Korištenje aplikacijskog okvira Entity Framework Core pri upravljanju bazom podataka

Kralj, Sebastian

SVEUČILIŠTE U ZAGREBU

PRIRODOSLOVNO–MATEMATIČKI FAKULTET

MATEMATIČKI ODSJEK

Sebastian Kralj

KORIŠTENJE APLIKACIJSKOG OKVIRA ENTITY FRAMEWORK CORE PRI UPRAVLJANJU BAZOM PODATAKA

Diplomski rad

Voditelj rada:
prof. dr. sc. Robert Manger

Zagreb, veljača, 2021.

Ovaj diplomski rad obranjen je dana _____ pred ispitnim povjerenstvom u sastavu:

1. _____, predsjednik
2. _____, član
3. _____, član

Povjerenstvo je rad ocijenilo ocjenom _____.

Potpisi članova povjerenstva:

1. _____
2. _____
3. _____

Sadržaj

Uvod

Cilj ovog rada je opisati i prikazati korištenje aplikacijskog okvira *Entity Framework Core* kod kreiranja i upravljanja bazom podataka. Osim teorijskog pregleda funkcionalnosti samog alata Entity Framework Core, u radu će se prikazati kreiranje aplikacije koristeći navedeni alat. Aplikacija koju ćemo izraditi biti će web kuharica, tj. služit će nam za pregled recepata i dodavanje novih u našu bazu recepata. Kod izrade aplikacije koristit ćemo *code first* pristup, tj. pristup kod kojeg prvo u kodu definiramo sve modele i veze između njih, a zatim koristeći neredbe iz Entity Framework Core-a po definiranim modelima kreiramo tablice u bazi.

Rad će biti podjeljen u četiri poglavlja. U prvom poglavlju proći ćemo kroz alate i tehnologije koje ćemo koristiti pri izradi aplikacije. Za svaki alat ćemo opisati ćemu služi i koje će nam funkcionalnosti biti potrebne pri izradi aplikacije. U drugom poglavlju detaljno ćemo obraditi sam alat Entity Framework Core. Proći ćemo kroz sve mogućnosti koje nam alat nudi: od kreiranja i konfiguracija tablica do metoda koje služe za upravljanje podacima u tim tablicama na bazi. Za svaku funkcionalnost navest ćemo konkretan kod iz naše aplikacije te objasniti ulogu napisanog koda u našoj aplikaciji. U trećem poglavlju simulirat ćemo rad naše aplikacije. Koristiti ćemo metode koje smo napisali u drugom poglavlju kako bismo novokreirane tablice popunili podacima. Nakon što ih napunimo podacima, provjerit ćemo na bazi da li su se sve tablice uspješno ispunile i da li su se sva svojstva definirana u kodu također preslikala na bazu. U četvrtom poglavlju spomenut ćemo koje smo prednosti uočili prilikom korištenja alata, te potencijalne mane alata. Također ćemo navesti i dva smjera u kojima bi se ovaj rad mogao nastaviti.

Poglavlje 1

Korišteni alati i tehnologije

U ovom poglavlju navest ćemo i ukratko predstaviti alate i tehnologije koje ćemo koristiti prilikom izrade aplikacije. U ovom poglavlju nećemo spomenuti alat *Entity Framework Core* pošto ćemo njega detaljnije promotriti te je zbog toga on izdvojen u zasebno poglavlje.

1.1 Microsoft Visual Studio

Microsoft Visual Studio je integrirano razvojno okruženje razvijeno od strane tvrtke *Microsoft*. Izraz integrirano razvojno okruženje znači da se radi o aplikaciji koja razvojnim inžinjerima pruža okolinu potrebnu za izradu softvera. Visual Studio se koristi za razvoj računalnih programa, web-stranica, web-servisa i mobilnih aplikacija. Od alata koji su korisni razvojnom inžinjeru za razvoj softvera valja izdvojiti *debugger* koji otkriva greške na nivou izvornog i strojnog koda te sugerira promjene u kodu koje poboljšavaju rad samog softvera. Osim debuggera, treba spomenuti i uređivač koda IntelliSense koji korisniku predlaže ostatak koda na temelju do tada napisanog koda.

