosti, sa svim ispravljenim problemima koje je imao .Net Framework. Preuzete su takođe verzije Identity-ja (koristi se za autentifikaciju i autorizaciju), Entity Framework-a (ORM koji koristimo za pristup bazi), kao i mnogi drugi drajveri i biblioteke – sve kako bi se povećala kompatibilnost i efikasnost novog .Net *framework*-a.

3.4.1 C#

C# je programski jezik opšte namene. Podržava imperativnu, objektno orijentisanu, funkcionalnu i deklarativnu paradigmu. Razvijen je od strane kompanije Microsoft i spada u grupu novijih programskih jezika. Sintaksno veoma podseća na C i C++ jezike, ali je njegova znatno jednostavnija. Takođe, obezbeđeno je i automatsko rukovanje memorijom što ga čini sigurnijim za programiranje.

3.5 MySQL

MySQL je SQL sistem za upravljanjem relacionom bazom podataka. MySQL je originalno razvijan za manipulaciju vrlo velikih baza podataka, mnogo je brži od postojećih rešenja i uspešno se koristi u visoko zahtevnim okruženjima. Pristupačnost, brzina i sigurnost čine MySQL vrlo pogodnim za pristupanje bazama podataka preko interneta.

4. Opis trenutnog rešenja

Kao što je već rečeno, ovaj sistem je realizovan kao SPA(*Single Page Application*) veb aplikacija. U daljem tekstu će biti detaljno opisano početno rešenje, kao i njegovi nedostaci koji su rešeni u okviru ovog rada, o čemu će biti više reči u 5. poglavlju.

4.1 Arhitektura rešenja

4.1.1 Klijentska strana

Osnovni zadatak prednje strane jeste da korisnicima na što intuitivniji način ponudi lagodno korišćenje aplikacije i najjednostavniji način da izvrše željenu akciju. O okviru ove veb aplikacije prednja strana je realizovana pomoću *Angular framework-a*.

4.1.2 Serverska strana

Serverska strana je realizovana pomoću mikroservisne arhitekture u *.NET Core*-u. Ne postoji jedna tačna definicija mikroservisa iz razloga što se oni konstantno razvijaju zajedno sa industrijom i njenim potrebama. Sistem se najčešće deli u mikroservise na osnovu funkcionalnosti. Prilikom podele potrebno je vrlo dobro razumevanje sistema. Ukoliko ih podelimo na manje servise nego što je potrebno, dobijamo veću zavisnost između njih, što svakako nije cilj. Ovakav stil arhitekture omogućava potpuno nezavisno razvijanje servisa, samim tim oni mogu biti razvijani uz pomoć različitih tehnologija. Mikroservisna arhitektua donosi određene prednosti u odnosu na monolitnu arhitekrutu,a najznačajnije su: modularnost, skalabilnost i distributivni razvoj. Modularnost olakšava razumevanje samih servisa, njihov razvoj i testiranje. Zbog podele na servise, gubi se prevelika kompleknost monolitnih aplikacija. Skalabilnost omogućava da se svaki servis nezavisno skalira, čime ne utiče na ostatak sistema. Distributivni razvoj se odnosi na prednost nezavisnosti mikroservisa, što omogućava da više timova u potpunosti nezavinost razvija, testira i pušta u rad servise.

Ova aplikacija je podeljena na 3 mikroservisa: *AirlineMicroservice*, *RentACarMicroservice* i *UserMicroservice*. Svaki od mikroservisa je implementiran tako da prati načela odabrane arhitekture, pod tim se smatra da svaki mikroservis ima svoju bazu podataka i slabo je spregnut sa drugim servisima. Još jedna prednost ove arhitekture jeste da u slučaju da jedan od servisa iznenada prestane da radi, ostatak aplikacije može neometano da nastavi da radi.

Svaki od servisa je kontejnerizovan kako bi se olakšalo njegovo održavanje i skaliranje, za ovaj proces je odabrano da se koriste *Docker* kontejneri. Proces započinje generisanjem *Docker* fajla. U ovom tekstualnom dokumentu je definisan način na koji je potrebno generisati sliku kontejnera. Slika predstavlja paket sa svim zavisnostima i informacijama koje su neophodne kako bi se generisao sam kontejner. Najčešće se sastoji iz više osnovnih slika na koje se dodaju dodatna podešavanja definisana *Docker* fajlom. Sam kontejner predstavlja jednu instancu slike, tj. jednu instancu same aplikacije ili u našem slučaju jednog servisa. Korišćenjem kontejnera je drastično olakšan proces skaliranja i održavanja instanci servisa, jer se uvek instanciraju u odnosu na istu sliku. Za lakše upravljanje kontejnerima korišćen je alat za orkestraciju kontejnera *Docker Compose*.

Mikroservisi su realizovani kao veb aplikacije. Svaki servis sadrži kontrolere koji su zasnovani na REST (*Representational state transfer*) arhitekturi i čija je uloga obrada zahteva i prosleđivanje podataka. Osnovni koncept REST arhitekture jeste resurs kojim je predstavljen svaki entitet na vebu. Entiteti su nedvosmisleno identifikovani svojim URI-jem (Uniform resource locator) i nad njima je moguće vršiti jednostavne operacije, koje odgovaraju HTTP metodama, poput: GET (čitanje), POST (kreiranje), PUT (ažuriranje) i DELETE (brisanje). Za razmenu podataka je odabran JSON (*JavaScript Object Notation*), ali je moguće koristiti i XML (*Extensible Markup Language*) kao format podataka.

