Razvoj mobilne i web aplikacije za pomoć kod razgledavanja kulturne baštine

Zubak, Ivan

Sveučilište u Rijeci – Odjel za informatiku
Diplomski studij, modul Informacijski i komunikacijski sustavi

Ivan Zubak

Razvoj mobilne i web aplikacije za pomoć kod razgledavanja kulturne baštine

Diplomski rad

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

Rijeka, prosinac 2019.

Sadržaj

1. Uvod

U diplomskom će radu, kao teorijska podloga, biti opisana osnovna arhitektura klijent – poslužitelj sustava i osnovne karakteristike mobilnih aplikacija. Naglasak diplomskog rada je na praktičnom dijelu koji se sastoji od opisa projekta, analize korištenih alata te opisa implementacije samog sustava uz prikaz programskog koda, slika zaslona i objašnjenja ključnih elemenata implementacije.

Cilj je ovog rada kroz implementaciju navedenog sustava proučiti i opisati proces izrade kompletnog programskog rješenja koje se sastoji od web aplikacije, mobilne aplikacije i aplikativnog programskog sučelja za komunikaciju između njih.

Očekivani rezultat diplomskog rada su web aplikacija te *Andriod* i *iOS* mobilne aplikacije. S obzirom na to da će biti korišten *cross-platform*[1] pristup za izradu mobilnih aplikacija, o mobilnim aplikacijama u daljnjem radu govorit će se u jednini.

Primjena sustava je da služi kao pomoć pri razgledavanja kulturne baštine, web aplikacija koja služi kao administracija sustava koristila bi se unutar turističke zajednica nekog grada ili regije, dok bi mobilna aplikacija bila namijenjena turistima koji posjećuju taj grad ili regiju, a žele na interaktivan način obići i objediniti sve pripadne lokalitete kulturne baštine. Detaljniji opis načina funkcioniranja sustava bit će opisan u poglavlju *O projektu*.

Odabrane tehnologije za razvoj navedenog sustava su:

- *ASP.NET Core* radno okruženje za pozadinski dio (engl. *backend*) i *Angular* za korisničko sučelje (engl. *frontend*) web aplikacije
- *Ionic* radno okruženje za razvoj hibridnih mobilnih aplikacija

S obzirom na to da se detaljno ne opisuju svi pojmovi vezani uz razvoj softvera za potrebe boljeg razumijevanja ovog rada potrebno je predznanje programerske paradigme, objektno-orijentiranog programiranja, razumijevanje funkcioniranja dinamičkih web aplikacija te mobilnih aplikacija.

[1] proizvod ili sustav koji radi na više različitih platformi ili operacijskih sustava, a koristi jedinstveni programski kod

2. Klijent – poslužitelj arhitektura

Web aplikacije se, prema osnovnoj strukturi, dijele na dvije skupine: *peer-to-peer* (P2P) i klijent – poslužitelj arhitektura.

Mnogo su popularnije web aplikacije temeljene na klijent – poslužitelj arhitekturi. Te aplikacije sastoje se od klijentskog i poslužiteljskog (eng. *Server*) dijela.

Arhitektura klijent – poslužitelj je arhitektura računalne mreže u kojoj mnogi klijenti traže i primaju uslugu od centraliziranog poslužitelja. Računala klijenta pružaju sučelje koje omogućuje klijentu da zahtijeva usluge poslužitelja i da prikazuje rezultate koje poslužitelj vraća. Poslužitelji čekaju da zahtjev stigne od klijenta, a zatim odgovore na zahtjev prema definiranim protokolima za obradu zahtjeva [1].

U ovoj je arhitekturi klijent taj koji inicira komunikaciju s poslužiteljem. Klijentski se dio aplikacije nalazi na korisnikovom uređaju (mobitel, tablet, stolno računalo…) u obliku web preglednika. Poslužiteljski dio web aplikacije u većini slučajeva radi na snažnijim računalima koja su stalno spojeni na mrežu uz maksimalnu dostupnost.

Distribuirana arhitektura *peer-to-peer* mreža se sastoji od sudionika koji dijele dio svojih resursa (kao što su primjerice tvrdi disk ili memorija) koje je moguće iskoristiti međusobnim povezivanjem. Ti su mrežni resursi dostupni drugim sudionicima mreže bez potrebe za središnjim upravljačkim jedinicama kao što su poslužitelji. Sudionici mreže su ravnopravni, tj. svi sudionici posjeduju jednaka prava uzimanja i davanja resursa [2].

Za potrebe ovog rada korisnije su web aplikacije temeljene na arhitekturi klijent – poslužitelj.

Razvoj web aplikacija u arhitekturi klijent – poslužitelj može se podijeliti na dvije poznate cjeline:

- programiranje na strani poslužitelja (eng. *Backend development*) – razvoj svih komponenti potrebnih za rad aplikacije na poslužitelju
- programiranje na strani klijenta (eng. *Frontend developement*) – razvoj svih komponenti potrebnih za prikaz aplikacije krajnjem korisniku.

Kroz projektni dio diplomskog rada proći će se proces izrade web aplikacije kroz obje cjeline.

3. Mobilne aplikacije

Mobilna aplikacija, najčešće zvana samo aplikacija, vrsta je aplikacijskog softvera dizajniranog za pokretanje na mobilnom uređaju kao što je pametni telefon ili tablet. Mobilne aplikacije su najčešće male, samostalne softverske jedinice s ograničenom funkcijom [3].

S obzirom na stanje na tržištu kada govorimo o razvoju aplikacija za mobilne operativne sustave dovoljno je zadržati se na razvoju mobilne aplikacije za *Android* i *iOS*. Tržišni udio *Android* operativnog sustava u studenom 2019. iznosi 75.3%, dok tržišni udio *iOS* operativnog sustava iznosi 22.9%. Sljedeći je *KaiOS* sa tek 0.49% [4].

Do nedavno u obzir se mogao uzeti i *Windows Mobile* operativni sustav, no *Microsoft* je 2019. napustio projekt razvijanja ovog operativnog sustava te se od prosinca 2019. gasi podrška za *Windows Mobile* uređaje [5].

U kontekstu razvijanja mobilnih aplikacija, kada se žele podržati i *Android* i *iOS* operativni sustavi postoje tri opcije [6]: nativni pristup, hibridni pristup ili razvoj mobilne web aplikacije (Slika 1).

Slika 1 - vrste mobilnih aplikacija
(https://qph.fs.quoracdn.net/main-qimg-5e57fd09720d68cdac71fe9c5139c4f4)

- **Nativne[2] aplikacije** – nativne su aplikacije dizajnirane da budu izvorne jednom operativnom sustavu. Nativne aplikacije funkcioniraju brzo i intuitivno jer su razvijene specifično za platformu.
    - Prednosti:
        - nativne aplikacije rade najbrže
        - lako se distribuiraju putem trgovina aplikacijama (*Google Play*, *Apple Store*)
        - interaktivne i intuitivne
        - lako se pristupa svim značajkama uređaja.
    - Mane:
        - razvijene za jednu platformu
        - skupe su za razvoj i održavanje

[2] Izvorne

- **Hibridne aplikacije** – tehnologija hibridnih aplikacija funkcionira na način da se razvija web aplikacija koja se pokreće u komponenti koja se koristi za prikazivanje web sadržaja unutar nativne aplikacije – *WebView*. Hibridne se aplikacije razvijaju koristeći standardne web tehnologije.
    - Prednosti:
        - brzo i lako se izrađuju zbog korištenja web tehnologija
        - jedna aplikacija za obje platforme
        - mogu pristupiti većini značajki uređaja
        - mogu se distribuirati putem trgovina aplikacijama.
    - Mane:
        - lošije performanse od nativnih aplikacija.

- **Mobilne web aplikacije** – responzivne[3] web stranice mijenjaju dizajn kada im se pristupa preko pretraživača mobilnog uređaja.
    - Prednosti:
        - jedna aplikacija za web stranicu i mobilnu aplikaciju
        - nema potrebe za instalacijom aplikacije
        - najlakše im se pristupa (www)
        - uvijek ažurirane.
    - Mane:
        - ovisne o brzini interneta
        - najlošiji performans
        - nemogućnost pristupa značajkama mobilnog uređaja.

Mobilne web aplikacije nadilaze svoje mane pretvarajući se u Progresivne Web Aplikacije (*PWA*), no u kontekstu ovog rada neće biti riječ o njima.

Kroz projektni dio diplomskog rada proći će se proces izrade hibridne mobilne aplikacije koristeći *Ionic* radni okvir.

[3] Prilagođene različitim veličinama ekrana

4. Opis projekta

Projektni je dio diplomskog rada izrada web aplikacije te *Andriod*/*iOS* mobilnih aplikacija koje služe kao pomoć pri razgledavanju lokaliteta kulturne baštine.

Web aplikacija je aplikacija koja, zaštićena prijavom administratora (npr. djelatnik turističke zajednice), omogućuje kreiranje kategorija kulturne baštine (npr. crkve, katedrale, spomenici…) i kreiranje lokaliteta kulturne baštine (npr. katedrala svetog Vida). Lokalitet se sastoji od informacija kao što su ime lokaliteta, kategorija lokaliteta, kraći opis, detaljan opis, geografska lokacija i slike. Web aplikacija omogućuje kreiranje, brisanje i uređivanje informacija o kategorijama i lokalitetima te učitavanje slika. Za pojedini lokalitet kreira se *QR kod* koji je potrebno skenirati kroz mobilnu aplikaciju kako bi se dobile informacije o lokalitetu.

Mobilna aplikacija omogućuje registraciju i prijavu korisnika, pregled do sada neotkrivenih lokaliteta, skeniranje *QR koda*, navigaciju te pregled otkrivenih lokaliteta.

Mobilna aplikacija funkcionira na način da ju korisnik (turist) instalira na svoj uređaj te se registrira. Prilikom ulaska u mobilnu aplikaciju korisnik vidi kartu regije u kojoj se nalazi te na njoj lokalitete koje do sada nije otkrio. Klikom na pojedini lokalitet korisnik može koristeći svoju zadanu aplikaciju za navigaciju (npr. *Google Maps* na *Android* uređajima ili *Maps* na *iOS* uređajima) otići do pojedinog lokaliteta gdje će pronaći *QR kod* te ga kroz aplikaciju skenirati. Skeniranjem *QR koda* korisnik otkriva pojedini lokalitet. Prije skeniranja korisnik o lokalitetu ima informacije o tome kako se lokalitet zove, gdje se nalazi i glavnu sliku tog lokaliteta. Otkrivanjem *QR koda* korisnik uz prethodne informacije dobiva kratki opis lokaliteta, detaljan opis lokaliteta i sve slike. Korisnik mobilne aplikacije, osim otkrivanja novih lokaliteta, može pregledavati lokalitete koje je do sada otkrio.

Smisao projekta je zamišljen na način da je posjetitelju nekog grada ili regije u cilju, kroz interaktivan način - „moraš skupit sve“, obići lokalitete kulturne baštine grada ili regije u kojoj se nalazi.

Također, u sklopu web aplikacije administrator ima mogućnost pregleda osnovne statistike nad korisnicima mobilnih aplikacija i lokaliteta – broj lokaliteta, broj korisnika, broj korisnika po državama, godinama i spolu. Koliko je puta koji lokalitet otkriven po mjesecima u aktivnoj godini te koliko je puta koji lokalitet otkriven općenito.

5. Korištene tehnologije

U ovom će poglavlju biti opisane tehnologije korištene za razvoj ovog sustava, to su programski okviri *ASP.NET Core* i *Angular* za razvoj web aplikacije te *Ionic* za razvoj mobilne aplikacije (Slika 2).

U nastavku su definirani osnovni elementi svake od korištenih tehnologija uz jednostavne primjere kako bi se razumjelo kako koji programski okvir u načelu funkcionira. Složeniji će primjeri, potrebni za implementaciju ovog sustava, biti pojašnjeni kroz primjer implementacije sustava.

Korištene verzije programskih okvira su *.NET Core 2.2*, *Angular 8* i *Ionic 5.4*.

Slika 2 - korištene tehnologije

5.1. ASP.NET Core

ASP.NET Core [7] je *cross-platform* razvojni okvir visokih performansi i otvorenog koda, koju održavaju *Microsoft* i *.NET* zajednica na *GitHubu*. *ASP.NET Core* koristi se za izradu modernih web aplikacija. Koristeći *ASP.NET Core* mogu se razvijati web aplikacije i servisi, Internet stvari (eng. *Internet of Things*) aplikacije i *backend* sustavi za mobilne aplikacije. S obzirom na to da je ASP.NET Core cross-platform možemo razvijati aplikacije na *Windows*, *macOS* ili *Linux* operativnim sustavima.

Prednosti *ASP.NET Core* razvojnog okvira su:

- jedinstven pristup razvoju web aplikacija i aplikativnog programskog sučelja (eng. *API – Application Programming Interface*)
- *Razor pages* omogućuju pisanje *C#* koda na *HTML* stranicama
- mogućnost razvijanja i pokretanja na *Windows*, *macOS* i *Linux* operativnim sustavima
- otvoreni kod i pristupačna zajednica
- jednostavna integracija s modernim, klijentskim radnim okruženjima i bibliotekama (*Angular*, *React*, *Vue*)
- lagan, vrlo učinkovit i modularan *HTTP* cjevovod

- mogućnost hostinga na sljedećim poslužiteljima: *Kestrel, IIS, HTTP.sys, Nginx, Apache, Docker*
- mogućnost korištenja *ASP.NET Core MV*C za izradu web aplikacija i API-ja

ASP.NET Core MVC koristi *MVC* obrazac koji olakšava testiranje i ubrzava razvoj web aplikacija.

5.1.1. MVC obrazac (*Model – View - Controller*)

Arhitektonski uzorak (eng. *Architectural pattern*) MVC [8] razdvaja aplikaciju u tri cjeline: *Models, Views* i *Controllers*. Ovaj obrazac omogućuje da se postigne razdvajanje problema. Pomoću ovog obrasca korisnički zahtjevi (eng. *Requests*) preusmjeravaju se na kontroler koji je odgovoran za rad s modelom kako bi dohvatio rezultat upita. *Controller* odabire *View* koji će se prikazati korisniku i pruža mu sve podatke modela koji su mu potrebni.

Model u MVC aplikaciji predstavlja svu logiku povezanu s podacima s kojima korisnik radi. Na primjer, objekt Kupac će dohvatiti informacije o kupcu iz baze podataka, manipulirati tim podacima pa proslijediti na renderiranje ili ažurirati i vratiti natrag u bazu podataka. U *ASP.NET Core MVC* aplikacijama *Model* se koristi usko vezano s *Entity Framework* programskim okvirom. *Entity Framework* je *object-relational mapper*[4] koji omogućuje da se radi s bazom koristeći *.NET* objekte. On eliminira potrebu za pisanjem većine programskog koda namijenjenog za pristup podacima u bazi kojeg bi programeri inače morali pisati.

View komponenta se koristi za prikaz korisničkog sučelja, *View* komponenta renderira podatke te ih prikazuje krajnjem korisniku.

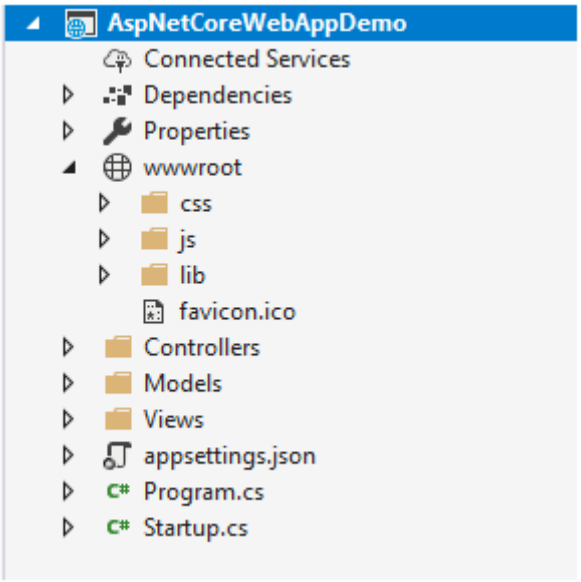
Controller funkcionira kao posrednik između *Model* i *View* komponenti, on procesira zahtjeve koji dolaze na poslužitelja, dohvaća podatke iz modela te ih prosljeđuje na pripadajuću *View* komponentu.

Bitno je naglasiti da kada se koristi *ASP.NET MVC* pristup s nekim od klijentskih radnih okruženja kao što je *Angular* onda *View* komponenta nije potrebna. Interakcija s korisnikom vrši se kroz *Angular* aplikaciju koja kroz zahtjeve prema *Controller*-u dohvaća potrebne podatke preko *Model* komponente. Odnosno, poslužiteljski dio web aplikacije funkcionira kao API koji vraća zahtijevane podatke u željenom formatu (najčešće *JSON*).

[4] Programski alat za pretvaranje podataka između nekompatibilnih tipova podataka pomoću objektno-orijentiranih programskih jezika

5.1.2. Struktura projekta

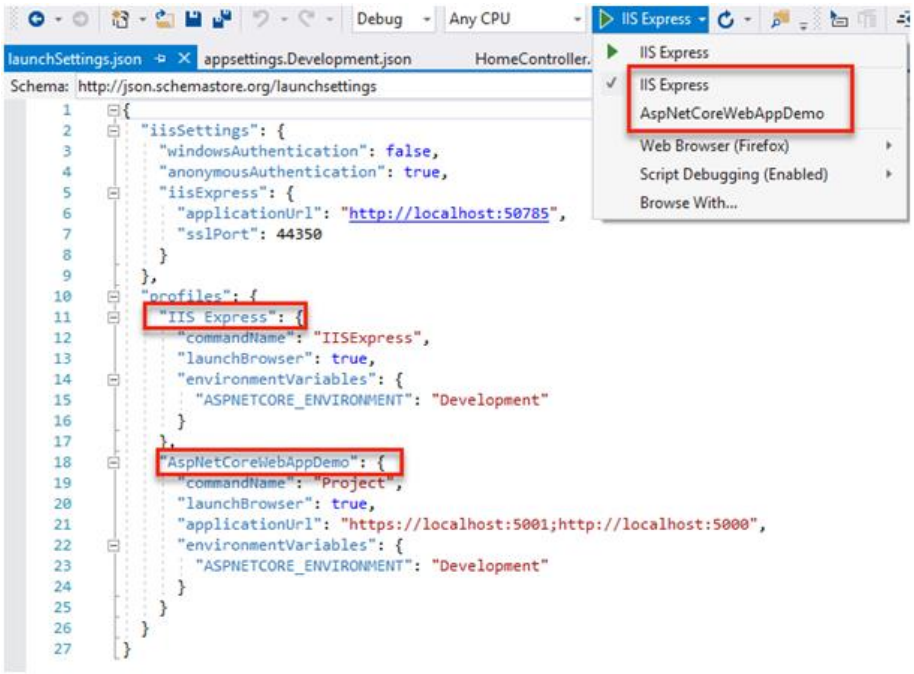
Kada generiramo *ASP.NET Core* aplikaciju koristeći *Visual Studio* dobije se osnovna struktura projekta [9] (Slika 3).



Slika 3 - struktura ASP.NET Core MVC aplikacije

Direktorij *Dependencies* sadrži sve potrebne *.dll* datoteke biblioteka o kojima je aplikacija ovisna kako bi mogla funkcionirati.

U direktoriju *Properties* nalazi se datoteka *launchSettings.json* (Slika 4) koja opisuje kako se projekt pokreće te zasebnu konfiguraciju prilikom pokretanja različitih servera.



Slika 4 - sadržaj launchSettings.json datoteke

Direktorij *wwwroot* sadrži statični sadržaj kao što je *CSS, Javascript* datoteke, *Bootstrap* i ostale *biblioteke*. Bitno je napomenuti da kada se aplikacija izrađena u *Angular* programskom okviru priprema za produkcijsko okruženje sav razvojni kod se pretvara u četiri *javascript* skripte i jednu *index.html* datoteku koja se postavlja u *wwwroot* direktorij.

Controller, *Models* i *Views* direktoriji sadrže pripadne klase i datoteke. U projektu koji radi s *Angular* programskim okvirom *View* direktorij se može obrisati.

Datoteka *appsettings.json* koristi se za spremanje informacija o podacima za spajanje na bazu, API ključeva te ostalih podataka o postavkama aplikacija.

I na kraju, dvije klase *Program.cs* i *Startup.cs*. *Program.cs* (Slika 5) je ulazna točka aplikacije, ona pokreće *Startup.cs (Slika 6)* klasu u kojoj se odrađuje definirana konfiguracija aplikacije.

```
namespace AspNetCoreWebAppDemo
{
    public class Program
    {
        public static void Main(string[] args)
        {
            CreateWebHostBuilder(args).Build().Run();
        }

        public static IWebHostBuilder CreateWebHostBuilder(string[] args) =>
            WebHost.CreateDefaultBuilder(args)
                .UseStartup<Startup>();
    }
}
```

Slika 5 - Program.cs klasa

```
namespace AspNetCoreWebAppDemo
{
    public class Startup
    {
        public Startup(IConfiguration configuration)...

        public IConfiguration Configuration { get; }

        // This method gets called by the runtime. Use this method to add services to the container.

        public void ConfigureServices(IServiceCollection services)...

        // This method gets called by the runtime. Use this method to configure the HTTP request pipeline.

        public void Configure(IApplicationBuilder app, IHostingEnvironment env)...
    }
}
```

Slika 6 - Startup.cs klasa

ConfigureServices metoda služi za definiranje servisa. Pomoću njega se aktiviraju klase koje se žele koristiti bilo gdje u aplikaciji. U drugoj će klasi biti potrebno dodati tu klasu u konstruktoru i moći će se koristiti.

