Razvoj mobilne messenger aplikacije

Trbojević, Milan

Master's thesis / Diplomski rad

2020

Degree Grantor / Ustanova koja je dodijelila akademski / stručni stupanj: **University of Rijeka / Sveučilište u Rijeci**

Permanent link / Trajna poveznica: https://urn.nsk.hr/urn:nbn:hr:195:047719

Rights / Prava: In copyright / Zaštićeno autorskim pravom.

Download date / Datum preuzimanja: **2024-03-29**

Sveučilište u Rijeci
Fakultet informatike
i digitalnih tehnologija

Repository / Repozitorij:

Repository of the University of Rijeka, Faculty of Informatics and Digital Technologies - INFORI Repository

zir.nsk.hr

DIGITALNA KNJIŽNICA

Image not found or type unknown

Sveučilište u Rijeci - Odjel za informatiku

Diplomski studij informatike

Informacijsko-komunikacijski sustavi

Milan Trbojević

Razvoj mobilne *messenger* aplikacije

Diplomski rad

Mentorica: izv. prof. dr. sc. Marina Ivašić-Kos

Rijeka, rujan 2020.

Sadržaj

1. Uvod 4
2. Arhitektura *messenger* aplikacije 5
2.1 Tijek procesa i tok podataka 5
2.2 Arhitektura klijentske strane aplikacije 6
2.3 Arhitektura serverske strane aplikacije 7
2.4 Verzioniranje aplikacije 9
3. Slanje obavijesti klijentu 10
3.1 Uvod 10
3.2 Analiza tijeka podataka i procesa 10
4. Analiza *messenger* aplikacije 11
4.1 Najbitnije stavke analize 11
4.2 Autorizacija korisnika 11
4.3 Upravljanje razgovorima 15
4.4 Stvaranje novog razgovora 16
4.5 Arhiviranje razgovora 19
4.6 Komunikacija između korisnika (slanje poruka) 19
4.7 Uređivanje korisničkog profila 23
4.8 Obavijesti (notifikacije) 24
5. Postavljanje aplikacije u produkcijsko okruženje 26
5.1 Postavljanje aplikacije na *Azure Cloud* 26
6. Prijedlog nadogradnje messenger aplikacije 27
6.1 *End-End* Enkripcija 27
6.2 Dijeljenje lokacije 28
6.3 Dijeljenje medijskog sadržaja 28
6.4 Lista bliskih korisnika 28
6.5 Pretraga poruka 28
6.6 Uređivanje/prilagodba izgleda razgovora 28
6.7 Višejezičnost 29
6.8 Pregled aktivnosti trenutno prijavljenih korisnika 29
6.9 Dvostruka autentifikacija i prijava putem *Google ili Facebook računa* 29
8 . Zaključak 30
9. Literatura 31

9.1 Popis slika 33
10. Prilozi 34
10.1 Smjernice prema repozitoriju koda 34
10.2 Instalacija i konfiguriranje radnog okruženja 34
10.2.1 Postavljanje *.Net Core Web API* radnog okruženja 34
10.2.2 Postavljanje *Ionic* aplikacije 35
10.2.3 Instalacija *SignalR* biblioteke 37
10.2.4 Konfiguriranje Rest API servisa na *Azure Cloudu* 38

1. Uvod

U ovom diplomskom radu glavni zadatak je razviti stabilnu mobilnu aplikaciju, koja će omogućiti razmjenjivanje tekstualnog i medijskog sadržaja između korisnika u realnom vremenu (popularnog naziva *Messenger*). Za razvijanje aplikacije koristiti će se *.NET Core[1]* za serverski dio aplikacije, koji će imati ulogu *web* servisa za manipuliranje i proslijeđivanje podataka između korisnika. Za klijentski dio aplikacije koristiti će se *Angular[3] Javascript* radnim okruženjem *s Ionic* bibliotekom koja je prilagođena za izradu mobilnih *cross-platform* aplikacija. Cilj diplomskog rada je prvenstveno primjena ovih tehnologija i njihovo zajedničko povezivanje u izradi aplikacije koju većina nas svakodnevno koristi.

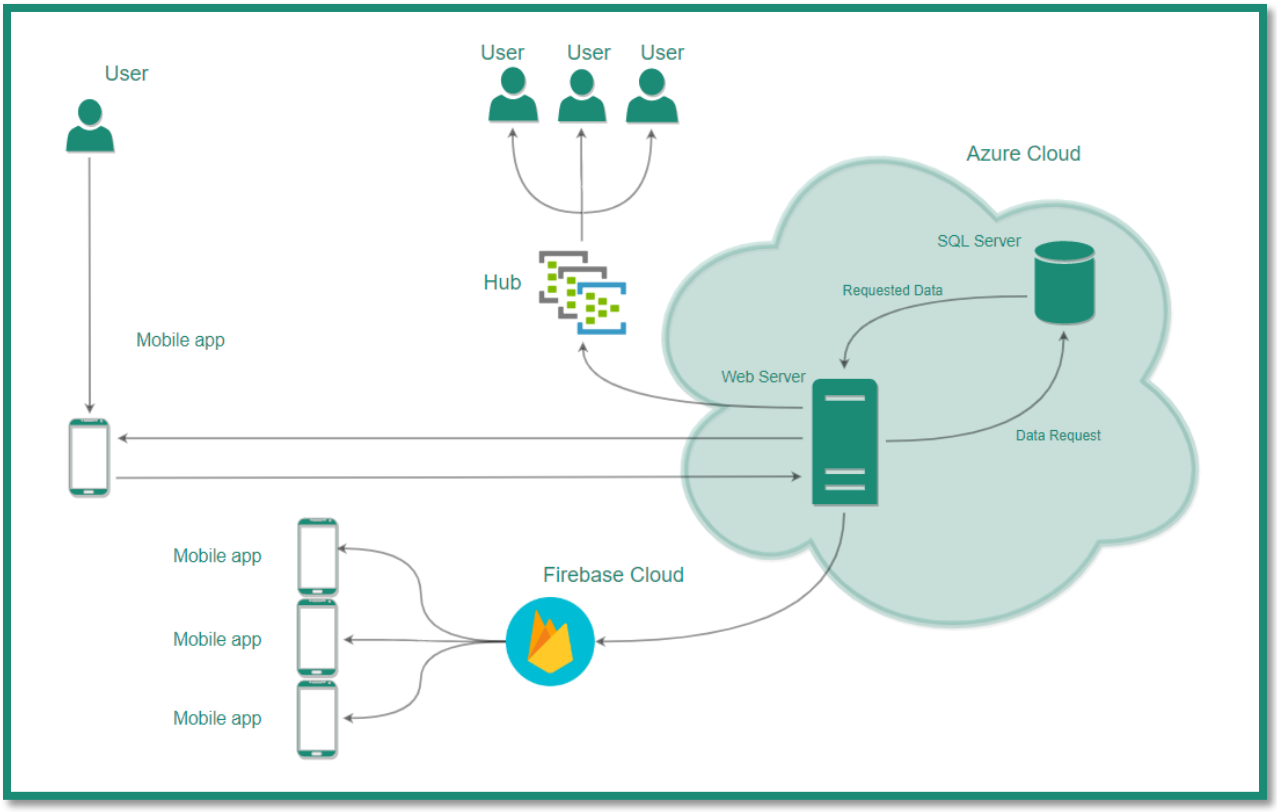
Što se tiče ostalih tehnologija koje će se koristiti trenutno je prije samog istraživanja zbog svoje popularnosti *Google-ov Firebase Cloud Messaging[2]* jedna od opcija, ali će se ta opcija još istražiti te će konačni odabir biti obrazložen u diplomskom radu.

Osim korištenog pristupa postoje i popularne, često korištene alternative kao što su na klijentskoj strani *React[4]* i *Vue* radna okruženja [32] te *Ruby on Rails* i neka *PHP* radna okruženja poput *Laravela* ili *Symphony-a* na serverskoj strani. Samim odabirom druge tehnologije vjerojatno bi se i sama arhitektura razlikovala od spomenute. Također pored različitog odabira tehnologije moguće je odabrati i različite pomoćne servise koji omogućavaju isti tip *real-time* komunikacije koji je kroz ovu aplikaciju praktički ručno isprogramiran i posložen. Pomoćni servisi koji bi to omogućavali imaju razvijene vlastite *Rest API* servise koji upravljaju zahtjevima i obavijestima sa klijentske strane aplikacije.

Primjeri koda iz aplikacije biti će u *Angular-u* (klijentski dio aplikacije*)* dok će serverski dio aplikacije biti u *.Net Core-u.*

2. Arhitektura *messenger* aplikacije

2.1 Tijek procesa i tok podataka



Slika 1 - Arhitektura messenger aplikacije

Na slici *(Slika 1)* prikazan je model/dijagram messenger aplikacije koja će se razvijati. Na dijagramu možemo primjetiti dijelove arhitekture gdje se radi o korisnicima sustava, dijelovima sustava messenger aplikacije i vanjskim sustavima. Korisnici sustava messenger aplikacije pristupaju iniciranjem zahtjeva putem mobilne aplikacije. Glavni dio messenger sustava je web server na kojem se nalazi Rest API[6] servis koji prihvaća klijentske zahtjeve i po potrebi šalje zahtjeve za podatcima prema SQL serveru[7], gdje se nalazi baza podataka. Baza podataka vraća dohvaćene podatke prema web serveru.

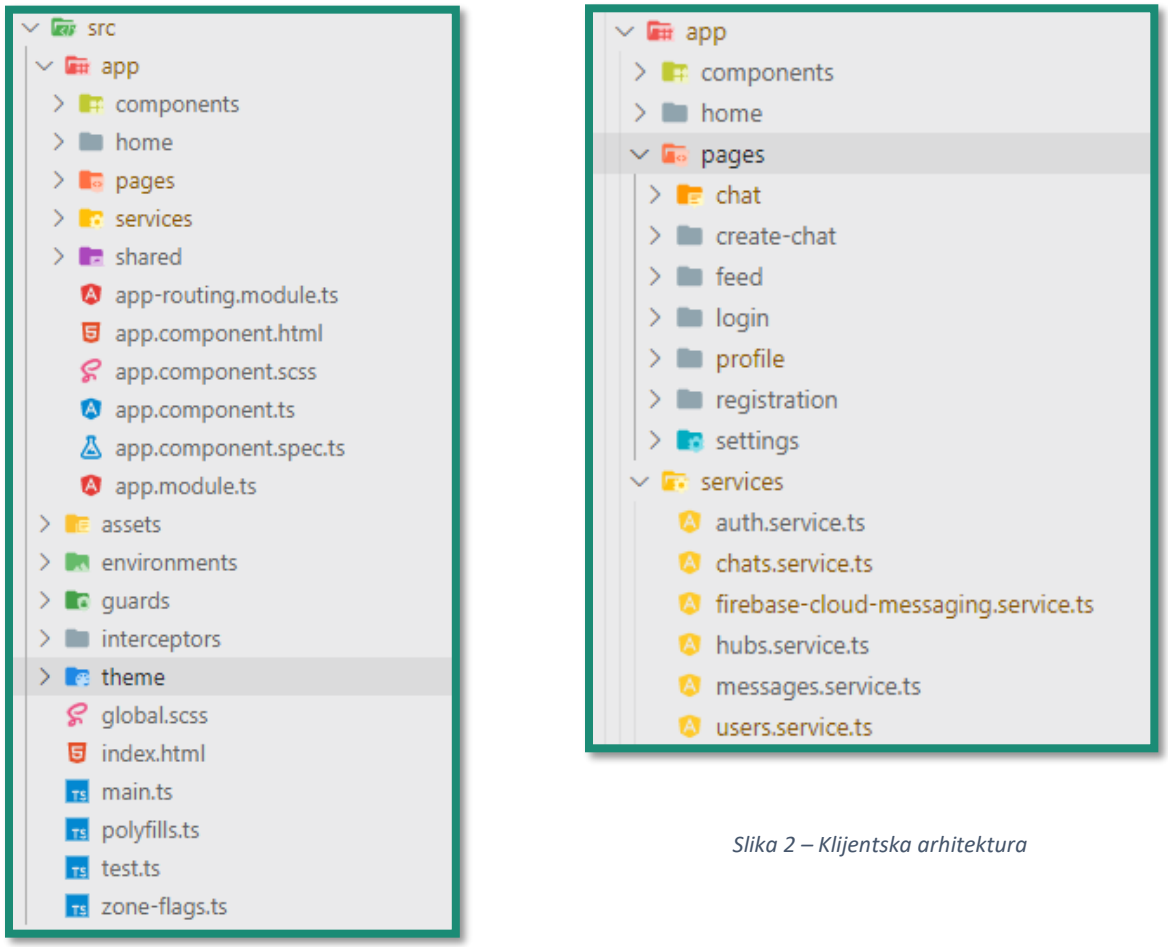
Kada je web server primio podatke od baze podataka on ih proslijeđuje korisniku sustava u mobilnu aplikaciju, odnosno klijentski dio messenger aplikacije. U isto vrijeme svoju ulogu dobivaju vanjski sustavi, koji se nakon spremanja poruke u bazu podataka aktiviraju u sustavu. Postoje dva vanjska sustava: sustav koji je zadužen za slanje poruka kao i sustav za obavijesti na korisnikovom mobilnom uređaju. Sustav za slanje poruka (channel[8] ili hub[8]) je sustav na koji je korisnik sustava automatski registriran pomoću connectionId vrijednosti koja označava vrijednost od niza slova i brojki koja zamjenjuje korisnika i služi kako bi se identificiralo korisnika koji šalje poruku. Ako su korisnici uspješno registrirani na hubu oni će prilikom slanja poruke u realnom vremenu vidjeti novu poruku u razgovoru u kojem se nalaze.

Osim registriranja korisnika na channelu ili hubu, potrebno je registrirati i sam uređaj sa kojeg se šalju poruke na Firebase Cloud (vanjski servis)[9]. Proslijeđivanje poruka odnosno obavijesti o pristignutoj poruci također se šalje posebnim kanalima na koje su predhodno registrirani
korisnici koji se nalaze u tim razgovorima.

Bitno je još napomenuti kako se iz priloženog može zaključiti kako komunikacija između mobilne messenger aplikacije i Cloud servera nije nužno dvosmjerna. Korisnik iako nije poslao zahtjev prema nekom od servera može dobiti obavijest ili podatke.

2.2 Arhitektura klijentske strane aplikacije

Angular[4] je radno okruženje koje se koristi za izradu *SPA* ili *single-page* aplikacija koristeći *HTML* i *Typescript*. *Angular* je napisan u *Typescriptu*. Pomoću *Typescripta* implementira jezgrene i dodatne funkcionalnosti kao skup *Typescript* biblioteka koje uključuju, odnosno radi se njihov *import* unutar aplikacije. Arhitektura se zasniva na posebnim *Angular* modulima odnosno *NgModules* omogućuju opsrkbljivanje *(provider)* potrebno za kompajliranje dodataka za *Angular* komponente.



Slika 2 – Klijentska arhitektura

Klijentska strana u ovoj *messenger* aplikaciji najsličnija je arhitekturi spomenute *Angular* aplikacije, ako ne računamo direktorije u kojima se nalaze izvršne datoteke, biblioteke i dodatni moduli. Jezgra aplikacije je kao i uvijek *src* direktorij.

Unutar njega se nalazi *app* folder *(Slika 2)* koji sadržava:

- Komponente *(components)* – dijelovi koda koji su višenamjenski i ne predstavljaju jedinstvene stranice već su ukomponirani u drugim stranicama.
- Stranice *(pages)* – jedinstvene stranice gdje svaka od njih posjeduje vlastitu rutu odnosno adresu na kojoj se nalazi.
- Servisi *(services)* – klase koje omogućuju komunikaciju prema serverskoj strani aplikacije.
- Pomoćne klase *(shared)* – u ovom se direktoriju nalaze pomoćne klase koje služe za mapiranje podataka koji su stigli sa serverske strane aplikacije.