Pored kontrolera, svaki servis sadrži modele podataka koji se koriste za rad sa bazom podataka, kao i DTO (*Data transfer objects*) modele za razmenu podataka sa klijentskom stranom. Potreba za DTO objektima ogleda se u tome što u bazu nije moguće direktno upisati liste prostih tipova podataka i jer je datume lakše slati u obliku *string*-a preko mreže.

4.2 Uloge i aktivnosti

Zbog kompleksnosti sistema, definisano je 5 vrsta korisnika:

- Neregistrovani korisnici
- Registrovani korisnici
- Sistem administratori
- Administratori avio-kompanija
- Administratori rent-a-car kompanija

Sistem ima mogućnost prijave pomoću korisničkog imena ili *email*-a i lozinke, a nudi i opciju logovanja pomoću već postojećih profila na *Google*-u i *Facebook*-u.

Svaki od korisnika ima specifične mogućnosti, neke od njih deli više tipova korisnika. Mogućnost koju dele svi korisnici osim sistem administratora jesu filtriranje i sortiranje letova, *rent-a-car* servisa i vozila po zadatim kriterijumima. Ovim je olakšana potraga za željenim informacijama i postignuto ubrzanje samog procesa.

4.2.1 Neregistrovani korisnici

Korisnici moraju da se registruju kako bi mogli da koriste napredne mogućnosti aplikacije. U procesu registracije od korisnika se traži da unese pored ličnih podataka i korisničko ime, email i lozinku. Nakon uspešne registracije korisnik prima *email*, koji sadrži *link* za verifikaciju i aktivaciju profila.

4.2.2 Registrovani korisnici

Registrovani korisnici imaju mogućnost rezervacije usluga, koje mogu biti rezervacija samo avionske karte, samo vozila ili i avionske karte i vozila. Postupak rezervacije avionskih karata se sastoji iz izbora leta, nakon čega korisnik unosi podatke o putnicima i vrši odabir sedišta za svakog putnika. Na samom kraju procesa, nudi se mogućnost dodatnog izbora usluga *rent-a-car*-a na odredišnoj lokaciji. Proces rezervacije vozila se sastoji iz izbora *rent-a-car* servisa, a potom izbora vozila za odabranu lokaciju i period. Korisnici takođe mogu da se povežu sa svojim prijateljima, a prijateljstvo je obostrano. Nakon što je zahtev za prijateljstvo prihvaćen, korisnik je u mogućnosti da ih pozove na putovanje. Oni su takođe u

mogućnosti da ocene letove, avio-kompanije, vozila i *rent-a-car* kompanije nakon završetka rezervacije. Svakim putovanjem korisnik koji je napravio rezervaciju leta ostvaruje poene, koje može da iskoristi kako bi umanjio cenu pri sledećoj rezervaciji. Korisnici imaju pregled svojih rezervacija, a u slučaju da žele da ih otkažu u mogućnosti su to da urade 3 časa pre poletanja ili 48 časova do vremena rezervacije vozila.

4.2.3 Sistem administratori

Administratori sistema su u mogućnosti da definišu popuste, tj. vrednost jednog poena, kao i procenat popusta za vozila koja su na brzom biranju. Pored toga sistem administrator je u mogućnosti da kreira profile drugih administratora.

4.2.4 Avio i rent-a-car administratori

Oba tipa administratora imaju mogućnost pregleda prosečnih ocena same kompanije i letova/vozila. Imaju mogućnost uvida u zaradu u određenom periodu, kao i grafički prikaz broja prodatih karata za odabrani period na nivou dana, nedelje ili meseca.

4.2.4.1 Administratori avio-kompanija

Administratori avio-kompanija su zaduženi za kreiranje profila svojih avio-kompanija, kao i za njihovo održavanje, a pored toga su u mogućnosti da dodaju, menjaju i brišu letove. Mogu da definišu popuste za pojedinačna sedišta, nakon čega se ta sedišta ne mogu izabrati pri redovnoj rezervaciji, već su jedino dostupna na stranici avio-kompanije gde se nalaze u odeljku za *fast-ticekts*.