Configure metoda se koristi za specificiranje kako aplikacija reagira na HTTP zahtjeve. Npr. u *Configure* metodi će se definirati zadani *Controller*, koristi li web aplikacija autentifikaciju, koristi li statične datoteke, *CORS*[5] pravila i slično.

Zadana ruta *MVC* aplikacije je *Index* metoda u *HomeController*-u jer je tako definirano u *Configure* metodi (Slika 7). Parametrom *template* u konfiguraciji rute konfiguriramo putanju do metoda u *Controlleru*.

```
app.UseMvc(routes =>
{
    routes.MapRoute(
        name: "default",
        template: "{controller=Home}/{action=Index}/{id?}");
});
```

Slika 7 - definicija zadane rute

Kada se korisnik navigira na */Home/Index .NET Core* će automatski pozvati metodu *Index* u *HomeController* kontroleru.

Index metoda u *HomeController*-u, prikazana na slici 8, vraća *View Index.cshmtl* koji se može naći na putanji */Views/Home/*.

```
namespace AspNetCoreWebAppDemo.Controllers
{
    public class HomeController : Controller
    {
        public IActionResult Index()
        {
            return View();
        }

        public IActionResult Privacy()...

        [ResponseCache(Duration = 0, Location = ResponseCacheLocation.None, NoStore = true)]
        public IActionResult Error()...
    }
}
```

Slika 8 - HomeController

Kada se *ASP.NET Core* aplikacija koristi s *Angular* programskim okvirom za klijentski prikaz kao zadana putanju definirat će se *Controller* koji pokazuje na *wwwroot* gdje se nalazi *Angular* aplikacija pripremljena za produkcijsko okruženje.

[5] *Cross-Origin Resource Sharing* je mehanizam koji koristi dodatna HTTP zaglavlja kako bi se pretraživačima dalo do znanja da daju aplikaciji koja pripada jednoj domeni pristup odabranim resursima s druge domene..

5.2. Angular

Angular [10] je programski okvir otvorenog koda za razvoj klijentskih aplikacija koristeći *HTML* i *Typescipt* koju održavaju *Google* i *Angular* zajednica na *GitHubu*. *Angular* aplikacija temelji se na SPA (*Single-page application*) arhitekturi što znači da tijekom interakcije s korisnikom trenutna stranica se dinamički prepisuje (eng. *rewrite*) umjesto da se učitava nova stranica kao što je to slučaj u klasičnom pristupu.

Typescript [11] je programski jezik razvijen od strane *Microsofta*, *Typescript* je nadskup *Javascripta* koji dodatno omogućuje statično pisanje (eng. *Static Typing*), klase i sučelja (eng. *interface*).

Kao što je već napomenuto u prethodnom poglavlju, kada se *Angular* aplikacija priprema za produkcijsko okruženje sav programski kod se pretvara u *Javascript*.

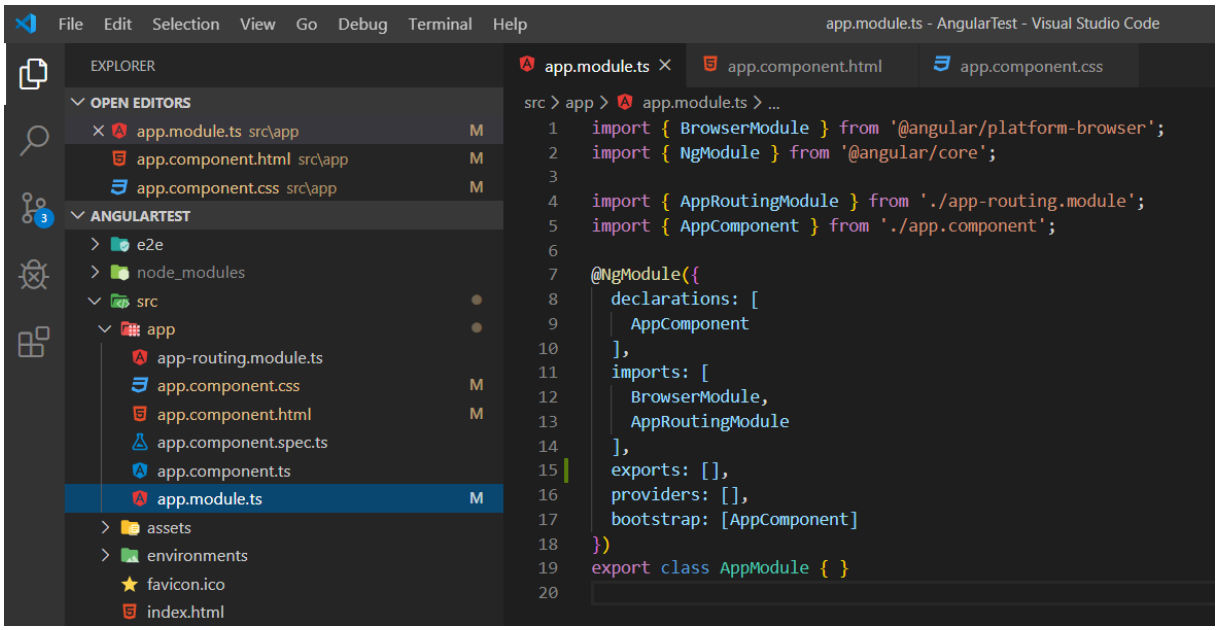
5.2.1. Angular arhitektura

Za potrebe izrade *Angular* aplikacije potrebno je poznavati osnovne *Angular* elemente [12]: module (eng. *Modules*), komponente (eng. *Components*), servise (eng. *Services*) i usmjeravanje (eng. *Routing*).

Najvažniji element *Angular* aplikacije su moduli. Oni se koriste za grupiranje povezanih komponenti i servisa. Ono što *Typescript* klasu čini modulom je definicija metapodataka koji se definira pomoću @ngModule dekoratora. ***ngModule*** je dekoratorska funkcija koja uzima objekt čija svojstva opisuju modul. Neki od najvažnijih svojstava kojima se opisuje modul su:

- *declarations* – klasa prikaza (eng. *view classes*) koje pripadaju modulu; klase prikaza mogu biti komponente, direktive i cijevi
- *exports* – deklaracije koje trebaju biti vidljive i upotrebljive u predlošcima komponenata iz drugih modula
- *imports* – moduli čije su izvozne klase (eng, *exports*) potrebne predlošcima komponenata deklariranih u ovom modulu
- *providers* – popis servisa (*services*) koje ovaj modul pruža.

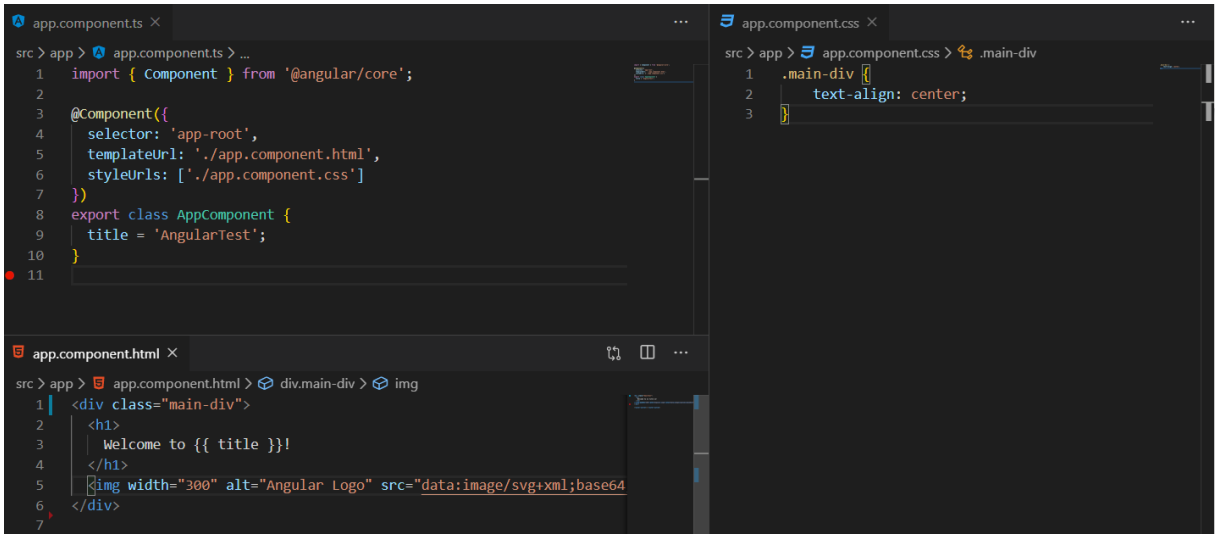
Slika 9 prikazuje *app.module.ts* datoteku, *app.module.ts* datoteka predstavlja korijenski modul *Angular* aplikacije.



Slika 9 - app.module.ts datoteka

Komponente su osnovni građevni blokovi (eng. *Building blocks*) *Angular* aplikacije. Svaka komponenta definira klasu koja sadrži logiku u *Typescript* formatu, pogled u *HTML* predlošku i pripadni *CSS* za taj *HTML* predložak. Svaka *Angular* aplikacija sastoji se mnogo komponenti koje su ili međusobno povezane ili samostalno odrađuju definiranu logiku. *Angular* automatski stvara, osvježava i uništava komponente kako se korisnik kreće kroz aplikaciju.

Slika 10 prikazuje korijensku komponentu s pripadnom *Typescript* klasom te pripadnim *HTML* predloškom i *CSS*-om.

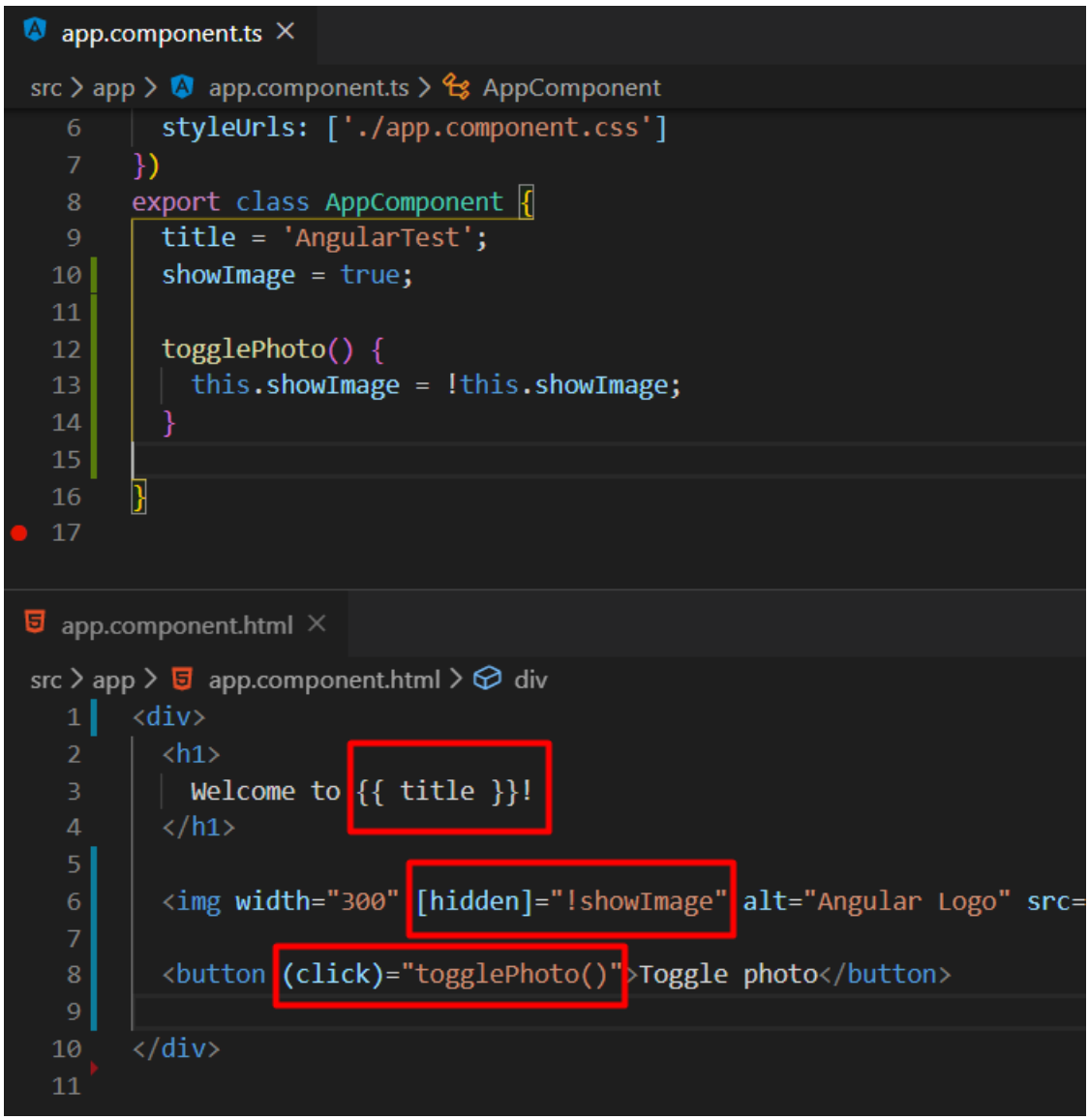


Slika 10 - korijenska komponenta s pripadnom Typescript klasom, HTML predloškom i CSS-om

Ono što *Typescript* klasu čini komponentom je definicija metapodataka koji se definiraju pomoću *@Component* dekoratora čiji objekt sadrži svojstva: *selector*, *templateUrl* i *styleUrl*. Kada unutar *HTML* datoteke, modulom povezane, komponente pozovemo *<app-root></app-root> Angular* će na to mjesto inicijalizirati komponentu s *app-root selectorom*. *templateUrl* i *styleUrl* su poveznice na pripadne *HTML* odnosno *CSS* datoteke.

Govoreći o komponentama, svakako je bitno naglasiti dva pojma koji omogućuju manipulaciju prikaza podataka unutar *HTML* dokumenta: vezanje podataka (eng. *Data binding*) i direktive (eng. *Directives*)

Vezanje podataka je mehanizam koji omogućuje povezivanje *Typescript* varijabli s *HTML* predloškom. Vezanje podataka postiže se korištenjem posebnih oznaka koje *Angular* prepoznaje.



Slika 11 - primjer vezanja podataka

Na slici 11 vidimo načine vezanja podataka:

- u komponenti postoji varijabla title, što znači da se u *HTML* predlošku te komponente može ispisati ta varijabla koristeći znakove za interpolaciju {{ *imeVarijable* }}; prilikom prikazivanja ove komponente u pregledniku ispisat će se naslov: *Welcome to AngularTest*
- u komponenti postoji varijablu *showImage*, što znači da se u *HTML* predlošku komponente koristiti ta varijabla; u ovom slučaju *hidden* svojstvo *img* elementa ovisi o

vrijednosti *showImage* varijable; prilikom prikazivanja ove komponente u pregledniku slika neće imati *hidden* svojstvo

- u komponenti postoji funkciju *toggleImage()* što znači da se može vezati klik događaj (eng. *Click event*) uz funkciju; prilikom klika na tipku slika će se sakriti.

Direktive u smislu manipulacije podacima u *HTML* predlošku mogu biti strukturalne i atributne. Strukturalne direktive mijenjaju izgled stranice tako što dodaju ili brišu elemente. Najkorištenije strukturalne direktive su **ngIf* i **ngFor*.

app.component.ts

```
src > app > app.component.ts > ...
1   import { Component } from '@angular/core';
2
3   @Component({
4     selector: 'app-root',
5     templateUrl: './app.component.html',
6     styleUrls: ['./app.component.css']
7   })
8   export class AppComponent {
9     humans = [{name: 'Ivan', age: 23}, {name: 'Ana', age: 24}, {name: 'Josip', age: 36}];
10  }
```

app.component.html

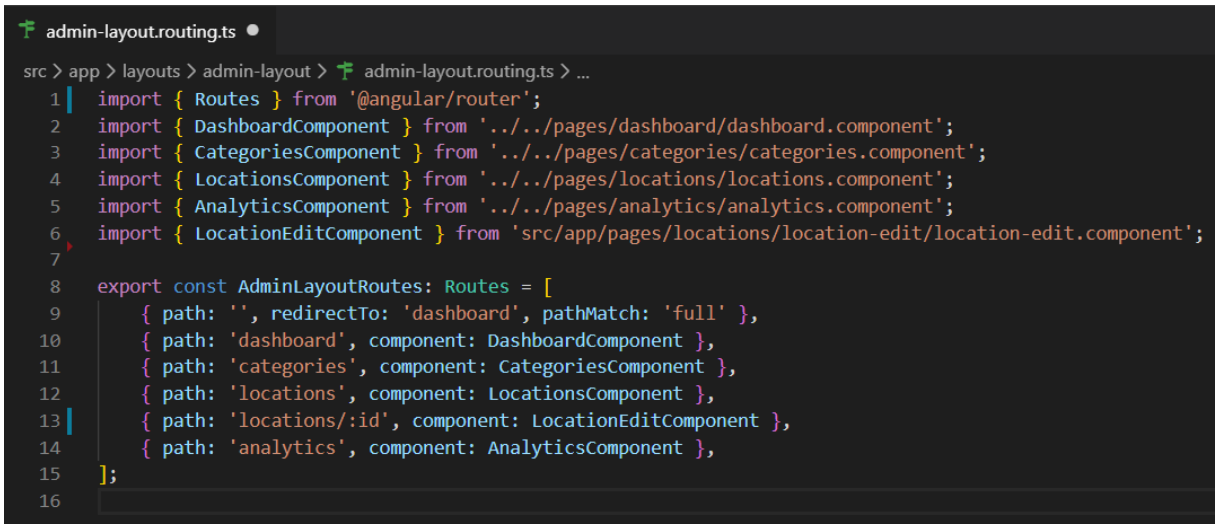
```
src > app > app.component.html > ...
1   <div *ngIf="humans">
2     <ul>
3       <li *ngFor="let human of humans">
4         {{ human.name }}, {{ human.age }}
5       </li>
6     </ul>
7   </div>
8
```

Slika 12 - strukturalne direktive

HTML predložak na slici 12 prikazuje upotrebu strukturalnih direktiva. **ngIf*, u ovom primjeru, provjera postoji li objekt *humans*, ako postoji prikazat će vezani * * element. **ngFor* direktiva ponovit će *<li>* element ovisno o broju elemenata polja *humans* te ispisati ime i godine svakog elementa polja. Atributne direktive utječu na izgled ili ponašanje postojećih elemanata. Atributne direktive **ngStyle* i **ngClass* dodjeljuju stil odnosno klasu pripadajućem elementu.

Servisi služe kako bi aplikaciji omogućili funkcionalnosti koje prema konvenciji ne pripadaju komponentama, npr. logika za dohvaćanje podataka sa servera ili zapisivanje poruka o greškama u konzolu. Iako programera ništa ne sprječava da napravi cijelu aplikaciju bez servisa to se ne preporuča jer komponente ne bi trebale raditi ništa više nego li omogućiti interakciju između korisnika i logike aplikacije. Ne koristeći servise *Angular* aplikacija gubi na modularnosti.

Usmjeravanje služi za navigaciju unutar *Angular* aplikacije. *Angular* može interpretirati *URL* adresu Internet preglednika te iskoristiti tu informaciju za navigaciju do definirane komponente.



Slika 13 - primjer klase usmjeravanja

Na slici 13 prikazan je jednostavan primjer definiranih ruta *Angular* aplikacije. *Routes* polje sastoji se od objekata koji definiraju rute. Kada se korisnik navigira na stranicu čija putanja glasi npr. http://localhost/categories *Angular* će pokrenuti *CategoriesComponent* komponentu unutar *<router-outlet></router-outlet>* elementa pripadajućeg modula.

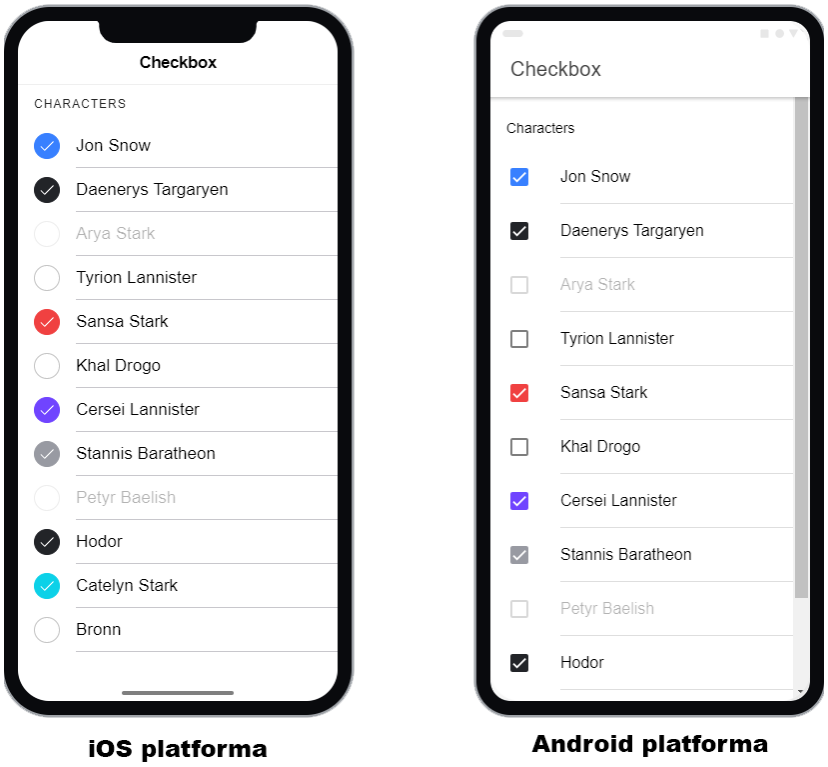
Poziv za navigaciju unutar komponenti može se definirati ili u *Typescript* klasi (*this.router.navigate(['/login']);*) ili u *HTML* predlošku (*Categories*).