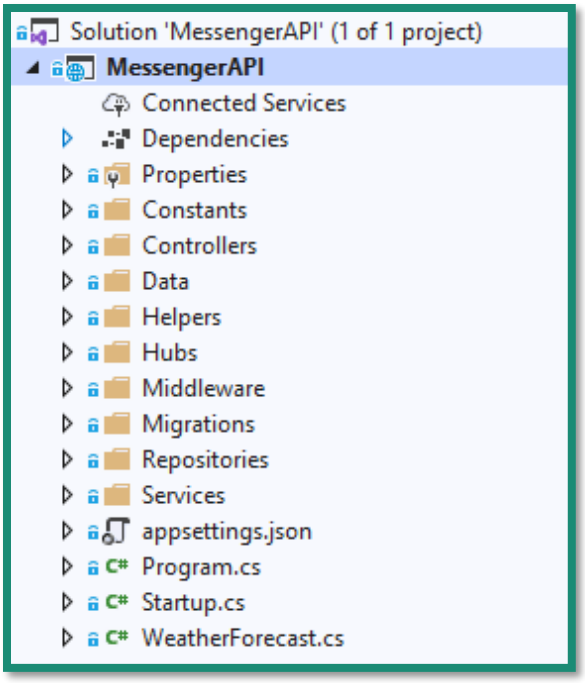
Osim ovih direktorija tu se nalazi i glavna *app* komponenta sa svojim modulima i *routingom* gdje se mogu mjenjati ili dodavati *routing* postavke te dodavati novi moduli i biblioteke koji će se korisiti na razini cijele aplikacije.

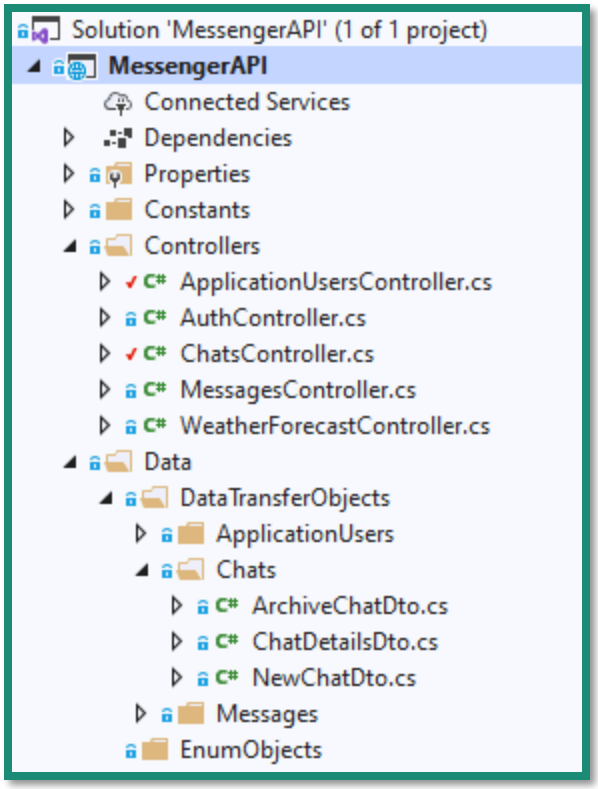
Assets direktorij sadrži slike i ikone aplikacije. Unutar *enviroments* direktorija nalaze se produkcijske i testne postavke aplikacije. *guards[11]* direktorij sadrži *auth.guard.ts* datoteku koja se brine za zaštitu ruta unutar aplikacije te autentificira svaki zahtjev prema njima. *Interceptors [12]* je direktorij koji također sadrži samo jednu datoteku koja automatizira slanje *tokena* za autentifikaciju prema serverskoj strani aplikacije pri svakom zahtjevu koji zahtjeva autentifikaciju. Od ostalih dijelova arhitekture bitno je istaknuti *package.json, google-service.json i config.xml* datoteke koje imaju konfiguracijsku svrhu.

2.3 Arhitektura serverske strane aplikacije

Serverska strana aplikacije posjeduje također jednu uobičajenu *Rest API* arhitekturu. Datoteke koje su izdvojene iz direktorija kao i kod klijentske strane aplikacije su konfiguracijske, ovdje se radi o *Startup.cs, Program.cs i app.setttings.json* datotekama.

Redoslijedom od gore prema dolje *(Slika 3)* imamo *Properties* direktorij koji sadrži postavke za *publish* odnosno sve potrebno kako bi naša aplikacija bez poteškoća bila objavljena na *Cloud* serveru aplikacije. *Constants* direktorij je uobičajen u njemu se nalaze klase koje sadržavaju konstantne tj. hardkodirane vrijednosti. U *Controllers[13]* direktoriju imamo sve kontrolere odnosno najbitnije datoteke koje su zbog *.Net Core* radnog okruženja i *MVC* arhitekture ujedno i rute na koje se šalju zahtjevi sa klijentske strane aplikacije. Sudeći po tome da se ovdje radi o razvijanju *Rest API* servisa, slijedeći standard i praksu, svi kontroleri kao rezultat obrade zahtjeva vraćaju korisniku podatke u JSON formatu.





Slika 3 – Serverska arhitektura

Data direktorij odnosi se na klase koje su putem *Entity* radnog okruženja *[14]* povezane sa bazom podataka. Unutar tog direktorija nalazi se *DataTransferObjects* direktorij, u njemu se nalaze pomoćne klase za manipulaciju podataka. *EnumObjects* direktorij sadrži enumeratore, ili vrijednosti koje su slične konstantnim, ali se zapisuju u bazu podataka u numeričkom obliku. *Models* je glavni poddirektorij u njemu se nalaze sve klase koje ujedno preslikavaju izgled tablica u bazi podataka.

Izmjenom klase u *Models* direktoriju potrebno je napraviti migraciju prema bazi podataka i tada će doći do promjene i na samoj bazi podataka. Osim poddirektorija tu se nalazi i *Model.cs* klasa čije stanje uvijek mora odgovarati stanju baze podataka. Nadalje u toj datoteci možemo definirati neke dodatne postavke vezane uz primjerice: nazivanje tablica, primarnih i vanjskih ključeva, atributa ili postavljanje relacija.

Helpers direktorij sadržava pomoćne klase za lakše i brže mapiranje podataka koji dođu u zahtjevu sa klijentske strane kao i pristup konfiguracijskim postavkama.

U *Hubs* direktoriju nalaze se *hub* kontroleri koji su odgovorni za slanje podataka prema *hubu* aplikacije koji dalje prosljeđuje poruke prema korisnicima prijavljenim na *hub* ili *channel* tj. kanal za zaprimanje poruka i obavijesti.

Migrations direktorij je iznimka jer se u njega datoteke dodavaju generički putem *Entity* radnog okruženja. Svaka od tih datoteka zabilježava „migraciju“ *[15]* odnosno promjene na modelu podataka poslane prema bazi podataka. U ovom direktoriju u svakom trenutku možemo vidjeti cijelu povijest promjena i po potrebi napraviti *revert* migracije odnosno povratak na neko od predhodnih stanja modela podataka.

Repositories je direktorij koji sadržava osnovne funkcije/metode koje su zadužene za slanje podataka prema bazi podatka. Uglavnom su to operacije pisanja, brisanja, ažuriranja ili čitanja podataka. U arhitekturi je zamišljeno da svaka od klasa unutar *Models* direktorija, kojoj pripada odgovarajuća tablica, ima svoju *Repository* datoteku.

Services direktorij sadrži *manager* datoteke koje su zadužene za upravljanje podacima, dohvaćanje podataka od strane odgovarajuće *repository* datoteke i proslijeđivanje odgovora *(response-a)* prema kontroleru.

Ideja razdvajanja *manager, repository i kontroler* datoteka je u *Single Responsibility* načelu objektnog programiranja*[16]* gdje je svaki dio zadužen samo za jednu funkcionalnost i nije ovisan o više dijelova aplikacije.

2.4 Verzioniranje aplikacije

U softverskom inženjerstvu sustav za verzioniranje[33] predstavlja skup sistemskih klasa odgovornih za upravljanje promjenama nad napisanim kodom, dokumentima, ili bilo kojoj vrsti informacije. Promjene su obično identificirane pomoću nekog oblika šifre, datuma i naravno korisničko imena korisnika koji je utjecao na tu promjenu. Oduvijek postoji potreba za boljim i organiziranijim načinom pisanja i upravljanjem starijih i novijih verzija. No, pravu popularnost sustavi za verzioniranje su stekli kada je počela era kompjuterizacije. U današnje vrijeme najčešće su sustavi za verzioniranje u obliku web platforme (*cloud* platforme*)* sa kojom možete putem komande linije komunicirati sa svog računala, koristeći primjerice *SSH* protokol ili sličan pristup.

Tijekom razvijanja aplikacije koristi se *git[29]* sustav za verzioniranje aplikacija. Kako se radi o dva odvojena sustava koja međusobno komuniciraju na *GitHub[28]* profilu postavljena je mobilna aplikacija i *Rest API* servis koji sa njom komunicira. Kada se radi o nešto kompleksnijoj arhitekturi aplikacije, kao što je ova *messenger* aplikacija koju smo razvijali, korisno je služiti se nekom vrstom *version-control[30]* sustava jer tada u bilo kojem trenutku sa sigurnošću znamo koje smo izmjene napravili i kada.

Najbolja praksa bi bila da nakon svake isprogramirane značajke napravimo *push[31]* novog koda prema *Github* repozitoriju.

3. Slanje obavijesti klijentu

3.1 Uvod

Klijent-server model komunikacije je distribuirana aplikacijska struktura koja proslijeđuje zahtjeve između opskrbljivača koji sadrže neki oblik resursa, popularnog naziva *serveri* i inicijatora tih zahtjeva, popularnog naziva *klijenti*. Komunikacija se najčešće odvija putem internetske mreže na odvojenom *hardveru*, no ponekad oni mogu pripadati i istoj *hardverskoj* arhitekturi.

Ukoliko klijentska aplikacija ne radi na *realtime* principu tada ona funkcionira na način da klijent pošalje zahtjev prema serveru i server odgovara u određenom obliku. Ako je aplikacija *real-time* to znači da klijent može dobiti obavijest ili podatke od servera neovisno je li poslao zahtjev za tim podatcima ili ne. Bitno je naravno da je korisnik prethodno „pretplaćen“ na te obavijesti ili poruke, jer u praksi, podatci su puno češće proslijeđeni samo određenim tj. „pretplaćenim“ korisnicima a ne svim korisnicima aplikacije. Izuzezak je primjerice *live-chat* u kojem sudjeluje mnogo korisnika, ili svi prijavljeni korisnici u sustav bez neke prethodne autentifikacije ili neke vrsta posebne „pretplate“.

Za ostavrivanje asinkronog komunikacije imeđu servera i klijenata, odnosno slanja obavijesti klijentima koristi se *SignalR* biblioteka. *SignalR* je i klijentska i serverska biblioteka otvorenog koda koja omogućava kao što smo rekli da server šalje asinkrone obavijesti prema klijentu. Unutar klijentske biblioteke nalaze se *Javascript* komponente koje omogućuju lakšu komunikaciju i postavljanje pretplate na serverskoj strani aplikacije.

3.2 Analiza tijeka podataka i procesa

SignalR Hubs API servis omogućava pozivanje metoda prema spojenim klijentima sa servera. U serverskom kodu, mogu se definirati metode koje poziva klijent. Dok su na klijentskoj strani definirane metode koje poziva server. Sve je to omogućeno pozivom imena *eventa,* za ostalo se brine *SignalR* biblioteka i omogućuje da komunikacija između klijenta i servera u pravom smislu bude *real-time.*

Prilikom spajanja klijenta na *hub* on dobiva *connectionId* koji je zapravo vrsta *tokena* preko kojeg se klijent/korisnik može identificirati na *hubu* te se on nalazi na serverskoj strani aplikacije. *ConnectionId* nam je najkorisniji sa klijentske strane aplikacije, i za to se pobrinuo *SignalR* jer ima ugrađenu funkciju za dohvaćanje *connectionId-a* korisnika, tako da jednostavnim zahtjevom možemo u odgovoru dobiti *connectionId*. Da ne posjedujemo *connectionId* bilo bi nam vrlo teško znati primjerice, kada je korisnik odjavljen sa *huba* odnosno kada je napustio razgovor u našoj *messenger* aplikaciji.

4. Analiza *messenger* aplikacije

Messenger aplikacija je oblik platforme koji omogućuje instantnu izmjenu poruka između korisnika. Najčešće se koristi putem mobilnih uređaja odnosno mobilnih aplikacija. Glavna prednost je što je komunikacija omogućena samim pristupom internetu, vrlo je brza i ne ovisi o vrsti uređaja na kojem se koristi. *Messenger* aplikacije su jedne od najbrže rastućih oblika aplikacija u današnje vrijeme [34].

Serverski dio aplikacije zamišljen je kao *Rest API* servis u *.Net Core* tehnologiji koji će prema klijentu aplikacije proslijeđivati podatke u *JSON [19]* obliku. Razlog odabira je predhodno iskustvo u *C#* programskom jeziku kao i u izradi *ASP .Net web* aplikacija. Jedna od glavnih prednosti nove *.Net Core* tehnologije je mogućnost postavljanja aplikacije na *Linux* i *Windows* server.

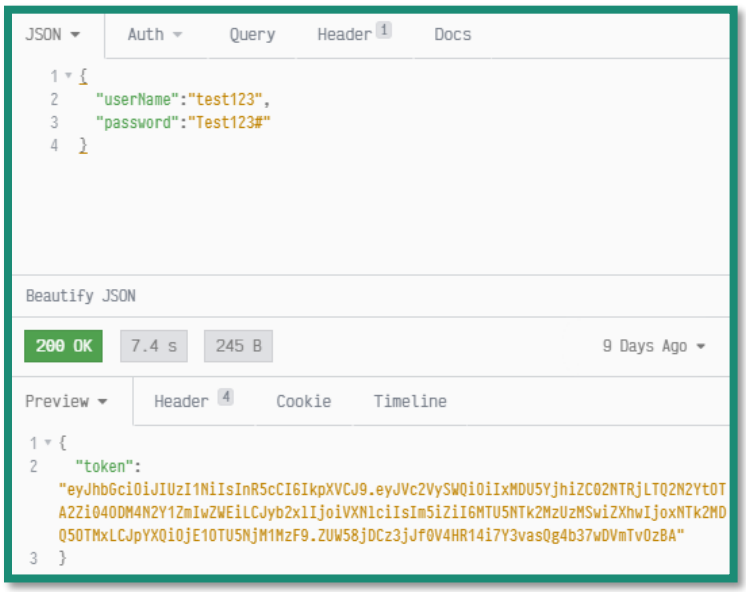
Klijentski dio aplikacije je *web* aplikacija u *Javascriptu,* preciznije, *Angular* radnom okruženju. Radno okruženje dodatno koristi *Ionic[20]* biblioteku. Glavni razlog je mogućnost korištenja komponenti i funckionalnosti za mobilne aplikacije. Čime zapravo dobivamo mobilnu *cross-platform* aplikaciju koju razvijamo kao i svaku drugu *web* aplikaciju.