4.2.4.2 Administratori rent-a-car kompanija

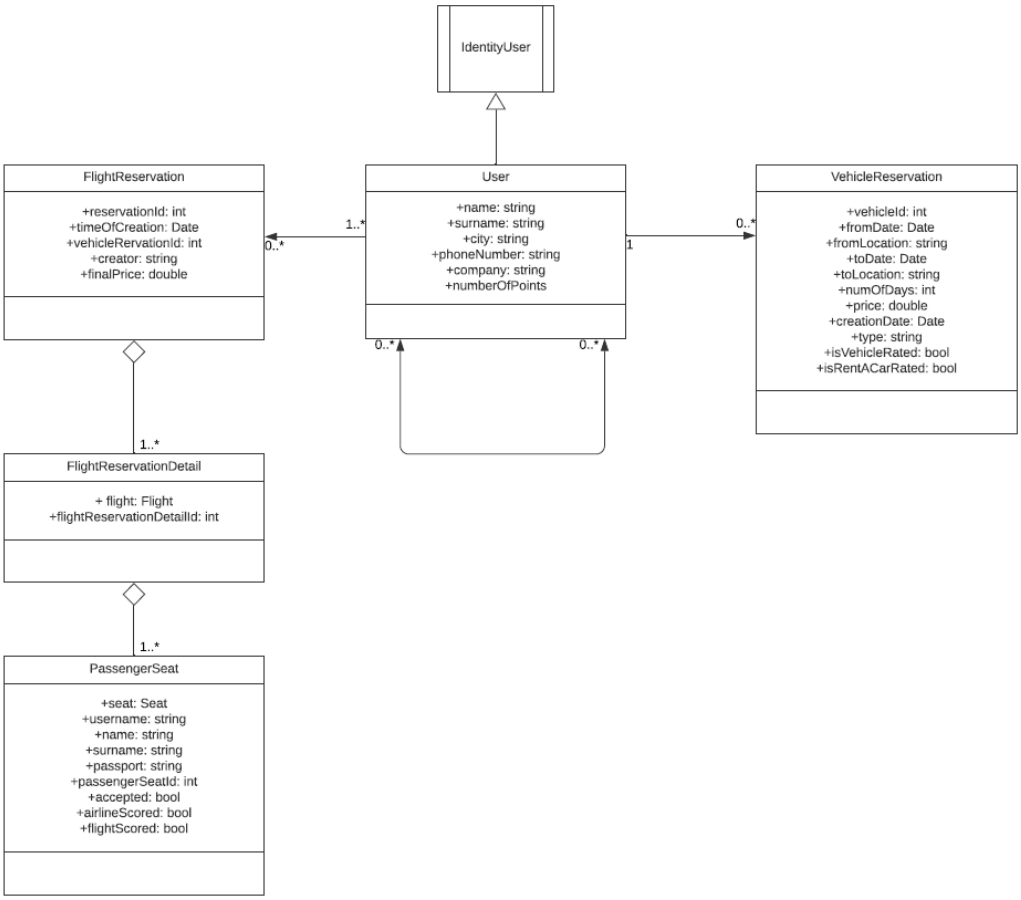
Administratori *rent-a-car* servisa su takođe zaduženi za kreiranje i održavanje profila svojih kompanija, kao i za dodavanje, izmenu i brisanje vozila. Administrator može da definiše koja se vozila nalaze na popustu. Kao i kod avio-kompanije. Vozila koja su na popustu se mogu odabrati samo pri rezervaciji avionskih karata.

4.3 Modeli

U okviru ovog podpoglavlja će biti opisani neki od ključnih modela za sistem. Na slici 4.3.1 prikazana je klasa *User* i njen odnos sa klasama koje modeluju rezervacije. Ove klase će biti detaljnije objašnjene u daljem tekstu.

U okviru Listinga 4.3.1 može se videti kako je modelovan korisnik pomoću klase User. Važno je napomenuti da se u okviru listinga navode modeli korišćeni na serverskoj strani radi lakše preglednosti. Za modelovanje korisnika odlučeno je da se koristi IdentityUser kako bi se olakšao rad sa osetljivim podacima, jer je ta klasa deo EntityFramework Core NuGet Package-a. Olakšice koje donosi ova klasa jesu automatsko *hash*-ovanje lozinki, mogućnost generisanja tokena za izmenu lozinke ili potvrdu *email* adrese. Pored toga postoje ugrađene funckije za izmenu lozinke kao i podešavanja za kriterijume koja lozinka mora da ispuni.

Pored predefinisanih polja dodato je, ime, prezime, adresa, broj telefona, kompanija i broj osvojenih poena.



Slika 4.3.1 Klasni dijagram za prikaz korisnika i rezervacija

Klasa *User* se koristi da opiše korisnika.

```
public class AppUser : IdentityUser
{
        public string Name { get; set; }

        public string Surname { get; set; }

        public string City { get; set; }

        public string Company { get; set; }

        public bool IsFirstLogIn { get; set; }

        public List<Friend> FriendsA { get; set; }

        public List<Friend> FriendsB { get; set; }

        public int NumberOfPoint { get; set; }
}
```

Listing 4.3.1 Klasa User - korisnik

Klase FlightReservation, FlightReservationDetail i PassengerSeat zajedno opisuju rezervaciju avionskih karata. Odlučeno je da se na ovaj način modeluje rezervacija da bi na jednostavan način mogao da se iskoristi isti model i za rezervaciju povratnog leta.

