MOBILNA APLIKACIJA ZA VOĐENJE TRENINGA

Popović, Ivan Zvonimir

VISOKO UČILIŠTE ALGEBRA

ZAVRŠNI RAD

MOBILNA APLIKACIJA ZA VOĐENJE TRENINGA

Ivan Zvonimir Popović

Zagreb, veljača 2020.

„Pod punom odgovornošću pismeno potvrđujem da je ovo moj autorski rad čiji niti jedan dio nije nastao kopiranjem ili plagiranjem tuđeg sadržaja. Prilikom izrade rada koristio sam tuđe materijale navedene u popisu literature ali nisam kopirao niti jedan njihov dio, osim citata za koje sam naveo autora i izvor te ih jasno označio znakovima navodnika. U slučaju da se u bilo kojem trenutku dokaže suprotno, spreman sam snositi sve posljedice uključivo i poništenje javne isprave stečene dijelom i na temelju ovoga rada“.

U Zagrebu, datum.

Ivan Zvonimir Popovic

Predgovor

Zahvaljujem se svom mentoru, profesoru Bojanu Fulanoviću na smjernicama i uputa koje mi je davao tokom izrade završnog rada. Također se želim zahvaliti obitelji i djevojci na iskazanoj podršci pri pohađanju Visokog učilišta Algebra te uspješnom završetku istoga. Na posljetku, želim se zahvaliti Visokom učilištu Algebra na prenesenom znanju tokom ove tri godine.

Prilikom uvezivanja rada, Umjesto ove stranice ne zaboravite umetnuti original potvrde o prihvaćanju teme završnog rada kojeg ste preuzeli u studentskoj referadi

Sažetak

Rad opisuje izradu softverskog rješenja namijenjenu za praćenje i menadžment treninga. Unutar rada su opisani principi i pristupi korišteni pri izradi baze podataka, *web* aplikacijsko programskog sučelja (engl. *application programming interface,* skraćeno API) te mobilne aplikacije, kao i interakcije između istih. Cilj ovog rada je prikazati, kroz praktični i teorijski dio kako izgraditi sustav koji može komunicirati sa više vrsta klijenata te kako izgraditi mobilnu aplikaciju koja komunicira s istim. Tokom izrade rada su korištene najnovije, trenutno dostupne, tehnologije kao što su ASP.NET Core, Entity Framework Core i React Native.

Ključne riječi: menadžment treninga, mobilna aplikacija, Entity Framework Core, Code First pristup, ASP.NET Core, *web* API, React Native

Abstract

This paper describes the development of a software solution that is meant for training managment and monitoring. Principles and approaches are described used to create the database, application programming interface and mobile application, as well as the interactions between them. The aim of this pape is to show, through practcal and theoretical part, how to build a system that can communicate with multiple types of clients and how to build a mobile application that communicates with it. The latest, currently available technologies such as ASP.NET Core, Entity Framework Core, and React Native were used in the development process.

Keywords: training managment, mobile application, Entity Framework Core, Code First approach, ASP.NET Core, *web* API, React Native

Sadržaj

1. Uvod

Unutar IT industrije većina poslova se izvršava sjedeći, te zahtijeva veći mentalni napor dok je fizički zanemariv. Nakon osam sati sjedenja na poslu većina ljudi ostatak dana provede bez fizičke aktivnosti. Zbog mentalne iscrpljenosti postoji manjak motivacije za kretanje ili treniranje, što za posljedicu može dovesti do raznih zdravstvenih problema. Stoga cilj ovog rada je napraviti softversko rješenje u obliku mobilne aplikacije za menadžment treninga, koje omogućuje jednostavno praćenje i menadžment treninga kako bi uvećao šanse za trening te podignuo motivaciju korisnicima same aplikacije. Sustav se sastoji od više komponenata te je napravljen kako bi se, u budućnosti, omogućilo razvijanje više klijentskih aplikacija neovisno o tehnologijama u kojima će biti rađene. U nastavku rada slijedi detaljniji opis korištenih tehnologija te načina izrade svake komponente zasebno.

2. Arhitektura aplikacije

Arhitektura završnog rada se sastoji od tri različite komponente:

- Baza podataka
- Web API
- Mobilna aplikacija

Baza podataka se koristi za skladištenje samih podataka u formatu tablica sa stupcima i redcima. Klijent, u ovome slučaju mobilna aplikacija koja služi za prikaz podataka, koristi *web* API kako bih dohvatio, dodao, izmijenio ili izbrisao podatke iz baze podataka. Prednost ovakve arhitekture je što klijentska aplikacija nije ograničena na specifičnu tehnologiju, kao ni na broj klijenata.

2.1. Podatkovni sloj

Podatkovni sloj je osnova svake aplikacije. Unutar podatkovnog sloja ulazi sve od spremanja podataka unutar baze podataka do načina na koji to radimo. Baza podataka koja je korištena unutar ovog projekta je Microsoft SQL Server (skraćeno MSSQL). Jezik koji se koristi unutar MSSQL-a je Transact SQL koji omogućuje pisanje upita prema bazi podataka. Transact SQL se može koristiti za pisanje jednostavnih kao i složenijih SQL upita koji imaju mogućnost promjene samog programskog toka [1]. Stvaranje i održavanje koda baze podataka je zahtjevan i izazovan posao za programera. Kako bismo olakšali i ubrzali izradu aplikacija, unutar ovog projekta se koristi ORM. Objektno relacijski maperi (engl. *Object-Relational Mapper,* skraćeno ORM) omogućuje programerima da se fokusiraju na arhitekturu i dizajn aplikacije te se brine za generiranje samog koda i stvaranja baze podataka iz proslijeđenog konteksta Na taj način osoba koja dizajnira i izgrađuje aplikaciju se ne mora brinuti, niti poznavati jezik potreban za kompletnu izgradnju i dizajn baze podataka, nego se može u potpunosti fokusirati na izradu aplikacije.

2.1.1. *Code First* pristup

Takozvani *Code First* pristup pisanja, to jest konstruiranja baze podataka, nam omogućuje da na brz i efikasan način definiramo tablice te relacije između njih, kao i tipove podataka unutar samih tablica. Sve počinje sa kreiranjem i definiranjem domenskih razreda, dok kod

konvencionalnog pristupa se prvo kreira i definira baza podataka sa svojim entitetima te se nakon toga ti entiteti preslikavaju u razrede unutar same aplikacije kako bih se kasnije mogli međusobno mapirati (engl. *mapping*). Nakon izrade domenskih razreda definiraju se dodatne veze (poput više-prema-više) između razreda te opcionalne restrikcije nad svojstvima razreda.

Neke od benefita *Code First* pristupa su:

- Brzina razvijanja aplikacije – nema brige o stvaranju baze podataka, programeri koji nemaju pozadinu programiranja u istoj mogu odmah početi s izgradnjom same aplikacije
- Jednostavnost – bazu podataka je moguće održavati i ažurirati kroz jednostavno uređivanje domenskih razreda
- Logika ostaje unutar koda – kod konvencionalnog pristupa, logika često završi unutar procedure same baze podataka

2.1.1.1 Definiranje razreda

U *code first* pristupu definiranjem razreda stvaramo ujedno i buduće tablice u bazi. S obzirom na svojstva te postavljena ograničenja razreda, kasnije u procesu generiranja baze iz razreda, dobiti ćemo tablice koje odgovaraju postavljenim pravilima unutar istih.

Unutar koda (Kôd 2.1 Definicija razreda) su definirane ukupno tri varijable. Varijabla *Id* predstavlja jedinstveni identifikator koji se dodjeljuje svakom računu (engl. *Account),* te je tipa *int* što predstavlja cjelobrojnu vrijednost. Varijabla *UserName* predstavlja korisničko ime vrste *string,* koji predstavlja tekst. Varijabla *Password* je također tipa *string* te predstavlja šifru korisnika koja će se kasnije koristiti za prijavu u sustav.

```
public class Account
      {
            public int Id { get; set; }
            [Required]
            [StringLength(50)]
            public string UserName { get; set; }
            [Required]
            [StringLength(50)]
            public string Password { get; set; }
      }
```

Kôd 2.1 Definicija razreda

2.1.1.2 Postavljanje ograničenja na razrede

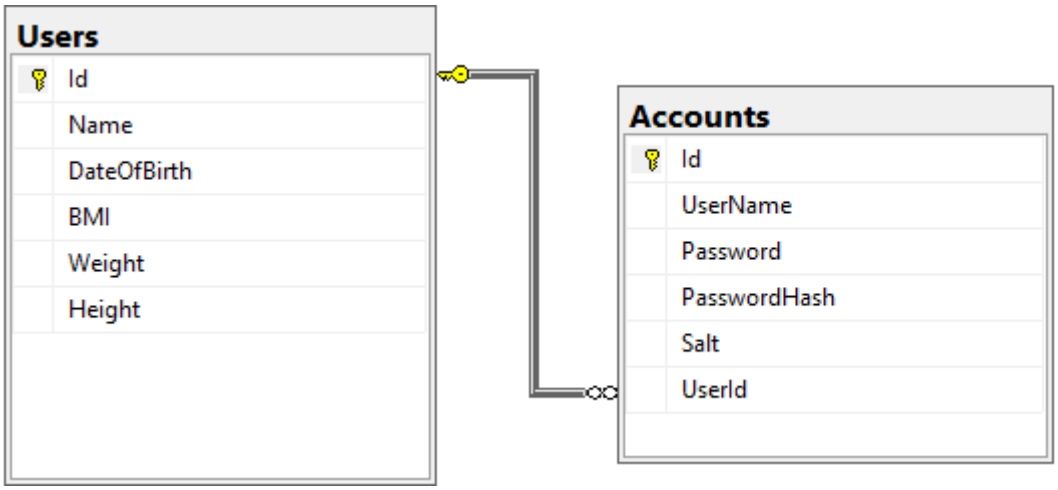
Za postavljanje ograničenja, na određene varijable razreda, koristimo takozvane podatkovne anotacije (engl. *dana annotations).* Glavna svrha podatkovnih anotacija je validacija korisnikovog unosa podataka. Na slici (2.1 Razred *Account*), iznad varijable *UserName,* postavljen je atribut pod imenom *StringLength* koji nam omogućuje postavljanje maksimalnog broja znakova, te je u unutar istog proslijeđen broj pedeset što znači da u varijablu *UserName* ne može biti spremljeno više od pedeset znakova. Iznad varijable *Password* je postavljen atribut *Required* koji govori da varijabla ne smije biti bez vrijednosti (engl. *null*), to jest da varijabla *Password* mora sadržavati barem jedan znak. Ovo su samo neki od mogućih atributa koje je moguće postaviti nad imena varijabli i time ograničiti te validirati korisnikov unos.