Postoje tri verzije Visual Studia: Community, Professional i Enterprise. Community je namjenjen korištenju u privatne svrhe, najčešće za učenje jezika i otkrivanja novih funkcionalnosti. Verzija je besplatna i dostupna svima preko Microsoftove web-stranice. Verzije Professional i Enterprise su namijenjene za izradu profesionalnih softvera. Professional verzija se najčešće koristi u manjim firmama, dok je Enterprise namjenjena većim korporacijama. U svom radu koristiti ću Visual Studio Community 2019, verziju 16.8.2.

Visual Studio podržava razne programske jezike. U svom radu ću koristiti programski jezik `C#` i aplikacijski okvir .NET framework. Oboje je ugrađeno u osnovnu verziju programa i nije potrebno instalirati nikakve dodatne servise. Od dodatnih servisa, koji nisu

uključeni u osnovnu verziju programa, koristiti ću paket *nuget*. Pomoću njega možemo lakše instalirati web-pakete. Ti web-paketi su dijelovi koda koji su napisali i objavili neki drugi inženjeri i mi ih koristimo u svom projektu. Na nama je da te pakete uključimo u naš kod i onda pozovemo određenu metodu iz tog paketa koja odradi određeni posao za nas.

1.2 .NET Core i C#

.NET Core je besplatno, *open-source* aplikacijsko sučelje za Windows, Linux i macOS operacijske sustave. Kreator sučelja je trvtka Microsoft. Prva verzija sučelja nastala je 2016. godine. Ta verzija zvala se .NET Core 1.0. Nakon nje izdano je još 7 verzija. Najnovija je nazvana .NET 5 i izdana je u studenom 2020. godine. U ovom radu koristit ćemo verziju .NET Core 3.1, jer je u vrijeme kretanja izrade rada ta verzija bila najnovija.

Jedna od osnovnih zadaća .NET Core-a je kompajliranje koda. To je proces kod kojeg se izvorni kod pretvori u **CIL** (Common Intermediate Language). CIL je skup instrukcija koje se spremaju u asemblije. Ti asembliji služe kod pokretanja aplikacija. Tada se ti asembliji u **CLR**-u (Common Language Runtime-u, koji možemo shvatiti kao virtualni stroj) pomoću **JIT**-a (just-in-time kompajlera) pretvara u strojni kod kojeg zatim stroj može pročitat i pretvorit u aplikaciju.

.NET Core u potpunosti podržava C# jezik. C# je jezik razvijen 2000. godine u Microsoftu. Radi se o objektno-orijentiranom programskom jeziku. Jedna od karakteristika jezika je tzv. garbage collecting, tj. sam se brine o bespotrebnom curenju memorije. Također je type-safe, što znači da se kompajler sam brine o tipovima unutar klasa. Trenutno je do sada izdano 11 verzija jezika C#. Koristit ćemo verziju C# 8.0, izdanu 2019. godine.

1.3 REST API

REST (Representational state transfer) je stil softverske arhitekture koja definira skup pravila koji se koriste kod izrade web-servisa. Servisi koji se pridržavaju principa iz REST-a nazivaju se **RESTful** web-servisi.

REST se prvi puta spominje 2000. godine. Nastao je sa svrhom da poboljša kominikaciju između komponenata u složenim sustavima. Posljedica toga bila je da su sustavi nastali u skaldu s REST-om imali bolje performanse, bili skalabilni i lakši za održavati. U današnje vrijeme REST je načešći arhitektonski oblik prilikom kreiranja web-aplikacije. Zbog toga ga često nazivaju jezikom interneta. U ovom radu pridržavat ćemo se pravila REST-a da bismo izradili **RESTful API**.

RESTful API je stil arhitekture za aplikacijsko programsko sučelje (API) koje koristi HTTP zahtjeve za pristup i korištenje podataka. Svaki taj zahtjev sadržavat će jednu od ključnih riječi: **GET**, **POST**, **PUT** ili **DELETE**. GET ćemo koristiti prilikom dohvata podataka, POST prilikom kreiranja novog podatka, PUT kod ažuriranja postojećeg podatka i DELETE prilikom brisanja podataka.