Klasa FlightReservation, prikazana na listingu 4.3.2., definiše koji korisnik je kreirao rezervaciju i kada, koja je ukupna cena i da li postoji rezervacija vozila koja je vezana za rezervaciju koja se kreira.

```
public class FlightReservation
{
        [Key]
        public int ReservationId { get; set; }

        public List<FlightReservationDetail> FlightReservationDetails
{ get; set; }

        public DateTime TimeOfCreation { get; set; }

        public string Creator { get; set; }

        public int VehicleReservationId { get; set; }

        public double FinalPrice { get; set; }

        public FlightReservation() { }
}
```

Listing 4.3.2 Klasa FlightReservation- osnova rezervacije leta

FlightReservationDetail, predstavljena na listingu 4.3.3, se kreira za svaki let koji se sadrži u rezervaciji. Za svaki od letova je potrebno definisati identifikacioni broj leta i ime aviokompanije.

```
public class FlightReservationDetail
{
        [Required]
        public FlightReservation FlightReservation { get; set; }

        [Key]
        public int FlightReservationDetailId { get; set; }

        public int FlightId { get; set; }

        public string AirlineName { get; set; }

        public List<PassengerSeat> PassengerSeats { get; set; }

        public FlightReservationDetail() { }
}
```

Listing 4.3.3 Klasa FlightReservationDetail – svaki let u okviru rezervacije

Poslednja klasa koja se koristi u procesu modelovanja leta jeste PassengerSeat, definiciju ove klase se može videti na listingu 4.3.4. Ova klasa se koristi da opiše svakog putnika i njemu adekvatno sedište. Za svakog putnika je potrebno da se popuni identifikacioni broj sedišta, korisničko ime korisnika ili u slučaju da putnik nema profil na veb stranici njegovo ime, prezime i broj pasoša. Za svakog putnika na svakom letu je potrebno voditi evidenciju da li je prihvatio zahtev za putovanje i da li je ocenio avio-kompaniju i let.

```
public class PassengerSeat
{
        [Required]
        public FlightReservationDetail FlightReservationDetail { get;
      set; }

        [Key]
        public int PassengerSeatId { get; set; }

        public int SeatId { get; set; }

        public string Username { get; set; }

        public string Name { get; set; }

        public string Surname { get; set; }

        public string Passport { get; set; }

        public bool Accepted { get; set; }

        public bool AirlineScored { get; set; }

        public bool FlightScored { get; set; }

        public PassengerSeat() { }

}
```

Listing 4.3.4 Klasa PassengerSeat – predstavlja putnika na letu

Klasa VehicleReservation predstavlja rezervaciju vozila, prikazanu na listingu 4.3.5. Rezervacija vozila je znatno jednostavnije implementirana iz razloga što nema više putnika niti više spojenih rezervacija vozila u jednom vremenskom periodu. Za rezervaciju vozila je potrebno uneti identifikacioni broj vozila, na kojoj lokaciji će biti preuzeto vozilo, kao i gde će biti vraćeno, datum preuzimanja i vraćanja, cenu, korisničko ime osobe koja je kreirala rezervaciju, datum kreiranja rezervacije i da li je ocenjeno vozilo i rent-a-car servis.

```
public class VehicleReservation
    {
        [Key]
        public int ReservationId { get; set; }

        public int VehicleId { get; set; }
        public DateTime FromDate { get; set; }

        public string FromLocation { get; set; }

        public DateTime ToDate { get; set; }

        public string ToLocation { get; set; }

        public double Price { get; set; }

        public string UserName { get; set; }

        public DateTime CreationDate { get; set; }

        public bool IsVehicleRated { get; set; }

        public bool IsRentACarRated { get; set; }
    }
```

Listing 4.3.5 Klasa VehicleReservation – rezervacija vozila

4.4 Problem

Neki od servisa zahtevaju podatke koje se nalaze na drugim servisima, kako oni ne bi bili spregnuti, komunikacija je realizovana preko HTTP zahteva. Iako HTTP zahtevi mogu da se podese da budu asinhroni, ne predstavljaju najbolji način za realizaciju komunikacije među servisima. U okviru narednog poglavlja će biti detaljno razrađen problem i opisan način na koji je prevaziđen.

5. Opis rešenja problema

Za realizaciju razmene poruka odabran je RabbitMQ, u okviru ovog poglavlja će biti opisan i biće priloženi primeri kako je on korišćen u okviru apliakcije.

5.1 RabbitMQ

RabbitMQ je *open source* softver koji ima ulogu posrednika za razmenu poruka (engl. *message broker*), koji je originalno implementirao AMQP (engl. *Advanced Message Queuing Protocol*), a naknadno je proširen da podrži i STOMP (engl. *Streaming Text Oriented Messaging Protocol*) protokol, MQTT (engl. *MQ Telemetry Transport*) protokol i njima slične. Kod je napisan Erlang programskim jezikom i napravljen je na OTP (engl. *Open Telecom Platform*) *framework* za ispade i klastere. Klijentske biblioteke koje omogućavaju korišćenje RabbitMQ-a postoje za sve veće programske jezike. Koristi se kako bi rasteretio komunikaciju između servera u slučaju mikroservisne arhitekture.

Arhitektura RabbitMQ- a je veoma jednostavna, kao što se može videti na slici 5.1. Postoje dve vrste korisnika proizvođači (engl. *producer*) i potrošači (engl. *consumer*). Producer kreira poruke i dostavlja ih *message broker*-u. Consumer-i se pretplate na *broker*-a i čekaju da budu obavešteni da postoji poruka. Poruka ostaje na redu u okviru *broker*-a sve dok je neki *consumer* ne obradi.