2.1.1.3 Uspostava veza između razreda

Općenito u bazama podataka ono što je potrebno tokom izrade samih tablica, koje sadrže informacije u formatu stupaca i redaka, je definirati i relacije između samih tablica. Relacije se ostvaruju pomoću primarnih i stranih ključeva. Postoji nekoliko vrsta relacija:

- jedan-prema-jedan
- jedan-prema-više
- više-prema-više

Jedan-prema-jedan relacija označava da redak u jednoj tablici može imati samo jedan odgovarajući redak u drugoj tablici i obrnuto. Na slici (Slika 2.1 Relacija jedan-prema-jedan) su prikazani tablica korisnici (engl. *Users) te* tablica računi (engl. *Accounts).* Međusobno su spojeni tako da je na tablici računi stavljen strani ključ *UserId,* koji predstavlja primarni ključ korisnika. Na taj način povezujemo točno jednog korisnika sa točno jednim računom i obrnuto.



Slika 2.1 Relacija jedan-prema-jedan

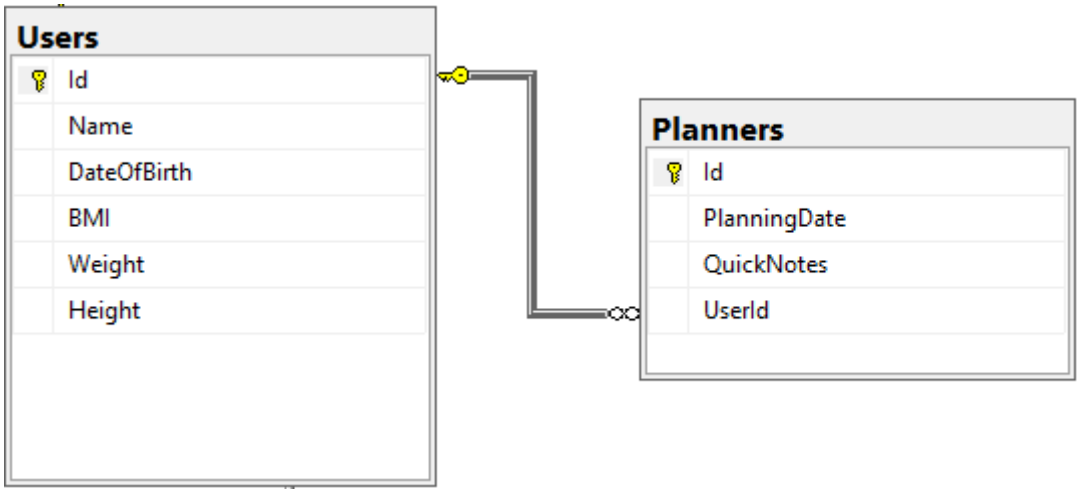
Unutar *Code-First* pristupa relaciju jedan-prema-jedan stvaramo kako je prikazano u kodu(Kôd 2.2 Relacija jedan-prema-jedan). Unutar domenskog razreda *Account* se postavlja varijabla *User, tipa User*. Domenski razred *User* sadrži sve varijable kao i tablica *Users.* Ono što smo de facto napisali unutar koda (Kôd 2.2 Relacija jedan-prema-jedan) je da domenski razred *Account* ima na sebi jednog *Usera* i obrnuto, baš kako i sama slika (Slika 2.1 Relacija jedan-prema-jedan) prikazuje. Postavljanjem varijable *User* unutar domenskog razreda *Account* ostvarujemo vezu jedan-prema-jedan prilikom generiranje baze podataka.

```
public class Account
      {
            public int Id { get; set; }
            [Required]
            [StringLength(50)]
            public string UserName { get; set; }
            [Required]
            [StringLength(50)]
            public string Password { get; set; }
            [StringLength(500)]
            public string PasswordHash { get; set; }
            [StringLength(50)]
            public string Salt { get; set; }
            public virtual User User { get; set; }
      }
```

Kôd 2.2 Relacija jedan-prema-jedan

Jedan-prema-više relacija označava da redak u jednoj tablici može imati više odgovarajućih redaka u drugoj tablici, ali redak iz druge tablice može samo imati jedan odgovarajući redak

iz prve tablice. Na slici(Slika 2.2 Relacija jedan-prema-više) su prikazani tablica Korisnici (engl. *Users*) te tablica Planeri (engl. *Planners*). Međusobno su spojeni tako da je na tablici *Planners* stavljen strani ključ *UserId* koji predstavlja primarni ključ korisnika.



Slika 2.2 Relacija jedan-prema-više

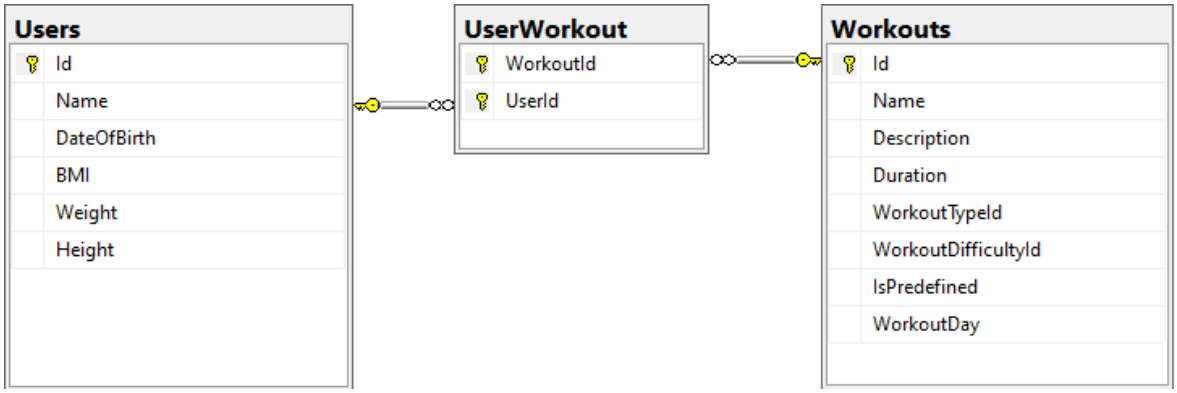
Iako na prvi pogled relacija jedan-prema-više izgleda identično, barem što se može iz slike (Slika 2.1 Relacija jedan-prema-jedan) i slike(Slika 2.2 Relacija jedan-prema-više) izvući, kao i relacija jedan-prema-jedan, postoji razlika kod definiranja domenskih razreda unutar *Code First* pristupa izrade baze podataka. Naime, kada definiramo relaciju jedan-prema-više, unutar domenskog razreda ne postavljamo varijablu koja predstavlja jedan entitet, već kolekciju entiteta. Kao što je vidljivo iz koda (Kôd 2.3 Relacija jedan-prema-više), domenski razred *User* sadrži kolekciju entiteta vrste *Planner*, te na taj način uvjetujemo i definiramo da odnos između tih entiteta je jedan-prema-više.

```
public class User
      {
            public int Id { get; set; }
            [Required]
            [StringLength(60)]
            public string Name { get; set; }
            [Required]
            public DateTime DateOfBirth { get; set; }
            [Required]
            public double BMI { get; set; }
            [Required]
            public double Weight { get; set; }
            [Required]
            public double Height { get; set; }
```

```
            public virtual ICollection<Planner> Planners {
get; set; }
        }
```

Kôd 2.3 Relacija jedan-prema-više

Relacija više-prema-više označava da redak iz jedne tablice može imati više odgovarajućih redaka u drugoj tablici i obrnuto . Na relaciju više-prema-više se može gledati kao dvije jedan-prema-više relacije, spojene sa dodatnom tablicom. Na slici (Slika 2.3 Relacija više-prema-više) su prikazani tablica *Users* te tablica *Workouts te su* međusobno spojene sa novom tablicom pod imenom *UserWorkout.* Unutar tablice *UserWorkout* stavljeni su strani ključevi *WorkoutId,* koji referencira primarni ključ *Id* iz tablice Workouts, te *UserId,* koji referencira primarni ključ *Id* iz tablice *Users*.



Slika 2.3 Relacija više-prema-više

Unutar *Code First* pristupa relaciju više-prema-više definiramo stvaranjem novog domenskog razreda *UserWorkout.* Unutar istog definiramo varijable *Workout* te *User, tipa Workout* i *User,* kao što je vidljivo u kodu(Kôd 2.4 Relacija više prema više), koji podatcima odgovaraju podatcima tablica na slici(Slika 2.3 Relacija više-prema-više). Kako bi ostvarili traženu relaciju, unutar domenskih razreda *User* i *Workout* definiramo varijablu koja sadrži kolekciju entiteta tipa *UserWorkouts,* kako je i prikazano unutar koda (Kôd 2.5 Definicija kolekcije entiteta za relaciju više-prema-više) za razred *User*

```
public class UserWorkout
        {
            public int WorkoutId { get; set; }
            public virtual Workout Workout{ get; set; }
            public int UserId{ get; set; }
            public virtual User User{ get; set; }
```

```
        }
```

Kôd 2.4 Relacija više prema više

```
public class User
        {
                public int Id { get; set; }
                [Required]
                [StringLength(60)]
                public string Name { get; set; }
                [Required]
                public virtual ICollection<UserWorkout>
UserWorkouts { get; set; }
        }
```

Kôd 2.5 Definicija kolekcije entiteta za relaciju više-prema-više

2.1.2. Korištene tehnologije

Kako bi se baza generirala iz prethodno definiranih domenskih razreda, zajedno sa svim postavljenim svojstvima i restrikcijama koristimo ORM. ORM omogućuje rad s bazom podataka koristeći objekte, to jest domenske razrede, te eliminiraju potrebu za pisanje većinu koda koji služi isključivo za pristup podatcima iz baze. ORM koji se koristi unutar ovog sustava je *Entity Framework Core* (skraćeno EF Core). EF Core je *Microsoft-ov* ORM, te je zapravo lagana (engl. *lightweight*) verzija, otvorenog koda (engl. *open-source*) *Entity Framework-a* [2]. Neke od prednosti EF Core-a nasprem EF-a su:

- Veća brzina izvršavanja upita [3]
- Mogućnost izvršavanja na bilo kojem operacijskom sustavu (engl. *cross-platform*)
- Mogućnost proširivosti (engl. extensibility)

2.1.2.1 Kreiranje konteksta

Podatkovni kontekst je ono što nam omogućuje slanje upita na samu bazu podataka te spremanje entiteta unutar baze podataka. Kontekst se stvara kreiranjem novog razreda koji nasljeđuje razred *DbContext*. Jednom kreiran, kontekst sadržava popis svih entiteta koji će kasnije biti prevedeni u tablice unutar baze podataka, kao i dodatne konfiguracije koje će se izvršiti tokom stvaranja baze podataka. Unutar koda (Kôd 2.6 Stvaranje konteksta za bazu podataka) vidimo popisane entitete prema čijim varijablama će EF Core izgrađivati bazu. Svi entiteti su potpisani kao takozvani *DbSet*-ovi, što je razred koji nam omogućuje

izvršavanje operacija nad danom tablicom. Neke od bitnijih su *create, read, update* i *delete* (skraćeno *CRUD*) operacije. Unutar metode *OnModelCreating* (Kôd 2.6 Stvaranje konteksta za bazu podataka) postavljamo dodatna pravila i ograničenja na svojstva ili relacije između domenskih razreda, to jest budućih tablica. Tako je dodatno definirana relacija više-prema-više između razreda *Workout* i *User*, ali i postavljeno ograničenje *Unique,* tako da unutar varijable *UserName,* koje predstavlja korisničko ime koje će se koristiti tokom prijave u aplikaciju, može biti spremljena samo kombinacija teksta koja do sada u bazi podataka ne postoji.

```
public class ApiDbContext : DbContext
      {
            public
ApiDbContext(DbContextOptions<ApiDbContext> options) :
base(options) { }
            public DbSet<WorkoutType> WorkoutTypes{ get; set;
}
            public DbSet<WorkoutDifficulty>
WorkoutDifficulties { get; set; }
            public DbSet<Exercise> Exercises { get; set; }
            public DbSet<Workout> Workouts { get; set; }
            public DbSet<Planner> Planners { get; set; }
            public DbSet<User> Users { get; set; }
            public DbSet<Account> Accounts { get; set; }

            protected override void
OnModelCreating(ModelBuilder modelBuilder)
            {
                  modelBuilder.Entity<UserWorkout>()
                  .HasKey(uw => new { uw.UserId,
uw.WorkoutId});
                  modelBuilder.Entity<UserWorkout>()
                        .HasOne(uw => uw.Workout)
                        .WithMany(w => w.UserWorkouts)
                        .HasForeignKey(uw => uw.WorkoutId);
                  modelBuilder.Entity<UserWorkout>()
                        .HasOne(uw => uw.User)
                        .WithMany(u => u.UserWorkouts)
                        .HasForeignKey(uw => uw.UserId);

                  modelBuilder.Entity<Account>()
```

```
                .HasIndex(a => a.UserName)
                .IsUnique();
        }
    }
```

Kôd 2.6 Stvaranje konteksta za bazu podataka

2.1.2.2 Migracije

Zbog potrebe mijenjanja razrednih klasa tokom razvoja aplikacija, mora se osigurati konzistencija baze podataka koji sadrži presliku razrednih klasa u obliku tablica. Kako bi se osigurala konzistencija baze podataka sa domenskim razredima koriste se migracije (engl. *Migrations*). Migracije unutar EF Core-a nam omogućuju da nakon napravljenih promjena na domenskim razredima napravimo novu migraciju koja sadržava zadnje promijenjeno stanje istih. Migracije se rade inkrementalno s obzirom na napravljene promjene. Dodavanje nove migracije se izvršava naredbom *add-migration,* unutar *Visual Studio Power Shell-a,* prikazanoj u kodu (Kôd 2.7 Dodavanje nove migracije) te kao parametar prima ime migracije. Ime migracije se kasnije može koristiti kako bih se baza vratila u stanje točno te migracije.

```
Add-Migration InitialMigration
```

Kôd 2.7 Dodavanje nove migracije

Nakon što je migracija dodana mogu se vidjeti točne promjene s obzirom na prošlu migraciju, to jest stanje, kako je prikazano u kodu (Kôd 2.8 Detalji dodane migracije). Metode *Up* i *Down* služe EF Core-u kako bih znao što se u toj iteraciji migracije promijenilo te koje promjene mora napraviti u slučaju povratka (engl. *revert*) na tu migraciju.

```
public partial class addAccount : Migration
    {
        protected override void Up(MigrationBuilder
migrationBuilder)
        {
            migrationBuilder.AddColumn<double>(
                name: "Weight",
                table: "ExerciseWorkout",
                nullable: false,
                defaultValue: 0.0);
        }
```

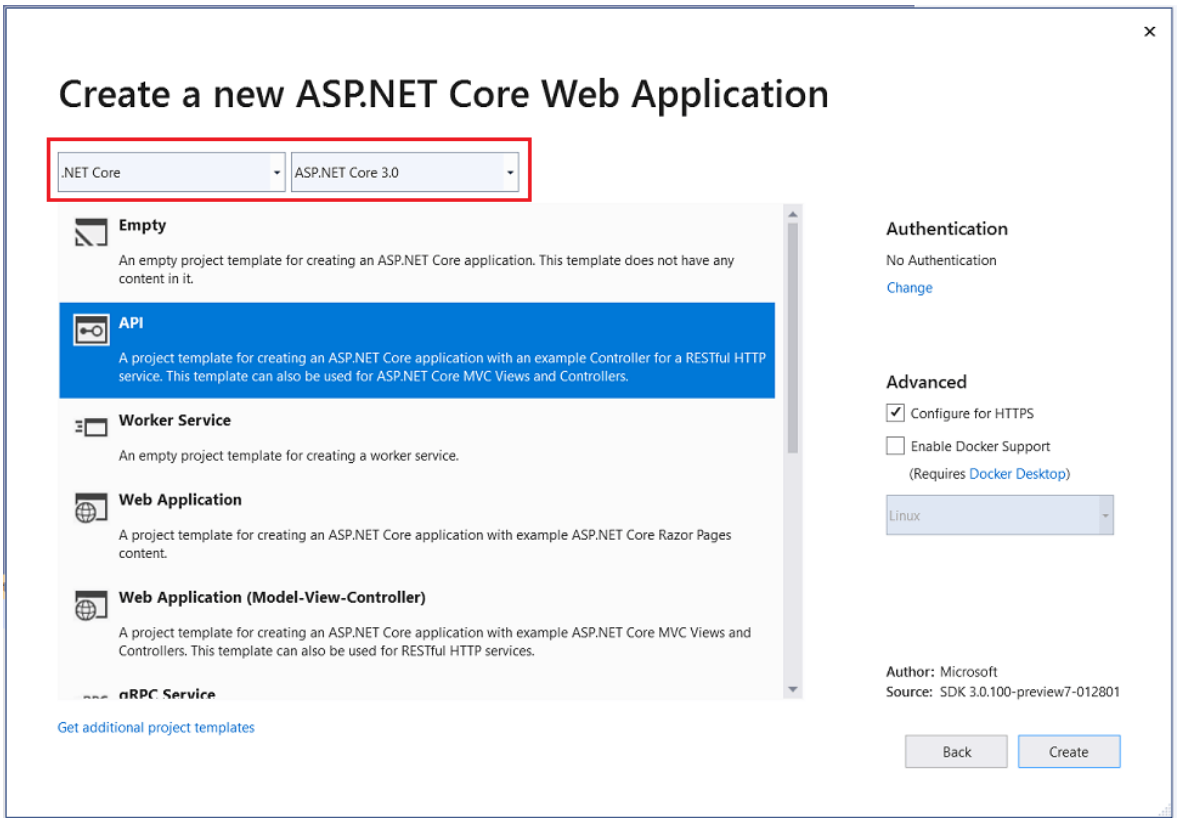
```
        protected override void Down(MigrationBuilder
migrationBuilder)
        {
            migrationBuilder.DropColumn(
                name: "Weight",
                table: "ExerciseWorkout");
        }
    }
```

Kôd 2.8 Detalji dodane migracije

Proces stvaranja baze podataka te izvršavanje potrebnih upita započinje naredbom *update-database*. Naredba *update-database* generira potrebne skripte koji bismo inače ručno pisali koristeći *Data definition language* (skraćeno DDL) te pokreće iste unutar specificirane baze podataka. Na kraju izvršavanja naredbe *update-database* dobivamo gotovu bazu podataka s tablicama i relacijama koji su preslika domenskih razreda i definiranih relacija unutar istih.