4.1 Najbitnije stavke analize

1. Autorizacija korisnika – analizirati će se sve najbitnije stavke vezane uz autoriziranje korisnika i sigurnost. Što uključuje registraciju, prijavu i autoriziranje korisnika tijekom korištenja aplikacije.
2. Upravljanje razgovorima – analizirati će se upravljanje razgovorima koji se sastoje od poslanih poruka između dvoje ili više korisnika. Analiza će uključivati stvaranje novog razgovora, priključivanje u postojeći razgovor, upravljanje korisnicima na temelju admin-korisnik uloga te uređivanje razgovora.
3. Poruke – analizirati će se cijeli tijek slanja poruke i primanja poruke između korisnika istog razgovora, kao i prikaz poslanih poruka u razgovoru i njihova uloga.
4. Korisnički profil – analizirati će se upravljanje korisničkim profilom te izmjena postavki *messenger* aplikacije.
5. Dodatne funkcionalnosti – analiza dodatnih funkcionalnosti koje su poveznica između navedenih točaka koje su analizirane.

4.2 Autorizacija korisnika

Kako bi komunikacija između serverske i klijentske strane naše *messenger* aplikacije bila sigurna bitno je da osiguramo neku vrstu autentifikacije, gdje će klijent prilikom slanja zahtjeva biti provjeren od strane *Rest API* servisa. Autentifikacija se vrši koristeći *JWT (JSON Web Token) [21]* koji je dodijeljen korisniku prilikom uspješne autentifikacije putem *Rest API* servisa. Kako bi se klijent autentificirao potrebno je da na adresu *https:44306/auth/login* pošalje pristupne podatke koji se sastoje od korisničkog imena i lozinke *(Slika 4).*

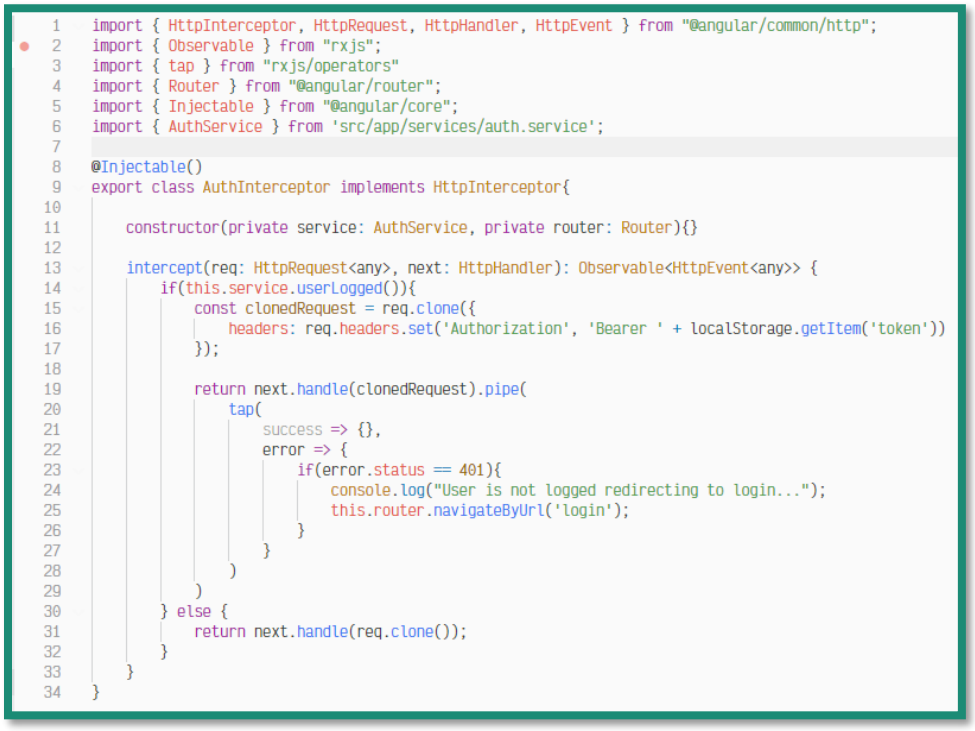


Slika 4 - Slanje zahtjeva za autorizaciju

Klijent u odgovoru ili *response-u* dobiva *token*. Nadalje, generirani *token* služi za autentifikaciju korisnika pri svakom zahtjevu prema *Rest API* servisu aplikacije koji zahtjeva autentifikaciju.

Na klijentskoj strani aplikacije nakon zahtjeva prema serverskoj strani dobivamo *token* za autorizaciju budućih zahtjeva. S obzirom da *token* moramo koristiti zbog autorizacije prilikom svakog slijedećeg zahtjeva nakon prijave u aplikaciju, logičan slijed bi bio da dobiveni *token* spremimo za buduću upotrebu. Tako ćemo *token* spremiti u lokalnu memoriju, pa iščitavati *token* iz lokalne memorije kada budemo imali potrebu. U *Angular* radnom okruženju postoji mogućnost direktnog pristupanja lokalnoj memoriji, omogućeno je i da putem *intreceptora* automatiziramo svakom zahtjevu dodavanje autorizacijskog zaglavlja odnosno naš autorizacijski *token.*

Osim *interceptora (Slika 5)* postoji i *guard* koji štiti naše rute u aplikaciji. On ih štiti tako što kod svakog pristupa nekoj drugoj ruti aplikacije provjerava vjerodostojnost *tokena* za autorizaciju. *Guard* i *interceptor* su zapravo moduli koje je vrlo lako moguće uključiti u bilo koji dio koda. U ovom slučaju oni su uključeni u glavne module komponenti gdje su specificirane rute u aplikaciji.



```
import { HttpInterceptor, HttpRequest, HttpHandler, HttpEvent } from "@angular/common/http";
import { Observable } from "rxjs";
import { tap } from "rxjs/operators"
import { Router } from "@angular/router";
import { Injectable } from "@angular/core";
import { AuthService } from 'src/app/services/auth.service';

@Injectable()
export class AuthInterceptor implements HttpInterceptor{

    constructor(private service: AuthService, private router: Router){}

    intercept(req: HttpRequest<any>, next: HttpHandler): Observable<HttpEvent<any>> {
        if(this.service.userLogged()){
            const clonedRequest = req.clone({
                headers: req.headers.set('Authorization', 'Bearer ' + localStorage.getItem('token'))
            });

            return next.handle(clonedRequest).pipe(
                tap(
                    success => {},
                    error => {
                        if(error.status == 401){
                            console.log("User is not logged redirecting to login...");
                            this.router.navigateByUrl('login');
                        }
                    }
                )
            )
        } else {
            return next.handle(req.clone());
        }
    }
}
```

Slika 5 - Kod interceptora

Pored *interceptora* i *guarda (Slika 7)*koristi se i pomoćni servis za autentifikaciju *(auth.service.ts)*. Kao što je već napomenuto u samom opisu arhitektura, klijentska strana aplikacije koristi servise koji su odvojeni od koda komponente i služe za komunikaciju sa serverskom stranom aplikacije. Ovaj servis za autentifikaciju omogućuje lakši pristup metodama za prijavu, odjavu i provjeru korisnika (*Slika 6*) .



```
import { Injectable } from '@angular/core';
import { environment } from 'src/environments/environment';
import { HttpClient } from '@angular/common/http';
import { NavController } from '@ionic/angular';

@Injectable({
  providedIn: 'root'
})
export class AuthService {

  authUrl = environment.apiUrl + '/auth';

  constructor(private http: HttpClient,
              private navCtrl: NavController) { }

  loginUser(loginData: undefined){
    return this.http.post(this.authUrl + "/login/", loginData);
  }

  userLogged(){
    return localStorage.getItem('token') != null;
  }

  logOut(){
    localStorage.setItem('token', '');
    this.navCtrl.navigateBack('/login');
  }
}
```

Slika 6 - Kod autorizacijskog servisa

```typescript
import { Injectable } from '@angular/core';
import { CanActivate, ActivatedRouteSnapshot, RouterStateSnapshot, Router } from '@angular/router';

@Injectable({
  providedIn: 'root'
})

export class AuthGuard implements CanActivate {

  constructor(private router: Router){}

  canActivate(
    next: ActivatedRouteSnapshot,
    state: RouterStateSnapshot): boolean {
      if (localStorage.getItem('token') != null) {
        return true;
      } else {
        this.router.navigateByUrl('/login');
        return false;
      }
  }
}

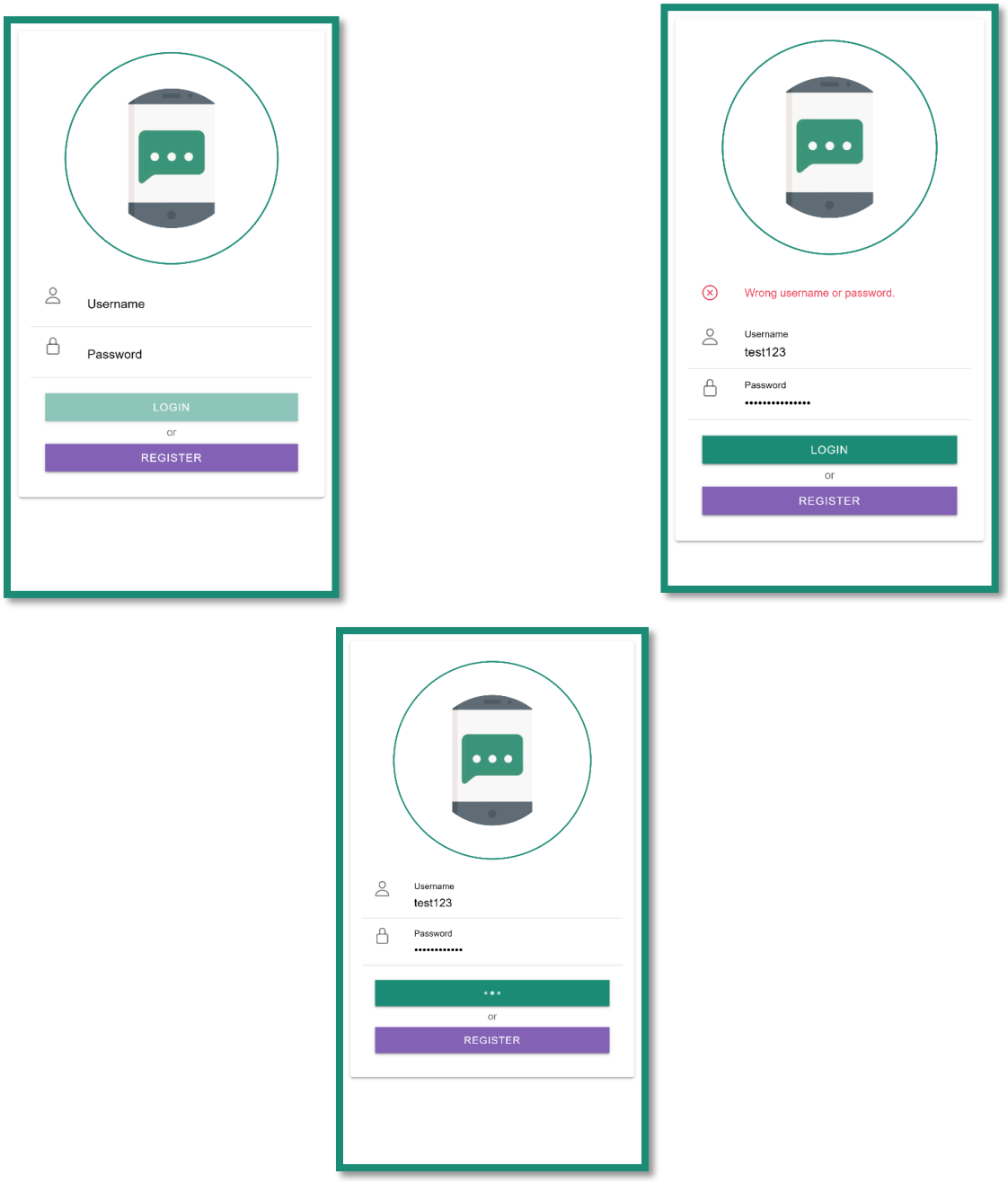
```

Slika 7 - Kod guarda

Kada bi objašnjeni dio koda preslikali u aplikaciju odnosno promatrali sa strane korisnika mobilne aplikacije, proces bi imao slijedeći redoslijed:

Korisnik otvara aplikaciju na mobilnom uređaju gdje je preusmjeren na početnu stranicu aplikacije koja je ujedno i stranica za prijavu *(Slika 8)*. Stranica za prijavu u aplikaciju ne posjeduje autorizacijsku provjeru *tokena* tako da je pristup dopušten. Na početnoj stranici nalazi se opcija gdje korisnik odabire prijavu ili se registrira ukoliko već ne posjeduje korisnički profil. Ukoliko korisnik upiše pristupne podatke i stisne gumb *Login* tada klijentska strana aplikacije odnosno kod *login.page.ts* komponente šalje podatke prema *auth.service.ts* klasi koja je zadužena za komunikaciju sa serverskom stranom aplikacije. Serverska strana aplikacije prihvaća podatke na *api/auth/login adresi Rest API* servisa.

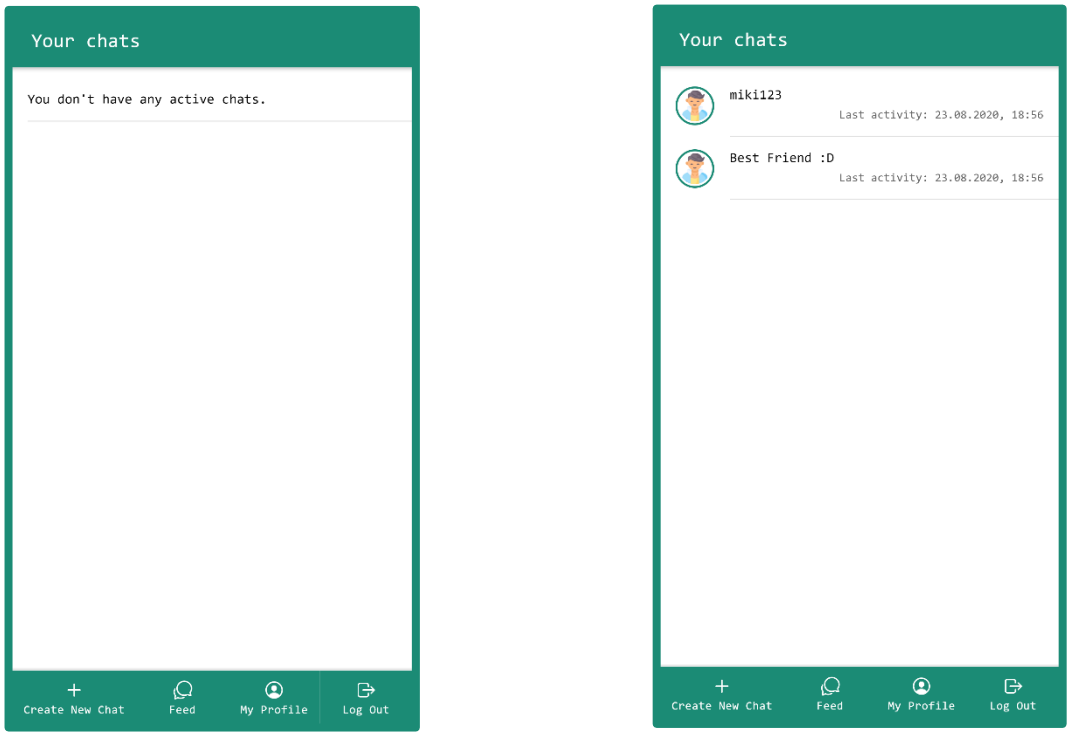
Preko *Auth managera Login* metoda kontrolera traži korisnika sa poslanim pristupnim podacima. Ukoliko korisnik sa pristupnim podacima ne postoji serverska strana aplikacije vraća status kod *401* ili *Unauthorized* što bi značilo da se korisniku zabranjuje daljni pristup dok ne upiše točne pristupne podatke. Tada klijentska strana ispisuje poruku o neuspješnosti i korisnik je dužan ponovno unijeti pristupne podatke. Ako su pristupni podaci ispravni, tada se putem metode *Auth managera* generira *token* i vraća u *JSON* obliku. Klijentska strana preuzima odgovor, *token* sprema u lokalnu memoriju i preusmjerava korisnika prema stranici sa aktivnim razgovorima.