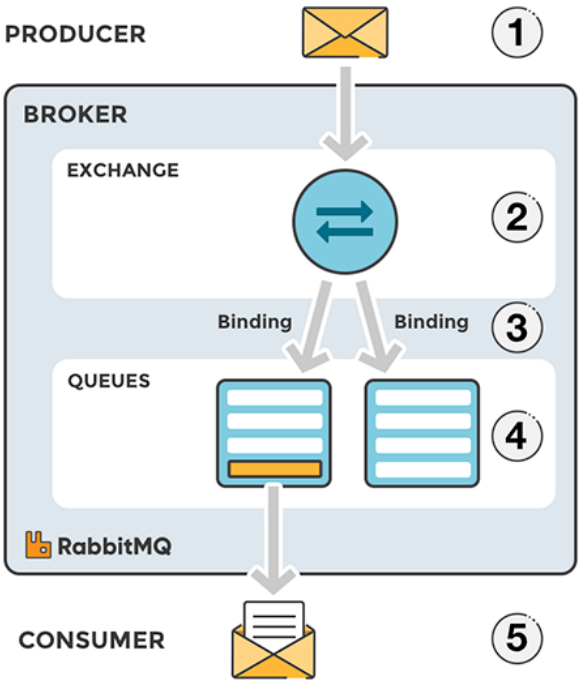
Slika 5.1.1 RabbitMQ

Korišćenje reda za čuvanje poruka omogućava veb serverima da brže odgovore na zahteve klijenata, umesto da izvršavaju zadatke koji zahtevaju dobavljanje velikog broja podataka, za koje bi potencijalno moglo da treba duži vremenski period da se izvrše. Takođe, jedna od dobrih primena reda jeste kada želimo da jednu istu poruku primi veći broj primaoca ili kada želimo da balansiramo količinu posla između više servera.

Poruke se ne objavljuju (engl. *publish*) direktno na redu, nego producer pošalje poruku na razmenu (engl. *exchange*). Razmena je zadužena za rutiranje poruka na različite redove uz pomoć *binding*-a i ključeva rutiranja. *Binding* je veza između reda i razmene.

Tok poruke u okviru RabbitMQ-a izgleda ovako (slika 5.2):

1. Producer objavi poruku razmeni. Kada se kreira razmena neophodno je definisati njen tip.
2. Razmena prihvata poruku i preuzima odgovornost za njeno rutiranje. Rutiranje se vrši uz pomoć atributa poruke, kao što je ključ rutiranja.
3. Binding mora biti kreiran između razmene i reda.
4. Poruka ostaje na redu dok je ne obradi consumer
5. Consumer obrađuje poruku



Slika 5.1.2 Tok poruke

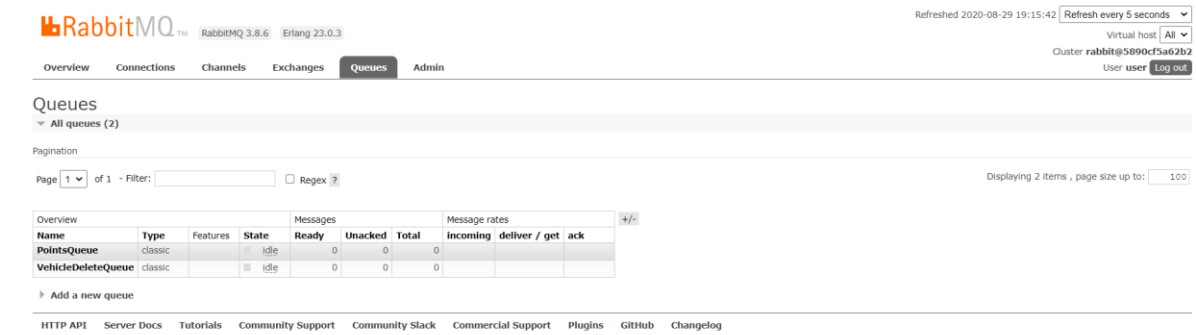
5.2 Implementacija

U ovkiru ovog rešenja iskorišćen je RabbitMQ kako bi se ostvarila komunikacija između mikroservisa na najjednostavniji način. Kako je ovo rešenje prebačeno u kontejnere, kreiran je dodatni kontejner koji koristi sliku RabbitMQ, listing 5.2.1.

```
rabbitmq:
    container_name: rabbitmq
    ports:
        - 5672:5672
        - 15672:15672
    environment:
        - RABBITMQ_DEFAULT_USER=user
        - RABBITMQ_DEFAULT_PASS=password
    image: rabbitmq:3-management
```

Listing 5.2.1 RabbitMQ kontejner

Na ovaj način pomoću porta 15672 može se pristupiti RabbitMQ interfejsu kako bi se pratili redovi i poruke na njima, prikaz se vidi na slici 5.2.1. U slučaju da je potrebno, na ovoj stranici je moguće ručno uklanjati nizove i poruke, kao i ručno dodati poruke u svrhe testiranja. Postoji mogućnost praćenja protoka poruka, broja kanala i konekcija, kao i mnoge druge mogućnosti koje nisu bile potrebne pri realizaciji ovog projekta.



Slika 5.2.2 Prikaz interfejsa RabbitMQ

U narednim poglavljima će biti detaljno objašnjeni slučajevi upotrebe, a sada će biti prikazan sam mehanizam razmene poruka, iz razloga što je on isti u svim slučajevima, jedina razlika jeste sam sadržaj poruke koji se menja.