Prazna putanja (*path: ' '*) predstavlja logiku prema kojoj će zadana putanja biti postavljane na određenu komponentu npr. *http://localhost/* će u ovom slučaju odvesti korisnika na *DashboardComponent* komponentu. Također, može se definirati i *wildcard*[6] putanja koja se aktivira kada putanja ne zadovoljava niti jednu od definiranih ruta. *wildcard* putanja (*path: '**'*) često se koristi za „404 – stranica nije pronađena“ slučajeve.

[6] Bilo koja

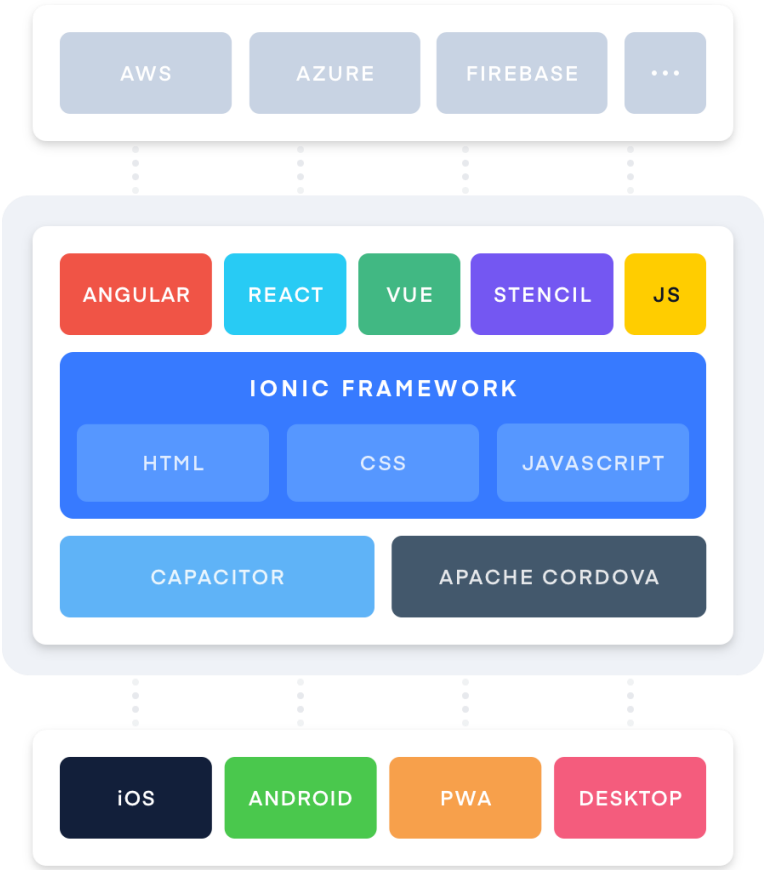
5.3. *Ionic*

Ionic [13] programski je okvir otvorenog koda za razvijanje hibridnih mobilnih aplikacija. *Ionic* pruža alate i komponente za izgradnju mobilnog sučelja s nativnim izgledom. Definicija nativnog izgleda znači da komponente koje iza sebe imaju isti programski kod drugačije će izgledati ovisno o platformi na kojoj se pokreću (slika 14).



Slika 14 - izgled Ionic komponenti na različitim operativnim sustavima (https://ionicframework.com/docs/api/checkbox)

Prema službenoj *Ionic* dokumentaciji [14], definicija *Ionic* programskog okvira kaže: *Ionic* je *HTML5* programski okvir za razvoj hibridnih mobilnih aplikacija. Zamislite *Ionic* kao *frontend* programski okvir koji upravlja izgledom, dojmom i interaktivnim sučeljima kako bi vaša aplikacija radila i izgledala onako kako bi trebala. *Ionic* možete zamisliti kao „*Bootstrap* za nativno“, ali s podrškom za širok spektar uobičajenih mobilnih komponenti, glatkih animacija i prekrasnog dizajna. Shema *Ionic* programskog okvira prikazana je na slici 14.



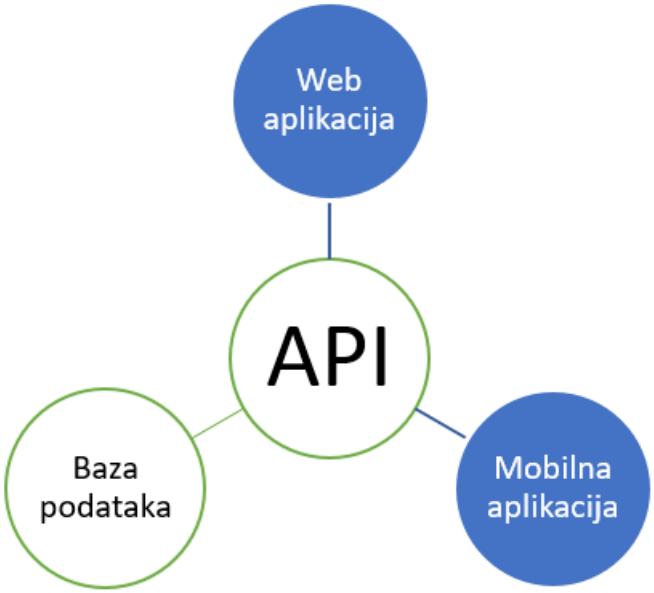
Slika 15 - Ionic framework
(https://ionicframework.com/)

Ionic radno okruženje [14] funkcionira sa svim *Javascript* radnim okruženjima. *Ionic* radno okruženje zahtjeva nativni omotač (eng. *Native Wrapper*) kako bi mogao raditi na mobilnim uređajima. Najzastupljeniji nativni omotač je *Apache Cordova*. *Cordova* funkcionira na način da uzima web aplikaciju i prikazuje je unutar nativne *WebView* komponente. *WebView* je komponenta aplikacije kao što je na primjer tipka ili tekst koja se koristi za prikazivanje web sadržaja unutar nativne aplikacije. Na *WebView* možemo gledati kao na pretraživač bez ijednog standardnog elementa korisničkog sučelja koji se može navigirati samo unutar definirane domene. Web aplikacija koje se pokreće unutar *WebViewa* je poput bilo koje druge aplikacije koja bi se izvodila unutar mobilnog preglednika – može otvoriti dodatne *HTML* stranice, izvršiti *Javascript* kod, komunicirati s udaljenim poslužiteljima i slično.

Da bi *Ionic* aplikacija pristupila nativnim komponentama kao što su kamera, lokacija, slanje SMS-a i slično mora se razviti *Cordova* dodatak (eng. *Cordova plugin*) koji se sastoji od implementacije tog nativnog koda u svakoj od platformi koja se želi pokriti te *Javascript* sučelja koje izvršava funkcije nativnog koda. Na sreću, zajednica koja se okupila oko *Cordove* je jako velika te postoje brojni gotovi dodaci za gotovo svaku nativnu značajku. Za razvoj hibridne mobilne aplikacije koristit će se *Angular* kao *Javascript* radno okruženje te *Apache Cordova* za pokretanje aplikacije na Android, odnosno *iOS* uređajima te za pristup nativnim komponentama (*QR kod* i pokretanje zadane aplikacije za navigaciju).

6. Arhitektura sustava

Sustav se sastoji od tri komponente: web aplikacije, mobilne aplikacije i aplikativnog programskog sučelja za komunikaciju s bazom podataka.



Slika 16 - arhitektura sustava

Slika 16 prikazuje kako su komponente međusobno povezane. Komunikacija između komponenti je dvostrana, odnosno i web i mobilna aplikacija mogu dohvaćati i ažurirati podatke iz baze podataka. Manipulacija nad podacima ograničena je skupom metoda unutar API-ja. Metode unutar API-ja zaštićene su autentifikacijom pomoću korisničkog imena i lozinke.

Administrator (korisnik koji pristupa preko web aplikacije) ima definirane metode za:

- autentifikaciju – prijava (POST)
- kategorije – kreiranje (POST), dohvaćanje svih (GET), ažuriranje (PUT) i brisanje (DELETE) kategorija
- lokalitete - kreiranje (POST), dohvaćanje svih (GET), dohvaćanje jednog (GET), ažuriranje (PUT), brisanje (DELETE) lokaliteta, dohvaćanje (GET) *QR koda* lokaliteta
- slike – kreiranje (POST), dohvaćanje (GET), brisanje (DELETE)
- analitiku – dohvaćanje (GET):
    - broj lokaliteta
    - broj korisnika
    - broj danas otkrivenih lokaliteta
    - broj korisnika po spolu
    - broj korisnika po starosnim skupinama
    - broj korisnika po državama
    - broj otkrivenih lokaliteta po mjesecima
    - broj koliko je puta koji lokalitet otkriven.

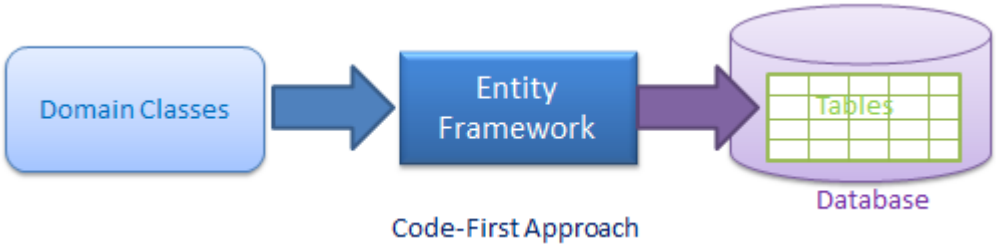
Korisnik (korisnik koji pristupa preko mobilne aplikacije) ima definirane metode za:

- autentifikaciju – registracija (POST), prijava (POST)
- otkrivanje lokaliteta (POST)
- dohvaćanje neotkrivenih lokaliteta (GET)
- dohvaćanje otkrivenih lokaliteta (GET).

Metode su zaštićene digitalno potpisanim tokenom koji se dodjeljuje administratoru ili korisniku prilikom prijave. Token koji je dodijeljen administratoru u sebi sadrži i dodatan podatak (rola) čime su zaštićene metode kojima pristupa web aplikacija.

6.1. Baza podataka

Prvo će se definirati struktura baze podataka. *ASP.NET Core MVC* omogućuje korištenje već spomenutog *Entity Framework*-a koji omogućuje da se radi s bazom koristeći *.NET* klase. *Entity Framework* omogućuje korištenje *Code-First* pristupa. *Code-First* pristup [15] definira skup pravila prema kojem se prvo definiraju klase za domenske entitete, a *Entity Framework* od tih klasa kreira tablice u bazi. Ovaj je proces brži od uobičajenog pristupa u kojem se prvo kreiraju tablice u bazi podataka, a zatim klase. Slika 17 prikazuje shemu funkcioniranja *Code-First* pristupa.



Slika 17 - Code-First pristup
(https://www.entityframeworktutorial.net/code-first/what-is-code-first.aspx)

Korištena baza podataka je *MicrosoftSQL* baza (*MSSQL*).

Domenske klase u ovom kontekstu su modeli iz MVC pristupa.

Za pristup bazi, *Entity Framework* mora dobiti konfiguracijska pravila. Unutar *Startup.cs* klase (Slika 18) potrebno je definirati konfiguracijska pravila.

```
0 references | 0 exceptions
public void ConfigureDevelopmentServices(IServiceCollection services)
{
    services.AddDbContext<DataContext>(optionsAction: options =>
    {
        options.UseLazyLoadingProxies();
        options.UseSqlServer(Configuration.GetConnectionString(name: "DBConnection"));
    });

    ConfigureServices(services);
}
```

Slika 18 - konfiguracijska pravila unutar Startup.cs klase

AddDbContext metoda zahtjeva klasu u kojoj su definirana pravila za kreiranje tablica (*DataContext.cs*). *AddDbContext* metodi mora se proslijediti tekstualni parametar koji sadrži podatke u spajanju na bazu. Naredba

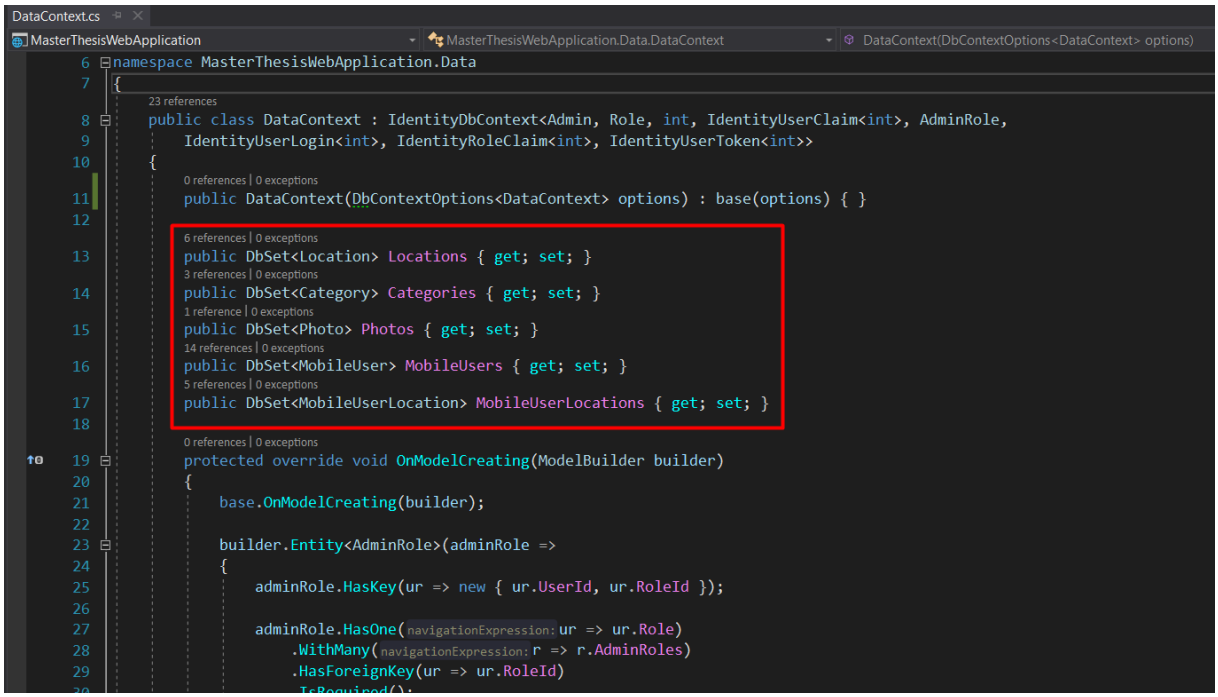
```
options.UseSqlServer(Configuration.GetConnectionString("DBConnection"));
```

dohvaća tekstualni parametar iz *appsettings.json* datoteke (Slika 19).

Slika 19 - DBConnection tekstualni niz u appsettings.json datoteci

Naredba `options.UseLazyLoadingProxies();` omogućuje da prilikom izvršavanja upita ne moramo dodatno uključivati povezane entitete već se oni uključiti automatski ako su potrebni.

Klasa *DataContext.cs* (Slika 20) sadrži pravila za kreiranje tablica.



Slika 20 - DataContext.cs klasa

Svako svojstvo vrste *DbSet* predstavlja tablicu u bazi. *DataContext.cs klasa* (Slika 20) u ovom projektu nasljeđuje *IdentityDbContext* zbog što je korišten *Identity* sustav za registraciju/prijavu administratora, no više o tome u nastavku.

Unutar *DbContext* klase može se i nadjačati (eng. *Override*) *OnModelCreating* metoda kako bi se definirala dodatna pravila. *OnModelCreating* metodu je potrebno nadjačati za definiranje više prema više (eng. *many-to-many*, M:M) veza među entitetima.

Model *Category.cs* vidljiv je na slici 21, svako ne *virtual* svojstvo (eng. *Property*) definirat će stupac u tablici.

```
6  namespace MasterThesisWebApplication.Models
7  {
       9 references
8      public class Category
9      {
           2 references | 0 exceptions
10         public int Id { get; set; }
           2 references | 0 exceptions
11         public string Name { get; set; }
           0 references | 0 exceptions
12         public DateTime DateCreated { get; set; }
           0 references | 0 exceptions
13         public virtual ICollection<Location> Locations { get; set; }
14     }
15 }
16
```

Slika 21 - Category.cs model

```
7  namespace MasterThesisWebApplication.Models
8  {
       19 references
9      public class Location
10     {
           4 references | 0 exceptions
11         public int Id { get; set; }
           3 references | 0 exceptions
12         public string Name { get; set; }
           0 references | 0 exceptions
13         public string ShortDescription { get; set; }
           0 references | 0 exceptions
14         public string LongDescription { get; set; }
           0 references | 0 exceptions
15         public double Lat { get; set; }
           0 references | 0 exceptions
16         public double Lng { get; set; }
           0 references | 0 exceptions
17         public DateTime DateCreated { get; set; }
           0 references | 0 exceptions
18         public int CategoryId { get; set; }
           1 reference | 0 exceptions
19         public virtual Category Category { get; set; }
           1 reference | 0 exceptions
20         public virtual ICollection<Photo> Photos { get; set; }
           1 reference | 0 exceptions
21         public virtual ICollection<MobileUserLocation> MobileUserLocations { get; set; }
22     }
23 }
```

Slika 22 - Location.cs model

Poveznica između modela *Location.cs (Slika 22)* i *Category.cs* su svojstvo *Locations* u *Category.cs* modelu i svojstva *CategoryId* i *Category* u *Location.cs* modelu.
Ovakva sintaksa će *Entity Framework*-u dati do znanja da se radi o jedan prema više ili više-prema-jedan (*one-to-many/many-to-one*, 1:M, M:1) vezi između entiteta. Odnosno, jedan lokalitet može pripadati samo jednoj kategoriji, a jedna kategorija može sadržavati više lokaliteta.

```
6  namespace MasterThesisWebApplication.Models
7  {
       7 references
8      public class Photo
9      {
           2 references | 0 exceptions
10         public int Id { get; set; }
           0 references | 0 exceptions
11         public string Url { get; set; }
           0 references | 0 exceptions
12         public string PublicId { get; set; }
           0 references | 0 exceptions
13         public int LocationId { get; set; }
           0 references | 0 exceptions
14         public virtual Location Location { get; set; }
15     }
16 }
17
```

Slika 23 - Photo.cs model

Na slici 23 prikazan je Photo.cs model. Za slike koristit će se servis u oblaku (eng. *Cloud service*) pa je u bazu dovoljno zapisati poveznicu na sliku koja je javno dostupna na webu. Kao i u odnosu lokaliteta i kategorija, jedna fotografija pripada jednom lokalitetu, a jedan lokalitet može imati više fotografija.

```
6  namespace MasterThesisWebApplication.Models
7  {
       14 references
8      public class MobileUser
9      {
           3 references | 0 exceptions
10         public int Id { get; set; }
           3 references | 0 exceptions
11         public string Username { get; set; }
           2 references | 0 exceptions
12         public byte[] PasswordHash { get; set; }
           2 references | 0 exceptions
13         public byte[] PasswordSalt { get; set; }
           0 references | 0 exceptions
14         public string FirstName { get; set; }
           0 references | 0 exceptions
15         public string LastName { get; set; }
           4 references | 0 exceptions
16         public string Country { get; set; }
           3 references | 0 exceptions
17         public string Gender { get; set; }
           1 reference | 0 exceptions
18         public DateTime DateOfBirth { get; set; }
           0 references | 0 exceptions
19         public DateTime CreatedAt { get; set; }
           2 references | 0 exceptions
20         public virtual ICollection<MobileUserLocation> MobileUserLocations { get; set; }
21     }
22 }
23
```

Slika 24 - MobileUser.cs model

Više-prema-više (*Many-to-Many*, M:M) vezs između entiteta se definira na način da, u ovom primjeru, modeli *MobileUser.cs (Slika 24)* i *Location.cs* imaju zajedničko svojstvo

MobileUserLocations vrste *ICollection*[7] modela *MobileUserLocation* unutar kojeg su definirane poveznice na oba modela (Slika 25).

```
6  namespace MasterThesisWebApplication.Models
7  {
       7 references
8      public class MobileUserLocation
9      {
           6 references | 0 exceptions
10         public int MobileUserId { get; set; }
           1 reference | 0 exceptions
11         public virtual MobileUser MobileUser { get; set; }
           6 references | 0 exceptions
12         public int LocationId { get; set; }
           3 references | 0 exceptions
13         public virtual Location Location { get; set; }
           3 references | 0 exceptions
14         public DateTime CreatedAt { get; set; }
15
           2 references | 0 exceptions
16         public MobileUserLocation()
17         {
18             CreatedAt = DateTime.Now;
19         }
20     }
21 }
22
```

Slika 25 - MobileUserLocation.cs

Kako bi *Entity Framework* znao kreirati više-prema-više vezu potrebno je pomoću *DataContext* klase unutar *OnModelCreating* metode to definirati (Slika 26).

```
builder.Entity<MobileUserLocation>(MobileUserLocations =>
    {
        MobileUserLocations.HasKey(ml => new {ml.LocationId, ml.MobileUserId});

        MobileUserLocations.HasOne(navigationExpression: ml => ml.Location)
            .WithMany(navigationExpression: l => l.MobileUserLocations)
            .HasForeignKey(ml => ml.LocationId)
            .IsRequired();

        MobileUserLocations.HasOne(navigationExpression: ml => ml.MobileUser)
            .WithMany(navigationExpression: m => m.MobileUserLocations)
            .HasForeignKey(ml => ml.MobileUserId)
            .IsRequired();
});
```

Slika 26 - definiranje veze više-prema-više

[7] Nadskup liste u *.NET-u*

Dakle, *Entity Framework* će kreirati pivot tablicu *MobileUserLocations* (Slika 27) koja će sadržavati ID korisnika, ID lokaliteta i vrijeme kada je ta veza nastala.

dbo.MobileUserLocations [Data]

Max Rows: 1000

MobileUserId	LocationId	CreatedAt
1006	19	09-Dec-19 18:17:31
2007	19	09-Dec-19 20:00:55
1006	1017	21-Jan-01 00:00:00
1006	1019	08-Dec-19 21:14:33
1006	1020	08-Dec-19 21:15:42
NULL	*NULL*	*NULL*

Slika 27 - MobileUserLocations tablica

Veza između modela *Admin.cs* i *Role.cs* koji predstavljaju administratora i njegovu rolu također ima više-prema-više vezu pa je rezultat te veze *AdminRoles* pivot tablica. Ono po čemu se modeli *Admin.cs* i *Role.cs* razlikuju od ostalih modela je to što je za njih korišten *Identity* sustav. *Identity* sustav omogućava brzo dodavanje funkcionalnosti prijave i registracije te brine o korisnicima, zaporkama, podacima o profilu i rolama. Omogućuje email potvrdu registracije, prati broj pokušaja prijave i prijave putem maila ili društvenih mreža te još mnogo korisnih funkcionalnosti. *Identity* sustav je u projektu korišten minimalno – samo za potrebe prijave administratora i dodjeljivanje njemu pripadne role. Za korisnika mobilne aplikacije nije korišten *Identity* sustav zbog želje za boljim proučavanjem funkcioniranja procesa prijave i registracije.