Slika 8 - Prikaz stranice za prijavu

4.3 Upravljanje razgovorima

Nakon prijave korisnik dolazi na stranicu sa aktivnim razgovorima između njega i drugih korisnika aplikacije *(Slika 9)*. Osim popisa nedavnih razgovora možemo primjetiti i donji izbornik.



Slika 9 - Prikaz naslovne stranice sa listom aktivnih razgovora

U donjem izborniku aplikacije postoje tri opcije:

1. Stvaranje novog razgovora *(Create New Chat)* – opcija pomoću koje stvaramo novi razgovor.
2. Naslovna stranica *(Feed)* – opcija koja nas preusmjerava na naslovnu stranicu sa popisom razgovora.
3. Moj profil *(My Profile)* – opcija koja nam omogućava pregled i izmjenu korisničkih podataka.
4. Odjava *(Log Out)* – opcija za odjavu iz aplikacije.

4.4 Stvaranje novog razgovora

Odabirom prve opcije donjeg izbornika *Create New Chat* preusmjereni smo na stranicu *create-new-chat.page.html* koja se sastoji od: naslova i tipke za povratak, polja za pretragu korisnika, liste trenutno dostupnih korisnika aplikacije, liste odabranih korisnika za razgovor, polja za upis imena novog razgovora i tipke za potvrdu odabranih opcija.

Nakon što pretragom pronađemo korisnika kojeg želimo dodati u naš novi razgovor, klikom na ikonu plusa dodajemo korisnika u odabrane korisnike *(Slika 12)*.

Kada smo dodali sve korisnike koje želimo u razgovoru, odabiremo naziv razgovora *(Slika 10)*. Ako nismo razgovoru dali naziv tada se uzima standardni naziv razgovora, koji se sastoji od imena svih korisnika koji

sudjeluju u razgovoru odvojenih zarezom. Nakon stvaranja novog razgovora preusmjerni smo na stranicu razgovora gdje možemo započeti komunikaciju sa ostalim sudionicima.

```
createNewChat(){
  this.chatSubmitted = true;

  let chat = new Chat();

  chat.users = this.selectedUsers;
  chat.name = this.chatForm.controls["chatName"].value;

  this.chatsService.createChat(chat).subscribe(
    (response: any) =>{
      this.chatSubmitted = true;
      this.navCtrl.navigateForward("/chat/" + response.chatId);
    });

}
```

Slika 10 - Kod klijentske metode za slanje podataka prema serveru

```
/// <summary>
/// Create new chat.
/// </summary>
/// <param name="data"></param>
/// <returns></returns>
[HttpPost]
[Route("PostNewChat")]
0 references
public async Task<IActionResult> PostNewChat(NewChatDto data)
{
    var userId = User.FindFirst("UserId")?.Value;

    if (string.IsNullOrEmpty(userId)) return BadRequest("Logged User Not Found.");

    data.AdminId = userId;

    var response = await _chatsManager.ProcessNewChat(data);

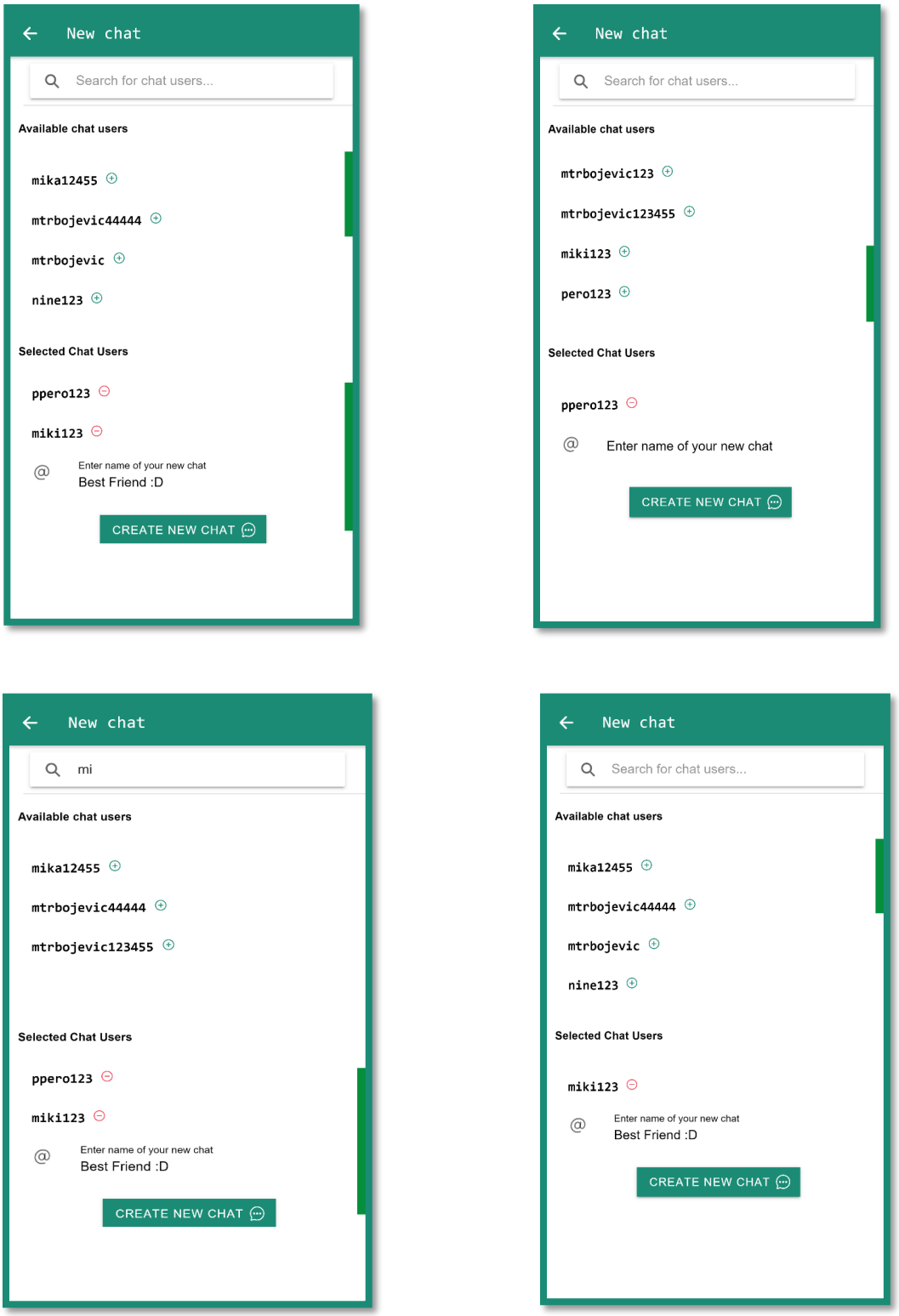
    if (!response.Success) return BadRequest(response.ErrorMessage);

    var chatId = response.Chat.ChatId.ToString();

    return Ok(new { chatId });
}
```

Slika 11 - Kod serverske metode za stvaranje novih razgovora

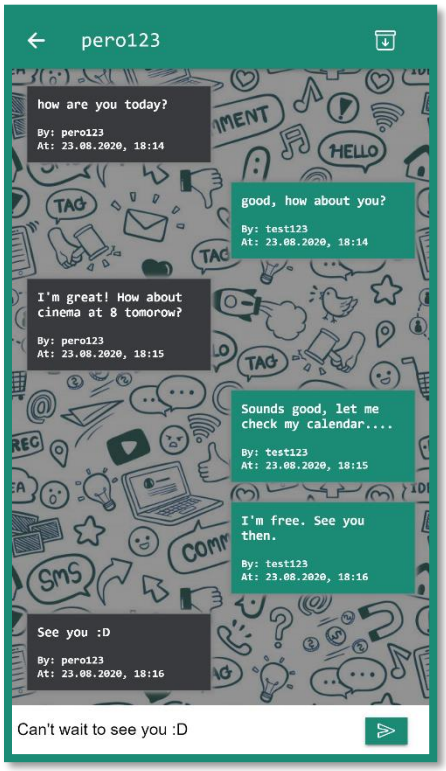
Sa serverske strane aplikacije u bazu podataka se sprema naziv razgovora, pripadajuće vremenske oznake, koje označavaju vrijeme stvaranja razgovora, vrijeme završetka razgovora i vrijeme arhiviranja razgovora. Prikaza metoda u *JSON* objektu vraća *id* novostvorenog razgovora kako bi lakše korisnika preusmjerili na stranicu novog razgovora *(Slika 11)*.



Slika 12- Prikaz stranice za stvaranje novog razgovora

4.5 Arhiviranje razgovora

U postavkama razgovora moguće je izbrisati razgovor *(Slika 13)*. U pozadini rada aplikacije to ne znači trajno brisanje već samo arhiviranje pri čemu je i dalje moguće napraviti povratak starog razgovora. Arhiviranje je jednostavan proces gdje korisnik putem klijentske strane aplikacije i *chat.service.ts* datoteke šalje zahtjev prema serverskoj strani aplikacije. Na serverskoj strani aplikacije, pronalazi se razgovor i vrijednost *IsArchived* u bazi podataka se postavlja na 1 ili *true*. Što bi značilo da pri slijedećem izlistavanju razgovora na naslovnoj stranici isti neće biti vidljiv.



Slika 13 - Prikaz razgovora sa ikonom sa arhiviranje

4.6 Komunikacija između korisnika (slanje poruka)

Kako bi ostvarili komunikaciju između korisnika bitno je da prilikom ulaska svakog korisnika u razgovor zabilježimo ulazak svakog od korisnika pomoću *connectionId-a* ili šifre gdje ćemo moći upravljati njihovim sesijama unutar razgovora *(Slika 15)*.

Prilikom učitavanja razgovora dohvaćamo sve potrebne postavke za postavljanje kanala za komunikaciju ili *hub-a* (može se nazivati i *channel)* između korisnika. Ovaj dio konfiguriranja razgovora obavlja *hubs.service.ts* datoteka.

```
5 references
public class MessageHub : Hub
{
    0 references
    public Task SendPublicMessage(Message message)
    {
        return Clients.All.SendAsync("receiveMessage", message);
    }

    0 references
    public async Task SendMessageToChat(Message message, string chatname)
    {
        await Clients.Group(chatname).SendAsync("receiveMessage", message);
    }

    0 references
    public string GetConnectionId() => Context.ConnectionId;
}
```

Slika 14- Kod hub controller-a za slanje asinkronih poruka prema klijentima

```
startConnection(chatId: string){
  this.hubConnection = new SignalR.HubConnectionBuilder()
                        .withUrl(environment.messageHubUrl)
                        .configureLogging(SignalR.LogLevel.Information)
                        .build();

  this.addChatMessagesListener();

  this.hubConnection.start().then(() => {
    console.log("Connection started");
    this.hubConnection.invoke('getConnectionId').then((connectionId) => {
      console.log("ConnectionId: " + connectionId);
      this.joinChatHub(connectionId, chatId);
    }).catch(error =>
      {
        console.log("Error while joining chat: " + error);
        return;
      })
  }).catch(error =>
    {
      console.log("Error while starting connection: " + error);
      return;
    }
  );
}
```

Slika 15 - Kod klijentske servisne datoteke za komunikaciju sa hub kontrolerom

Tijekom tog procesa dobivamo već spomenuti c*onnectionId (Slika 15)*. Ako smo dobili uspješan odgovor prilikom dohvaćanja *connectionId-a (Slika 14)* pridužujemo korisnika kanalu za komunikaciju.

Nakon toga dohvaćamo sve podatke o razgovoru zajedno sa porukama koje pripadaju tom razgovoru, i dodatne podatke o trenutno prijavljenom korisniku odnosno u ovom slučaju nama. Dodatni podatci o prijavljenom korisniku nam najviše služe kako bismo lakše prikazali korisniku njegovu ulogu u razgovoru.

Primjerice, koje poruke su poslane sa njegove strane, a koje od strane drugih korisnika. Nakon završenog procesa konfiguriranja pretplaćeni *(subscribe)* smo na sve obavijesti koje su povezane sa tim kanalom za komunikaciju.

```
subscribeToChatEvents() {
  this.hubsService.getReceivedMessage().subscribe((response: any) => {
    this.messages.push(response as Message);
    this.scrollToBottom();
  });
}

sendMessage(sendMessageForm: NgForm) {

  this.sendMessageForm.controls["chatId"].setValue(this.chatId);

  this.messagesService.sendChatMessage(sendMessageForm.value).subscribe(
    (response: any) => {
      this.sendMessageForm.reset();
      this.scrollToBottom();
    },
    () => {
      this.errorSendingMessage = true;
    }
  );
}
```

Slika 16 - Kod servisa za slanje sadržaja poruke

Prilikom slanja poruke putem *messages.service.ts* datoteke *(Slika 16)* šaljemo sadržaj poruke prema *Rest API* servisu koji sprema poruku u bazu podataka i veže je uz razgovor. Nakon uspješnog spremanja poruke u bazu podataka potrebno je svim korisnicima pretplaćenim na kanal proslijediti istu poruku, kako bi u realnom vremenu bili obavješteni o novoj poruci u razgovoru *(Slika 17)*. Bitno je naglasiti kako korisnika koji je poslao poruku moramo izuzeti od slanja obavijesti. Jer je poruka na njegovom mobilnom uređaju prilikom klika na gumb *Send* koji šalje poruku u razgovor odmah prikazana, što bi rezultiralo dupliciranjem iste poruke na korisnikovom ekranu mobilnog uređaja.

```
/// <summary>
/// Process new message
/// to existing chat.
/// </summary>
/// <param name="messageData"></param>
/// <returns></returns>
[HttpPost]
[Route("SendChatMessage")]
0 references
public async Task<IActionResult> SendChatMessage(PostMessageToChatDto messageData)
{
    var userId = User.FindFirst("UserId")?.Value;

    if (string.IsNullOrEmpty(userId)) return BadRequest();

    messageData.UserId = userId;

    var response = await _messagesManager.ProcessChatMessage(messageData);

    if (!response.Success) return BadRequest(response.ErrorMessage);

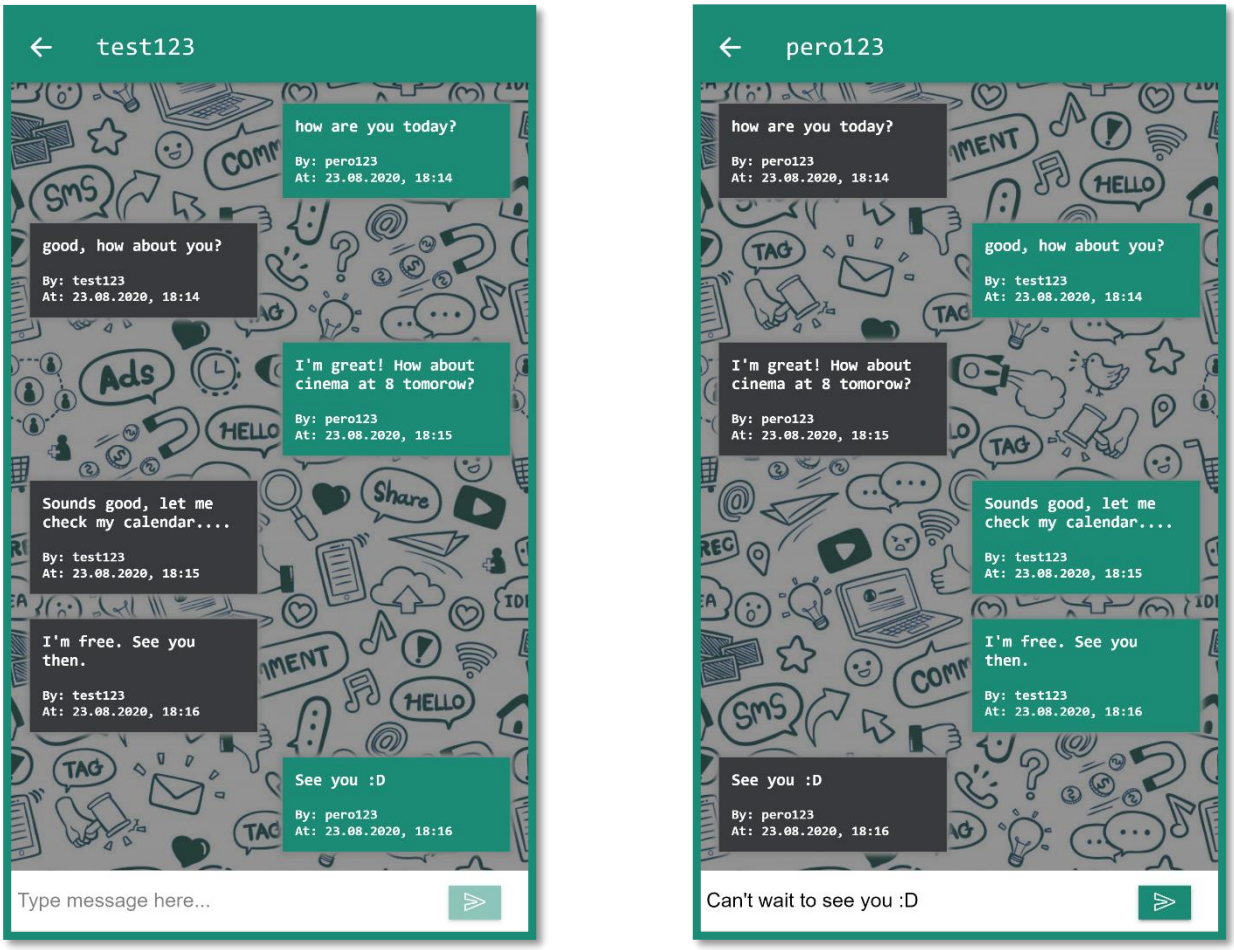
    var groupName = messageData.ChatId.ToString();
    var senderId = messageData.UserId;

    await _hub.Clients
        .GroupExcept(groupName, senderId)
        .SendAsync("receivemessage", response.Message);

    return Ok(response.Message);
}
```