2.2. Sloj poslovne logike

Za poslovnu logiku sustava, kao i dohvaćanje zapisa iz baze podataka te serviranje istih klijentima, zadužen je *Web Application Programming Interface* (skraćeno Web API). Tehnologija korištena za razvoj Web API-a u ovome projektu je ASP.NET Core, razvijena od strane Microsofta, koja podržava izvršavanje na bilo kojem sustavu (engl. *cross platform)* te je drastično bolje optimizirana, postižući velike brzine i performanse, od prethodne Microsoft-ove tehnologije ASP.NET [4]. Web API, izveden pomoću ASP.NET Core-a, koristi C# kao programski jezik. Koristeći grafičko sučelje Visual Studio-a možemo napraviti novi ASP.NET Core Web API kao što je prikazano na slici (Slika 2.4 Stvaranje novog Web API-a). Prilikom izgradnje samog Web API-a koristili su se *Representational State Transfer* (skraćeno REST) principi koji opisuju kako bi dva računalna sustava trebali međusobno komunicirati. REST principi su ustvari preporuke i ograničenja izgradnje *RESTful web* servisa, pod kojim Web API i spada. Unutar REST arhitekture, glavna reprezentacija podatka, ili podataka, se zove resurs (engl. *resource*)



Slika 2.4 Stvaranje novog Web API-a

2.2.1. Web API

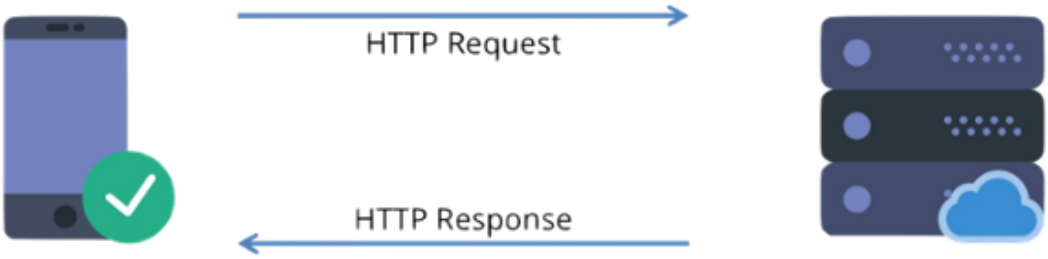
Glavna zadaća Web API-a je posluživanje podataka klijentima. Kako bih znao kome treba poslužiti podatke klijent prvo mora napraviti zahtjev (engl. *request*) prema poslužitelju, to jest Web API-u. *Hypertext Transfer Protocol* (skraćeno HTTP) je najpoznatiji protokol koji se koristi kako bih se zahtjev poslao na poslužitelj, te kako bi poslužitelj vratio odgovor (engl. *response*) klijentu. Klijent šalje zahtjev prema poslužitelj pomoću HTTP protokola, te nakon uspješno zaprimljenog zahtjeva, poslužitelj obrađuje zahtjev te nazad klijentu šalje odgovor kako je i prikazano na slici (Slika 2.5 Slanje zahtjeva i primanje odgovora). Zahtjev se šalje na određenu adresu, to jest na određeni *Uniform Resource Locator* (skraćeno URL). URL je vrsta URI-a koja počinje sa deklaracijom protokola koji bi se trebao koristiti kako bi se locirao specifični resurs na mreži (engl. network) [5]. *Uniform Resource Identifier* (skraćeno URI) je jedinstveni identifikator resursa koji identificira resurs po lokaciji, imenu ili oboje, te je korišten kako bismo pristupili specifičnom resursu [6].

URL sadrži sljedeće:

- Protokol korišten od strane klijenta i poslužitelja (HTTP u ovom slučaju)
- Ime domene ili *Internet Protocol* adresu (skraćeno IP)

- *Transmission Control Protocol* (skraćeno TCP) broj porta na kojem poslužitelj sluša nadolazeće zahtjeve klijenata
- Putanju i ime datoteke zahtijevanog resursa

Primjer validnog URL-a je http://www.algebra.hr/cjelozivotno-obrazovanje/programi-obrazovanja/.



Slika 2.5 Slanje zahtjeva i primanje odgovora

Kada god se napravi zahtjev prema određenom URL-u, na primjer kada unutar adresne kučice preglednika unesemo URL i stisnemo idi, preglednik automatski prevodi URL u zahtjev prema specificiranom protokolu te ga šalje prema poslužitelju. Sam zahtjev na server, poslan preko HTTP protokola, sadržava sljedeće informacije:

- *Request-line* koji sadrži metodu zahtjeva, putanju zahtjeva te protokol kojim se šalje
- Nula ili više zaglavlja (engl. *headers*)
- Praznu liniju koja ukazuje na kraj zaglavlja
- Opcionalno tijelo poruke (engl. *message-body*)

2.2.1.1 Metode

Web API-u pristupamo različitim metodama koje se specificiraju prilikom svakog zahtjeva prema istome. Metode korištene unutar projekta, ujedno i najčešće korištene metode, su :

- *GET* – dohvat zapisa sa poslužitelja
- *POST* – stvaranje novog zapisa na poslužitelju
- *PUT* – ažuriranje postojećeg zapisa na poslužitelju
- *DELETE* – brisanje postojećeg zapisa na poslužitelju

Gore navedene metode odgovaraju CRUD operacijama, te se koriste u tu svrhu. Na primjer *GET* zahtjev prema ruti /korisnik/2 će dohvatiti sve informacije vezane za korisnika kojem je ID (jedinstveni identifikator) jednak broju dva. *PUT* ili *DELETE*

zahtjevi na istu rutu će ažurirati ili obrisati odabranog korisnika. *POST* zahtjev na rutu /korisnik/ će dodati novog korisnika.

2.2.1.2 Rute

Rute (engl. *route)* su odgovorne za mapiranje URI zahtjeva na krajnje točke (engl. *endpoints*) te za slanje (engl. *dispatch*) nadolazećih zahtjeva prema krajnjim točkama. Rute, unutar ASP.NET Core Web API-a, možemo definirati na razini kontrolera (engl. *controller*), kako je prikazano u kodu (Kôd 2.9 Definicija kontrolera), te na razini pojedine metode, kako je prikazano na kodu (Kôd 2.10 Definicija GET metode). Definiranjem rute definiramo način na koji će klijentska aplikacija pristupati pojedinom resursu. Dobra praksa za slaganje imena ruta je korištenje imenica kao reprezentacija pojedinog resursa. Postavljanje rute na razini kontrolera omogućuje svim ostalim metodama, koje se nalaze unutar tog kontrolera, automatski prefiks iste. To znači da će metoda *GetUser* unutar kontrolera *UserController* imati rutu definiranu na vrhu kontrolera kao prefiks, a ostatak će se samo nadodati s obzirom na definiciju rute iznad same metode. Stoga ruta za pristup pojedinom korisniku će biti /api/[controller]/{userId} gdje je *controller* ime samog kontrolera, u ovom slučaju *User*, a *{userId}* jedinstveni identifikator pojedinog korisnika. Iz perspektive klijenta ruta će izgledati kao */api/user/2*, te pošto je specificirana metoda pri pristupu krajnje točke *GetUser* metoda *GET*, ova krajnja točka služi za dohvaćanje pojedinog korisnika prema njegovom jedinstvenom identifikatoru.

```
[Route("api/[controller]")]
[ApiController]
public class UserController : ControllerBase
{
      public UserController()
      {}
}
```

Kôd 2.9 Definicija kontrolera

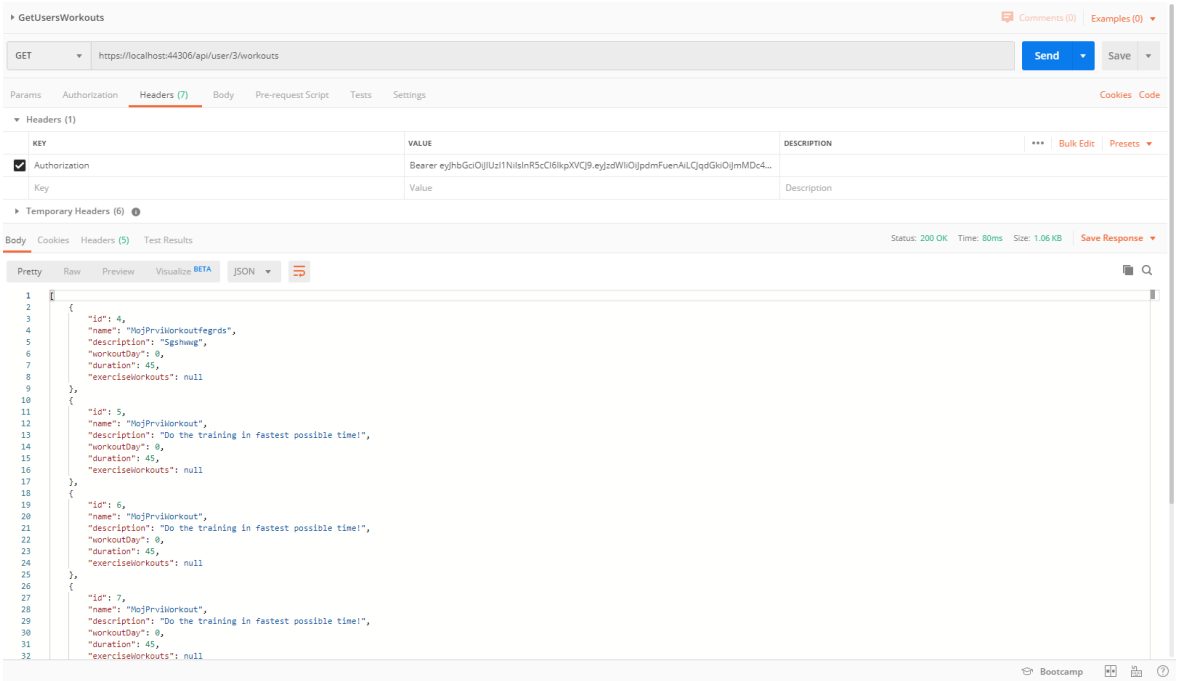
```
[HttpGet("{userId}")]
public IActionResult Get(int userId)
{
      var user = _apiDbContext.Users.FirstOrDefault(u
      => u.Id == userId);
      if (user == null)
            return NotFound("User not found!");
      return Ok(user);
```

```
}
```

Kôd 2.10 Definicija GET metode

2.2.1.3 Testiranje krajnjih točaka

Nakon izrade web API-a potrebno je testirati krajnje točke kako bismo bili sigurni da validacije na istima rade te da vraćaju željene podatke u točnom formatu. Format koji se koristio za slanje i primanja objekata unutar ovog projekta je JavaScript Object Notation (skraćeno JSON). Za testiranje funkcionalnosti Web API-a koristio se Postman. Postman je aplikacija koja omogućuje jednostavno slanje zahtjeva prema poslužitelju te primanje odgovora poslužitelja. Na slici (Slika 2.6 Postman GET zahtjev) imamo primjer slanje GET zahtjeva prema poslužitelju. Prvo se definira metoda, u ovom slučaju GET, te url na koji će se poslati GET zahtjev. Nakon toga se specificiraju potrebna zaglavlja. Jedino zaglavlje u ovom primjeru je zaglavlje *Authorization* koje u sebi ima definiranu vrijednost tokena koji je potreban za autentikaciju korisnika pri pristupanju bilo kojem resursu poslužitelja. Nakon klika na gumb pošalji, ako je zahtjev uspješno prošao te je zadovoljio sve uvjete postavljene od strane poslužitelja, onda nazad dobivamo odgovor koji je prikazan, u donjem dijelu slike (Slika 2.6 Postman GET zahtjev), unutar tijela (engl. *body*) poruke samog odgovora.