Za korištenje *Identity* sustava dovoljno je unutar *startup.cs* klase definirati *Identity* servis te kreirati potrebne modele koji nasljeđuju *IdentityUser* odnosno *IdentityRole* klasu kao što je prikazano na slikama 29-31.

```
IdentityBuilder builder = services.AddIdentityCore<Admin>(setupAction: opt =>
{
    opt.Password.RequireDigit = false;
    opt.Password.RequiredLength = 8;
    opt.Password.RequireUppercase = false;
    opt.Password.RequireNonAlphanumeric = false;
});
builder = new IdentityBuilder(builder.UserType, typeof(Role), builder.Services);
builder.AddEntityFrameworkStores<DataContext>();
builder.AddRoleValidator<RoleValidator<Role>>();
builder.AddRoleManager<RoleManager<Role>>();
builder.AddSignInManager<SignInManager<Admin>>();
```

Slika 28 - definicija Identity servisa u Startup.cs klasi

```
4  namespace MasterThesisWebApplication.Models
5  {
       12 references
6      public class Admin : IdentityUser<int>
7      {
           1 reference | 0 exceptions
8          public virtual ICollection<AdminRole> AdminRoles { get; set; }
9      }
10 }
11
```

Slika 29 - Admin model

```
       9 references
6      public class Role : IdentityRole<int>
7      {
           1 reference | 0 exceptions
8          public virtual ICollection<AdminRole> AdminRoles { get; set; }
9      }
10 }
11
```

Slika 30 - Role model

```
3  namespace MasterThesisWebApplication.Models
4  {
       4 references
5      public class AdminRole : IdentityUserRole<int>
6      {
           1 reference | 0 exceptions
7          public virtual Admin Admin { get; set; }
           1 reference | 0 exceptions
8          public virtual Role Role { get; set; }
9      }
10 }
11
```

Slika 31 - AdminRole Model

Kako bi *Entity Framework* pretvorio modele u tablice potrebno je napraviti migraciju. Migracija predstavlja način za sinkronizaciju sheme baze podataka s modelima u projektu. Shema rada *Entity Framework*-a prikazana je na slici 32.

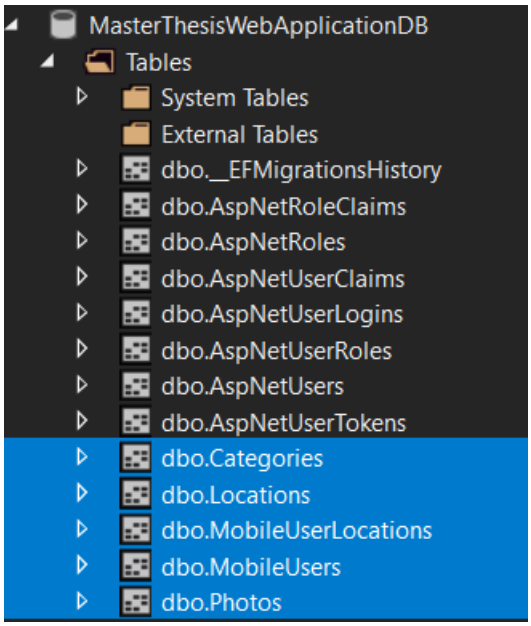
Slika 32 - Entity Framework
(https://www.entityframeworktutorial.net/efcore/entity-framework-core.aspx)

Naredba:
> *dotnet ef migrations add MyFirstMigration*
kreirat će prvu migraciju koju možemo pregledati, a zatim primijeniti na bazu koristeći

naredbu:
> *dotnet ef database update*

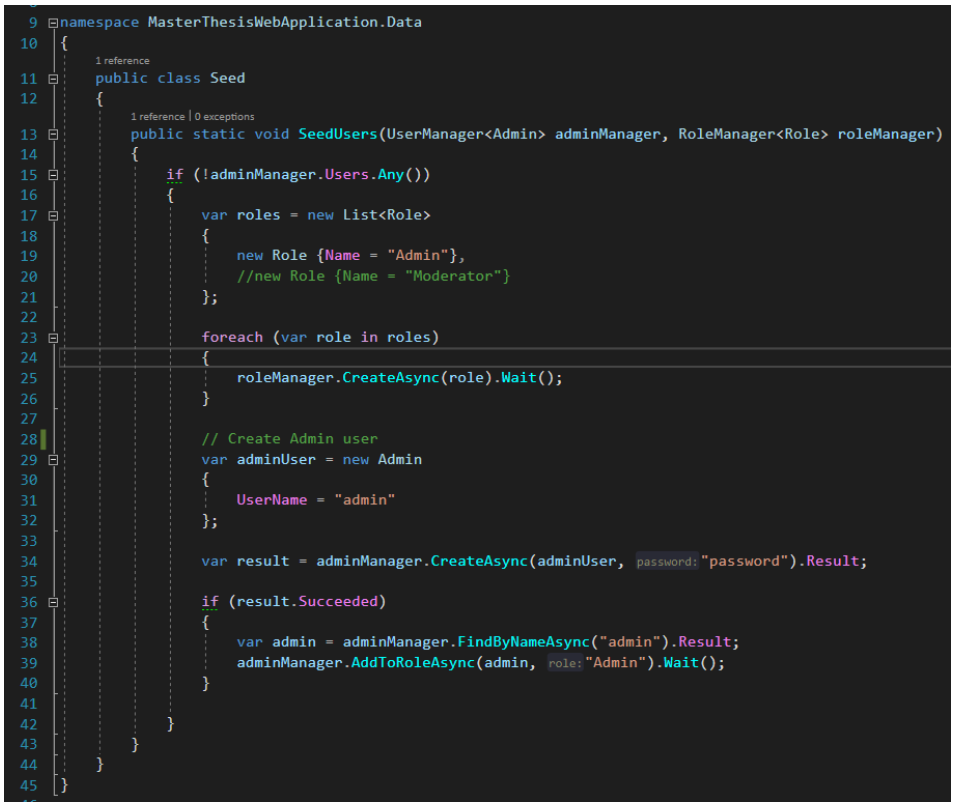
Postupak dodavanja migracije i primjenjivanja na bazu potrebno je napraviti svaki puta kada radimo promjenu u modelu. Dodavanje novih migracija ne utječe na podatke koji su već spremljeni u bazi podataka. *Entity Framework* je samostalan programski okvir te nudi mnogo opcija kojih se ovaj projekt nije dotakao.

Nakon primjene kreirane migracije nad bazom, uspješno su kreirane sve potrebne tablice (Slika 33). Tablice s prefiksom *dbo.AspNet* su tablice koje je generirao *Identity* sustav.



Slika 33- shema baze podataka

S obzirom na to da za administratora nije predviđena mogućnost registracije, on se kreira prilikom prvog kreiranja baze podataka koristeći *Seed.cs* klasu (Slika 34) koja kreira administratora ukoliko on ne postoji.



```
namespace MasterThesisWebApplication.Data
{
    1 reference
    public class Seed
    {
        1 reference | 0 exceptions
        public static void SeedUsers(UserManager<Admin> adminManager, RoleManager<Role> roleManager)
        {
            if (!adminManager.Users.Any())
            {
                var roles = new List<Role>
                {
                    new Role {Name = "Admin"},
                    //new Role {Name = "Moderator"}
                };

                foreach (var role in roles)
                {
                    roleManager.CreateAsync(role).Wait();
                }

                // Create Admin user
                var adminUser = new Admin
                {
                    UserName = "admin"
                };

                var result = adminManager.CreateAsync(adminUser, password: "password").Result;

                if (result.Succeeded)
                {
                    var admin = adminManager.FindByNameAsync("admin").Result;
                    adminManager.AddToRoleAsync(admin, role: "Admin").Wait();
                }

            }
        }
    }
}
```

Slika 34 - Seed.cs klasa

SeedUsers metoda će, koristeći *Identity* sustav, kreirati rolu *Admin*, kreirati administratora s korisničkim imenom *admin* i zaporkom *password* te mu dodijeliti *admin* rolu.

6.2. API

6.2.1. *Repository pattern*

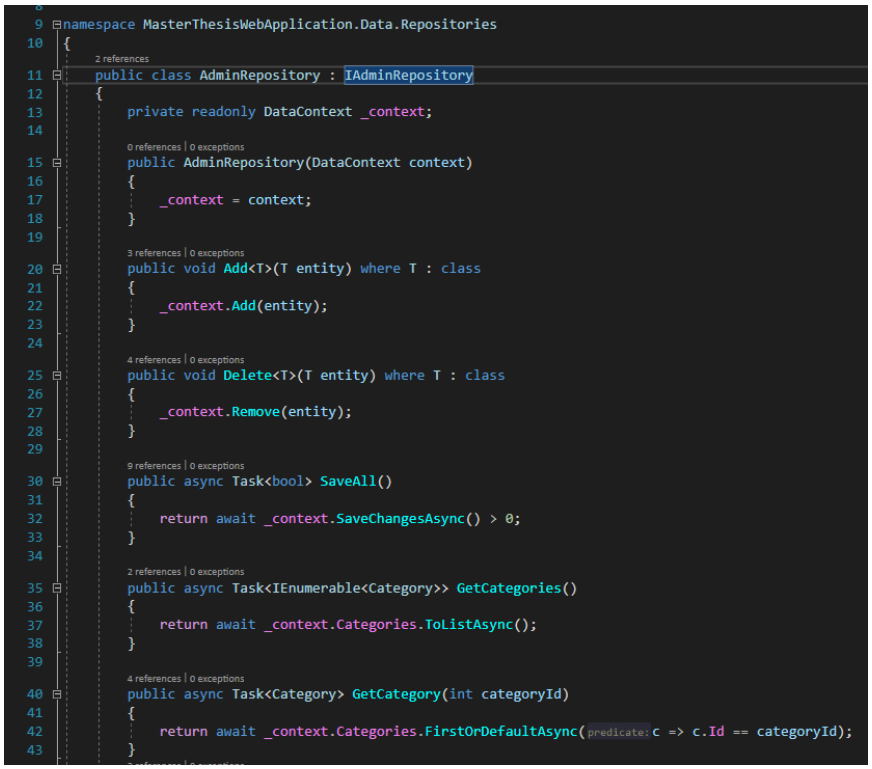
Uz već definirani *MVC* koji razdvaja aplikaciju na tri komponente, za izradu ovog API-ja korištena je još jedna razina apstrakcije. Uzorak repozitorija [16] (eng. *Repository pattern*) funkcionira na način da se kreira sučelje (eng. *interface*) te repozitorij koji predstavlja implementaciju tog sučelja. U repozitorij, iz kontrolera, premješta se sva logiku vezanu uz komunikaciju s bazom podataka. Kontroler je povezan sa sučeljem te preko njega dohvaća potrebne podatke. Ovaj način rada omogućava lakše testiranje i bolju modularnost aplikacije. Također, ako se promjeni baza nad kojom API radi morat će se samo ponovno implementirati metode iz sučelja. Programski kod u kontroleru i sučelju neće se mijenjati.

IAdminRepository.cs (Slika 35) predstavlja sučelje sa svim metodama potrebnim za dohvaćanje podataka za web aplikaciju.

```
namespace MasterThesisWebApplication.Data
{
    12 references
    public interface IAdminRepository
    {
        3 references | 0 exceptions
        void Add<T>(T entity) where T : class;
        4 references | 0 exceptions
        void Delete<T>(T entity) where T : class;
        9 references | 0 exceptions
        Task<bool> SaveAll();
        2 references | 0 exceptions
        Task<IEnumerable<Category>> GetCategories();
        4 references | 0 exceptions
        Task<Category> GetCategory(int categoryId);
        3 references | 0 exceptions
        Task<bool> CategoryByNameExist(string categoryName);
        2 references | 0 exceptions
        Task<IEnumerable<Location>> GetLocations();
        6 references | 0 exceptions
        Task<Location> GetLocation(int locationId);
        3 references | 0 exceptions
        Task<Photo> GetPhoto(int photoId);
        // Dashboard
        2 references | 0 exceptions
        Task<int> GetNumberOfLocations();
        2 references | 0 exceptions
        Task<int> GetNumberOfUsers();
        2 references | 0 exceptions
        Task<int> GetNumberOfTodayDiscoveredLocations();
        // Analytics
        2 references | 0 exceptions
        Task<List<int>> UsersGenderCount();
        2 references | 0 exceptions
        Task<List<int>> UsersAgeCount();
        2 references | 0 exceptions
        Task<List<int>> UsersCountryCount();
        2 references | 0 exceptions
        Task<List<int>> GetScansByMonth();
        2 references | 0 exceptions
        Task<IEnumerable<LocationWithScanCount>> GetLocationsWithTimesScanned();
    }
}
```

Slika 35 - IAdminRepository.cs sučelje

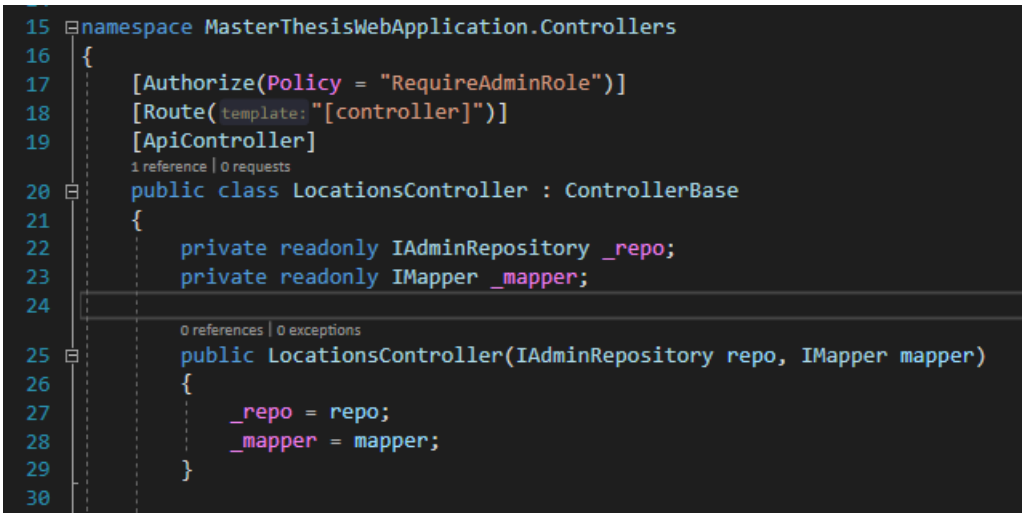
AdminRepository.cs (Slika 36) je klasa koja implementira *IAdminRepository* sučelje.



```
namespace MasterThesisWebApplication.Data.Repositories
{
    2 references
    public class AdminRepository : IAdminRepository
    {
        private readonly DataContext _context;

        0 references | 0 exceptions
        public AdminRepository(DataContext context)
        {
            _context = context;
        }

        3 references | 0 exceptions
        public void Add<T>(T entity) where T : class
        {
            _context.Add(entity);
        }

        4 references | 0 exceptions
        public void Delete<T>(T entity) where T : class
        {
            _context.Remove(entity);
        }

        9 references | 0 exceptions
        public async Task<bool> SaveAll()
        {
            return await _context.SaveChangesAsync() > 0;
        }

        2 references | 0 exceptions
        public async Task<IEnumerable<Category>> GetCategories()
        {
            return await _context.Categories.ToListAsync();
        }

        4 references | 0 exceptions
        public async Task<Category> GetCategory(int categoryId)
        {
            return await _context.Categories.FirstOrDefaultAsync(predicate: c => c.Id == categoryId);
        }
```

Slika 36 - AdminRepository.cs klasa

U konstruktoru kontrolera instancira se sučelje, te tako kontroler ima pristup metodama sučelja što je prikazano na slikama 37 i 38.



```
namespace MasterThesisWebApplication.Controllers
{
    [Authorize(Policy = "RequireAdminRole")]
    [Route(template: "[controller]")]
    [ApiController]
    1 reference | 0 requests
    public class LocationsController : ControllerBase
    {
        private readonly IAdminRepository _repo;
        private readonly IMapper _mapper;

        0 references | 0 exceptions
        public LocationsController(IAdminRepository repo, IMapper mapper)
        {
            _repo = repo;
            _mapper = mapper;
        }
```

Slika 37 - Instanciranje IAdminRepository.cs klase u konstruktoru kontrolera

```
[HttpGet(template: "{locationId}", Name = "GetLocation")]
0 references | 0 requests | 0 exceptions
public async Task<IActionResult> GetLocation(int locationId)
{
    var location = await _repo.GetLocation(locationId);

    if (location == null)
        return NoContent();

    var locationToReturn = _mapper.Map<LocationToReturnDto>(location);

    return Ok(locationToReturn);
}
```

Slika 38 - pozivanje metode repozitorija iz kontrolera

6.2.2. Autentifikacija i autorizacija korisnika

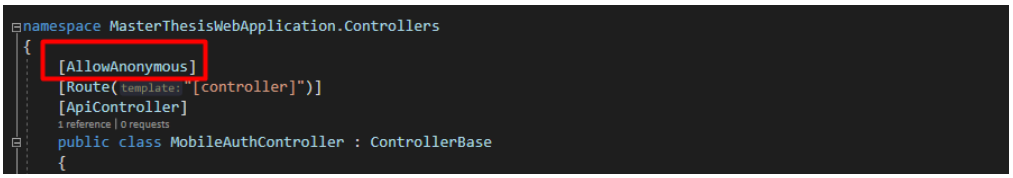
.NET Core programski okvir ima funkcionalnost za jednostavnu provjeru autorizacije i autentifikacije korisnika. U *startup.cs* klasu dovoljno je dodati *AddAuthorization* i *AddAuthentication* servise (Slika 39) te će *.NET Core* provjeravati svaki zahtjev koji dolazi prema metodama u kontroleru na temelju konfiguracije u servisima.

```
services.AddAuthorization(options =>
{
    options.AddPolicy(name: "RequireAdminRole", configurePolicy: policy => policy.RequireRole("Admin"));
});

services.AddAuthentication(JwtBearerDefaults.AuthenticationScheme)
    .AddJwtBearer(options =>
    {
        options.TokenValidationParameters = new TokenValidationParameters
        {
            ValidateIssuerSigningKey = true,
            IssuerSigningKey =
                new SymmetricSecurityKey(
                    Encoding.ASCII.GetBytes(Configuration.GetSection(key: "AppSettings:TokenKey").Value)),
            ValidateIssuer = false,
            ValidateAudience = false
        };
    });
```

Slika 39 - dodavanje AddAuthorization i AddAuthentication servisa

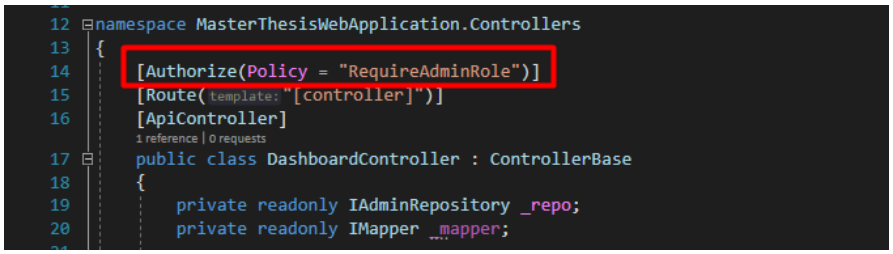
Zatim, koristeći napomene (eng. *Data Annotations*) iznad klasa i/ili metoda može se definirati tko im može, odnosno ne može pristupiti.