Slika 17 - Kod za spremanje poslane poruke i slanje preka hub controlleru

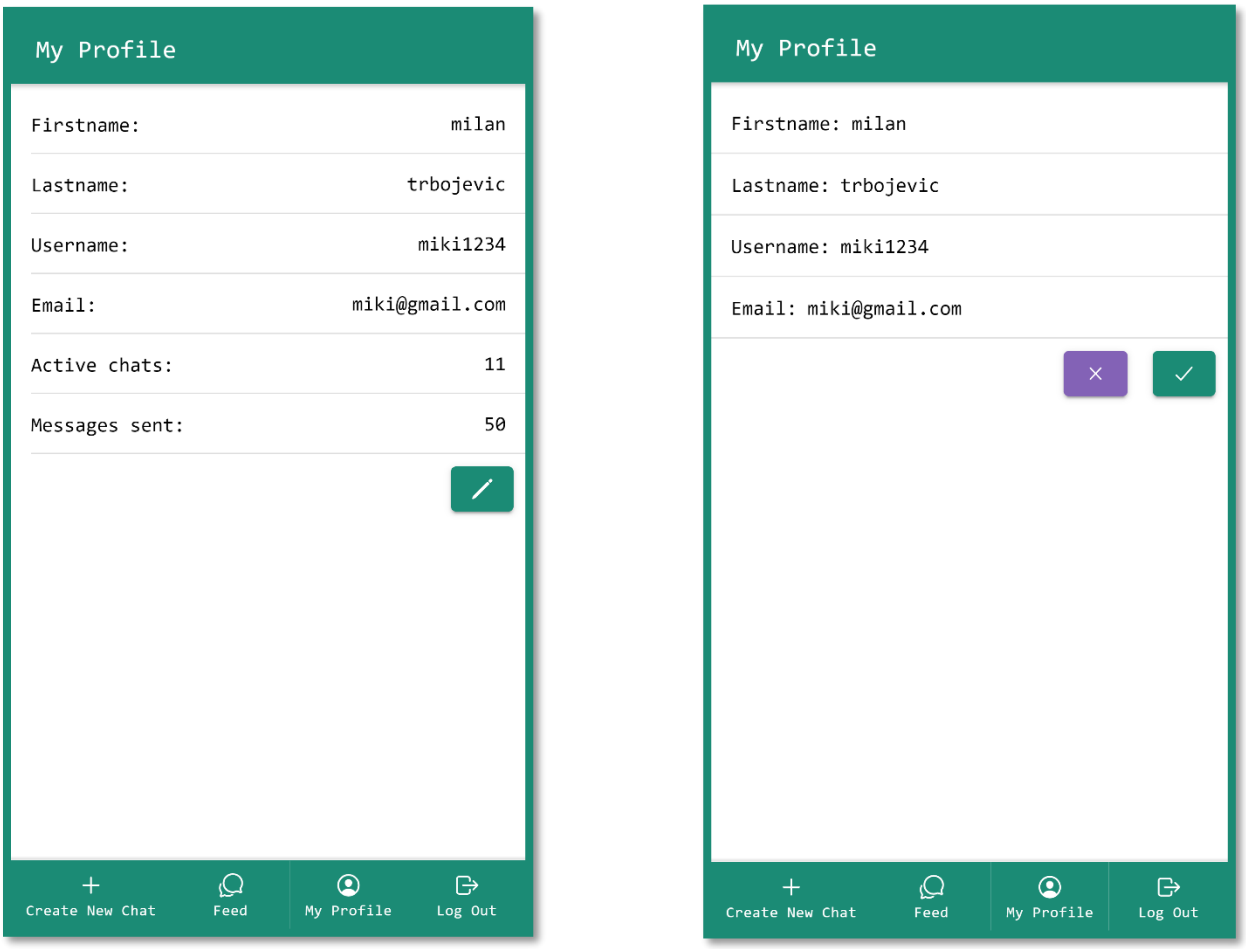
Kada poruka u obliku objekta uđe u kanal za komunikaciju tada će ona biti vidljiva svim pretplaćenim korisnicima *(Slika 18)*. Ukoliko se dogodila pogreška u procesu tada dobivamo obavijest u skočnom prozoru koja nam kaže kako slanje poruke nije uspješno.



Slika 18 - Prikaz stranice razgovora

4.7 Uređivanje korisničkog profila

Odabirom treće opcije *My Profile* otvaraju nam se osnovne informacije o našem profilu *(Slika 19)*. Klikom na gumb sa ikonom olovke podatci se pretvaraju u formu za izmjenu podataka. Klikom na gumb kvačice potvrđujemo izmjenu korisničkih podataka, dok klikom na gumb sa oznakom križa odustajemo do izmjene podataka i vraćamo na prikaz. Ažurirani podatci biti će vidljivi odmah te se nije potrebno ponovno prijavljivati u aplikaciju da bi se oni ažurirali u prikazu aplikacije.



Slika 19 - Prikaz stranice profila

4.8 Obavijesti (notifikacije)

Vanjski sustav *Firebase Cloud* je sustav koji će omogućiti registraciju uređaja na *Cloudu* na kojem je instalirana aplikacija. Za razliku od ostalih komponenti naše messenger aplikacije, ova komponenta koja će omogućiti slanje obavijesti korisnicima je nativna. Što znači da moramo instalirati dodatak(*plugin*) kako bi ona funkcionirala i pomoću nje pristupamo samom uređaju. Postupak registracije korisnika na *Firebase Cloudu* će nam pomoći da točno znamo kojem korisniku moramo poslati obavijest o novoj pristigloj poruci u razgovoru u kojem on sudjeluje. Prednost je što korisnik može dobiti obavijest ukoliko nije unutar same aplikacije, znači zaključao je mobilni uređaj ili obavlja rad u drugoj mobilnoj aplikaciji.

Prvi korak koji moramo napraviti je registrirati se na *Firebase Cloud*. To možemo napraviti i sa postojećim *Google* računom. Usluga *Firebase Clouda* koja nas zanima jest *Firebase Cloud Messaging* tako da ćemo nakon registracije preuzeti konfiguracijsku datoteku za *Cloud Messaging* i postaviti je u naš projekt. Tu datoteku će koristiti mobilna aplikacija prilikom komunikacije sa *Firebase Cloud* platformom, kako bi se ispravno autentificirala.

```
import { AngularFirestore } from '@angular/fire/firestore';
import { Platform } from '@ionic/angular';
import { Firebase } from '@ionic-native/firebase/ngx';
import { Injectable } from '@angular/core';

@Injectable({
  providedIn: 'root'
})
export class FirebaseCloudMessagingService {

  constructor(public firebaseNative: Firebase,
              public firestore: AngularFirestore,
              private platform: Platform) { }

  async getToken(){

    let token;

    if(this.platform.is('android')){
      token = await this.firebaseNative.getToken();
    }

    if(this.platform.is('ios')){
      token = await this.firebaseNative.getToken();
      await this.firebaseNative.grantPermission();
    }

    if(!this.platform.is('cordova')){
      //currently not needed
    }

    return this.saveTokenToFirestore(token);

  }

  saveTokenToFirestore(token: any) {
    if(!token) return;

    const devicesReference = this.firestore.collection('devices');

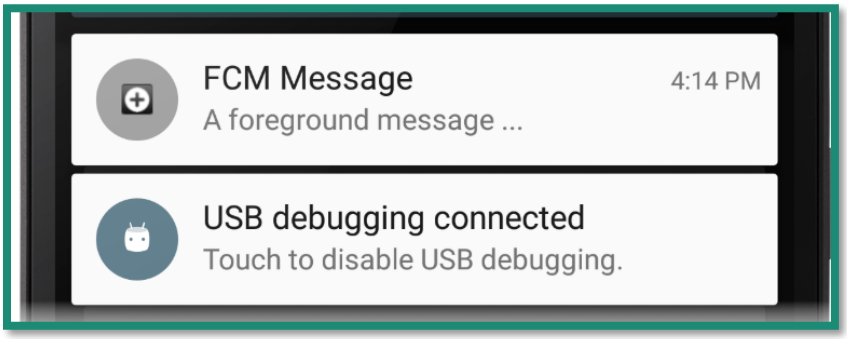
    const docData = {
      token,
      userId: 'testUser'
    }

    return devicesReference.doc(token).set(docData);
  }
```

Slika 20 - Kod FCMProvider servisa

U našem projektu napraviti ćemo FCMProvider (Firebase Cloud Messaging Provider) *(Slika 20)* koji je zapravo servisna datoteka naše mobilne aplikacije i putem nje ćemo slati zahtjev prema Firebase Cloud platformi. Vrlo je bitno prilikom korisnikove prijave u mobilnu aplikaciju upozoriti korisnika da će se koristiti značajka slanja obavijesti te da ih on može putem postavki onemogućiti u bilo kojem trenutku.

Nakon što *Firebase Cloud* proslijedi poruku prema korisniku, poruka će se korisniku prikazati na njegovom uređaju u obliku obavijesti(notifikacije) *(Slika 21)*. Klikom na obavijest korisnik je preusmjeren u razgovor u koji je pristigla nova poruka.



Slika 21 - Prikaz dospjele obavijest o pristigloj poruci

5. Postavljanje aplikacije u produkcijsko okruženje

Sudeći po planu izvedbe aplikacije, odnosno tijeku podataka i procesa navedenom u uvodnom poglavlju, odlučeno je kako će se produkcijsko okruženje serverskog dijela aplikacije nalaziti na *Azure Cloud [22]* platformi. Tamo će biti postavljen *Rest API* servis i *SQL* baza podataka. U ovom poglavlju biti će objašnjen cijeli proces postavljanja na produkciju kao i postavljanje okruženja u *Microsoft Visual Studio-u* za objavljivanje ažuriranja aplikacije (ili popularnije *publish*) direktno na *Azure Cloud.*

5.1 Postavljanje aplikacije na *Azure Cloud*

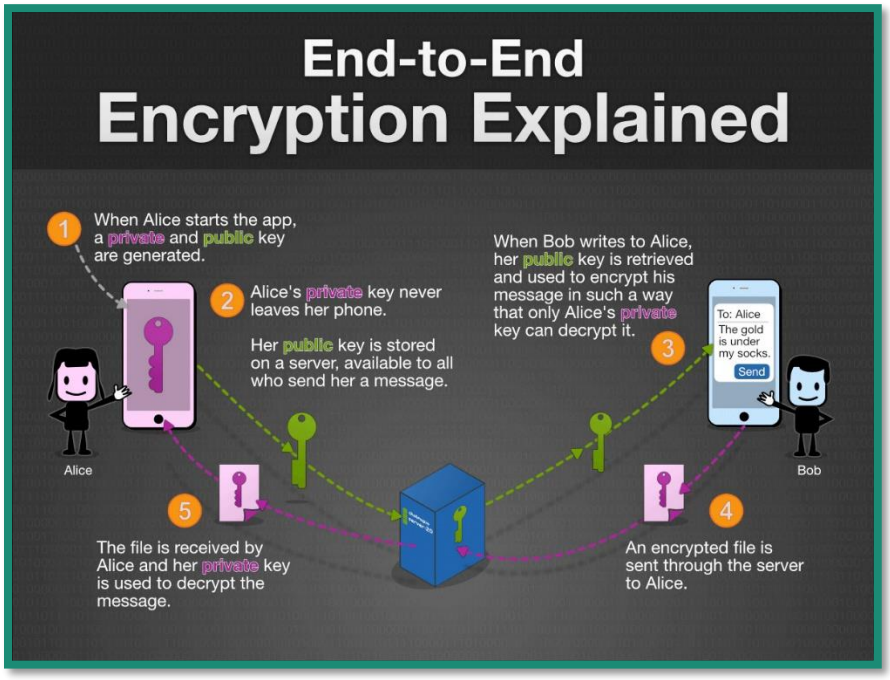
S obzirom da je *Azure Cloud* pod vlasništvom *Microsoft-a*, moguće se prijaviti u sustav putem bilo kojeg *Microsoft-ovog* računa. Za potrebe naše *messenger* aplikacije možemo koristiti i *Office 365[23] Microsoft-ov* račun koji nam je besplatno dodjeljen od strane sveučilišta. Naravno, treba izbjegavati korištenje računa od strane organizacija te bi bilo najbolje koristiti privatne račune ukoliko se radi o komercijalnoj aplikaciji. Naš račun od sveučilišta omogućava nam godinu dana besplatnog korištenja *Azure Cloud* proizvoda u iznosu od 170 eura ili 200 američkih dolara. Nama trenutno taj iznos ne predstavlja ništa osobito jer to nisu stvarni novci već je to samo kredit pomoću kojeg možemo zakupiti usluge na godinu dana, i time nas se ograničava da sa besplatnim planom korištenja ne zauzimamo previše resursa u infrasktrukturi.