Kao prvo u konfiguracionom fajlu je dodat segment za RabbitMQ, listing 4.2.2 kako bi izmena podataka, u slučaju da je to neophodno bila jednostavna.

```
"RabbitMq": {
    "Hostname": "rabbitmq",
    "VehicleQueueName": "VehicleDeleteQueue",
    "PointsQueueName": "PointsQueue",
    "UserName": "user",
    "Password": "password"
  }
```

Listing 5.2.2 Konfiguracioni fajl

U delu metode gde je potrebno poslati poruku se poziva metoda koja je implementirana na sledeći način (listing 5.2.3). Prvi korak koji je neophodno izvršiti jeste otvaranje konekcije ka RabbitMQ serveru pomoću ConnectionFactory-a. Pomoću ove konekcije je dalje moguće definisanje reda. Red se defiše pomoću metode QueueDeclare koja kreira red u slučaju da on već ne postoji. Ključni parametri koji se prosleđuju ovoj metodi jesu naziv i da li je izdržljiv (engl. durable). Nakon kreiranja reda, neophodno je podatke pretvoriti u JSON format i kodirati ih. Poslednji korak jeste objaviti poruku na red pomoću metode BasicPublish, čiji su ključni parametri ime reda i poruka.

public void updatePoints(string username, int points)

```
        {
            var factory = new ConnectionFactory() { HostName = _hostname,
UserName = _username, Password = _password };

            using (var connection = factory.CreateConnection())
            using (var channel = connection.CreateModel())
            {
               channel.QueueDeclare(queue:    _pointsQueueName,    durable:
false, exclusive: false, autoDelete: false, arguments: null);

               PointsClass pointsClass = new PointsClass(username, points);
               var json = JsonSerializer.Serialize(pointsClass);
               var body = Encoding.UTF8.GetBytes(json);

               channel.BasicPublish(exchange:        "",        routingKey:
_pointsQueueName, basicProperties: null, body: body);
            }
        }
```

Listing 5.2.3 Slanje poruka

Servisu koji treba da primi poruke je neophodno obezbediti da konstantno osluškuje da li je došla poruka na red. Ovo je omogućeno pomoću nasleđivanja apstraktne klase BackgroundService. Da bi ona zapravo bila pokrenuta u pozadini, klasu je neophodno definisati kao HostedService u startup fajlu (listing 5.2.4) Kada se klasa definiše kao HostedService ona se izvršava kao pozadinska nit. Mogućnost korišćenja osobine klase BackgroundService kao HostedService steknuta je implementacijom interfejsa IHostedService. Pored metoda StartAsync i StopAsync koje definiše interfejs, klasa BackgroundService dodatno definiše ExecuteAsync koja se izvršava u pozadini od trenutka kada se klasa koja nasleđuje BackgroundService podigne kao HostedService.

```
public void ConfigureServices(IServiceCollection services)
        {
            …

            services.AddHostedService<RabbitMQRead>();
        }
```

Listing 5.2.4 Dodavanje HostedService-a u Startup fajl

U okviru konstruktora, pored informacijama o RabbitMQ, iz konfiguracionog fajla je neophodno proslediti i IServiceProvider o kome će biti više reči kasnije u tekstu. Iz konstruktora je neophodno pozvati metodu koja će otvoriti konekciju ka RabbitMQ, kako bi mogli da osluškujemo red (listing 5.2.5). Otvaranje kanala je identično kao na strani pošiljaoca, s tim da je jedina razlika da se objekat kanala smešta u lokalnu promenljivu.

```
public class RabbitMQRead : BackgroundService
    {
        private IModel _channel;
        private IConnection _connection;
        private readonly string _hostname;
        private readonly string _queueName;
        private readonly string _username;
        private readonly string _password;
        private readonly IServiceProvider _service;

        public RabbitMQRead(IServiceProvider service,
IOptions<RabbitMQConfiguration> rabbitMqOptions)
        {
            _service = service;
            _hostname = rabbitMqOptions.Value.Hostname;
            _queueName = rabbitMqOptions.Value.QueueName;
            _username = rabbitMqOptions.Value.UserName;
            _password = rabbitMqOptions.Value.Password;
            InitializeRabbitMqListener();
        }

        private void InitializeRabbitMqListener()
        {
            var factory = new ConnectionFactory
            {
                HostName = _hostname,
                UserName = _username,
                Password = _password
            };

            _connection = factory.CreateConnection();
            _channel = _connection.CreateModel();
            _channel.QueueDeclare(queue: _queueName, durable: false,
exclusive: false, autoDelete: false, arguments: null);
        }
```

Listing 5.2.5 Implementacija BackgroundService-a

Ranije pomenuta metoda ExecuteAsync je implementirana tako (listing 5.2.6) da kada se primi poruka, prvo je neophodno da se dekodira nazad u string, a zatim da se izvrši deserijalizacija objekta.

```
protected override Task ExecuteAsync(CancellationToken stoppingToken)
        {
            stoppingToken.ThrowIfCancellationRequested();

            var consumer = new EventingBasicConsumer(_channel);
            consumer.Received += (ch, ea) =>
```

```
        {
                    var content =
    Encoding.UTF8.GetString(ea.Body.ToArray());
                    var pointsClass =
    JsonConvert.DeserializeObject<PointsClass>(content);

                    UpdatePoints(pointsClass.Username,
    pointsClass.Points).ConfigureAwait(false);

                _channel.BasicAck(ea.DeliveryTag, false);
        };

        _channel.BasicConsume(_queueName, false, consumer);

        return Task.CompletedTask;
      }
    }
```