Dodavanjem autentifikacije, za svaku metodu u kontroleru *.NET Core* će provjeravati postoji li digitalno potpisani token u zaglavlju zahtjeva i je li on validan. Ako token nije validan ili ne postoji *.NET Core* će vratit odgovor 401 *Unauthorized*. Ako želimo nekim metodama dopustiti pristup bez obzira na token. Zatim dodajemo *AllowAnonymous* napomenu iznad metode ili kontrolera (Slika 40). Na primjer, metodama za registraciju i prijavu bi se trebalo moći pristupiti bez prethodne autentifikacije.



Slika 40 - dodavanje napomena (1/2)

Slična stvar je s autorizacijom, nekom kontroleru ili metodi može se zabraniti pristup ako ne sadrže određenu rolu to se radi pomoću *[Authorize(Policy = "NazivRole")]* napomene. Na primjer, u ovom projeku svim kontrolerima vezanima uz web aplikaciju može pristupiti samo korisnik koji ima rolu Admin (Slika 41).



Slika 41 - dodavanje napomena (2/2)

6.2.3. Metode

S obzirom na količinu metoda, u radu će detaljnije biti prikazani samo primjeri metoda za *CRUD*[8] operacije nad lokalitetom. Na odabranim metodama može se vidjeti način funkcioniranja API metoda u *ASP.NET Core MVC* aplikacijama.

Za dohvaćanje svih lokaliteta šalje se *GET* zahtjev na */locations/* ako ne postoji niti jedan lokalitet, kao odgovor će doći HTTP Status 204 Nema sadržaja (eng. *No Content*).

Mapper klasa služi za mapiranje podataka između modela u bazi i modela kojeg se želi vratiti korisniku. *Map* metoda unutar <> prima ciljni model, a unutar () izvorni model. Dakle, ako u model ima deset parametara, a želi se korisniku vratiti samo pet onda ćemo koristiti *Mapper* koji će automatski mapirati podatke i poslati ih korisniku. *Mapper* klasi je bitno da se svojstva u modelima između kojih se mapiranje obavlja zovu jednako. Slika 42 prikazuje metodu za dohvaćanje svih lokaliteta u kontroleru, slika 43 prikazuje metodu za dohvaćanje svih lokaliteta iz baze podataka, a slika 44 prikazuje testiranje dohvaćanja svih lokaliteta koristeći HTTP klijent Postman.

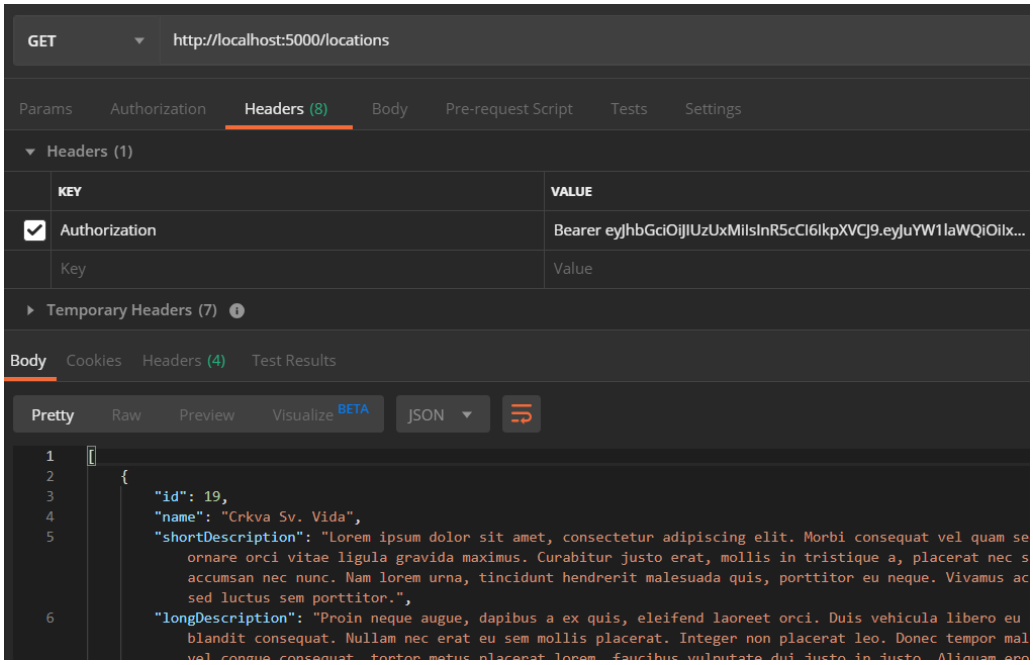
[8] Akronim riječi Create, Read, Update, Delete koje predstavljaju osnovne operacije nad bazom podataka

```
[HttpGet]
0 references | 0 requests | 0 exceptions
public async Task<IActionResult> GetLocations()
{
    var locations = await _repo.GetLocations();

    if (!locations.Any())
        return NoContent();

    var locationsToReturn = _mapper.Map<IEnumerable<LocationToReturnDto>>(locations);

    return Ok(locationsToReturn);
}
```

Slika 42 - metoda za dohvaćanje lokaliteta u kontroleru

```
2 references | 0 exceptions
public async Task<IEnumerable<Location>> GetLocations()
{
    return await _context.Locations.ToListAsync();
}
```

Slika 43 - metoda za dohvaćanje svih lokaliteta iz baze podataka



Slika 44 - testiranje dohvaćanja lokaliteta koristeći HTTP klijent Postman

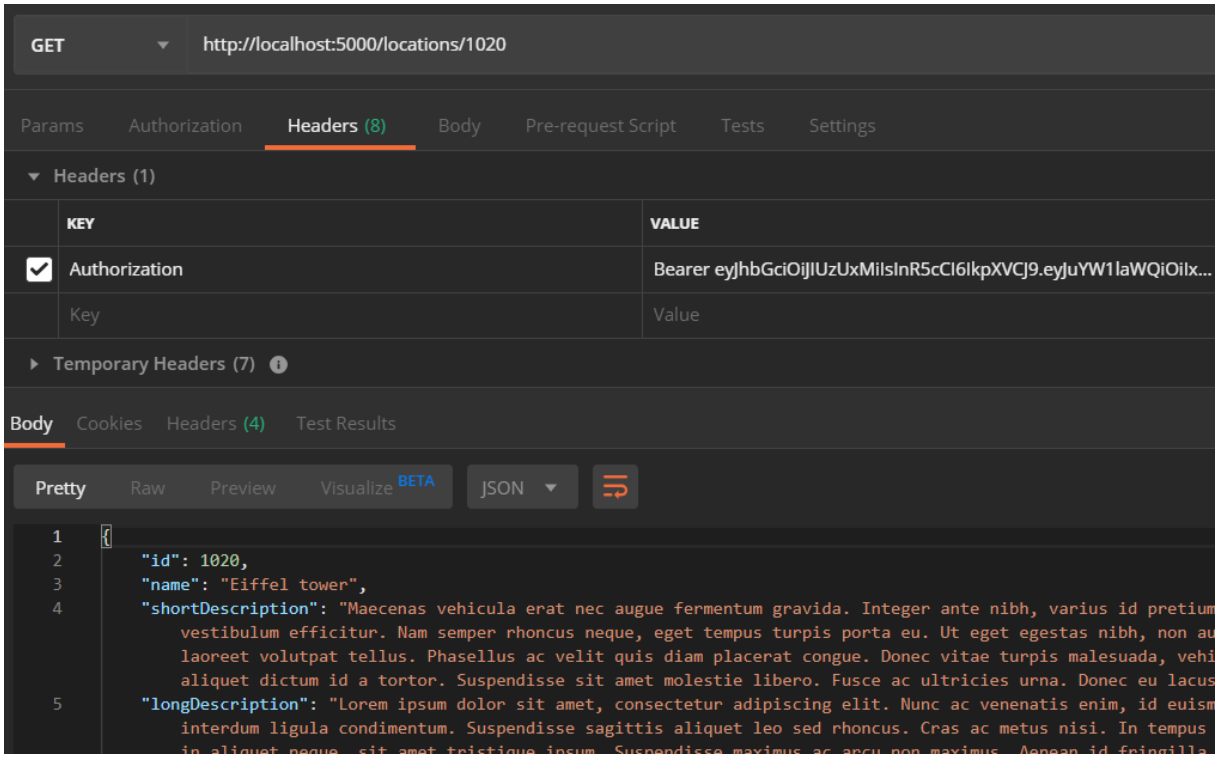
Za dohvaćanje određenog lokaliteta šalje se *GET* zahtjev na /*locations/id-lokaliteta* ako ne postoji taj lokalitet, kao odgovor će doći HTTP Status 204 Nema sadržaja (eng. *No Content*). Slika 45 prikazuje metodu za dohvaćanje lokaliteta u kontroleru, slika 46 prikazuje metodu za dohvaćanje lokaliteta iz baze podataka, a slika 47 prikazuje testiranje dohvaćanja lokaliteta koristeći HTTP klijent Postman.

```
[HttpGet(template: "{locationId}", Name = "GetLocation")]
0 references | 0 requests | 0 exceptions
public async Task<IActionResult> GetLocation(int locationId)
{
    var location = await _repo.GetLocation(locationId);

    if (location == null)
        return NoContent();

    var locationToReturn = _mapper.Map<LocationToReturnDto>(location);

    return Ok(locationToReturn);
}
```

Slika 45 - metoda za dohvaćanje lokaliteta u kontroleru

```
6 references | 0 exceptions
public async Task<Location> GetLocation(int locationId)
{
    return await _context.Locations.FirstOrDefaultAsync(predicate: l => l.Id == locationId);
}
```

Slika 46 - metoda za dohvaćanje lokaliteta iz baze podataka



Slika 47 - testiranje dohvaćanja lokaliteta koristeći HTTP klijent Postman

Za kreiranje novog lokaliteta šalje se *POST* zahtjev na */locations/*. Ova metoda kao parametar u tijelu zahtjeva (eng. *Request body*) prima objekt klase *LocationForCreationDto*, putem *Mapper* klase mapira podatke u *Location* model te spremi podatke u bazu, ako je sve u redu, metoda će vratiti novonastali *Location* objekt. Slika 48 prikazuje metodu za kreiranje novog lokaliteta u kontroleru, slika 49 prikazuje metodu za kreiranje novog retka u tablici, a slika 51 prikazuje testiranje kreiranja novog lokaliteta koristeći HTTP klijent Postman.

```
[HttpPost]
0 references | 0 requests | 0 exceptions
public async Task<IActionResult> CreateLocation([FromBody] LocationForCreationDto locationForCreationDto)
{
    var location = _mapper.Map<Location>(locationForCreationDto);
    _repo.Add(location);

    if (await _repo.SaveAll())
    {
        var locationToReturn = _mapper.Map<LocationToReturnDto>(location);
        return CreatedAtRoute(routeName: "GetLocation", routeValues: new { locationId = location.Id }, value: locationToReturn);
    }

    return BadRequest(error: "Creating the location failed.");
}
```

Slika 48 - metoda za kreiranje novog lokaliteta u kontroleru

```
3 references | 0 exceptions
public void Add<T>(T entity) where T : class
{
    _context.Add(entity);
}
```

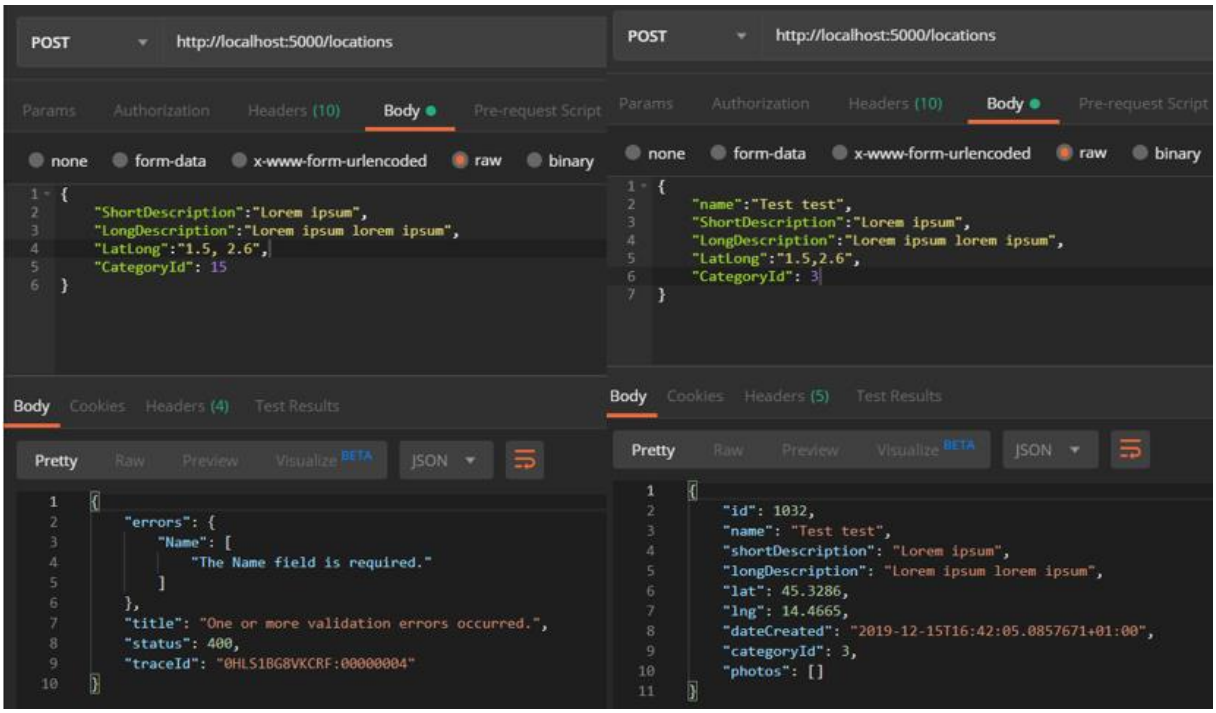
Slika 49 - metoda za kreiranje novog retka u tablici

```
namespace MasterThesisWebApplication.Dtos
{
    4 references
    public class LocationForCreationDto
    {
        [Required]
        0 references | 0 exceptions
        public string Name { get; set; }
        0 references | 0 exceptions
        public string ShortDescription { get; set; }
        0 references | 0 exceptions
        public string LongDescription { get; set; }
        0 references | 0 exceptions
        public double Lat { get; set; } = 45.3286;
        0 references | 0 exceptions
        public double Lng { get; set; } = 14.4665;
        1 reference | 0 exceptions
        public DateTime DateCreated { get; set; }
        [Required]
        0 references | 0 exceptions
        public int CategoryId { get; set; }
        0 references | 0 exceptions
        public LocationForCreationDto()
        {
            DateCreated = DateTime.Now;
        }
    }
}
```

Slika 50 - LocationForCreationDto.cs klasa

Klasa *LocationForCreationDto* (Slika 50) predstavlja model koji se očekuje kao parametar na *CreateLocation* metodi. ASP.NET Core će automatski validirati objekt prema napomeni

napisanoj iznad svojstva. U ovom objektu, svojstva *Name* i *CategoryId* su obavezna te će .NET Core automatski vratiti da je zahtjev neispravan uz odgovarajuću poruku.

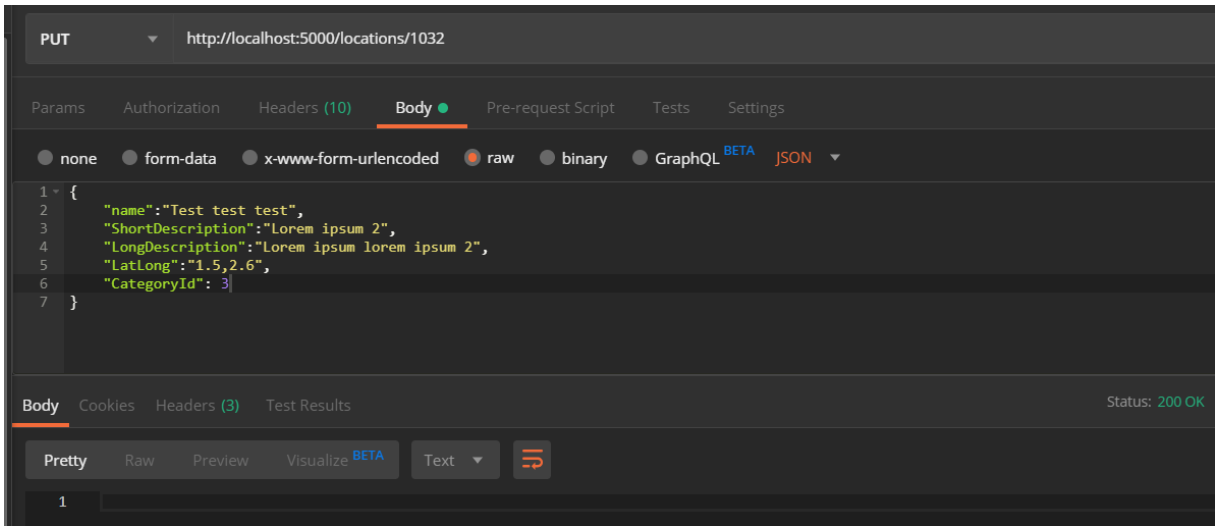


Slika 51 - testiranje kreiranja lokaliteta koristeći HTTP klijent Postman

Za ažuriranje postojećeg lokaliteta šalje se *PUT* zahtjev na */locations/id-lokaliteta*. Ova metoda kao parametar u tijelu zahtjeva prima objekt klase *LocationForCreationDto*. Prvo prema ID parametru dohvaća lokalitet, te ako postoji zamjenjuje njegove vrijednosti s novima te ponovno zapisuje u bazu. Slika 52 prikazuje metodu za ažuriranje postojećeg lokaliteta u kontroleru, a slika 53 prikazuje testiranje ažuriranje postojećeg lokaliteta koristeći HTTP klijent Postman.

```
[HttpPut(template: "{locationId}")]
0 references | 0 requests | 0 exceptions
public async Task<IActionResult> UpdateLocation(int locationId, [FromBody] LocationForCreationDto locationForCreationDto)
{
    var locationFromRepo = await _repo.GetLocation(locationId);

    if (locationFromRepo == null)
        return NoContent();

    _mapper.Map(locationForCreationDto, locationFromRepo);

    if (await _repo.SaveAll())
        return Ok();

    return BadRequest(error: "Updating the location failed.");
}
```

Slika 52 - metoda za ažuriranje postojećeg lokaliteta u kontroleru



Slika 53 - testiranje metode za ažuriranje koristeći HTTP klijent Postman

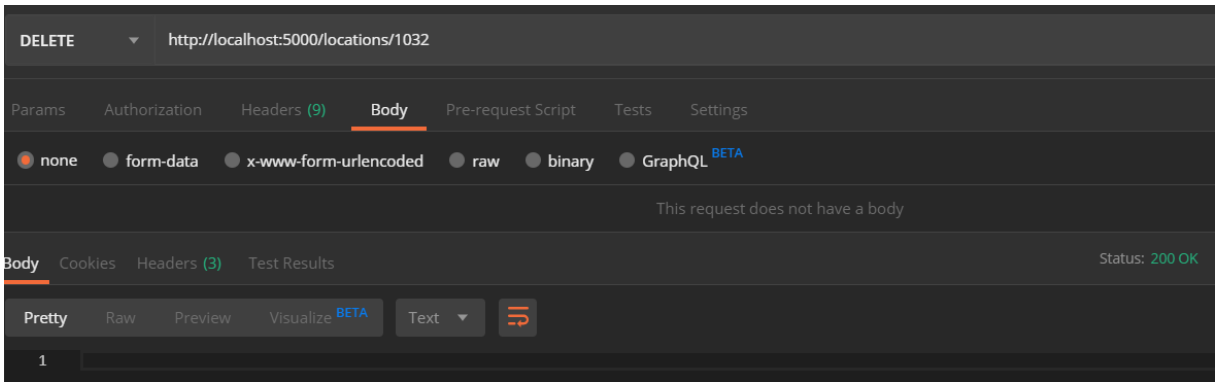
Za brisanje postojećeg lokaliteta šalje se *DELETE* zahtjev na */locations/id-lokaliteta*. Slika 54 prikazuje metodu za brisanje postojećeg lokaliteta u kontroleru, slika 55 prikazuje metodu za brisanje postojećeg retka u tablici, a slika 56 prikazuje testiranje brisanja postojećeg lokaliteta koristeći HTTP klijent Postman.

```
[HttpDelete(template: "{locationId}")]
0 references | 0 requests | 0 exceptions
public async Task<IActionResult> DeleteLocation(int locationId)
{
    var location = await _repo.GetLocation(locationId);

    if (location == null)
        return NoContent();

    _repo.Delete(location);

    if (await _repo.SaveAll())
        return Ok();

    return BadRequest(error: "Deleting location failed.");
}
```

Slika 54 - metoda za brisanje lokaliteta u kontroleru

```
4 references | 0 exceptions
public void Delete<T>(T entity) where T : class
{
    _context.Remove(entity);
}
```

Slika 55 - metoda za brisanje retka iz tablice



Slika 56 - testiranje metode brisanja lokaliteta koristeći HTTP klijent Postman

Popis ostalih metoda definiranih za web aplikaciju prikazan je u tablici 1.