6. Prijedlog nadogradnje messenger aplikacije

Analizirajući sve najbitnije stavke *messenger* aplikacije, moglo bi se reći da su one najbitnije pokrivene kako bi bio opravdan sam naziv aplikacije. Naravno da najpoznatije *messenger* aplikacije poput: *WhatsApp-a, Facebook Messenger-a, Viber-a* itd. Imaju puno više dodatnih funkcionalnosti nego ova *messenger* aplikacija. No, cilj nije bio da stvorimo konkurentnu *messenger* aplikaciju već da složimo stabilnu aplikaciju sa osnovnim funkcionalnostima i mogućnošću razmjene sadržaja u realnom vremenu. Svejedno bih naveo nekoliko mogućnosti koje mogu predstavljati nadogradnju na postojeći projekt i stvoriti bolji *UX (user-experience).*

6.1 *End-End* Enkripcija

Svjesni smo kako je sigurnost unutar *messenger* aplikacije jedna od bitnijih tehničkih stavki. Osim sadržaja koji se razmjenjuje moramo biti svjesni kako se tu radi i o našim privatnim informacijama i podatcima. Kako bi se osjećali sigurnije i bezbrižno slali poruke drugim korisnicima, bez straha od mogućeg presretanja poruke ili zloupotrebljavanja njenog sadržaja, jedna od mogućih nadogradnji koja se nameće je implementacija *End-End (end to end) [27]* enkripcije kako bi se zaštitio sadržaj poruke koji se šalje, na način da se poruka kriptira javnim ključem i pretvara u nečitljivi oblik. U tom nečitljivom obliku ona se šalje prema drugom korisniku koji pomoću tajnog ključa otključava/dekriptira poruku i tada mu se prikazuje pravi sadržaj *(Slika 22)*.



Slika 22 - Dijagram tijeka procesa End-End enkripcije
(https://cybersecop.com/news/2018/1/22/how-end-to-end-encryption-wors)

Primjer implementacije End-End enkripcije u messenger aplikaciji:

S obzirom da se *End-End* enkripcija zasniva na korištenju i manipulaciji javnog i tajnog ključa. Bitno je odrediti mjesto na kojem ćemo skladištiti i čuvati naše ključeve za komunikaciju. Privatni ključ koji koristi korisnik spremljen je na samom mobilnom uređaju gdje je pokrenuta aplikacija, dok su javni ključevi spremljeni u bazi podataka. Sam server tj. baza podtaka nije upoznata sa sadržajem poruke odnosno u bazi podataka se nalazi samo enkriptirani oblik poslane poruke. Također je bitno i da se ključevi s vremena na vrijeme mjenjaju. Što naravno nije u planu da promjenu napravi sam korisnik već bi naš *web* servis trebao imati preodređene vremenske periode kada će izvršavati izmjenu javnih ključeva.

6.2 Dijeljenje lokacije

Dijeljenje trenutne lokacije tijekom razgovora gdje će ostali sudionici biti u mogućnosti unutar razgovora saznati gdje se drugi korisnik razgovora točno nalazi prema poslanim koordinatama koje će *messenger* aplikacija pretvarati u koordinate označene na karti.

6.3 Dijeljenje medijskog sadržaja

Ideja je da se korisniku omogući dijeljenje slika,videa i dokumenata direktno iz same aplikacije kao i *upload* sadržaja iz galerije mobilnog uređaja. Osim slika i videa moguće je poslati i kratki glasovni zapis, što je vrlo korisno ako korisnik nije u mogućnosti u potpunosti odvratiti pažnju prema ekranu mobilnog uređaja. Valja napomenuti kako cijeli proces sa tehničke strane povlači cijeli jedan novi dio arhitekture skladištenja sadržaja koji se razmjenjuju. Jer se ipak sada radi o sadržajima koji zauzimaju znatno više memorijskog prostora od samog teksta.

6.4 Lista bliskih korisnika

Korisnik ima mogućnost da u bliske korisnike doda druge korisnike sa kojima najčešće komunicira što je vrlo korisno ako imate mnogo otvorenih razgovora istovremeno.

6.5 Pretraga poruka

Pretraga poruka unutar razgovora gdje je moguće pretražiti povijest poruka prema ključnoj riječi ili vremenskom razdoblju poslanih poruka.

6.6 Uređivanje/prilagodba izgleda razgovora

Kako bi se poboljšala prisnost između korisnika i aplikacije, bilo bi dobro omogućiti da korisnik sam uređuje izgled razgovora sa drugim korisnikom. Neke od stavki podložnih promjeni bi bila izmjena pozadine u razgovoru ili boja „oblačića“ odnosno prostora u koje je zapisan tekst poslane poruke. Time bi se na neki način aplikacija još više približila korisniku, jer tada bi teže došlo do zasićenja pri korištenju aplikacije.

6.7 Višejezičnost

Jedna vrlo korisna značajka čija funkcionalnost može biti usko povezana i sa samim korisnikom. Tako da bi svaki korisnik nakon postavljanja željenog jezika u aplikaciji imao svoj zadani jezik, te bi mu nakon same prijave i odabira jezika omiljeni jezik uvijek bio predefiniran.

6.8 Pregled aktivnosti trenutno prijavljenih korisnika

Uobičajena značajka danas popularnih *messenger* aplikacija je i pregled aktivnosti trenutno prijavljenih korisnika. Specifično se misli na to da u svakom trenutku prilikom komunikacije sa korisnikom znamo kada je taj korisnik zadnji puta bio aktivan u aplikaciji. Također je vrlo korisno imati i informaciju o tome je li poslana poruka u potpunosti isporučena prema mobilnom uređaju/mobilnoj aplikaciji drugog korisnika.

6.9 Dvostruka autentifikacija i prijava putem *Google ili Facebook računa*

Trenutno naša *messenger* aplikacija posjeduje samo jedan tip prijave u aplikaciju. Vrlo popularna i korisna značajka bi bila da nam je prijava omogućena putem nekog već postojećeg računa kako ne bi morali prolaziti kroz proces registracije. Korisnički podatci bi bili povučeni sa računa kojeg smo izabrali. Također kako bi povećali sigurnost tijekom same prijave u aplikaciju, jedan od načina može biti i implementacija dvostruke autentifikacije. Tako možemo tražiti potvrdu koda u tekstualno ili brojčanom obliku poslanog na email unutar 10 min od prijave kako bismo bili sigurni da se ne radi o zlouporabi korisničkih podataka.

8 . Zaključak

Cilj ovog diplomskog rada bio je samostalno projektirati i isprogramirati modernu *web* i mobilnu aplikaciju za veći broj korisnika. Počevši od planiranja arhitekture aplikacije, plana izvedbe, odabira tehnologija te naposlijetku dokumentiranja cijelog procesa i tijeka rada. Osim toga bilo je bitno i postaviti *Rest API* servis na server kako bi bila moblina aplikacija funkcionirala i bila dostupna korisnicima mobilnih uređaja neovisno o platformi i mreži koju koriste.

Što se tiče samih performansi, odabir tehnologija u ovom diplomskom radu i optimiziranost koda utjecala je da se slanje poruka odvija veoma brzo i u duhu jedne ozbiljne *real-time* aplikacije.

Performanse bi vjerojatno bile još izraženije i bolje da na serveru gdje se „vrti“ *Rest API* servis nemamo djeljive resurse sa drugim korisnicima *Azure Clouda*. No, uskoro bi se i to trebalo promijeniti te će se aplikacija prebaciti na ozbiljniji *plan* korištenja. Namjera je da se aplikacija i dalje razvija u svrhu stjecanja novih znanja i edukacije u razvijanju modernih *web* i mobilnih aplikacija.

Fokus je bio na stvaranju dobrog „kostura“ mobilne messenger aplikacije koja se može lako nadograditi i poboljšati kako bi bila spremna za komercijalnu upotrebu.

Nadam se da će ovaj diplomski rad mnogima pomoći u njihovim istraživanjima i učenju novih tehnologija ili načina rješavanja problema ili jednostavno proširiti vidike što ponovno dovodi do samoostvarenja u nekom smislu.

Zaključio bih sa time da sam mišljenja kako je i meni osobno, širenje znanja koje sam stekao pišući ovaj diplomski rad, pomoglo u trenutnom zaposlenju i ispunilo najbitniji cilj koji sam si zacrtao, a to je da radim posao kojim volim. Jer kao i što poslovica kaže: „Pronađite posao koji volite i nećete morati raditi niti jednog dana u svom životu.“

9. Literatura

[1] *.Net Core Wikipedia,* https://en.wikipedia.org/wiki/.NET_Core, pristupljeno 25.6.2020.

[2] *Firebase Cloud Messaging,* https://firebase.google.com/docs/cloud-messaging,

pristupljeno 25.6.2020.

[4] *Angular,* https://angular.io/, pristupljeno 30.6.2020.

[5] *React,* https://reactjs.org/, pristupljeno 30.6.2020.

[6] *Rest API,* https://restfulapi.net/, pristupljeno 30.6.2020.

[7] *Microsoft SQL Server,* https://hr.wikipedia.org/wiki/Microsoft_SQL_Server, pristupljeno 5.7.2020.

[8] *Using SignalR Hubs,* https://docs.microsoft.com/en-us/aspnet/core/signalr/hubs?view=aspnetcore-3.1, pristupljeno 7.7.2020.

[9] *Firebase,* https://firebase.google.com/, pristupljeno 15.7.2020.

[10] *SignalR,* https://dotnet.microsoft.com/apps/aspnet/signalr, pristupljeno 15.7.2020.

[11] *Using route guards,* https://medium.com/@ryanchenkie_40935/angular-authentication-using-route-guards-bf7a4ca13ae3, pristupljeno 18.7.2020.

[12] *Angular Interceptors,* https://angular.io/api/common/http/HttpInterceptor, pristupljeno 18.7.2020.

[13] *Controllers,* https://docs.microsoft.com/en-us/aspnet/core/mvc/controllers/actions?view=aspnetcore-3.1, pristupljeno 18.7.2020.

[14] *Entity Framework,* https://docs.microsoft.com/en-us/ef/, pristupljeno 19.7.2020.

[15] *Migrations,* https://docs.microsoft.com/en-us/ef/core/managing-schemas/migrations/?tabs=dotnet-core-cli, pristupljeno 20.7.2020.

[16] *Single Responsibility Principle,* https://en.wikipedia.org/wiki/Single-responsibility_principle, pristupljeno 22.7.2020.

[17] *Node Package Manager,* https://www.npmjs.com/, pristupljeno 22.7.2020.

[18] *NuGet Package Manager,* https://www.nuget.org/, pristupljeno 22.7.2020.

[19] *JSON,* https://www.json.org/json-en.html, pristupljeno 23.7.2020.

[20] *Ionic,* https://ionicframework.com/, pristupljeno 23.7.2020.

[21] *JSON Web Token,* https://jwt.io/, pristupljeno 24.7.2020.

[22] *Azure Cloud,* https://azure.microsoft.com/en-us/free/cloud-services/search/?&ef_id=CjwKCAjwm_P5BRAhEiwAwRzSO7HNWT9fKmvZJypzXM6a6Q5Rct5LX9FX6VJRqEsEyLcFy7rz4OJjPBoC3uEQAvD_BwE:G:s&OCID=AID2100638_SEM_CjwKCAjwm_P5BRAhEiwAwRzSO7HNWT9fKmvZJypzXM6a6Q5Rct5LX9FX6VJRqEsEyLcFy7rz4OJjPBoC3uEQAvD_BwE:G:s, pristupljeno 25.7.2020.

[23] *Office 365,* https://www.microsoft.com/hr-hr/microsoft-365/buy/microsoft-365-family, pristupljenom 25.7.2020.

[24] *What is an SSL Certificate,* https://www.cloudflare.com/learning/ssl/what-is-an-ssl-certificate/, pristupljeno 26.7.2020.

[25] *Microsoft Visual Studio,* https://visualstudio.microsoft.com/, pristupljeno 26.7.2020.

[26] *SQL Server Managment Studio,* https://docs.microsoft.com/en-us/sql/ssms/sql-server-management-studio-ssms?view=sql-server-ver15, pristupljeno 26.7.2020.

[27] *End-End Encryption,* https://en.wikipedia.org/wiki/End-to-end_encryption, https://www.gbnews.ch/what-is-end-to-end-encryption-e2ee/, pristupljeno 27.7.2020.

[28] *GitHub,* https://github.com/, pristupljeno 25.8.2020.

[29] *git,* https://git-scm.com/, pristupljeno 20.8.2020

[30] *Version control,* https://en.wikipedia.org/wiki/Version_control, pristupljeno 24.8.2020.

[31] *Pushing and Pulling from GitHub,* https://biologyguy.github.io/git-novice/05-push-pull/, pristupljeno 24.8.2020.

[32] *Radno okruženje,* https://en.wikipedia.org/wiki/Software_framework, pristupljeno 25.8.2020.

[33] *Version Control, https://en.wikipedia.org/wiki/Version_control,* pristupljeno 4.9. 2020.

[34] *Messaging Apps, https://en.wikipedia.org/wiki/Messaging_apps,* pristupljeno 5.9.2020.

9.1 Popis slika

Slika 1 - Arhitektura messenger aplikacije 5
Slika 2 – Klijentska arhitektura 6
Slika 3 – Serverska arhitektura 8
Slika 4 - Slanje zahtjeva za autorizaciju 12
Slika 5 - Kod interceptora 13
Slika 6 - Kod autorizacijskog servisa 13
Slika 7 - Kod guarda 14
Slika 8 - Prikaz stranice za prijavu 15
Slika 9 - Prikaz naslovne stranice sa listom aktivnih razgovora 16
Slika 10 - Kod klijentske metode za slanje podataka prema serveru 17
Slika 11 - Kod serverske metode za stvaranje novih razgovora 17
Slika 12- Prikaz stranice za stvaranje novog razgovora 18
Slika 13 - Prikaz razgovora sa ikonom sa arhiviranje 19
Slika 14- Kod hub controller-a za slanje asinkronih poruka prema klijentima 20
Slika 15 - Kod klijentske servisne datoteke za komunikaciju sa hub kontrolerom 20
Slika 16 - Kod servisa za slanje sadržaja poruke 21
Slika 17 - Kod za spremanje poslane poruke i slanje preka hub controlleru 21
Slika 18 - Prikaz stranice razgovora 22
Slika 19 - Prikaz stranice profila 23
Slika 20 - Kod FCMProvider servisa 24
Slika 21 - Prikaz dospjele obavijest o pristigloj poruci 25
Slika 22 - Dijagram tijeka procesa End-End enkripcije (https://cybersecop.com/news/2018/1/22/how-end-to-end-encryption-wors) 27
Slika 23 - Stvaranje novog .Net Core projekta 34
Slika 24 - Konfiguriranje novog .Net Core projekta 35
Slika 25 - Vrste Ionic predložaka 36
Slika 26 - Standardna stranica nove Ionic aplikacije 36
Slika 27 - Instalacija SignalR biblioteke putem NuGet Package Managera 37
Slika 28 - Poruka o uspješnosti instalacije SignalR biblioteke 37
Slika 29 - Azure Portal 38
Slika 30 - Proces postavljanja Rest API servisa na web server 39
Slika 31 - Prikaz prometa Rest API servisa u obliku grafikona 40
Slika 32 - Konfiguriranje SQL baze podataka na SQL server 41

10. Prilozi

10.1 Smjernice prema repozitoriju koda

Poveznice na javni repozitorij *Github* profila gdje se nalazi kod *Rest API* servisa i mobilne aplikacije:

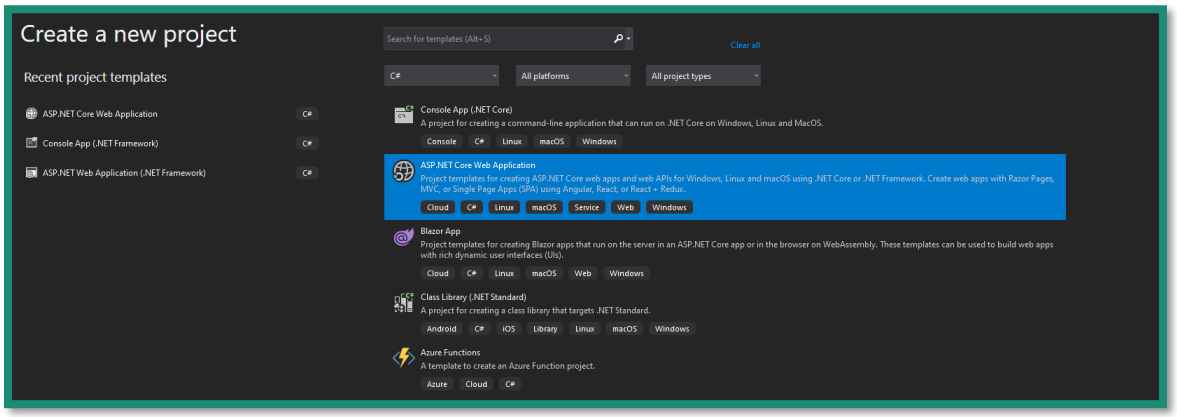
https://github.com/riverDeer12/messenger-api - poveznica za *REST API* servis.

https://github.com/riverDeer12/messenger - poveznica za mobilnu aplikaciju.