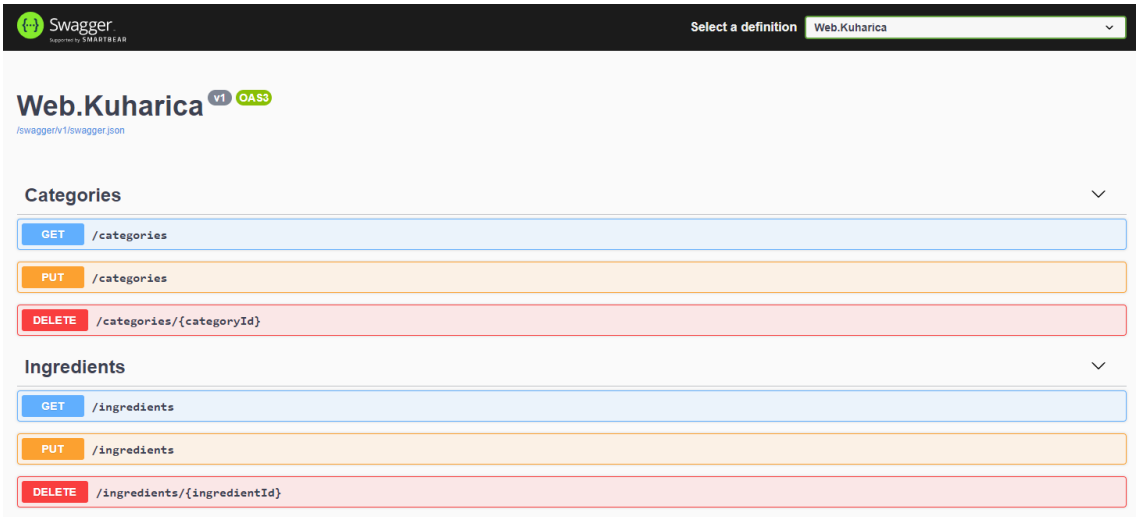
1.4 Model-view-controller

Model-view-controller(MVC) je oblik softverske arhitekture kod kojeg se nastoji odvojiti pojedine dijelove aplikacije u zasebne komponente. Dijelovi su sortirani u komponente prema zajedničkim namjenama. Naziv MVC dolazi od tri osnovna sloja koja se pojavljuju u toj arhitekturi. U model sloju nalaze se podaci i modeli koji će se koristiti u aplikaciji, te sama logika aplikacije. U view sloju nalaze se stvari potrebne za prikaz podataka. U tom sloju definira se izgleda aplikacije koji se zatim prikazuje krajnjem korisniku. Kontrolerski sloj upravlja zahtjevima koji se dobiju od korisnika aplikacije. On svaki zahtjev proslijedi određenoj metodi unutar modelskog sloja koja dalje provodi logiku.

Softver koji će nastati u sklopu ovog rada također će implementirati principe MVC arhitekture. Kada korisnik šalje zahtjev na naš API, taj zahtjev prvo dolazi do odgovarajućeg kontrolera. Kontroler, nakon što dobije zahtjev, poziva odgovarajući servis u aplikacijskom sloju koji ima ulogu provesti logiku aplikacije. Svaki servis uzima modele iz posebnog sloja koji osim aplikacijskih modela sadržava i modele na temelju kojih će se konstruirati baza podataka. Nakon što se dobiveni podaci u servisu preslikaju u modele, ti modeli se šalju u infrastrukturni dio aplikacije. U tom infrastrukturnom dijelu dolazi do slanja podataka prema bazi podataka i očekivanju odgovora od iste. Nakon što se dobije odgovor od baze podataka, taj odgovor se vraća natrag u servis. U servisu dolazi do preslikavanja podataka iz odgovora u modele koji se zatim vraćaju korisniku. Kako je u ovom radu naglasak na backend dio aplikacije, nećemo raditi korisnički dio aplikacije, nego ćemo koristiti alat Swagger UI koji će predstavljati prezentacijski sloj moje aplikacije.