Listing 2.5.6 Execute Async

Nakon toga se poziva metoda koja treba da izvrši neophodnu akciju sa podacima koji su pristigli. Svaka od metoda ima istu osnovu (listing 5.2.7), prvi korak je kreiranje IServiceScope opsega (engl. scope) pomoću IServiceProvider-a koji je prosleđen u konstruktoru. Zatim se novokreirani opseg koristi kako bi bilo moguće instancirati kontekste baze podataka. Ovi koraci su neophodni kako bi opseg mogao na kraju korišćenja da uništi objekat koji predstavlja kontekst baze podataka.

```
public async Task UpdatePoints(string id, int points)
    {
      using (IServiceScope scope = _service.CreateScope())
      {
        var _userManager =
scope.ServiceProvider.GetRequiredService<UserManager<AppUser>>();
        …
      }
    }
```

Listing 5.2.7 Metoda pozvana od strane ExecuteAsync-a

Ovime je opisan ceo postupak slanja i primanja poruka pomoću RabbitMQ-a, u naredom poglavlju će biti opisani pojedinačni sluačejvi korišćenja i zašto je odabrano da se tu vrši razmena poruka pomoću RabbitMQ-a, a ne na neki drugi način.

5.3 Slučajevi upotrebe

U okviru svih slučajeva upotrebe poruke se uvek šalju sa AirlineMicroservice-a.

5.3.1 Otkazivanje rezervacije vozila

Prvi slučaj korišćenja jeste pri otkazivanju rezervacije. Postoji više načina da dodje do otkazivanja rezervarcije leta: direktnim otkazivanjem, istekom pozivnica i odbijanjem pozivnica. U slučaju da se desila neka od ovih situacija obavezno je otkazati i rezervaciju vozila, ako je ona vezana za rezervaciju leta. Dok je rešenje bilo realizovano kao monolitna arhitekuta, nije postojala poteškoća pristupanju tabela vezanih za rent-a-car usluge, ali podelom na mikroservise, kao što je opisano u trećem poglavlju, ovakav način otkazivanja više nije moguć.

Nakon prelaska na mikroservisnu arhitekturu odabrano je da se ova komunikacija realizuje pomoću HTTP zahteva, na isti način kao komunikacija između klijenta i servera. Nedostatak ovakvog pristupa se ogleda u tome što u slučaju da je trenutno iz nekih razloga RentACarMicroservice nedostupan ili je iznenada došlo do prekida, poruka bi bila neuspešna i klijentima bi bila vraćena poruka da je otkazivanje neuspešno ili da ga nije moguće izvršiti.

Ovaj nedostatak se prevazilazi prelaskom na RabbitMQ jer on čuva poruke dok god se one ne obrade. Način rada da se poruke čuvaju omogućava neometan rada servisa nakon slanja poruke, jer ne postoji način da zna kada je poruka obrađena, jedini način jeste, da se servis koji je poslao poruku pretplati na neki drugi red i da čeka povratnu informaciju o uspešnosti zadatka. U slučaju koji se trenutno posmatra nije neophodno klijentu vratiti informaciju kada je rezervacija uklonjena. U listingu 5.3.1.1 će biti prikazana metoda na prijemnoj strani koja obrađuje informacije koje su stigle.

```
public async Task DeleteReservation(int id)
    {
      using (IServiceScope scope = _service.CreateScope())
      {
        var context = scope.ServiceProvider.GetRequiredService<DatabaseContext>();

        var vehicleReservation = await context.VehicleReservation.FindAsync(id);
        if (vehicleReservation == null)
        {
```

```
            return;
        }

        context.VehicleReservation.Remove(vehicleReservation);
        await context.SaveChangesAsync();

        var vehicle = await context.Vehicles.Include(vehicle =>
vehicle.UnavailableDates).FirstOrDefaultAsync(vehicle => vehicle.VehicleId ==
vehicleReservation.VehicleId);

        vehicle.UnavailableDates.ToList().ForEach(
            unavailableDate =>
            {
                if (unavailableDate.Date.Date >= vehicleReservation.FromDate.Date &&
unavailableDate.Date.Date <= vehicleReservation.ToDate.Date)
                {
                    context.Remove(unavailableDate);
                }
            }
        );

        await context.SaveChangesAsync();
    }
}
```

Listing 5.3.1.1 Otkazivanje rezervacije vozila

5.3.2 Uvećanje broja poena

Drugi slučaj upotrebe se dešava pri svakom kreiranju rezervacije nekog leta. Kako je samom specifikacijom definisano, pri kreiranju rezervacije korisniku se dodeljuju poeni koje on može kasnije da iskoristi za popuste. Da bi ova aktivnost mogla da se izvrši neophodan je pristup korisnicima kako bi im se uvećali poeni, ali isto tako i umanjili u slučaju da su ih iskoristili pri rezervisanju. Kada je arhitektura sistema bila realizovana kao monolitna, problem izmene poena nije predstavljao problem, ali nakon prelaska na mikroservisnu arhitekturu, direktan pristup tabeli više nije moguć.