10.2 Instalacija i konfiguriranje radnog okruženja

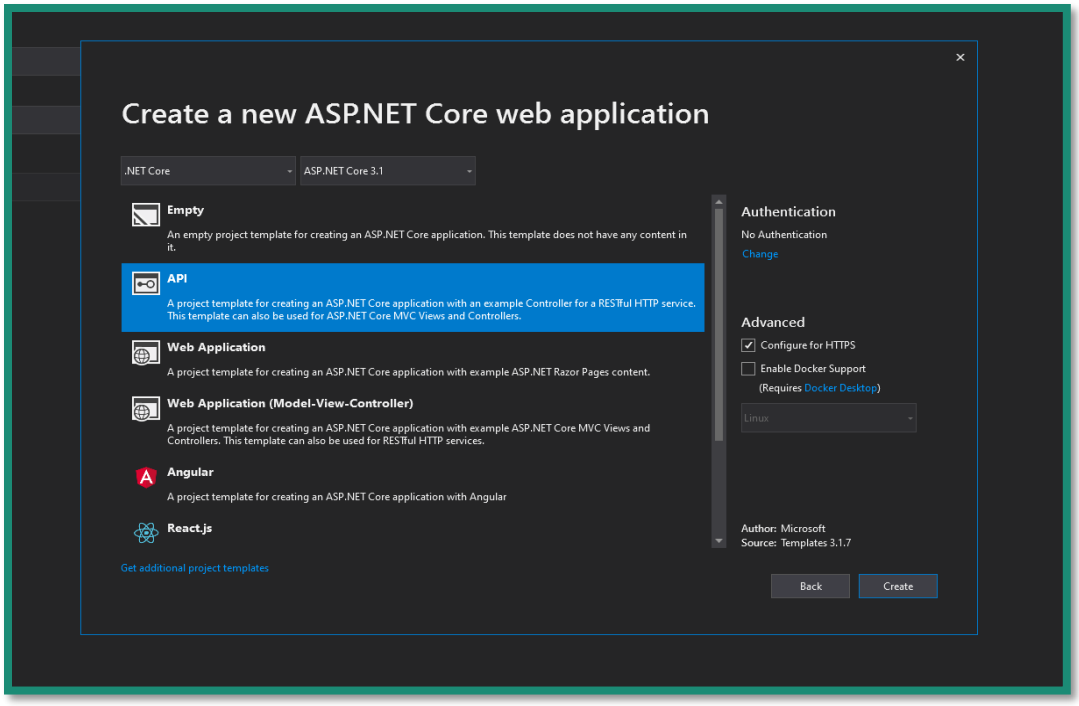
10.2.1 Postavljanje *.Net Core Web API* radnog okruženja

Za razvijanje serveskog dijela aplikacije kao alat korišten je *Microsoft Visual Studio*. Ukoliko ga nemate već predhodno instaliranog na Vašem računalu možete besplatno skinuti njegovu *Community* verziju (koja posjeduje dovoljan broj značajki potreban za razvijanje *messenger* aplikacije) putem ove poveznice: https://visualstudio.microsoft.com/vs/, dok ostale verzije sa nešto naprednijim značajkama imaju probni period nakon kojeg plaćate pretplatu. Prilikom stvaranja novog projekta imate već ponuđenu opciju za postavljanjem arhitekture koja se koristi pri izradi ove aplikacije (*Slika 23*).



Slika 23 - Stvaranje novog .Net Core projekta

Zatim odabiremo vrstu projekta, u ovom slučaju to je *.Net Core* aplikacija. Nakon definiranja naziva projekta u skočnom prozoru možemo odabrati vrstu aplikacije pomoću čega će biti složena i njena arhitektura *(Slika 24)*. Postavljanje projekta i arhitekture može potrajati nekoliko minuta. Nakon toga smo spremni za rad na serverskom dijelu naše *messenger* aplikacije.



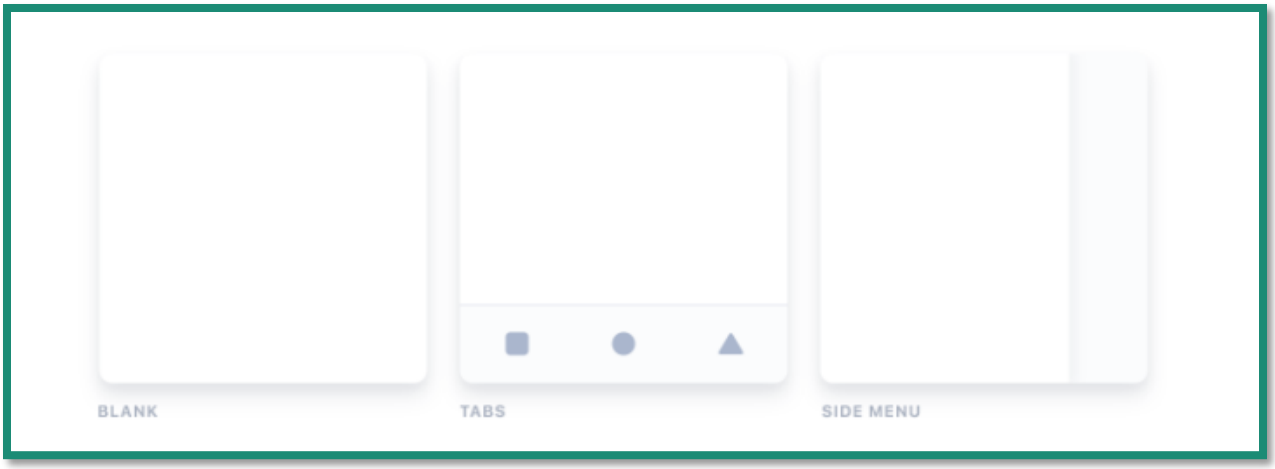
Slika 24 - Konfiguriranje novog .Net Core projekta

10.2.2 Postavljanje *Ionic* aplikacije

Kako bi putem komande linije (što je ujedno i najbrži način stvaranja novog projekta) mogli koristiti *Ionic* i *npm* naredbe. Potrebno je instalirati *Node.js*. Koji nam omogućuje globalne naredbe pomoću kojih možemo upravljati *Javascript* projektima. Naredbom *npm install -g @ionic/cli* postavljamo *ionic* naredbu kao globalnu u našem operacijskom sustavu. Pozicioniramo se u direktorij našeg operacijskog sustava gdje želimo stvoriti novi *Ionic* projekt. Zatim sa *ionic start <ime ionic aplikacije> <vrsta predloška>* stvaramo novi *ionic* projekt. Parametar *<vrsta predloška>* predstavlja predodređeni predložak ili *template* koji će se primijeniti na *ionic* aplikaciju nakon njenog stvaranja.

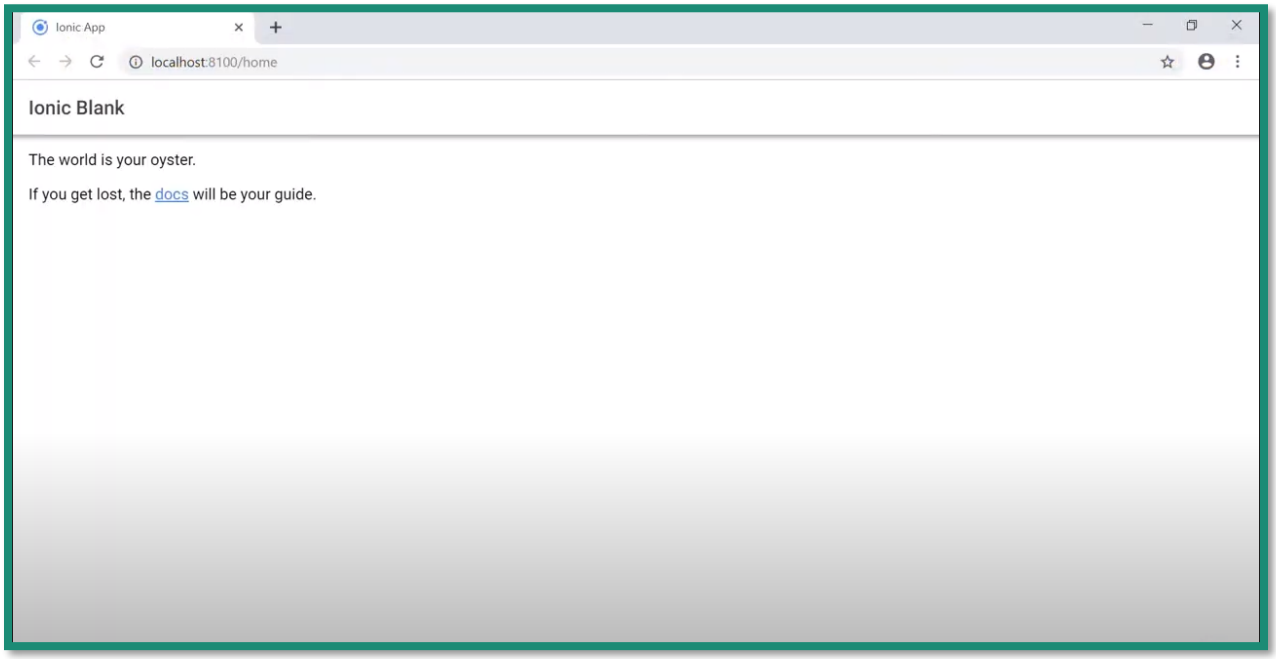
Za parametar *<vrsta predloška>* možemo odabrati:

- blank – prazan predložak.
- *tabs* – predložak sa uobičajenim izbornikom u donjem djelu ekrana karakterističan za mobilne *Android* i *iOS* aplikacije.
- *sidemenu* – predložak sa uobičajenim izbornikom u lijevom djelu ekrana karakterističan za mobilne *Android* i *iOS* aplikacije. Otvara se povlačenjem sa lijeva na desno ili klikom na gumb poznat i kao *hamburger button*. (*Slika 25*)



Slika 25 - Vrste Ionic predložaka

Naredbom *ionic serve* pokrećemo aplikaciju u *default-nom web* pregledniku. Otvaranjem početne stranice najčeće na adresi: *http://localhost:8100* možemo biti sigurni da smo uspješno postavili *Ionic* radno okruženje *(Slika 26)* .

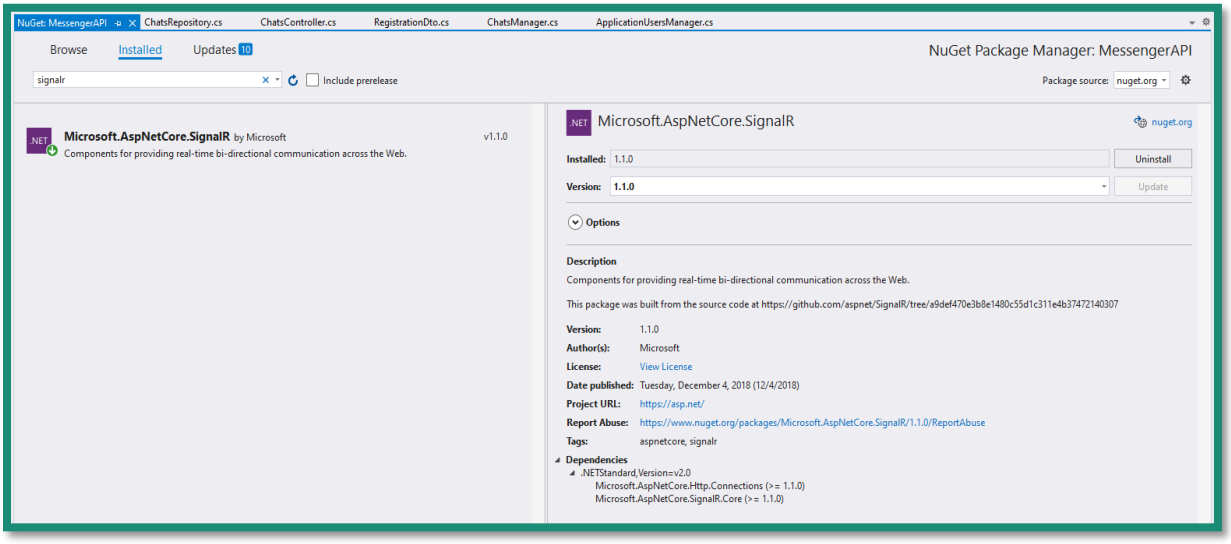


Slika 26 - Standardna stranica nove Ionic aplikacije

Bitno je još napomenuti kako će dodatni paketi potrebni za razvijanje *Rest API-a* i *Ionic* aplikacije biti objašnjeni prilikom analize značajki kod kojih su funkcionalnosti paketa korišteni.

10.2.3 Instalacija *SignalR* biblioteke

U našem slučaju *SignalR* biblioteku bilo je potrebno instalirati na *Rest API* servisu i u *Ionic* aplikaciji. Kako za oba radna okruženja postoje već prilagođeni *manager-i* za instalaciju biblioteka sama instalacija je vrlo jednostavna. Biblioteka je instalirana pomoću *Node Package Managera[17] (Slika 28)* sa klijentske strane aplikacije i pomoću *NuGet Package Managera[18] (Slika 27)* sa serverske strane aplikacije. Osim pokretanja jednostavnih jednolijiskih naredbi za instalaciju nije bilo potrebno dodatno konfiguriranje.