1.5 Swagger

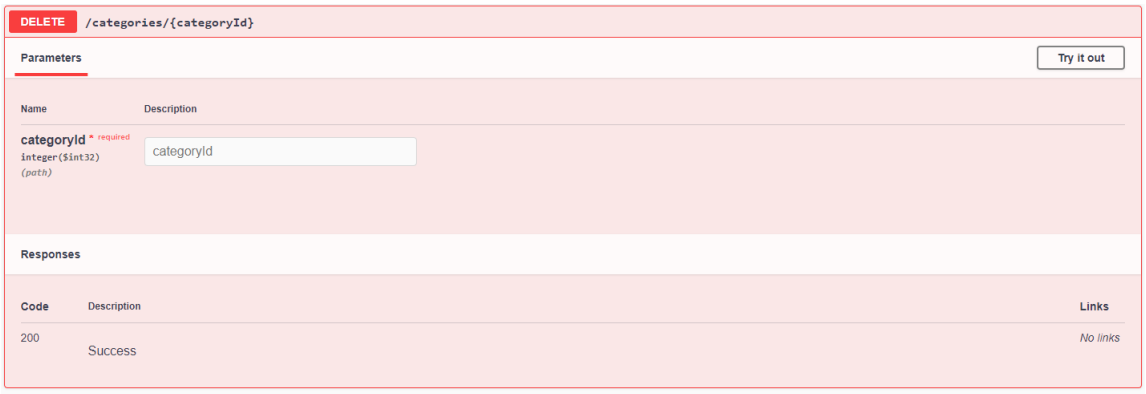
Swagger je jezik koji se koristi za opisivanje sučelja RESTful API-ja. Kod nas će biti konfiguriran tako da će prikazati sve metode na našim kontrolerima. Koristit ćemo Swagger UI koji će nam izgenerirati korisničko sučelje na temelju podataka koji se dohvate prilikom pokretanja aplikacije, a tiču se naših metoda na kontroleru.



Slika 1.1: Izgled Swagger UI-a na našoj aplikaciji

Na slici 1.1 vidimo dio korisničkog sučelja koje je Swagger UI izgenerirao za našu aplikaciju. U prikazu možemo vidjeti koje sve metode na kontrolerima postoje u našoj aplikaciji, koju ključnu riječ koristimo za dohvaćanje određene metode te na kojoj se ruti svaka metoda nalazi. Uzmimo za primjer kontroler *Categories*, na kojem se nalaze metode vezane za upravljanje kategorijama unutar naše aplikacije. Vidimo da je prva metoda na ruti */categories* i sadrži ključnu riječ **get**. Ona služi da bismo dobili popis svih kategorija iz baze. Također na tom kontroleru imamo metodu na ruti */categories* s ključnom riječi **put** i metodu na ruti */categories/{categoryId}*s ključnom riječi **delete**. U posljednjoj metodi categoryId je naveden u vitičastim zagradama zbog toga što mi tu metodu zovemo s nekim konkretnim id-em kojeg želimo izbrisati. Npr. ako bismo željeli izbrisati kategoriju s id-em 2, ruta bi nam glasila *categories/2*.

Osim pregleda svake metode, Swagger UI nam omogućuje da pritiskom tipke miša na metodu dobijemo detaljan prikaz o toj metodi.