Tablica 1 - popis preostalih metoda za web aplikaciju

Opis metode	Ruta ne serveru	Metoda vraća	Metoda prima
Prijava administratora	POST: /auth/	Token	AdminForLogin objekt
Dohvaćanje svih kategorija	GET: /categories/	Polje CategoryToReturn objekata	-
Dohvaćanje kategorije	GET: /categories/:id	CategoryToReturn objekt	Id kategorije
Kreiranje kategorije	POST: /categories/	CategoryToReturn objekt	CategoryForCreation objekt
Ažuriranje kategorije	PUT: /categories/	Status: 200 OK	CategoryForCreation objekt
Brisanje kategorije	DELETE: /categories/:id	Status: 200 OK	Id kategorije
Dodavanje slike	POST: /locations/photos/:id	PhotoToReturn objekt	Id lokaliteta
Dohvaćanje slike	GET: /locations/photos/:id	PhotoToReturn objekt	Id slike
Brisanje slike	DELETE: /locations/photos/:id	Status: 200 OK	Id slike
Broj korisnika po spolu	GET: /analytics/getUsersGenderCount	Polje s brojevnim vrijednostima	-
Broj korisnika po starosnim skupinama	GET: /analytics/getUsersAgeCount	Polje s brojevnim vrijednostima	-
Broj korisnika po državama	GET: /analytics/getUsersCountryCount	Polje s brojevnim vrijednostima	-
Broj otkrivenih lokaliteta po mjesecima	GET: /analytics/GetScansByMonth	Polje s brojevnim vrijednostima	-
Broj otkrivenih lokaliteta po mjesecima	GET: /analytics/GetLocationsWithTimesScanned	Polje LocationScanCount objekata	-
Broj lokaliteta	GET: /dashboard/getNumberOfLocations	Broj	-
Broj korisnika	GET: /dashboard/getNumberOfUsers	Broj	-

Broj danas otkrivenih lokaliteta	GET: /dashboard/getTodayDiscoveredLocations	Broj	-
Dohvaćanje QR koda	GET: /locations/getQRCode/:id	Tekst (QR kod u base64 formatu)	Id lokaliteta

Popis metoda definiranih za mobilnu aplikaciju prikazan je u tablici 2.

Tablica 2 - popis metoda za mobilnu aplikaciju

Opis metode	Ruta ne serveru	Metoda vraća	Metoda prima
Registracija korisnika	POST: /mobileauth/register	Status: 200 OK	MobileUserForRegisterDto objekt
Prijava korisnika	POST: /mobileauth/login	Token	MobileUserForLoginDto objekt
Otkrij lokalitet	/mobile/:userId/discoverLocation/:locationId	Status: 200 OK	userId, locationId
Dohvati otkrivene lokalitete	GET: /mobile/:userId/getDiscoveredLocations	Polje DiscoveredLocationDto objekata	userId
Dohvati neotkrivene lokalitete	GET: /mobile/:userId/getUndiscoveredLocations	Polje UniscoveredLocationDto objekata	userId

6.3. Web aplikacija

Klijentska strana web aplikacije, napravljena u *Angularu*, strukturirana je kroz dva modula i 12 komponenti.

Prvi modul je korijenski modul *app.module.ts* koji objedinjuje četiri komponente:

- Korijensku komponentu: *AppComponent* komponentu (Sve komponente se učitavaju u ovoj komponenti)
- Login stranicu: *Login* komponentu
- 404 stranicu: *PageNotFound* komponentu
- Okvir nadzorne ploče administratora: *AdminLayout* komponenta

Drugi modul je modul nadzorne ploče administratora *admin-layout.module*.ts koji objedinjuje osam komponenti koje se učitavaju unutar *AdminLayout* komponente:

- Nadzorna ploča: *Dashboard* komponenta
- Kategorije: *Categories* komponenta
- Lokaliteti: *Locations* komponenta
    - Kartica lokaliteta: *LocationCard* komponenta
    - Uređivanje lokaliteta: *LocationEdit* komponenta
    - Karta lokaliteta: *LocationMap* komponenta
    - Uređivanje slika: *LocationPhoto* komponenta
- Statistika: *Analytics* komponenta

Oba modula imaju zasebno definirano usmjeravanje.

```
export const appRoutes: Routes = [
    { path: 'login', component: LoginComponent },
    {
        path: '',
        component: AdminLayoutComponent,
        canActivate: [AuthGuard],
        children: [{
            path: '',
            loadChildren: () => import('./layouts/admin-layout/admin-layout.module').then(mod => mod.AdminLayoutModule)
        }],
    },
    { path: '**', component: PageNotFoundComponent, canActivate: [AuthGuard] }
];
```

Slika 57 - usmjeravanje korijenskog modula

Usmjeravanje korijenskog modula definirano je na slici 57**.** Na praznoj putanji dodana je provjera *canActivate* koja znači da će *Angular* aplikacija prije nego učita komponentu provjeriti logiku u navedenoj klasi – *AuthGuard*. Slika 58 prikazuje AuthGuard klasu.

```
@Injectable({
  providedIn: 'root'
})
export class AuthGuard implements CanActivate {

  constructor(private authService: AuthService, private router: Router, private toastr: ToastrService) { }

  canActivate(route: ActivatedRouteSnapshot, state: RouterStateSnapshot): boolean {
    if (this.authService.loggedIn()) {
      return true;
    }
    this.router.navigate(['/login']);
    return false;
  }
}
```

Slika 58 - AuthGuard.ts klasa

AuthGuard klasa kroz *canActivate* metodu provjerava je li korisnik ulogiran, te ako je vraća vrijednost true i učitava *AdminLayoutComponent* komponentu. Ukoliko korisnik nije ulogiran ili je token istekao vraća vrijednost *false* i vraća korisnika natrag na komponentu za prijavu. Metoda *loggedIn* (Slika 59) provjera je li korisnik ulogiran definirana je u servisu *AuthService*.

```
loggedIn() {
  const token = localStorage.getItem('token');
  return !this.jwtHelper.isTokenExpired(token);
}
```

Slika 59 - loggedIn metoda

jwtHelper klasa je iz *@auth0/angular-jwt* paketa preuzetog s *Github*-a, koja služi za provjeru validnosti tokena. Također, *@auth0/angular-jwt* sadrži definiciju preko koje dodajemo token u zaglavlje svakog zahtjeva prema serveru kako bi nas server uspješno autentificirao.

```
export const AdminLayoutRoutes: Routes = [
    { path: '', redirectTo: 'dashboard', pathMatch: 'full' },
    { path: 'dashboard', component: DashboardComponent },
    { path: 'categories', component: CategoriesComponent },
    { path: 'locations', component: LocationsComponent },
    { path: 'locations/:id', component: LocationEditComponent, resolve: {location: LocationEditResolver}},
    { path: 'analytics', component: AnalyticsComponent },
];
```

Slika 60 - usmjeravanje modula nazdorne ploče administratora

Usmjeravanje modula nadzorne ploče administratora prikazano je na slici 60**.** Parametar *resolve* predstavlja pravilo prema kojem će *Angular* prvo učitati definiranu vrijednost, a zatim učitati komponentu. Unutar *LocationEditResolver* klase definirana je *resolve* metoda (Slika 61). U ovom kontekstu to ima smisla jer *LocationEdit* komponenta ne može funkcionirati bez lokaliteta kojeg se uređuje.

```
@Injectable()
export class LocationEditResolver implements Resolve<Location> {
    constructor(private locationService: LocationService, private toastr: ToastrService, private router: Router) {}

    resolve(route: ActivatedRouteSnapshot, state: RouterStateSnapshot): Observable<Location> {
        return this.locationService.getLocation(route.params.id).pipe(
            catchError(error => {
                this.toastr.error('Problem retrieving your data');
                this.router.navigate(['/locations']);
                return of(null);
            })
        );
    }
}
```

Slika 61 - LocationEditResolver.ts klasa

U projektu je definirano pet servisa koji služe za komunikaciju sa serverom:

- *AuthService* – za slanje zahtjeva za prijavu korisnika
- *AnalyticsService* – za dohvaćanje podataka o statistikama
- *DashboardService* – za dohvaćanje podataka za nadzornu ploču
- *CategoryService* – za dohvaćanje i slanje podataka o kategorijama
- *LocationService* – za dohvaćanje i slanje podataka o lokalitetima

Na slici 62, primjera radi, prikazana je implementacija *CategoryService* servisa.

```
@Injectable({
  providedIn: 'root'
})
export class CategoryService {
  apiUrl = environment.baseUrl + 'categories/';

  constructor(private http: HttpClient) { }

  getCategories(): Observable<Category[]> {
    return this.http.get<Category[]>(this.apiUrl);
  }

  getCategory(id: number) {
    return this.http.get<Category>(this.apiUrl + id);
  }

  createCategory(category: Category) {
    return this.http.post(this.apiUrl, category);
  }

  updateCategory(id: number, category: Category) {
    return this.http.put(this.apiUrl + id, category);
  }

  deleteCategory(id: number) {
    return this.http.delete(this.apiUrl + id);
  }
}
```

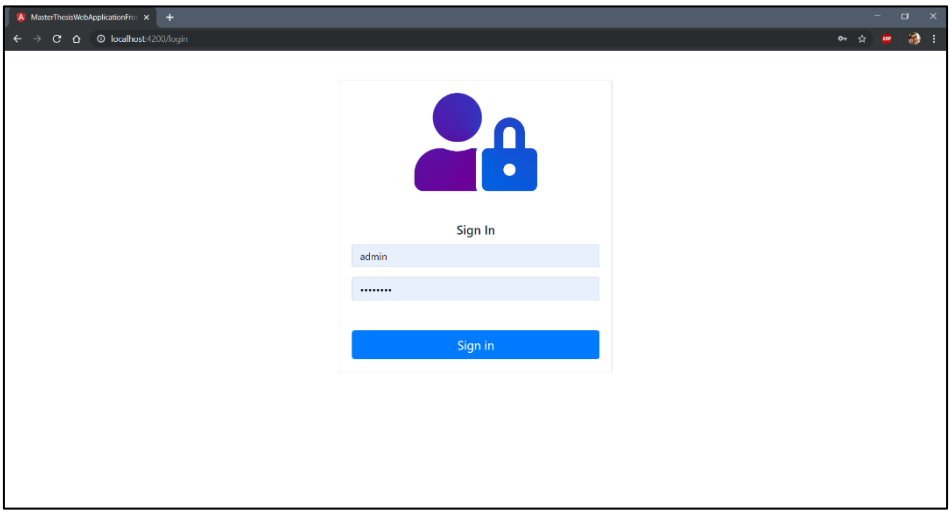
Slika 62 - CategoryService servis

Opcionalni i vanjski moduli koji su korišteni u projektu:

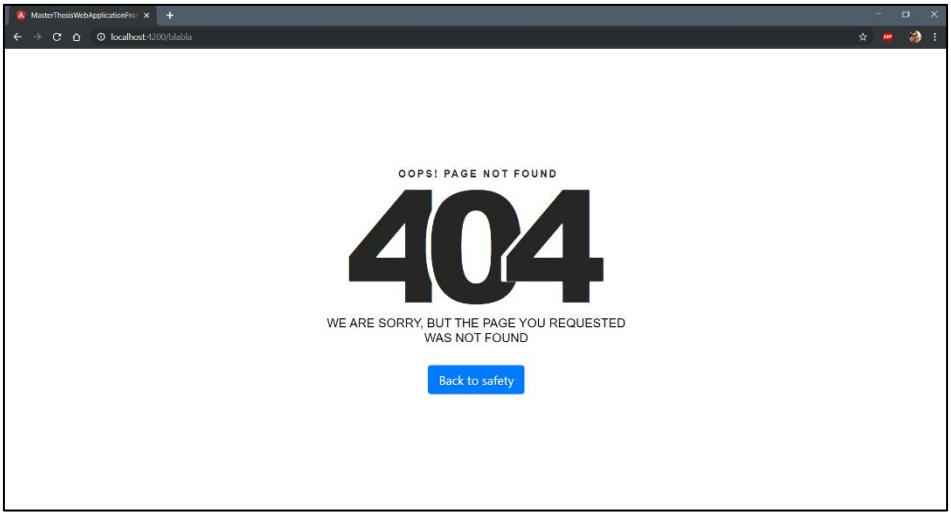
- *JwtModule (@auth0/angular-jwt)* koji dodaje token u svaki zahtjev prema serveru i omogućuje provjeru je li token istekao
- *HttpClientModule (@angular/common/http)* koji sadrži metode za slanje zahtjeva prema serveru
- *ToastrModule (ngx-toastr)* za prikaz poruka u projektu
- *ReactiveFormsModule (@angular/forms)* za kreiranje formi
- *TabsModule (ngx-bootstrap/tabs)* za korištenje Bootstrap kartica
- *ModalModule (ngx-bootstrap/modal)* za korištenje Bootstrap modala
- *AgmCoreModule (@agm/core)* za korištenje Google karti
- *ImageUploaderModule (ngx-image-uploader)* za učitavanje slika na server, ovaj modul je ovisan o FileUploadModule modulu
- *FileUploadModule (ng2-file-upload)*
- *ChartsModule (ng2-charts)* za prikaz grafova

6.3.1. Slike zaslona

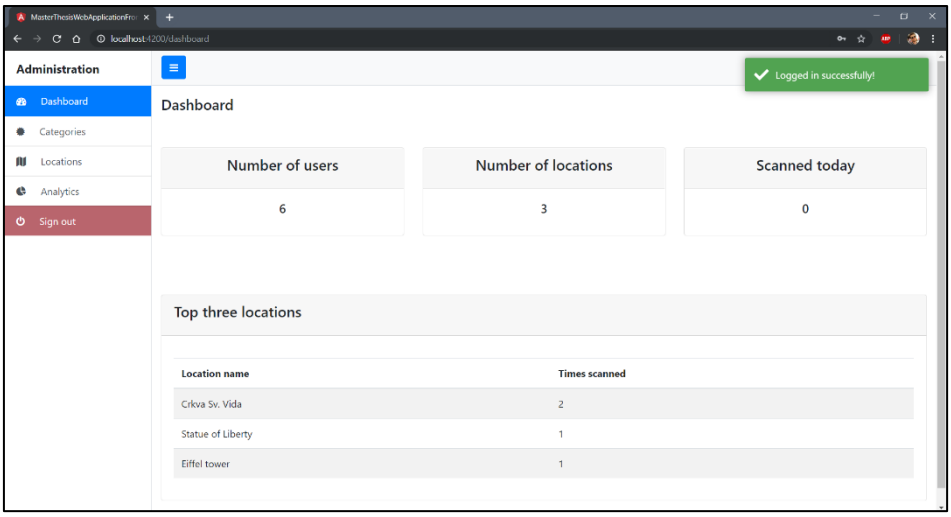
Na slikama 63-77 prikazane su slike zaslona implementirane web aplikacije.



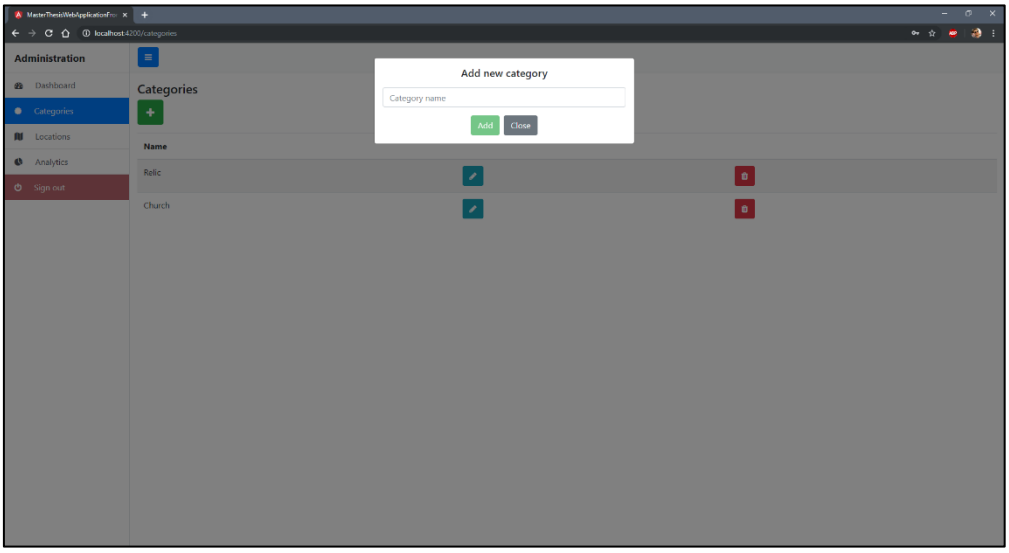
Slika 63 – prijava



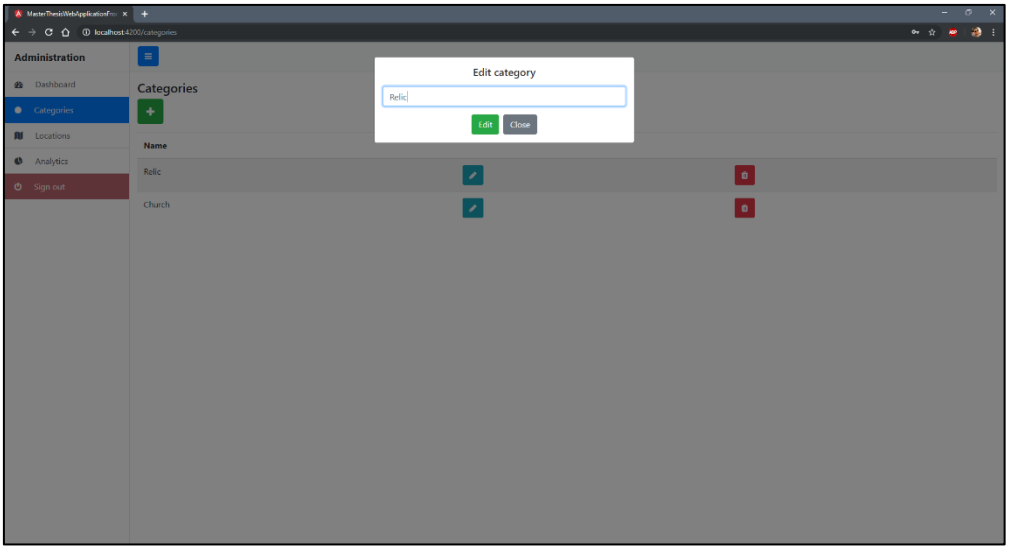
Slika 64 – 404 stranica nije pronađena



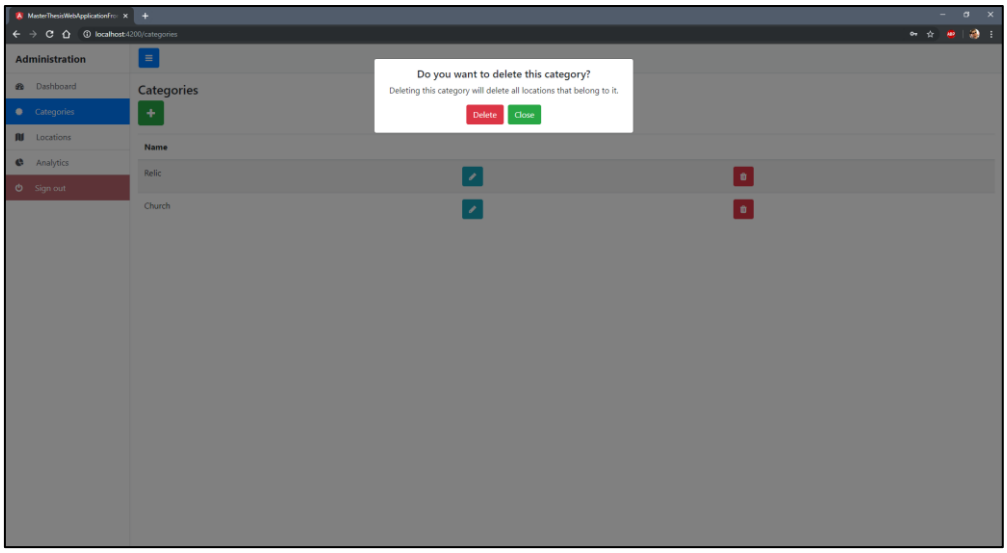
Slika 65 – nadzorna ploča administratora