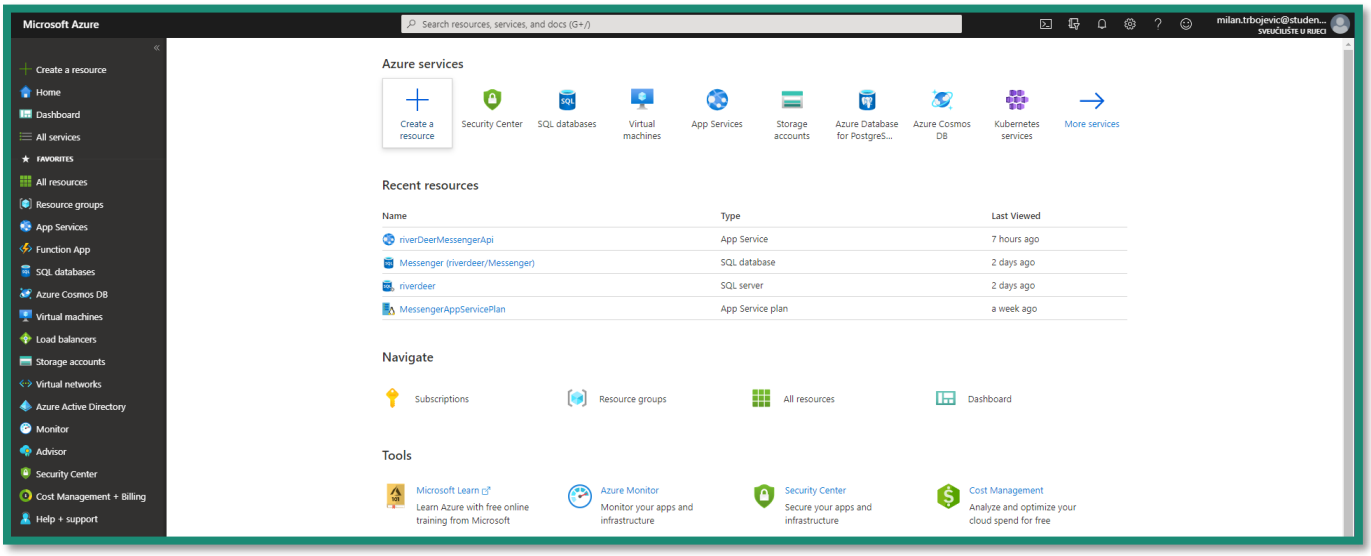
Slika 27 - Instalacija SignalR biblioteke putem NuGet Package Managera

```
+ signalr@2.4.1
added 2 packages from 2 contributors and audited 2 packages in 4.725s
found 0 vulnerabilities
```

Slika 28 - Poruka o uspješnosti instalacije SignalR biblioteke

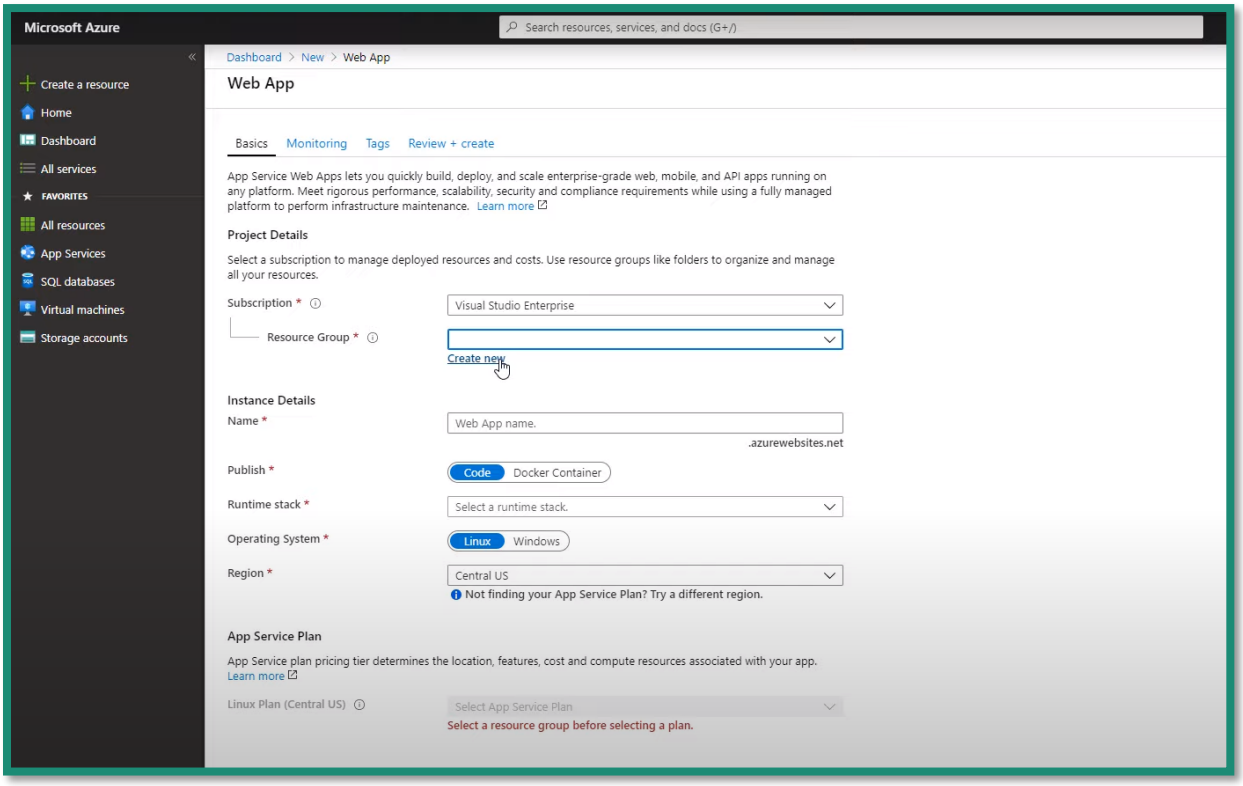
10.2.4 Konfiguriranje Rest API servisa na *Azure Cloudu*

Nakon prijave u sustav potrebno je još u obliku neke vrste dokumenta potvrditi identitet. To također može biti i kreditna kartica. Uz potvrdu identiteta još jednom stoji napomena kako na *Azure Cloud-u* ne postoji automatska naplata za korištenje usluga te da će nakon isteka besplatnog razdoblja biti ponuđeno produženje usluga, ali da se usluga neće automatski naplaćivati.



Slika 29 - Azure Portal

Potvrda identiteta traje nekoliko minuta. Nakon toga smo spremni za korištenje *Azure Cloud* usluga. Preusmjereni smo na *Azure Cloud Portal (Slika 29)*. Na portalu odabiremo vrstu proizvoda ili usluge koju želimo koristiti. Za našu *messenger* aplikaciju odabiremo *Web Service* i *SQL Database* uslugu.



Slika 30 - Proces postavljanja Rest API servisa na web server

Najprije je postavljen *Rest API* servis *(Slika 30)* tako što je odabrana *Web Service* uslugu na našem *Azure Cloud Portal-u*. Kada smo odabrali uslugu potrebno je specificirati naziv aplikacije, kojem skupu resursa će aplikacija pripadati i na kraju vrstu arhitekture *Cloud* servera. Kako bi iskoristili puni potencijal *.Net Core* radnog okruženja odabrali smo *Linux* server. Neki od razloga za odabir su:

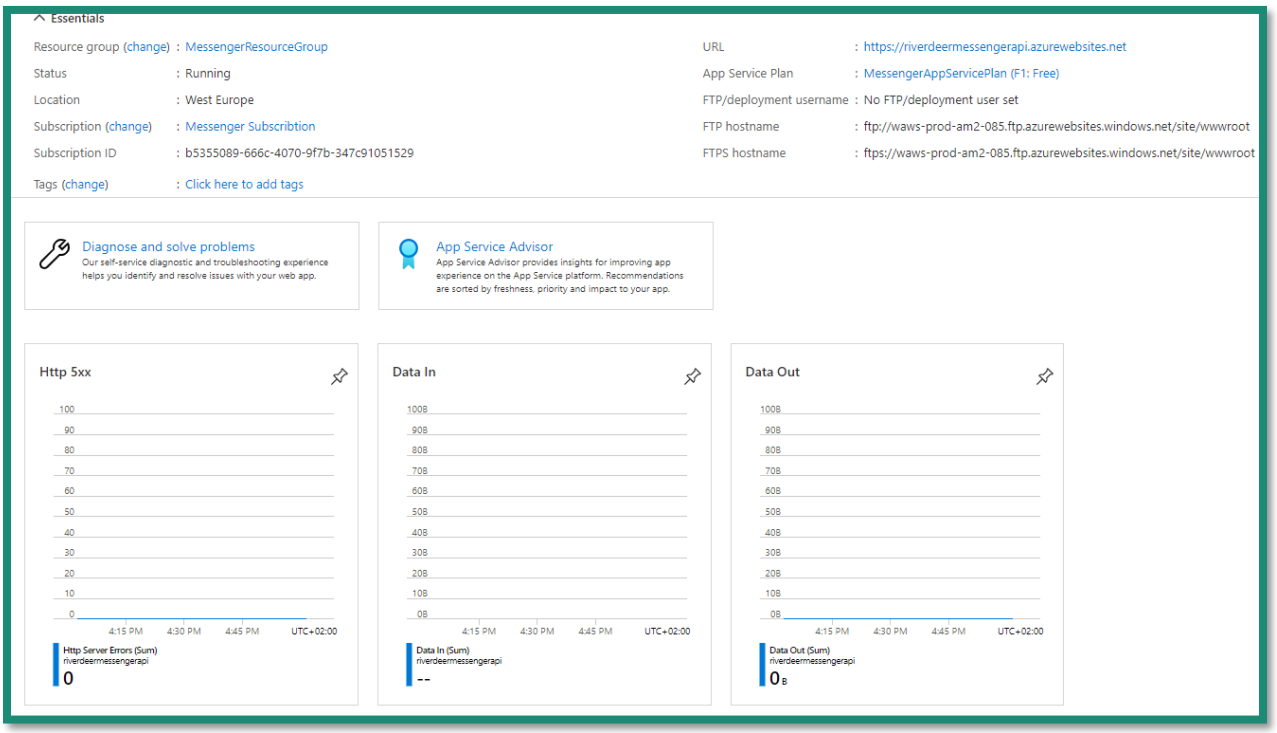
- Bolje performanse aplikacije – što nam je vrlo bitno zbog djeljive arhitekture sa drugim korisnicima koji također koriste besplatni plan *Azure Cloud* platforme.
- Veća portabilnost aplikacije – selidba na drugi *Linux* server, lakša je i jeftinija od one na *Windows* serveru.

Poslije odabira arhitekture proces pripreme okruženja u kojem će se nalaziti aplikacija traje nekoliko minuta. Po završetku procesa primjećujemo na našem portalu kada odaberemo *Web Service* uslugu da smo dobili i link na kojem se nalazi aplikacija. To je najčešće adresa u obliku [ime aplikacije].azurewebisites.net. Važno je napomenuti i da je zadano za aplikaciju da je zaštićena *SSL* certifikatom *[24]*.

Nakon što je postavljen Web Servis u *Microsoft Visual Studiu* postojeći profil povezujemo s *Azure Cloud-om* radi lakšeg ažuriranja aplikacije na produkciji. Prvo se moramo prijaviti sa istim *Microsoft-ovim* računom kao i na *Azure Cloud-u.* Nakon toga desnim klikom na ikonu aplikacije odabiremo opciju *Publish*.

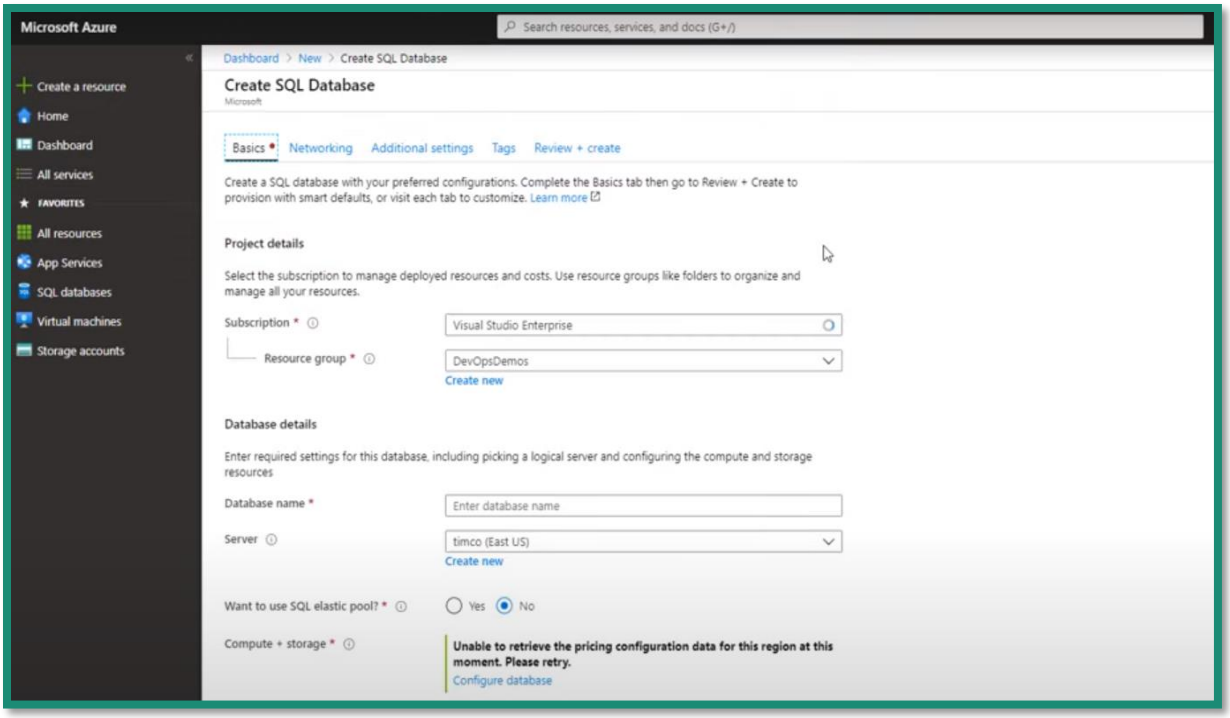
Nakon odabira stvaramo novi *Azure Cloud* profil. U podatcima upisujemo adresu aplikacije i povezujemo se sa *Azure Cloud* platformom.

Kada smo uspješno povezani tada klikom na *Publish* opciju, promjene na aplikaciji šaljemo na *Azure Cloud* odnosno produkcijsko okruženje aplikacije. U donjem prozoru ćemo dobiti sve potrebne informacije i poruke o uspješnosti/neuspješnosti procesa. Ukoliko je status *Successful* možemo biti sigurni da je aplikacija uspješno objavljena, što će potvrditi i automatsko otvaranje adrese aplikacije u zadanom *web* pregledniku našeg računala. Ako odemo na portal primjetiti ćemo dodatne grafikone koji sada prate aktivnosti našeg *Rest API* servisa *(Slika 31)*.



Slika 31 - Prikaz prometa Rest API servisa u obliku grafikona

Nakon postavljenog *Rest API* servisa potrebno je postaviti i *SQL* bazu podataka *(Slika 32)* kako bi serverska strana naše aplikacije u potpunosti funkcionirala. Nakon što odaberemo uslugu *SQL* baze podataka na našem portalu. Kao što je to bio slučaj sa *Web Service* uslugom, biramo naziv baze podataka te arhitekturu. Osim arhitekture odabire se i veličina diska za spremanje podataka. Kada smo odabrali željene opcije proces postavljanja moramo pričekati koju minutu.



Slika 32 - Konfiguriranje SQL baze podataka na SQL server

Kada je *SQL* baza podataka postavljena možemo se pripremiti na sinkronizaciju sa našom lokalnom bazom podataka. Postavljanje baze podataka možemo napraviti na 2 načina:

1. Postavljanje baze podataka kroz *Microsoft Visual Studio[25]* – postavljanje baze podataka vrlo je slično postavlju *Rest API* servisa. Gdje bazu podataka dodajemo kao zaseban projekt u naš *solution* gdje se nalazi i *Rest API* servis. Preko *Publish* opcije radimo spajanje na *Azure Cloud* i „objavljujemo“ shemu baze podataka na *Azure Cloud-u.*
2. Generiranje *SQL* skripte kroz *Microsoft SQL Server Managment Studio [26]* – pomoću ovog alata ili sličnog možemo izgenerirati *SQL* skriptu koja je zapravo *Create Query* ili upit koji stvara tablice za bazu podataka. Kada smo dobili skriptu možemo je kopirati i zalijepiti u *SQL Query* polje na *Azure Cloud SQL Database* usluzi i pokrenuti. Ako je sve u redu sa izgeneriranom skriptom. Ona bi trebala stvoriti sve tablice potrebne za rad aplikacije. Ovaj proces je nešto lakši i brži s obzirom na prvi proces ukoliko niste predhodno imali *SQL* i *Web API* projekt u *solution-u* svog *Rest API* servisa.

Za kraj jedna korisna napomena: prilikom generiranja skripte moguće je izgenerirati i *Insert* dio upita u kojem možemo imati i neke od lokalno spremljenih podataka, tako da kada izvršimo skriptu na *Azure Cloud-u* baza podataka nije prazna nego možemo imati testne podatke radi lakšeg tesitranja *messenger* aplikacije u cjelosti.