Slika 2.6 Postman GET zahtjev

2.2.1.4 Ngrok alat

Kako bismo omogućili mobilnoj aplikaciji slanje i primanje zahtjeva, prvo moramo Web API izložiti (engl. *expose*) preko javnog interneta, kako bismo mogli poslati zahtjev na isti od bilo kuda i preko bilo kojeg uređaja. Za potrebe projekta koristimo alat po imenu ngrok. Ngrok izlaže program, u ovom slučaju Web API, iza Network Address Translation (skraćeno NAT) i vatrozida (engl. *firewall*), javnom internetu preko osiguranih komunikacijskih tunela. Sve što je potrebno je skinuti program, te ga pokrenuti na računalu na kojem se pokreće i Web API te mu omogućiti port preko kojeg će komunicirati [8]. Ngrok se nakon toga spaja na ngrok oblak (engl. cloud), preusmjerava sve pozive prema toj adresi na lokalnu adresu Web API-a te nam pruža javnu adresu koju možemo koristiti za pristup svim ostalim metodama istog.

2.3. Prezentacijski sloj

Kao sloj prezentacije podataka dobivenih iz Web API-a unutar ovog projekta koristi se React Native radna okolina. React Native je tehnologija razvije od strane Facebook-a koja koristi JavaScript programski jezik za razvoj nativnih mobilnih aplikacija za Android i IOS operacijske sustave [7]. React Native koristi kombinaciju JavaScript programskog jezika i sintakse, koja izgledom najviše sliči EXtensible Markup Language-u (skraćeno XML), pod nazivom JSX. JSX omogućuje programerima pisanje komponenata (engl. components) koje sadrže metodu *render* unutar koje je logika renderiranja UI-a (engl. *User Interface*) kao i ostali programski algoritmi za funkcioniranje komponente koji su se do sada razdvajali u posebne datoteke, i time otežavali upravljanje događaja (engl. *event handling*), promjenu stanja (engl. *state*), te pripremu podataka za prikaz krajnjem korisniku.

2.3.1. Uspostava React Native radne okoline

Kako bismo započeli graditi React Native mobilnu aplikaciju prvo je potrebno uspostaviti radnu okolinu koja će nam omogućiti isto. Postoje dva načina inicijalizacije i razvoja bilo koje React Native mobilne aplikacije. Prvi je pomoću React Native sučelja naredbenog retka (engl. *Command Line Interface,* skraćeno CLI), drugi je pomoću Expo sučelja naredbenog retka. React Native CLI ima za prednost potpunu kontrolu nad projektom te se mogu dodavati nativni moduli pisani u programskom jeziku Java i C. Nedostatci istog su što se razvoj aplikacije mora raditi pomoću alata Android Studio (što ima za posljedicu set

dodatnih problema) te je vrlo kompleksna, i uzima značajan dio vremena, sama uspostava projekta. Sa druge strane Expo CLI omogućuje uspostavu novog projekta unutar minuta, dozvoljava jednostavno dijeljenje aplikacija pomoću QR coda te nije potrebno napraviti izgradnju (engl. *build*) aplikacije kako bi se aplikacija pokrenula na uređaju. Za potrebe projekta koristio se Expo CLI kako bi olakšao i ubrzao proces izrade React Native mobilne aplikacije.

2.3.1.1 Expo alati

Expo je skupina alata koja, osim što je besplatna i prati standard otvorenog koda (engl. *open source*), pomaže pri izgradnji nativnih Android i IOS aplikacija koristeći JavaScript i React programske jezike [9]. Expo se sastoji od pet, međusobno povezanih, alata:

- Expo razvojno okruženje (engl. Expo Development Environment, skraćeno XDE) – glavni alat koji se koristi za stvaranje, izgradnju te dijeljenje React Native aplikacije
- Expo sučelje naredbenog retka – sadrži punu snagu XDE samo u formatu naredbenog retka
- Expo klijent (engl. *client*) – mobilna aplikacija za Android i IOS operativne sustave. Omogućuje pokretanje projekta unutar mobilne Expo aplikacije, bez potrebe instalacije aplikacije koja se trenutno razvija. Isto tako sadrži opciju brzog osvježavanja (engl. *hot reload*).
- Expo Snack – omogućava pokretanje i izrade React Native aplikacije unutar preglednika
- Expo paket za razvijanje softvera (engl. *software development* kit, skraćeno SDK) – kolekcija JavaScript API-a koja dolazi sa svakim novim projektom napravljenim pomoću Expo-a

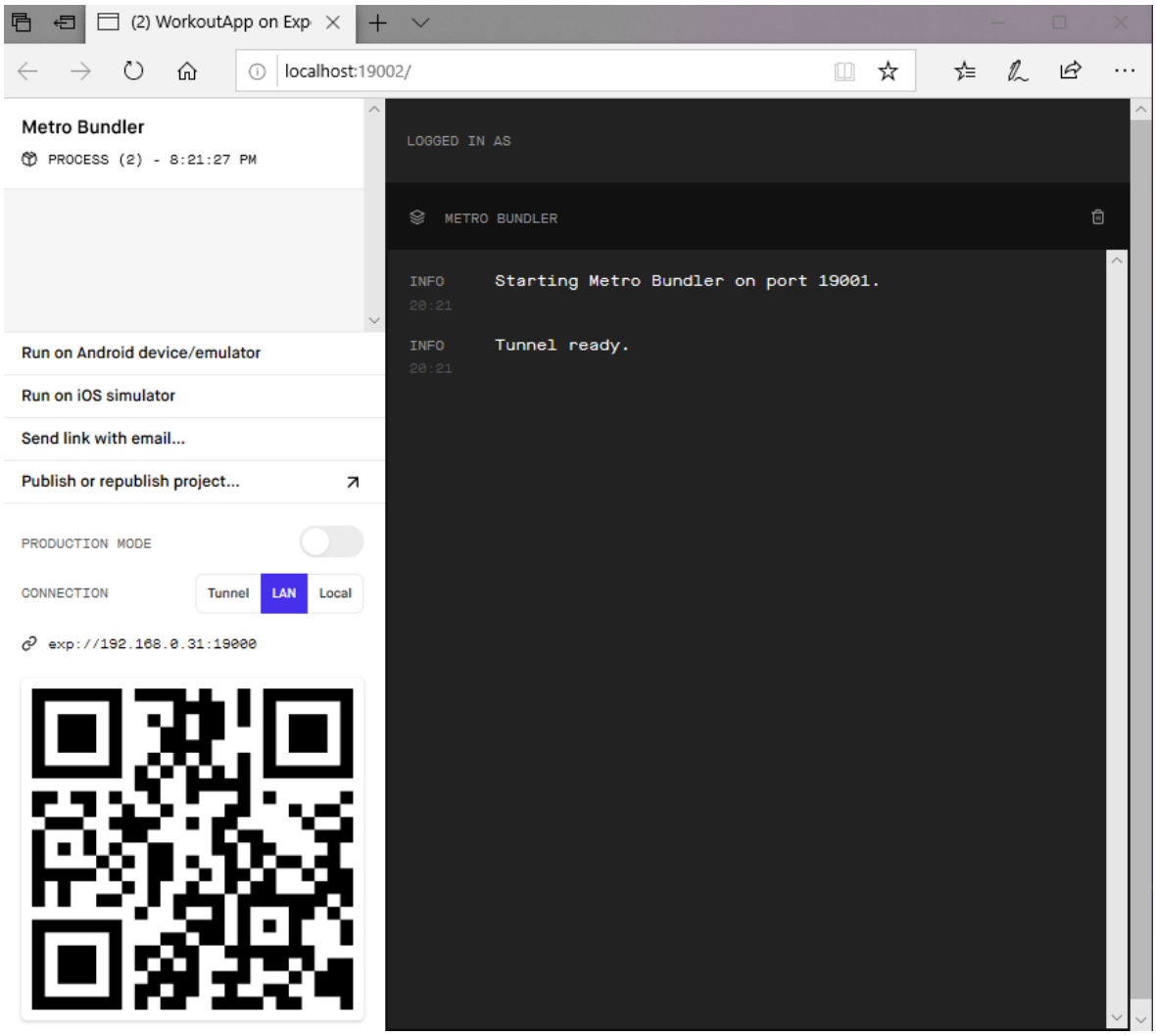
2.3.1.2 Pokretanje aplikacije

Kako bismo napravili i pokrenuli prvu Expo aplikaciju potrebno je zadovoljiti određene preduvjete. Potrebno je imati instaliran Node.js. To je JavaScript *runtime* koji omogućuje da napisani JavaScript kod se ne izvršava u pregledniku nego direktno u računalu, kao proces računala. Koristeći naredbu `npm install -g expo-cli` pokrećemo inicijalnu instalaciju Expo CLI-a. Za stvaranje i pokretanje novog React Native projekta pod nazivom MojProjekt pokrećemo naredbe kako je prikazano u kodu (Kôd 2.11 Inicijalizacija i pokretanje Expo aplikacije).

```
expo init AwesomeProject
cd MojProjekt
npm start
```

Kôd 2.11 Inicijalizacija i pokretanje Expo aplikacije

Nakon izvršenja naredbe *npm start* Expo CLI pokreće Metro Bundler, što je ustvari HTTP poslužitelj koji prevodi JavaScript kod aplikacije koristeći Babel i servira ga Expo aplikaciji. Babel je JavaScript prevoditelj koji prevodi noviju verziju JavaScript jezika na verziju koja je kompatibilna sa starijim verzijama preglednika [10]. Nakon Metro Bundlera podiže se i kontrolni panel za razvoj aplikacija unutar primarnog internet preglednika kao što je vidljivo na slici (Slika 2.7 Kontrolni panel za razvoj aplikacija unutar internet preglednika).



Slika 2.7 Kontrolni panel za razvoj aplikacija unutar internet preglednika

Sve što je nakon toga potrebno za pokretanje aplikacija na svom mobilnom uređaju je skinuti aplikaciju Expo sa Google PlayStore-a te skenirati QR kod prikazan unutar internet preglednika.