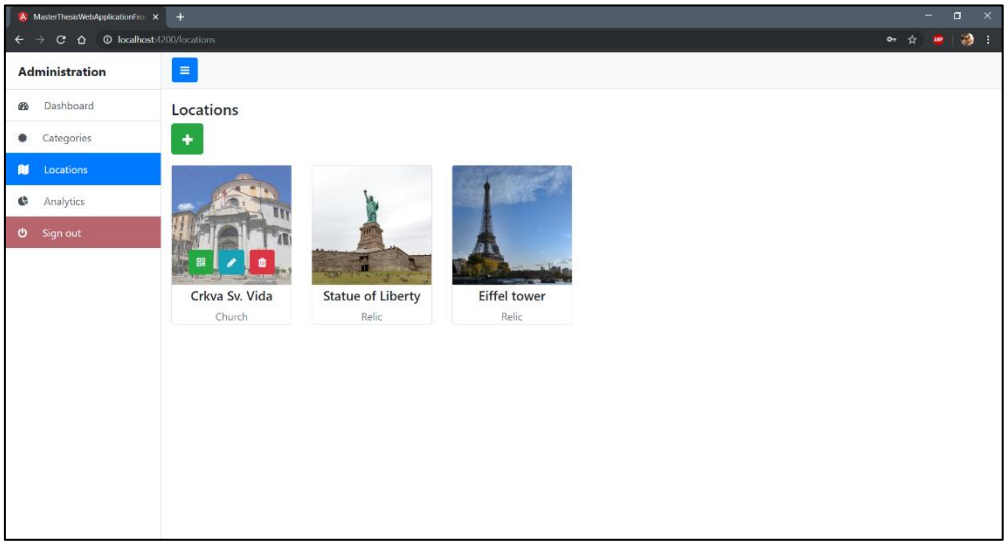
Slika 66 – dodavanje nove kategorije



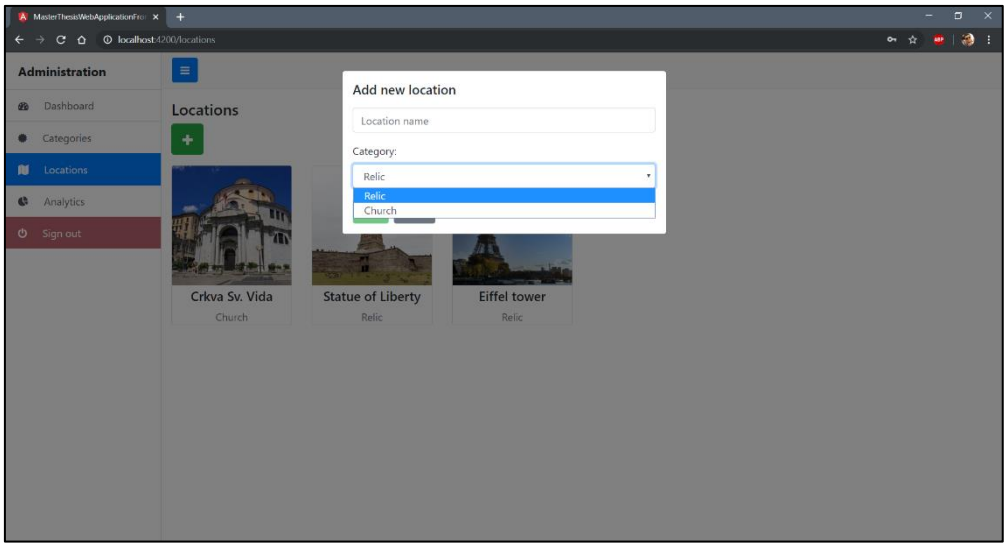
Slika 67 – uređivanje kategorije



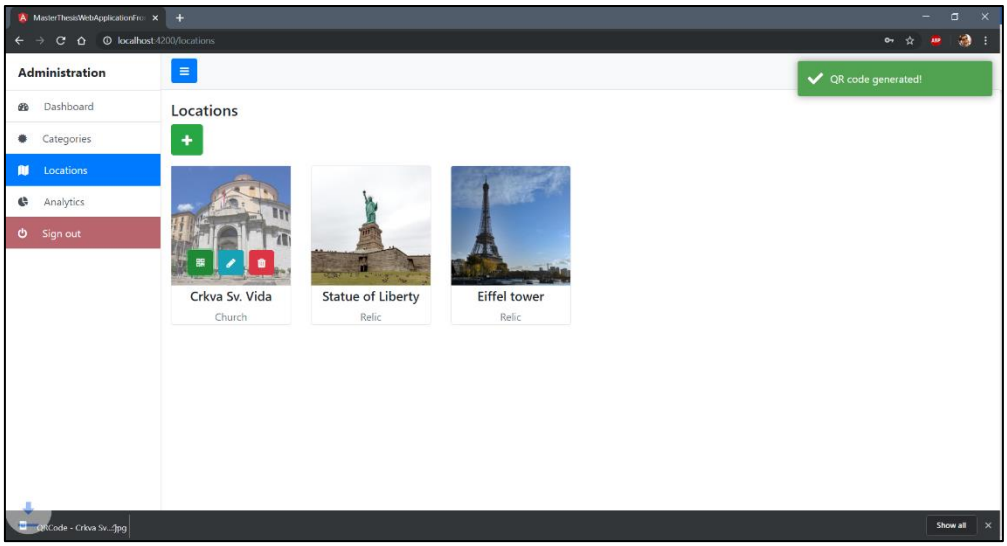
Slika 68 – brisanje kategorije



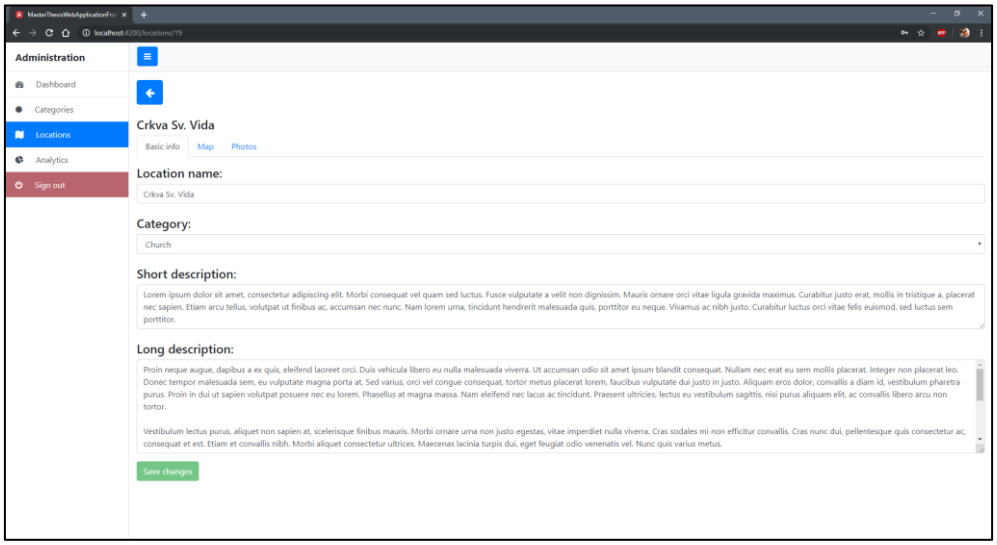
Slika 69 – pregled lokaliteta



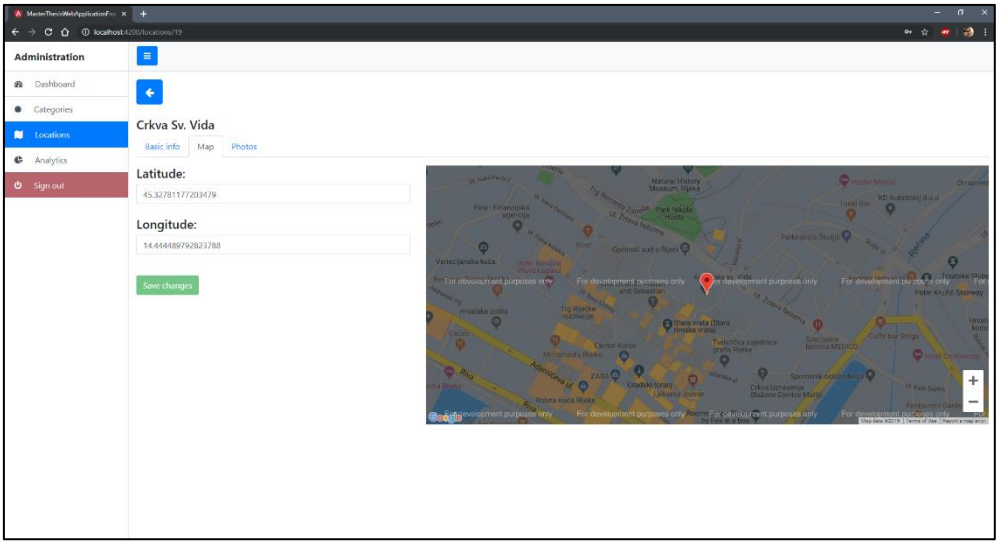
Slika 70 – dodavanje novog lokaliteta



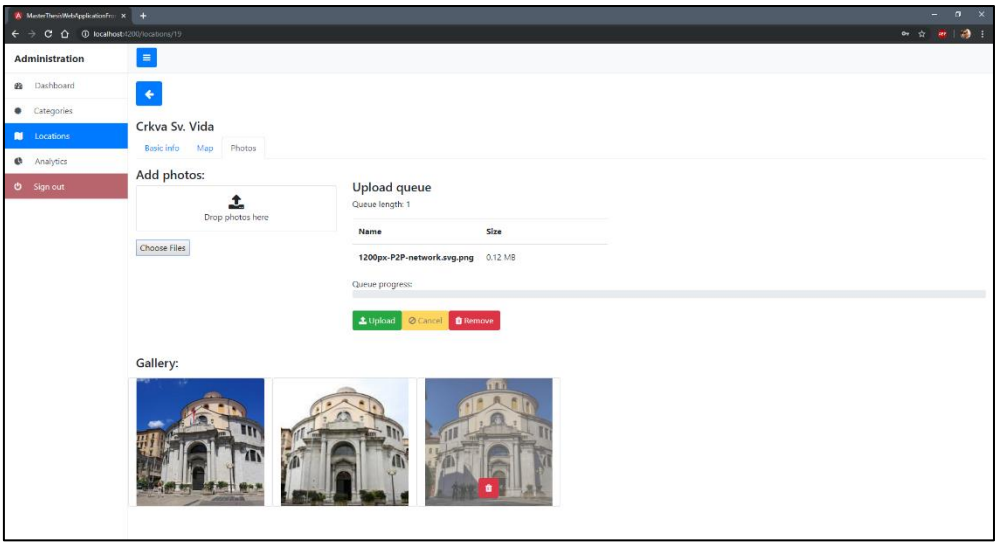
Slika 71 – generiranje QR koda za lokalitet



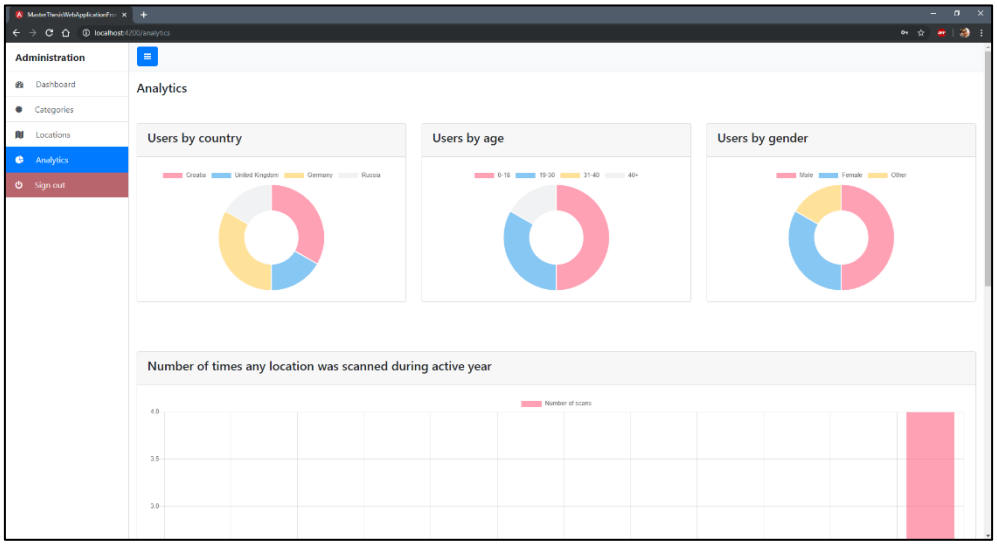
Slika 72 – uređivanje osnovnih informacija o lokalitetu



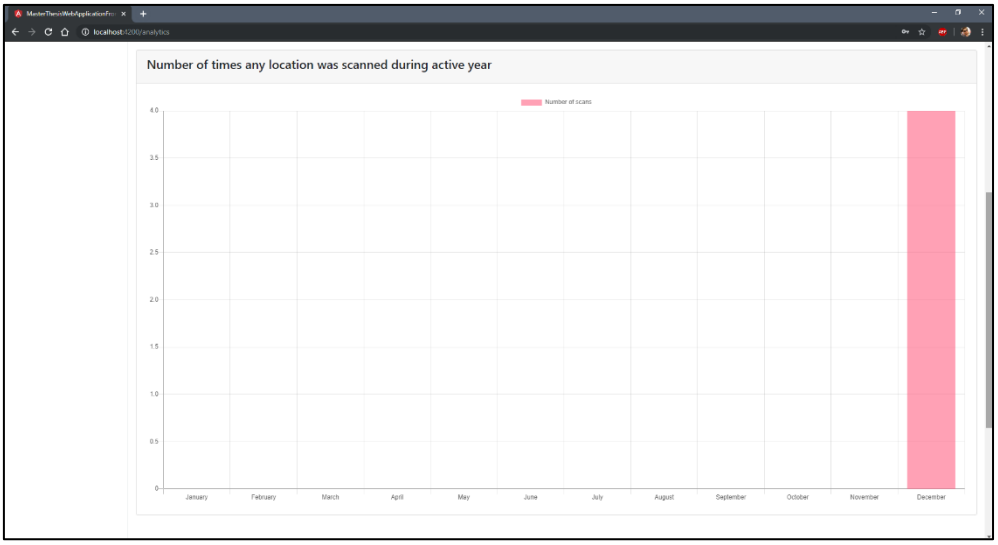
Slika 73 – uređivanje geografske lokacije lokaliteta



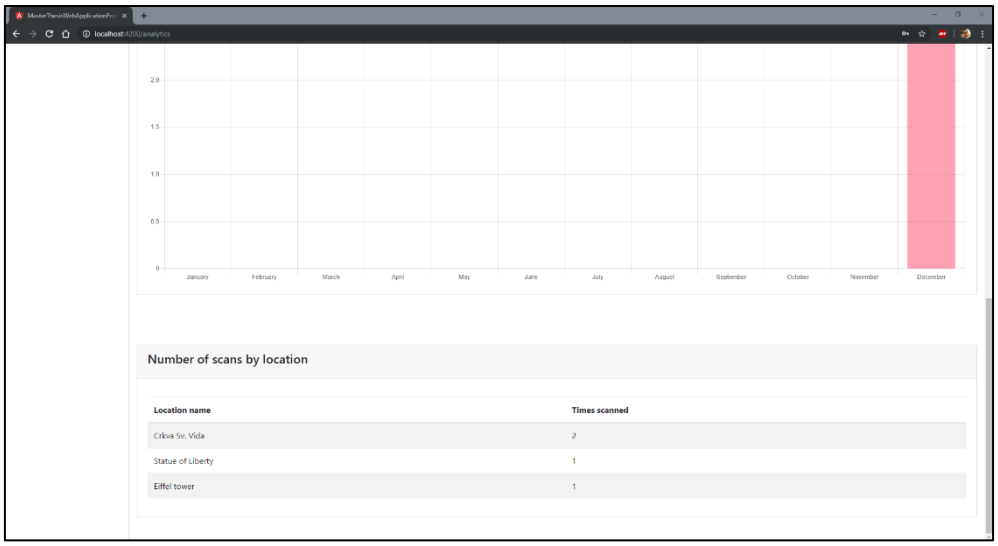
Slika 74 – dodavanje i brisanje slika lokaliteta



Slika 75 – statistika (1/3)



Slika 76 – statistika (2/3)



Slika 77 – statistika (3/3)

6.4. Mobilna aplikacija

Klijentska strana mobilne aplikacije, napravljena u *Ionic* programskom okviru koristeći *Angular*, strukturirana je slično kao i web aplikacija. kroz dva modula i 5 komponenti.

Prvi modul je korijenski modul *app.module.ts* koji objedinjuje tri komponente:

- Korijensku komponentu: *AppComponent* komponenta (Sve komponente se učitavaju u ovoj komponenti)
- Login stranicu: *Login* komponenta
- Komponentu s karticama: *Tabs* komponenta

Drugi modul je modul kartica *tabs.module*.ts koji objedinjuje četiri komponente koje se učitavaju unutar *Tabs* komponente:

- Neotkriveni lokaliteti: *Discover* komponenta
- Otkriveni lokaliteti: *Discovered* komponenta
    - Kartica lokaliteta: *LocationCard* komponenta
    - Informacije o otkrivenom lokalitetu: *LocationMoreInfo* komponenta
- Korisnički profil: *Profile* komponenta

Opcionalni i vanjski moduli koji su korišteni u projektu:

- *JwtModule (@auth0/angular-jwt)* koji dodaje token u svaki zahtjev prema serveru i omogućuje provjeru je li token istekao
- *HttpClientModule (@angular/common/http)* koji sadrži metode za slanje zahtjeva prema serveru
- *ReactiveFormsModule (@angular/forms)* za kreiranje formi
- *NgxGalleryModule (ngx-gallery)* za korištenje galerije
- *AgmCoreModule (@agm/core)* za korištenje Google karti
- *LaunchNavigator (@ionic-native/launch-navigator/ngx)* za pokretanje zadane aplikacije za navigaciju (Google Maps za Android, Maps za iOS)
- *BarcodeScanner (@ionic-native/barcode-scanner/ngx)* za skeniranje QR kodova

S obzirom da *Ionic* programski okvir omogućuje pretvaranje web aplikacije u nativnu mobilnu aplikaciju to je i učinjeno. Dakle, razvijena je još jedna *Angular* web aplikacija koja se razlikuje od uobičajene web aplikacije po tome što su korištene *Ionic* HTML komponente (slika 79), te *Cordova* dodaci za pristup nativnim komponentama.

Za korištenje nativne komponente skenera za *QR* kodove dovoljno je putem naredbenog retka pokrenuti:

```
> ionic cordova plugin add phonegap-plugin-barcodescanner
> npm install @ionic-native/barcode-scanner
```

Nakon toga, *Cordova* dodatak može se koristiti kao na primjeru slike 78.

```
import { BarcodeScanner } from '@ionic-native/barcode-scanner/ngx';

@Component({
  selector: 'app-discover',
  templateUrl: 'discover.page.html',
  styleUrls: ['discover.page.scss']
})
export class DiscoverPage implements OnInit {
  constructor(private barcodeScanner: BarcodeScanner) {}

  ngOnInit() {
  }

  scanQrCode() {
    this.barcodeScanner.scan().then(barcodeData => {
      console.log(barcodeData.text);
    }).catch(err => {
      console.log(err);
    });
  }
}
```

Slika 78 – korištenje nativnih komponenti

```
<img src="/assets/header.jpg" alt="" />
<ion-content class="ion-padding">
  <form *ngIf="showLoginForm" [formGroup]="loginForm" (ngSubmit)="login()">
    <ion-grid>
      <ion-row class="ion-justify-content-center">
        <ion-col class="ion-align-self-center" size-md="6" size-lg="5" size-xs="12">

          <div class="ion-padding">
            <ion-item>
              <ion-label position="stacked">Username:</ion-label>
              <ion-input name="email" type="email" formControlName="username" ngModel></ion-input>
            </ion-item>
            <ion-item>
              <ion-label position="stacked">Password:</ion-label>
              <ion-input name="password" type="password" formControlName="password" ngModel></ion-input>
            </ion-item>
          </div>
          <div class="ion-padding">
            <ion-button size="medium" type="submit" [disabled]="!loginForm.valid" expand="block">Login</ion-button>
            <ion-item-divider></ion-item-divider>
            <div class="ion-text-center">
              <p>or</p>
            </div>
            <ion-button size="medium" type="button" (click)="switchForm()" color="secondary" expand="block">Sign up
            </ion-button>
          </div>
        </ion-col>
      </ion-row>
    </ion-grid>
  </form>
```

Slika 79 – Ionic HTML komponente

6.4.1. Pokretanje na nativnim uređajima

Za potrebe pokretanja mobilne aplikacije na *Android* operativnom sustavom, na računalu je potrebno instalirati *Java Development Kit* 8 (*JDK* 8) i *Android Studio*. Nakon toga, potrebno je spojiti *Android* uređaj putem USB kabla s računalom, na uređaju omogućiti USB ispravljanje pogrešaka (eng. *USB Debugging mode*) te pokrenuti naredbu u naredbenom retku:

> *ionic cordova run android –device*

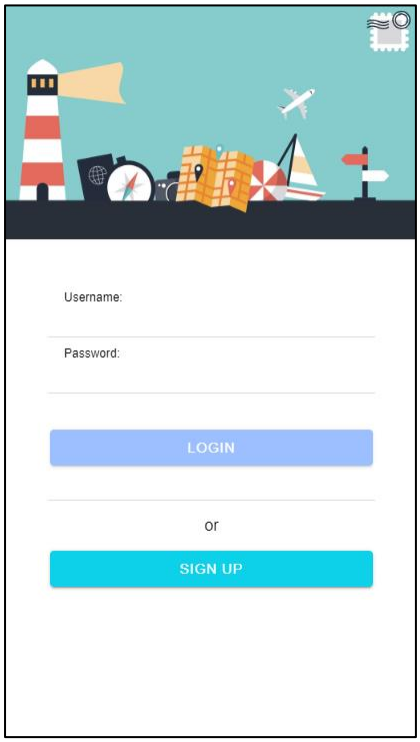
Za potrebe pokretanja mobilne aplikacije na *iOS* operativnom sustavom, potreban je minimalno *Xcode* 7, *iOS9* i *AppleID*. Nakon toga, potrebno je pokrenuti naredbu u naredbenom retku:

> *ionic cordova build ios –prod*

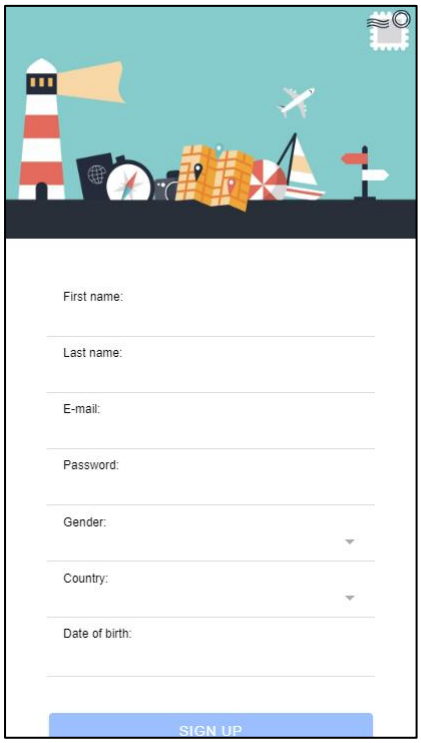
U *Xcode*-u otvoriti .xodeproj datoteku koja se nalazi u *platforms/ios/*, spojiti *iPhone* USB kablom i pokrenuti aplikaciju preko *Xcode*-a na spojenom uređaju.