Prelaskom na mikroservise i komunikacija je prebačena na HTTP zahteve. Isti se problemi javljaju kao i u prethodnom slučaju tako da neće biti ponovo opisivani.

Prednosti RabbitMQ koja je ovde došla do izražaja jeste mogućnost potpune nezavisnosti dva procesa. Povećanje ili umanjenje broja poena korisnika ne utiče na dalji postupak rezervacije, samim tim asinhron način komunikacije bez očekivanog odgovora jeste najefikasniji način rada. Postupak obrade informacija za ovu aktivnost se može videti u listingu 5.3.2.1.

```
public async Task UpdatePoints(string username, int points)
        {
            using (IServiceScope scope = _service.CreateScope())
            {
                var _userManager =
scope.ServiceProvider.GetRequiredService<UserManager<AppUser>>();

                var user = await _userManager.FindByNameAsync(username);
                user.NumberOfPoint += points;

                await _userManager.UpdateAsync(user);
            }
        }
```

Listing 5.3.2.1 Postupak promene poena

5.3.3 Slanje *email*-ova korisnicima

Poslednji slučaj korišćenja se javlja u trenutku kada je potrebno obavestiti prijatelje koji su pozvani na putovanje da imaju pozivnicu koju mogu da prihvate ili odbiju. Problem se javlja u trenutku kada je u odnosu na korisničko ime potrebno pronaći *email* na koji se šalje pozivnica. Kao i u prethodnim slučajevima javljaju se isti problemi i koristi se isto inicijalno rešenje.

Dobre osobine RabbitMQ u ovom trenutku dolaze dodatno do izražaja iz razloga što u slučaju da je pozvan veliki broj prijatelja i da takvih rezervacija u istom trenutku UserMicroservice bi bio preopterećen pristiglim HTTP zahtevima i došlo bi do otkaza u nekom trenutku. U slučaju upotrebe RabbitMQ-a servis je u mogućnosti da obrađuje poruke svojim tempom i na ovaj način su smanjene mogućnosti da dođe do otkaza. Ako se u nekom trenutku nakupi veliki broj poruka u redu moguće je podići nove instance UserMicroservice-a. Implementacija ove metode se nalazi na listingu 5.3.3.1.

```
public async Task SendEmailInvites(string username, FlightReservation
flightReservation)
        {
            using (IServiceScope scope = _service.CreateScope())
            {
                var _userManager =
scope.ServiceProvider.GetRequiredService<UserManager<AppUser>>();

                var user = await _userManager.FindByNameAsync(username);
                EmailService.SendInvite(user, flightReservation);
            }
        }
```

Listing 5.3.3.1 Postupak slanja pozivnica prijateljima

6. Zaključak

Projekat treba da predstavlja aplikaciju za rezervisanje avionskih karata i vozila iz *rent-a-car* servisa. Ova aplikacija je u potpunosti odgovorila na zahteve koji su bili postavljeni, kako je ovaj projekat za fakultet, nivo kompleksnosti je bio definisan u skladu sa znanjem i iskustvom studenata završne godine. Aplikacija je realizovana na način da se optimizuje proces dobavljanja resursa sa zadnje strane, vodi računa o konkurentnom pristupu, sigurnosti i bezbednosti, itd.

Pored svega toga, ako bismo želeli da ovu aplikaciju publikujemo, morali bismo da unesemo neke izmene u sistem i da ga optimizujemo za rad sa više klijenata. Prvenstveno bi bilo neophodno voditi više računa o privatnosti podataka klijenata i o zaštiti svih podataka od neželjenih napada. Bilo bi neophodno ponovo definisati modele kako bi bili optimizovaniji i pogodniji za čuvanje, izmenu i slanje na prednju stranu. Pored toga potrebno je optimizovati *docker compose* sa *load balancer*-om kako bi mogao da podiže više instanci ili da gasi instance po potrebi. Na ovaj način bi klijenti mogli da neometano koriste web aplikaciju.

Ovo su izmene koje bi bile neophodne kako bi ova aplikacija mogla da se koristi od strane više klijenata istovremeno. Ako bismo želeli da ova aplikacija bude stvarno upotrebljiva i da može da parira drugim rešenjima na tržištu, neophodno bi bilo dodati mogućnost *online* plaćanja.

Izmenom načina komunikacije između mikroservisa napravljen je prvi veliki korak u pravcu veće i lakše skalabilnosti rešenja. Pored toga olakšana je proširivost samog rešenja, npr. u slučaju da su novim servisima potrebni isti podaci kao nekom od postojećih, dovoljno je podesiti da poruka ima više od jednog primaoca, što RabbitMQ može da omogući.

U slučaju da bi se ova aplikacija pustila na tržište bilo bi neophodno njeno redovno održavanje i vremenom bi se dodavale nove funkcionalni na zahteve klijenata. Aplikacija je razvijana u savremenim tehnologijama koje se i same trenutno razvijaju i proširuju što je čini pogodnom za dalji razvoj.