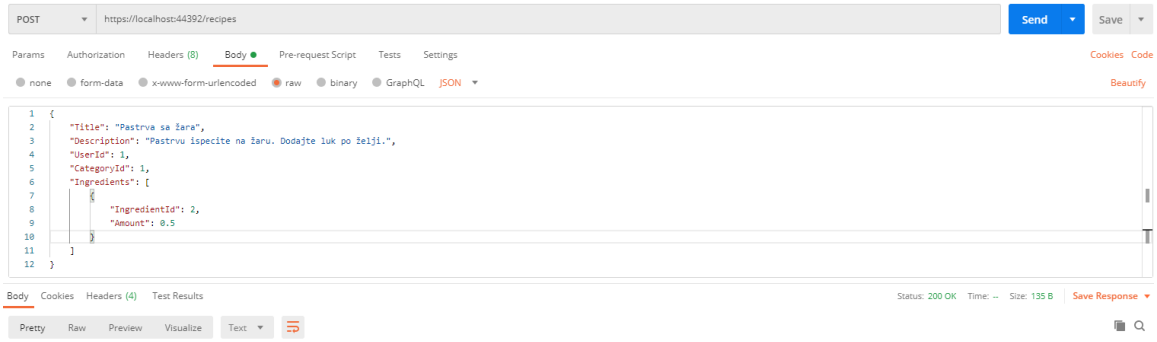


Slika 1.2: Detaljan prikaz metode DELETE /categories/{categoryId}

Na slici 1.2 vidimo detaljan prikaz metode DELETE. Osim prije spomenute rute i ključne riječi, ovdje vidimo zahtjev koji treba poslati te odgovor koji možemo dobiti. Konkretno na ovoj slici, vidimo da zahtjev od parametara mora sadržavati vrijednost za categoryId, dok u odgovoru dobimo samo status 200 i poruku ”Success” ukoliko je naš zahtjev prošao u redu. Pritiskom na gumb ”Try it out” možemo isprobati poslati zahtjev prema toj metodi. U praksi se Swagger UI koristi samo za dokumentaciju API-a, a ne za testiranje i slanje zahtjeva prema aplikaciji, jer za neke složenije upite korištenje ovog sučelja nije najspretnije rješenje. Zbog toga se za slanje zahtjeva najčešće koristi alat Postman.

1.6 Postman

Postman je alat koji služi za razvoj i testiranje API-a. Njegova glavna namjena je slanje zahtjeva prema API-u i prikaz odgovora kojeg dobimo. Vrlo je raširen upravo zbog svoje jednostavnosti, jer je korisničko sučelje vrlo jednostavno i nudi mnoštvo opcija za konfiguriranje poziva.



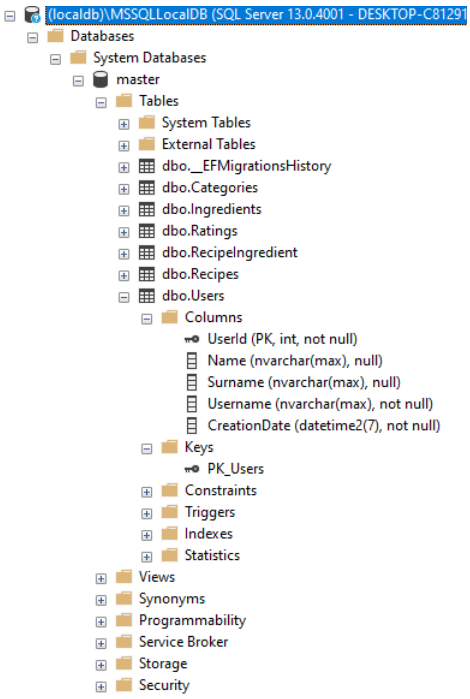
Slika 1.3: Primjer Postman poziva za metodu za dodavanje novog recepta

Na slici 1.3 vidimo primjer Postman poziva metode za dodavanje recepata. Prvo što trebamo odabrati je ključna riječ. Ovdje je odabrana ključna riječ POST jer se radi o dodavanju novog zapisa u bazu. Sljedeće što smo dodali je url na kojem se nalazi navedena metoda. U tijelo zahtjeva dodali smo informacije o podatku koji želi poslati na metodu. Zahtjev je napisan u JSON formatu i iz njega će se kasnije preslikati podaci u modele. Ako pogledamo donji dio slike, vidimo i odgovor koji smo dobili od API-a. Pošto smo dobili odogovor sa kodom 200, znamo da je naš zahtjev prošao u redu. Iz postmana još možemo isčitati veličinu odgovora, vrijeme trajanje zahtjeva i podatke koji su nam poslani u odgovoru. Kako ova metoda ne dohvaća nikakve podatke, nismo dobili nikakve dodatne informacije u tijelu odgovora.

1.7 Microsoft SQL Server Managment Studio 18

Do sada smo prošli kroz alate koji su nam bili potrebni za izradu same aplikaciju. No, naša aplikacija će koristiti bazu podataka za spremanje i dohvaćanje podataka. Da bismo vidjeli stanje na bazi koristit ćemo alat Microsoft SQL Server Managment Studio 18. Pomoću tog alata provjerit ćemo strukturu baze i tablica koje ćemo kreirati, podatke koji će se u tim tablicama nalaziti i kreirati diagram koji će nam reprezentirati tablice i veze između njih.