6.4.2. Slike zaslona

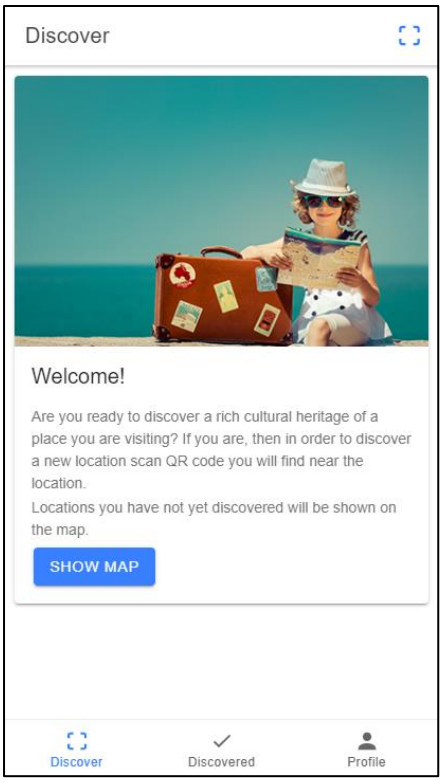
Na slikama 80-93 prikazane su slike zaslona implementirane mobilne aplikacije.



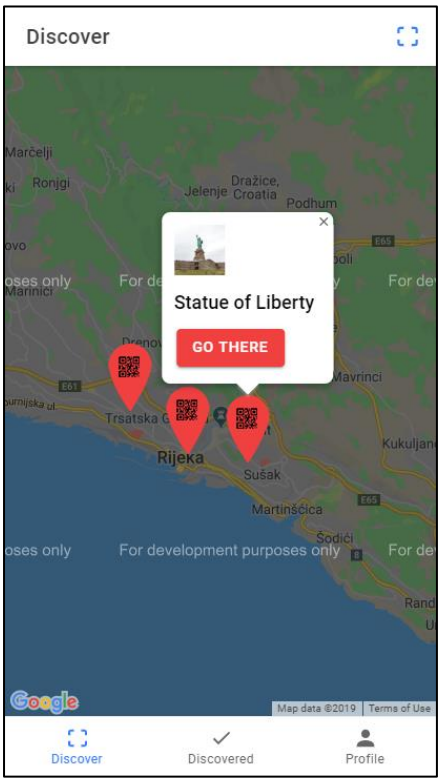
Slika 80 – login komponenta - prijava



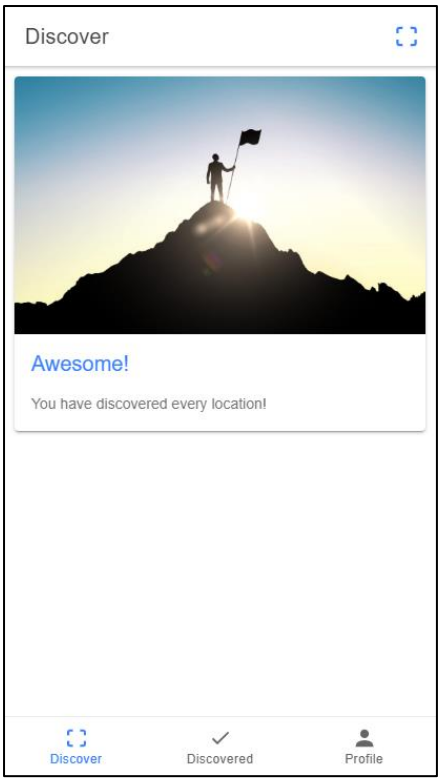
Slika 81 - login komponenta - registracija



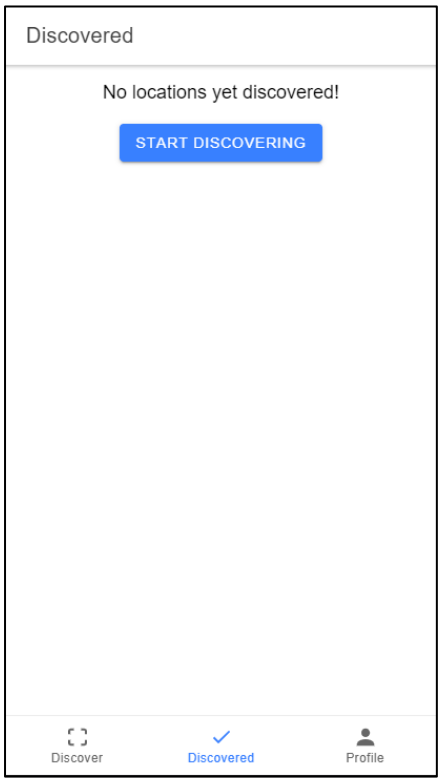
Slika 82 - discover komponenta – početni zaslon



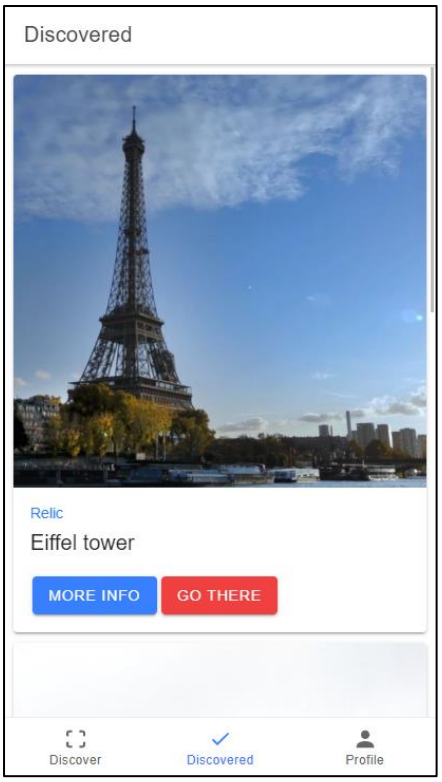
Slika 83 - discover komponenta - neotkriveni lokaliteti



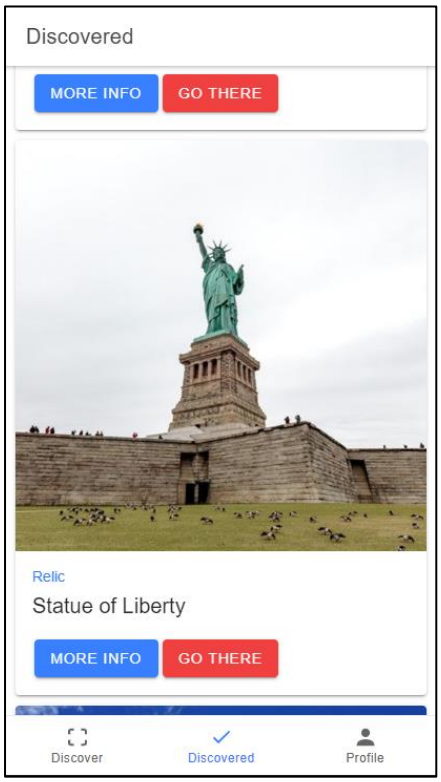
Slika 84 - discover komponenta - svi lokaliteti otkriveni



Slika 85 – discovered komponenta – niti jedan lokalitet otkriven



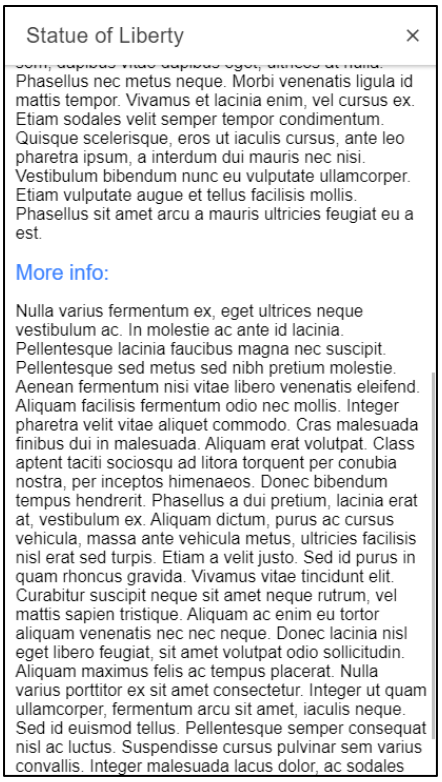
Slika 86 – discovered komponenta (1/2)



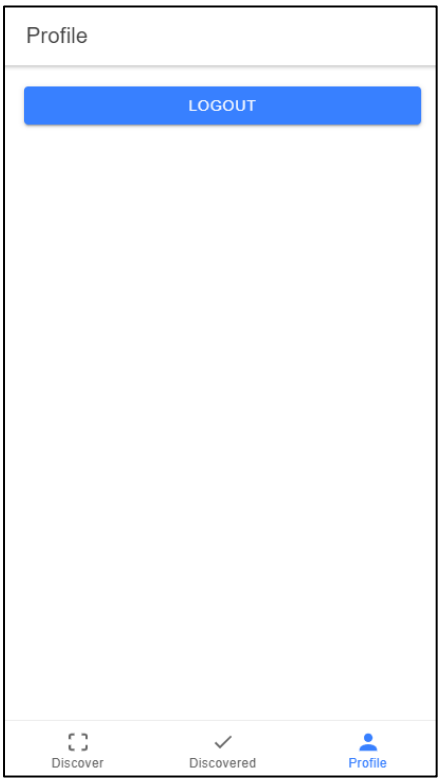
Slika 87 – dicovered komponenta (2/2)

Slika 88 – more-info komponenta (1/4)

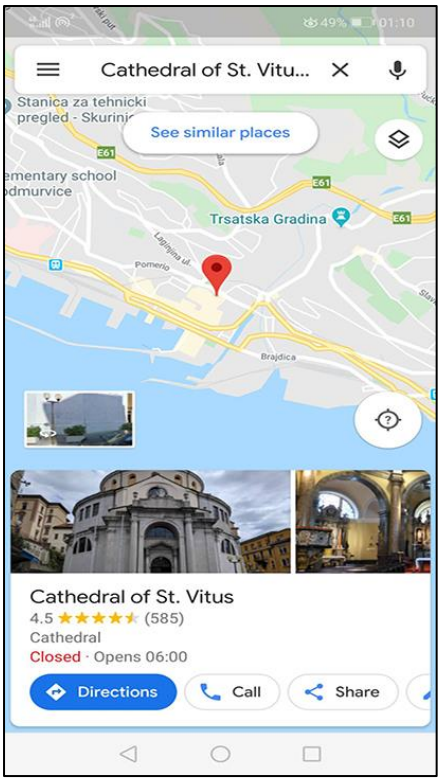
Slika 89 – more-info komponenta (3/4)



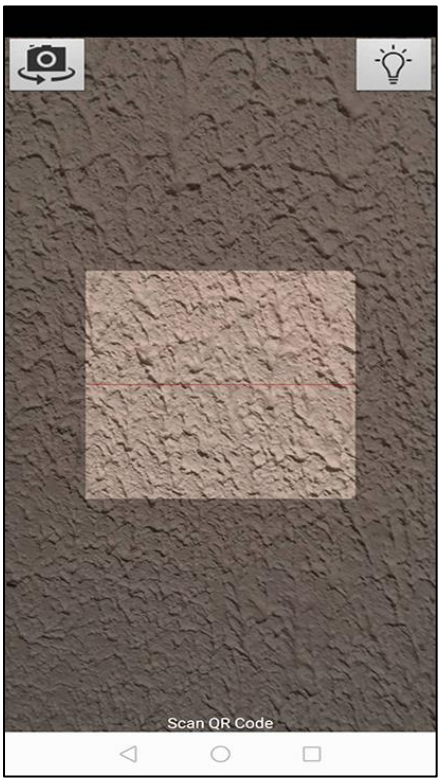
Slika 90 – more-info komponenta (4/4)



Slika 91 – profile koponenta



Slika 92 – pokretanje nativne komponente (Aplikacije za navigaciju)



Slika 93 – pokretanje nativne komponete (Kamera za skeniranje QR koda)

7. Budući rad na projektu

Neke funkcionalnosti i unaprjeđenja sustava koje će biti implementirane ako se nastavi raditi na ovom projektu:

Kreiranje super admina koji će imati mogućnosti kreiranja regija i kreiranja administratora. Svaki administrator bi pripadao jednoj regiji za koju može, kao i sada, uređivati lokalitete. Putem mobilne aplikacije, na temelju trenutne lokacije, bi se ustvrdilo u kojoj regiji se korisnik trenutno nalazi te bi mu se prikazivali relevantni lokaliteti. Na ovaj način, projekt bi mogao npr. pokriti sve turističke zajednice republike Hrvatske.

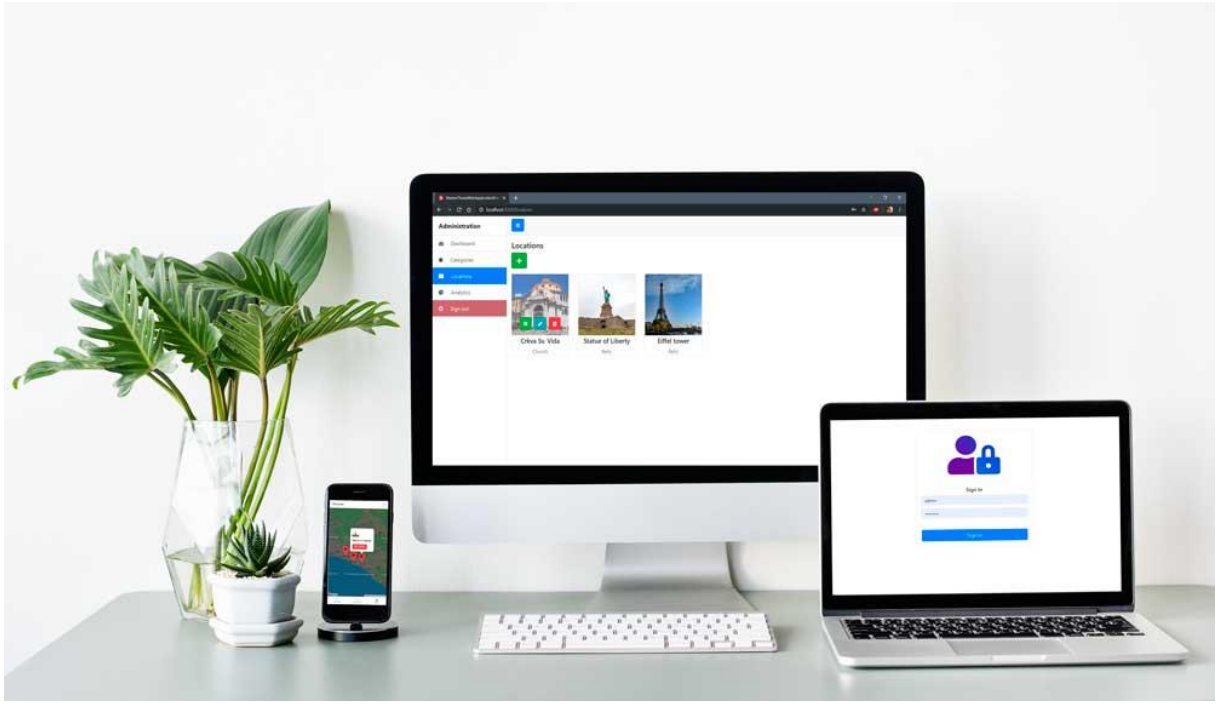
Dodatna zaštita načina otkrivanja lokaliteta – s obzirom da je QR kod slika, jedan korisnik može poslikati relevantne QR kodove i proslijediti ostalim korisnicima koji bi ih mogli skenirati. Dodatna zaštita bi bila da se prilikom procesa otkrivanja lokaliteta korisnika traži i trenutna lokacija te na temelju nje uspoređuje udaljenost lokaliteta i korisnika. Iako i ovo rješenje ne garantira validan način otkrivanja lokaliteta jer se lokacija uređaja može prilagođavati svakako je korak naprijed u odnosu na trenutnu situaciju.

Dodatne mogućnosti – funkcionalnost aplikacije može se proširiti da se osim lokaliteta kulturne baštine, mogu dodavati i ostale točke interesa koje bi bile zanimljive turistima: restorani, kafići, trenutni događaji u gradu i slično.

Dizajn aplikacija – Za dizajniranje aplikacije korištene su zadane postavke elemenata s minimalno konfiguracije CSS-a jer je naglasak projekta nije bio na izgledu već na funkcionalnosti.

8. Zaključak

Ideja rada bila je da kroz proces izrade ovog sustava prođem implementaciju kompletnog programskog rješenja koje se sastoji od mobilne aplikacije, web aplikacije i aplikativnog programskog sučelja za komunikaciju između njih. Maketa izrađenog sustava prikazana je na slici 94.



Slika 94 – maketa (eng. Mockup) izrađenog sustava

Odabrana tematika vezana uz kulturnu baštinu potaknuta je činjenicom da je 2020. godine Europska prijestolnica kulture upravo grad Rijeka te bih putem turističke zajednice grada, ako se pokaže obostrani interes, mogao pokrenuti projekt u produkcijsko okruženje. Modularnost je bila u fokusu tijekom izrade svakog dijela sustava te bi željene preinake na sustavu bile brzo implementirane bez da se narušava trenutna stabilnost sustava.

Odabrane tehnologije nisu bile u fokusu prilikom idejnog razmatranja projekta, već su odabrane zbog svoje popularnosti i velike zajednice ljudi koja ih koristi. Nisam imao puno iskustva s korištenjem ovih tehnologija te smatram da sam prilikom implementacije ovog sustava stekao znanje neprocjenjive vrijednosti za nekoga koga zanima razvoj softvera.

Sav programski kod dostupan je na mom Github profilu: https://github.com/zubi96/

Aplikativno programsko sučelje:
https://github.com/zubi96/Master-Thesis-Web-Application-API

Klijentska strana web aplikacije:
https://github.com/zubi96/Master-Thesis-Web-Application-Frontend

Mobilna aplikacija:
https://github.com/zubi96/Master-Thesis-Mobile-Application

9. Popis slika

10. Popis referenci

[1] Client-Server Arhitecture
https://www.britannica.com/technology/client-server-architecture
(pristupljeno: Studeni, 2019.)

[2] Peer-to-peer network
https://en.wikipedia.org/wiki/Peer-to-peer
(pristupljeno: Studeni, 2019.)

[3] Mobile applications
https://www.techopedia.com/definition/2953/mobile-application-mobile-app
(pristupljeno: Studeni, 2019.)

[4] Mobile Operating System Market Share Worldwide
https://gs.statcounter.com/os-market-share/mobile/worldwide
(pristupljeno: Studeni, 2019.)

[5] Microsoft ends support for Windows 10 Mobile
https://www.windowslatest.com/2019/12/10/microsoft-ends-support-for-windows-10-mobile
(pristupljeno: Studeni, 2019.)

[6] Types of mobile apps
https://nimbleworks.co.uk/types-of-apps.html
(pristupljeno: Studeni, 2019.)

[7] Introduction to ASP.NET Core
https://docs.microsoft.com/en-us/aspnet/core/?view=aspnetcore-3.1
(pristupljeno: Studeni, 2019.)

[8] Overview of ASP.NET Core MVC
https://docs.microsoft.com/en-us/aspnet/core/mvc/overview?view=aspnetcore-3.1
(pristupljeno: Studeni, 2019.)

[9] ASP.NET Core MVC Web Application (Project Structure)
https://medium.com/net-core/asp-net-core-mvc-web-application-project-structure-3ccaa244fa66
(pristupljeno: Prosinac, 2019.)

[10] Architecture overview
https://angular.io/guide/architecture
(pristupljeno: Studeni, 2019.)

[11] TypeScript
https://en.wikipedia.org/wiki/TypeScript
(pristupljeno: Prosinac, 2019.)

[12] Introduction to the Angular Docs
https://angular.io/docs
(pristupljeno: Prosinac, 2019.)

[13] What is Ionic Framework
https://ionicframework.com/docs/intro
(pristupljeno: Prosinac, 2019.)

[14] Ionic Framework Documentation
https://ionicframework.com/docs
(pristupljeno: Prosinac, 2019.)

[15] What is Code-First?
https://www.entityframeworktutorial.net/code-first/what-is-code-first.aspx
(pristupljeno: Prosinac, 2019.)

[16] Repository Pattern - A data persistence abstraction
https://deviq.com/repository-pattern/
(pristupljeno: Prosinac, 2019.)