3. React Native radna okolina

3.1.1. Prednosti

Pošto se React Native bazira na web tehnologijama, bilo tko tko je u području razvoja *web* aplikacija se vrlo lako može prebaciti na razvoj mobilnih aplikacija te poboljšati svoje vještine u istima. Mobilne aplikacija razvijene u React Native radnoj okolini omogućuju izvršavanje neovisno o operacijskom sustavu. To znači da se jednom napisana aplikacija, ne mijenjajući izvorni kod (engl. *source code*) može izvršiti jednako i na Android operacijskom sustavu kao i na IOS operacijskom sustavu. Do sada se moralo, za svaki od željenih operacijskih sustava, razviti zasebnu mobilnu aplikaciju u zasebnim jezicima i editorima. Takav pristup je bio skuplji, zbog količine programera i znanja potrebnih za izgradnju aplikacija na zasebnim platformama, te sporiji. Iako React Native ne može pokriti sve mogućnosti koje nudi na primjer programski jezik Swift za operacijski sustav IOS ili programski jezik Java za operacijski sustav Android, takvi slučajevi nisu česti te taj benefit nije dovoljan razlog za zamijeniti sve ostale benefite koje React Native nudi.

3.1.2. Navigacija ekrana

Kako bismo imali mogućnost prikaza više ekrana te navigaciju između istih unutar React Native mobilne aplikacije koristimo paket React Navigation. Paket, prije korištenja, je potrebno instalirati unutar direktorija samog projekta pomoću komandne linije (engl *Command Prompt*, skraćeno CMD). Instalaciju paketa vršimo pomoću naredbe `npm install react-navigation`. Nakon instalacije na raspolaganju, unutar projekta, imamo nekoliko vrsta navigacije. Navigacije koje su se koristile u ovom projektu su:

- Switch navigator – pokazuje samo jedan ekran, ne zadržava stanja (engl. *state*), koristi se za prijelaz između autentikacijskih ekrana prijava (engl. *signin*) i registracija (engl. *signup*) na ostale ekrane aplikacije dostupne jednom kada se korisnik uspješno registrirao ili prijavio
- Bottom tab navigator – jednostavan izbornik pozicioniran pri dnu ekrana koji omogućuje prelazak između različitih ruta
- Stack navigator – omogućuje tranziciju između ekrana aplikacije gdje je svaki novi ekran stavljen na vrh stoga (engl. *stack*)

Primjer korištenja iznad navedenih navigacija možemo vidjeti unutar koda (Kôd 3.1 React Native navigacija između ekrana). Imamo dva glavna toka (engl. *flow*) aplikacije, jedan je autentikacijski tok, a glavni tok koji predstavlja ostatak prozora. Kako bismo se navigirali između istih koristi se Switch navigator kao što je vidljivo unutar koda (Kôd 3.1 React Native navigacija između ekrana). Unutar Switch navigatora je stavljen Stack navigator kako bismo mogli navigirati između ekrana unutar autentikacijskog toka. Unutar glavnog toka je inicijalno kreiran Bottom Tab navigator koji postavlja na dno ekrana izbornik pomoću kojeg se pomičemo na ostale tokove sa ekranima. Ostali tokovi unutar Bottom Tab navigatora su definirani i kreirani pomoću već gore spomenutog Stack navigatora koji opet ima za zadaću navigaciju između ekrana, stavljajući ih na vrh stoga, unutar svakog od definiranih navigacijskih tokova.

```
const switchNavigator = createSwitchNavigator({
  logInFlow: createStackNavigator({
    SignUp: SignUpScreen,
    SignIn: SignInScreen
  }),
  mainFlow: createBottomTabNavigator({
    predefinedWorkoutsFlow: createStackNavigator({
      PredefinedMain: PredefinedMainScreen,
      PredefinedDifficulty: PredefinedDifficultyScreen,
      PredefinedDays: PredefinedDaysScreen,
      PredefinedExercises: PredefinedExercisesScreen,
      PredefinedExercisesDetails:
PredefinedExercisesDetailsScreen
    }),
    customWorkoutsFlow: createStackNavigator({
      WorkoutsMain: WorkoutsMainScreen,
      AddWorkout: AddWorkoutScreen,
      WorkoutExercises: WorkoutExercisesScreen,
      EditExercise: EditExerciseScreen,
      EditWorkout: EditWorkoutScreen,
      AddExercise: AddExerciseScreen
    }),
    userAccountFlow: createStackNavigator({
      AccountMain: AccountMainScreen
    })
  })
```

```
});
```

Kôd 3.1 React Native navigacija između ekrana

3.1.3. Komponente

Komponente nam u React Native-u pružaju određene gotove funkcionalnosti te ih se može smatrati gradivnim blokovima svakog ekrana. Neke od korištenih komponenata unutar ovog projekata su:

- View - osnovna komponenta izrade korisničkog sučelja
- Text – komponenta za prikaz teksta
- TextInput – komponenta za upis teksta pomoću tipkovnice
- StyleSheet – omogućuje izradu stilova za ostale komponente
- Button – gumb koji omogućuje rukovanje s događajima (engl. events) na isti
- FlatList – komponenta za prikazivanje (engl *rendering*) liste objekata

Kako bismo koristili bilo koju od komponenata, prilikom kreiranja nove datoteke sa datotečnim nastavkom .js, koja predstavlja ekran prvo moramo uvesti (engl. *import*) biblioteku koja ju i sadržava. Biblioteke uvodimo tako da prvo definiramo komponente koje želimo uvesti te nakon toga iz koje biblioteke ih uvodimo. Primjer uvođenja različitih biblioteka možemo vidjeti u kodu (Kôd 3.2 Dodavanje novih biblioteka).

```
import { View, StyleSheet, Text } from "react-native";
import { Input, Button } from "react-native-elements";
```

Kôd 3.2 Dodavanje novih biblioteka

Unutar koda (Kôd 3.3 Primjer JSX sintakse) možemo vidjeti primjer korištenja iznad navedenih komponenata te međusobne interakcije istih. Unutar View komponente je postavljena komponenta Input za unos imena. Svaka komponenta ima svoja svojstva koja se mogu definirati prilikom izrade iste. Odmah ispod Input komponente je postavljena Button komponenta koja za spremanje novo stvorenog imena. Button i Input komponente, u ovom slučaju, pripadaju biblioteci React Native Elements koja nudi različite komponente sa već primijenjenim stilovima. Sintaksa korištena unutar koda (Kôd 3.3 Primjer JSX sintakse) je JSX te se pomoću nje, na brz i efikasan način, definiraju rasporedi i funkcionalnosti komponenata.

```
<View style={styles.container}>
      <Input
        label="Name"
```

```
        autoCorrect={false}
        value={name}
        onChangeText={newName => {
          setName(newName);
        }}
      />
      <Button
        title="Save"
        onPress={() => {
          updateUserWorkout(workout.id, name, description,
duration);
        }}
      />
</View>
```

Kôd 3.3 Primjer JSX sintakse

3.1.4. Stilovi

Stilovi nam omogućuju vizualno uređivanje komponenata i njihovog sadržaja. Za uređivanje komponenata se koriste Cascading Style Sheets (skraćeno CSS). Kako bismo primijenili određeni stil na određenu komponentu, prvo što je potrebno je na toj komponenti dodati oznaku *style*. Nakon toga se ili unutar te oznake može postaviti vrijednost nekog svojstva (unutarnji CSS, engl. *inline*) ili se može dati referenca na lokaciju definiranog stila (vanjski CSS, engl. *external*). Primjer unutarnjeg CSS-a je prikazan u kodu (Kôd 3.4 Primjer unutarnjeg CSS-a).

```
<Text style={{ fontSize: 20 }}>Hello World!</Text>
```

Kôd 3.4 Primjer unutarnjeg CSS-a

Vanjski CSS unutar React Native-a definiramo unutar svake datoteke pomoću funkcije `StyleSheet.create`. Unutar iste možemo navoditi stilove za različite komponente te će se te iste komponente na pojedine stilove referencirati koristeći ime stila kako je i prikazano unutar koda (Kôd 3.5 Primjer vanjskog CSS-a).

```
<Text style={styles.style1}>Hello World!</Text>

const styles = StyleSheet.create({
  style1: {
      fontSize: 20,
      color: "blue",
```

```
        fontWeight: "bold"
    }
});
```

Kôd 3.5 Primjer vanjskog CSS-a

3.1.5. Komunikacija s Web API-em

Kako bi klijent, React Native mobilna aplikacija, komunicirao sa Web API-em potrebno je implementirati mogućnost slanja zahtjeva te primanja odgovora na klijentskoj strani. Za to se unutar projekta koristi Axios. Axios je HTTP klijent koji omogućuje slanje zahtjeva prema poslužitelju te primanje odgovora samog servera. Kako bismo koristili Axios prvo ga je potrebno instalirati, koristeći CMD unutar projektnog direktorija, naredbom `npm install axios`. Nakon toga unutar datoteke koja koristi Axios je potrebno uvesti Axios te napraviti instancu istog, prikazano u kodu (Kôd 3.6 Kreiranje Axios instance), kako bi se funkcije koje pruža mogle koristiti. Varijabla *baseURL* predstavlja bazu URL puta svakog od zahtjeva, to jest dio URL-a koji će svaki zahtjev koristiti, pa umjesto ponavljanja baznog URL-a na svakom mjestu gdje se šalje zahtjev prema poslužitelju možemo ga definirati tokom stvaranja instance Axios-a te kasnije stavljati samo dio URL-a koji je promjenjiv sa svakim zahtjevom.

```
import axios from "axios";

const instance = axios.create({
  baseURL: "http://503a7809.ngrok.io"
});
```

Kôd 3.6 Kreiranje Axios instance

Kako bismo napravili zahtjev prema poslužitelju potrebno je na instanci Axios-a, u ovom slučaju pod imenom *workoutApi*, pozvati željenu metodu (GET, POST, PUT, DELETE). Unutar koda (Kôd 3.7 Slanje GET zahtjeva pomoću Axios-a) je prikazano slanje GET zahtjeva prema lokaciji definiranoj unutar zagrada get metode. Metoda *then* se izvršava nakon zaprimljenog odgovora poslužitelja te unutar iste možemo specificirat koju akciju želimo napraviti ili funkciju pozvati pri uspješno vraćenom odgovoru sa poslužitelja. Na kraju metoda *catch* služi hvatanju bilo kakvih grešaka koje se mogu dogoditi u iznad opisanom procesu.

```
workoutApi
    .get(`api/user/${userId}/workouts`)
```

```
    .then(function(response) {
      console.log(response.data);
      dispatch({ type: "get_user_workouts", payload:
response.data });
    })
    .catch(function(error) {
      console.log(error);
    });
```

Kôd 3.7 Slanje GET zahtjeva pomoću Axios-a

Po istom principu se grade i sve ostale vrste poziva prema Web API-u te po potrebi pojedine metode se dodaju dodatni parametri unutar tijelo poslane poruke kao što se može vidjeti u kodu (Kôd 3.8 Slanje POST zahtjeva pomoću Axios-a) za primjer POST zahtjeva. U primjeru POST zahtjeva imamo dodatne parametre definirane koji se šalju na poslužitelj. Ime parametara mora odgovarati imenima na poslužitelju kako bih prošli potrebnu validaciju. U ovom slučaju na server šaljemo informacije pod svojstvima *name, description* i *duration* te po istom principu kao i za GET zahtjev čekamo odgovor poslužitelja unutar *then* funkcije te hvatamo eventualne pogreške unutar *catch* bloka.

```
workoutApi
    .post(`api/user/${userId}/workouts`, {
      name: workoutName,
      description: workoutDescription,
      duration: workoutDuration
    })
    .then(function(response) {
      dispatch({
        type: "add_user_workout",
        payload: response.data
      });
    })
    .catch(function(error) {
      console.log(error);
    });
```

Kôd 3.8 Slanje POST zahtjeva pomoću Axios-a

4. Autorizacija i sigurnost

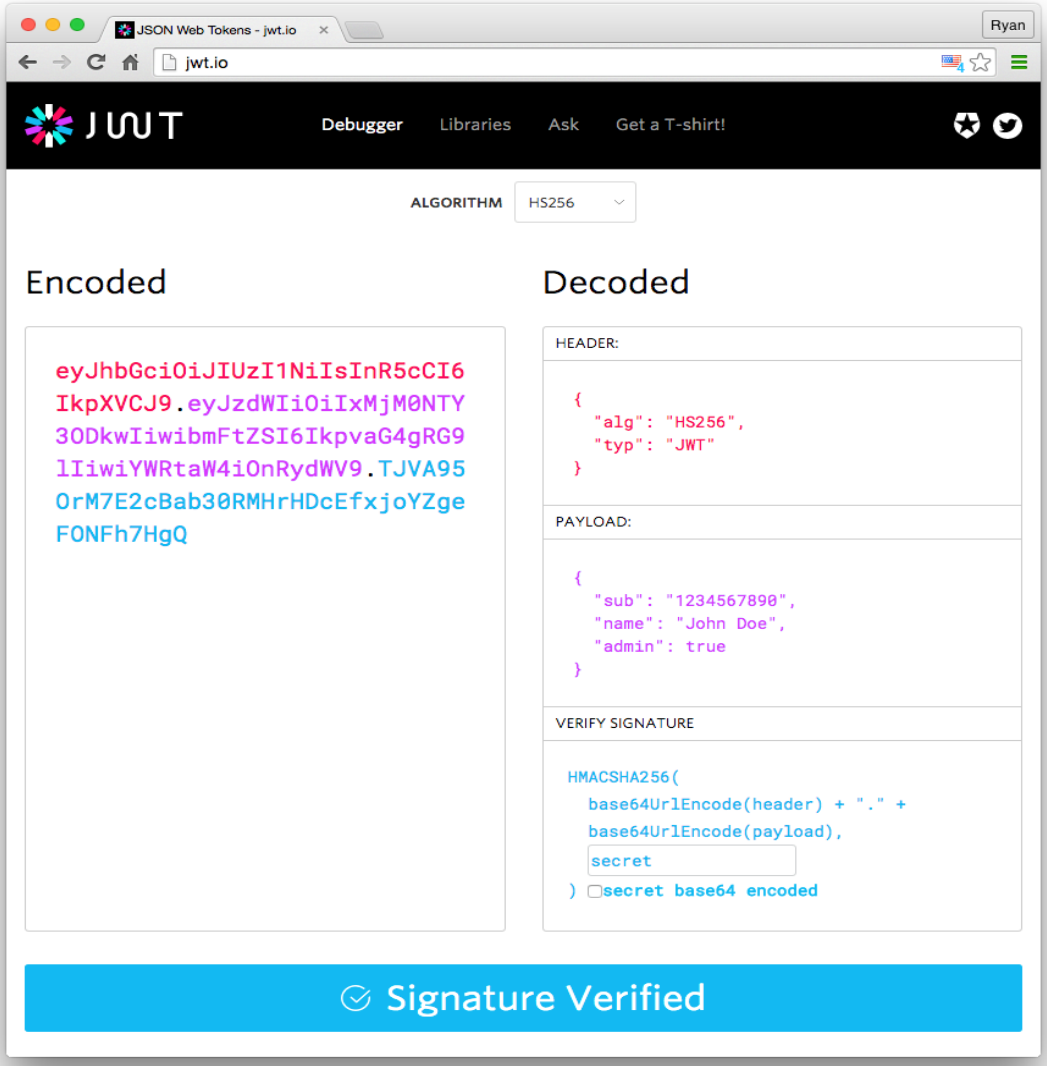
4.1. JWT autorizacija

JWT stoji za Json Web Token te je otvoreni standard (engl *open standard*) koji definira kompaktni način za sigurno slanje informacija između dva ili više sustava u obliku JSON objekata [11]. Pohranjena informacija unutar JWT može biti verificirana i može joj se vjerovat zato jer je digitalno potpisana tajnim ključem izdavača tokena. JWT mogu biti i kriptirani kako bih se sakrile informacije, no za potrebe projekta se koriste potpisani tokeni u svrhu autorizacije. Ideja je da pri registraciji ili prijavi korisnika, nakon uspješne prijave u sustav, Web API izda token te ga pošalje u svom odgovoru klijentu. Klijent nakon uspješnog primanja tokena isti sprema lokalno te ga koristi pri svakom sljedećem zahtjevu klijenta kao potvrdu identiteta kako je i prikazano na slici ().

JWT se sastoji od tri dijela odvojenih točkom:

- Header – zaglavlje se obično sastoji od dva dijela, tipa tokena (JWT) te algoritma sa kojim ga se potpisuje (HMAC, SHA256, RSA)
- Payload – sadrži izjave o entitetu (obično korisnik) te dodatne podatke
- Signature – potpis je kombinacija kodiranog header-a i payload-a sa Base64 formatom te tajnog ključa, sve to zajedno kodirano algoritmom specificiranim unutar zaglavlja

Kao što je vidljivo na slici (Slika 4.1 Dekodiranje JWT-a) vidimo da JWT token, prikazan na lijevoj strani u kodiranom formatu, ima 3 dijela odvojenih s točkom i obojanih različitim bojama zbog lakše razlikovanosti. Prvi dio predstavlja *header*, drugi dio *payload* te treći dio sam *signature*. S desne strane vidimo informacije koje JWT token sadrži jednom kada je dekodiran. Bitno je naglasiti da se token može dekodirati bez tajnog ključa kako bih se pročitali spremljeni sadržaji unutar istog, te ga unutar projekta koristimo samo kao autorizaciju pojedinog korisnika. Informacije unutar *payload* dijela JWT tokena možemo sami postaviti te na taj način prenijeti do klijenta potrebne podatke na siguran i kompaktan način. JWT token se generira samo jednom tokom prijave ili registracije za svakog od korisnika te se tokom stvaranja tokena određuje rok isteka. Proces stvaranja pojedinog tokena je opisan unutar koda (Kôd 4.1 Generiranje novog JWT-a).



Slika 4.1 Dekodiranje JWT-a

U *payload* dio stavljamo korisničko ime te jedinstveni identifikator korisnika kako bismo mogli u klijentskoj aplikaciji na jednostavan i brz način otkriti o kojem se korisniku točno radi prilikom slanja zahtjeva na web API. Nakon toga token se potpisuje tajnim ključem duljina od 32 karaktera, gdje je najbolja praksa generirati nasumičnu kombinaciju slova duljine 32 karaktera te ju spremiti i koristiti kao tajni ključ. Nakon definiranja tajnog ključa potreban je kompletan JWT token potpisati sa sigurnosnim algoritmom, u ovom slučaju *hashing* algoritmom HMAC-SHA256. Na kraju generiramo JWT token sa svim svojim svojstvima te ga šaljemo, ako je korisnik unio točne podatke pri registraciji ili prijavi, korisniku. Svaki put kada se korisnik odjavi iz aplikacije, ili istekne vrijeme trajanja tokena, JWT token spremljen na korisnikovom uređaju se briše te tokom nove prijave se generira novi za istog korisnika i ponovno sprema na uređaj korisnika. Ako korisnik ima na uređaju

spremljen JWT token aplikacija ga neće tražiti nego će ga automatizmom prepoznati kao prijavljenog korisnika, no ako token ne postoji na uređaju korisnik mora ili proći proces prijave ako se prethodno registrirao ili registracije ako nije.

```
public string GenerateAccessToken(int userId, string
username)
{
      var claims = new[]
      {
            new Claim(JwtRegisteredClaimNames.Sub, username),
            new Claim(JwtRegisteredClaimNames.Jti,
Guid.NewGuid().ToString()),
            new Claim(ClaimTypes.NameIdentifier,
userId.ToString()),
            new Claim(ClaimTypes.Name, username),
      };

      var key = new
SymmetricSecurityKey(Encoding.UTF8.GetBytes(_jwtSettings.Secr
etKey));
      var creds = new SigningCredentials(key,
SecurityAlgorithms.HmacSha256);

      var token = new JwtSecurityToken(
            _jwtSettings.AuthorityURL,
            _jwtSettings.AuthorityURL,
            claims,
            expires:
DateTime.Now.AddDays(_jwtSettings.ExpireTime),
            signingCredentials: creds);

      return new JwtSecurityTokenHandler().WriteToken(token);
}
```

Kôd 4.1 Generiranje novog JWT-a

Unutar React Native mobilne aplikacije spremamo dobiveni JWT token u memoriju uređaja naredbom `AsyncStorage.setItem("token", response.data.token)`. Iz odgovora poslužitelja izvlačimo podatak pod ključem token te ga spremamo pod istim ključem. Ako korisnik koji nije prijavljen pokuša pristupiti podatcima koje vraća upit na određeni URL, a u zaglavlje označeno oznakom *Authorization* token ne postoji, to znači da

korisnik nema prava pristupiti istome. Unutar *web* API-a iznad metode koje želimo zaštiti stavljamo atribut `[Authorize]`. Nakon uspješne registracije ili prijave korisnika u mobilnu aplikaciju poslužitelj mu automatski vraća novi JWT. Taj JWT Axios svaki puta prilikom zahtjeva zaštićenom resursu stavlja u zaglavlje pod oznaku *Authorization*. Na taj način se osiguravamo da samo prijavljeni korisnici mogu pristupiti zaštićenim metodama, te da ako netko slučajno pogodi URL ne može bez samog tokena dobiti nikakvu informaciju nazad od web API-a. Jedine dvije metode, unutar web API-a, koje nemaju atribut `[Authorize]` su metoda *Login* i metoda *Register*. Te dvije metode moraju biti dostupne svim korisnicima kako bih se bilo tko mogao pokušati ili registrirati ili prijaviti.

4.2. *Salt* i *hashing zaporki*

Hashiranje je proces pretvaranja teksta koji je čitljiv (engl. *plain text*) u probavljenu (engl. *digested*) verziju istoga. *Hash* funkcije za svaku kombinaciju unutar nekog teksta vraćaju jedinstvenu vrijednost. Promjenom i samo jednog znaka unutar postojane kombinacije mijenja se izgled kompletnog *hasha* koji funkcija vraća. Kako procesori dobivaju na snazi i brzini, *hash* funkcije generalne namjene poput MD5 i SHA1, koje su napravljene da u što bržem vremenu konvertiraju čitljivu lozinku u probavljenu verziju iste, su sve manje sigurne. Moderni poslužitelji već mogu izračunati MD5 hash brzinom od 330MB svake sekunde, što znači da ako korisnik ima lozinku koja je bez velikih slova, alfanumerička te duljine šest znakova, poslužitelj može isprobati sve kombinacije te lozinke unutar samo 40 sekundi [12]. Kako bismo se osigurali od krađe zaporki koristimo Bcrypt *hash* algoritam. Prilikom registracije korisnik specificira svoju lozinku. Umjesto spremanja čitljive verzije te lozinke unutar baze podataka, mobilna aplikacija prvo *hashira* lozinku te ju nakon toga šalje poslužitelju. Poslužitelj nakon toga sprema *hash* vrijednost korisničke lozinke. Unutar ovog projekta je dodan dodatni korak sigurnosti koji se zove *salting.* Problem kod spremanja *hashirane* verzije lozinke je ako napadač dobije pristup bazi podataka može metodom napada rječnikom (engl. *dictionary attack*) otkriti loše definirane, to jest jednostavne šifre. Napad rječnikom koristi datoteku koja sadrži riječi, fraze, te česte lozinke. Unutar datoteke te iste kombinacije su *hashirane* te se jednostavno uspoređuju sa originalnom *hashiranom* lozinkom korisnika. Ako su *hash* vrijednosti iz rječnika i iz baze podataka *hashiranih* lozinka jednake, napadač iz rječnika izvlači čitljivu (engl. *plain text*) lozinku korisnika. U slučaju da više korisnika koristi istu lozinku, *hash* tih lozinka će izgledati identično u bazi podataka. Pošto *hash* funkcija uvijek vraća isti *hash* za dvije identične kombinacije teksta,

napadači otkrivajući *hash* jedne lozinke mogu jednostavno usporediti isti sa ostalim hashevima lozinka unutar baze podataka, te ako su bilo koja dva identična, lozinka tog istog korisnika je identična prethodnome. Kako bismo izbjegli takve napade koristimo metodu *saltinga*. Umjesto spremanja čiste *hash* verzije lozinka unutar baze podataka, poslužitelj nakon što zaprimi *hash* verziju lozinke na nju nadoda nasumični tekst fiksne duljine zvan *salt*. Dodavanjem istog, *hash* funkcija generira potpuno drugačiji *hash*. Nakon toga se taj *hash* sprema u bazu zajedno sa čitljivom verzijom *salta*. Sada ako dvoje ili više korisnika ima istu lozinku, samim time i isti *hash* lozinke, zbog nasumično generiranog teksta koji je dodan na kraju lozinke zvan *salt*, hash ispada jedinstven. Time smo zaštitili lozinke korisnika čak i u slučaju curenja podataka. *Salting* proces ne zaustavlja napade na silu (engl. *brute force attack*). Zato koristimo Bcryt funkciju *hashiranja* lozinki jer je adaptivna funkcija, te se kroz određeno vrijeme broj iteracija može povećati kako bi algoritam bio otporan na sve veću snagu i brzinu procesora koji koriste napad na silu.

Zaključak

Code First pristup se ispostavio kao idealan način za razvijanje i ažuriranje aplikacije, a samim time i baze podataka. Jednostavno održavanje i lagane promjene kao i vraćanje na iste je omogućeno na vrlo koncizan način kroz migracije koje dolaze implementirane unutar EF Core-a. Za razvoj baze nije bilo potrebno znanje pisanja upita pomoću Transact SQL-a, već jednostavnim promjenama domenskih razreda se mogu manipulirati svojstva i izgledi tablica unutar same baze podataka. EF Core je optimiziran za maksimalnu brzinu izvršavanja upita na bazu te vraćanja podataka korisniku što ga čini idealnim kandidatom za sustav ove vrste i veličine.

ASP.NET Core *web* API ima odličnu podršku i predloške za kreiranje novog *web* API-a te koristeći C# kao bekend vrlo brzo se razvija logika za primanje i vraćanje podataka. Isto tako, zbog odabrane tehnologije i mogućnosti koje pruža ASP.NET Core, *web* API je optimiziran za brzinu izvršavanja zahtjeva te se može izvršavati na bilo kojoj platformi što mu daje odličnu sveukupnu iskoristivost u svijetu gdje se sve manje oslanja na ovisnost o operacijskim sustavima.

React Native, kao tehnologija odabrana za razvoj mobilne aplikacije, dobiva sve veću popularnost u mobilnom softverskom svijetu. Koristi nativne elemente te omogućuje istovremeni razvoj mobilne aplikacije i za Android i IOS operacijski sustav. Zbog JSX sintakse razvoj UI mobilne aplikacije je brz i jednostavan ukoliko programer ima prethodno iskustvo sa jezicima JavaScript, HTML i CSS. Koristeći Expo pojednostavljuje se proces inicijalne inicijalizacije i instalacije aplikacije, te se unutar par minuta dobije gotov projekt koji je spreman za razvoj.

Na kraju autentikacija bazirana na tokenima se ispostavila relativno jednostavna za implementaciju, s obzirom da ASP.NET Core *web* API i React Native imaju odličnu podršku za JWT, čime smo uspjeli zaštiti web API od anonimnih korisnika.

Popis kratica

ORM	*Object-Relational Mapper*	Objektno relacijski maper
EF	*Entity Framework*	Entity Framework
CRUD	Create, Read, Update, Delete	Izrada, Čitanje, Izmjena, Brisanje
DDL	*Data Definition Language*	Jezik definicije podataka
API	*Application Programming Interface*	Aplikacijsko programsko sučelje
REST	*Representational State Transfer*	Prijenos prikazanog stanja
HTTP	*Hypertext Transfer Protocol*	Hypertext prijenosni protokol
URL	*Uniform Resource Locator*	Jedinstveni resursni lokator
URI	*Uniform Resource Identifier*	Jedinstveni resursni identifikator
IP	*Internet Protocol*	Internet protokol
TCP	*Transmission Control Protocol*	Protokol kontrole prijenosa
JSON	*JavaScript Object Notation*	JavaScript objektna notacija
NAT	*Network Address Translation*	Prevoditelj mrežne adrese
XML	*EXtensible Markup Language*	Jezik za označavanje podataka
UI	*User Interface*	Korisničko sučelje
CLI	*Command Line Interface*	Sučelje naredbenog retka
XDE	*Expo Development Environment*	Expo razvojno okruženje
CMD	*Command Prompt*	Naredbeni redak
CSS	*Cascading Style Sheets*	Objektno relacijski maper
JWT	*Json Web Token*	JSON mrežni tokeni

Popis slika

Popis kôdova

Literatura

[1] WIKIPEDIA, Microsoft SQL Server, https://en.wikipedia.org/wiki/Microsoft_SQL_Server, veljača, 2020

[2] MICROSOFT, Entity Framework Core, https://docs.microsoft.com/en-us/ef/core/, veljača, 2020

[3] CHADGOLDEN, Entity Framework 6 vs Entity Framework Core 3 https://www.chadgolden.com/blog/comparing-performance-of-ef6-to-ef-core-3, siječanj, 2020

[4] MICROSOFT, Create web APIs with ASP.NET Core, https://docs.microsoft.com/en-us/aspnet/core/web-api/?view=aspnetcore-3.0, veljača, 2020.

[5] WIKIPEDIA, URL, https://en.wikipedia.org/wiki/URL, prosinac, 2019

[6] WIKIPEDIA, URL, https://en.wikipedia.org/wiki/Uniform_Resource_Identifier, prosinac, siječanj, 2020

[7] REACT NATIVE, React Native documentation, https://facebook.github.io/react-native/docs/getting-started, 2020.

[8] NGROK, Ngrok documentation, https://ngrok.com/docs, veljača, 2020

[9] EXPO, Expo documentation https://docs.expo.io/versions/latest/, veljača, 2020

[10] BABEL, What is Babel?, https://babeljs.io/docs/en/, veljača, 2020.

[11] JWT.IO, Introduction to JSON Web Tokens, https://jwt.io/introduction/, veljača, 2020

[12] CODEHALE, How to safely store a password, https://codahale.com/how-to-safely-store-a-password/, siječanj, 2010