Prva verzija alata izdana je 2005. godine. Razvila ga je tvrtka Microsoft kako bi Windows korisnicima olakšala korištenje baze podataka. No, namjena alata nije samo za dohvat i spremanje podataka u bazu. Alat korisnicima omogućuje upravljanje i konfiguranje baze. Ako korisnik ima prava, preko alata obavlja administraciju svih komponenti na Microsoft SQL Serveru.



Slika 1.4: Pregled baze podataka u programu MS SQL Server Managment Studio 18

Na slici 1.4 možemo vidjeti strukturu naše baze u navedenom alatu. Vidimo da naša baza sadrži 6 tablica s podacima. Ako proširimo jednu od tih tablica, dobijemo ispis stupaca u tablici i ključeva, ako postoje. Desnim klikom miša na bilo koju od tablica dobiju se dodatne opcije vezane uz tablicu. Također, u alatu se može upisivati kod u SQL jeziku da bismo dodavali, dohvaćali ili mijenjali podatke u tablicama. No, mi nećemo koristiti SQL naredbe za upravljanje podacima u bazi. Taj dio će umjesto nas odraditi Entity Framework Core.

Poglavlje 2

Entity Framework Core

2.1 O alatu

Entity Framework Core (EF Core) je alat koji olakšava razvojnim inžinjerima da pristupaju bazama podataka. Sam alat dizajniram je kao *object-relational mapper*, tj. kao neka vrsta preslikavača. Služi kao spona između dva svijeta: svijeta objekata i softverskog koda. Glavna namjena mu je ubrzati i olakšati razvoj softvera.

EF Core je izdan 2016. godine. Iako ga je razvila tvrtka Microsoft, zapravo se radi o alatu koji je upotrebljiv na više platforma, jer radi i na Windowsu, Linuxu i MacOS. Naziv Core dobio je zato jer je alat bio dio .NET inicijative, koja je za cilj imala prilagoditi razvoj ASP.NET aplikacija da bi se mogle razvijati na više platforma. Prije EF Core-a, postojala je verzija EF6.x, koja je omogućavala razvoj samo na Windowsima. U tijeku pisanja ovog rada, izdana je novija verzija Entity Framework-a nazvana EF Core 5.0 i ona bi trebala pratiti razvoj .NET 5 aplikacijskog sučelja.

Prije nego se krenemo upoznavati s alatom, objasnimo već spomenuti pojam object relational mapper. Prevedeno, to bi bio preslikavač iz objekta u relaciju i obratno. Objekte definiramo u našem kodu. Nakon što definiramo sve objekte, pozovemo alat EF Core. On na temelju definiranih objekata izgenerira relacije koje zatim naša baza može pročitati i zapisati. Pritom ovdje pod pojmom relacija mislimo na tablicu na našoj bazi. Ako očekujemo odgovor od baze, baza ga šalje u obliku relacije. Entity Framework na temelju toga popuni naše objekte s informacijama. Developeru se čini da cijelo vrijeme radi sa objektima, dok se u pozadini ti objekti preslikaju u relacije i obratno. To omogućuje da inžinjer razvija aplikaciju koja koristi SQL bazu podataka bez da uopće poznaje SQL jezik.

2.2 Kreiranje modela

Kako smo odabrali code first pristup, znači da nam je baza podataka u početku prazna. Potrebo je u kodu definirati objekte. Na temelju tih objekata alat Entity Framework Core će kreirati tablice u bazi. U literaturi se često objekti koji imaju atribute ali nemaju metode nazivaju entiteti, pa ću ih odsada i ja tako nazivati.

Kako radimo aplikaciju koja bi trebala simulirati web-kuharicu, potrebno nam je šest tablica u bazi. Jedna u kojoj će biti kategorije jela, jedna u kojoj će biti namirnice, jedna u kojoj ćemo bilježiti ocjenjivanje recepata, jedna u kojoj će se spremati recepti, jedna u kojoj će biti spremljeni korisnici i jedna vezna tablica, koja će spajati recepte i namirnice. Stoga treba kreirati šest objekata. Pogledajmo npr. kako izgleda klasa koja predstavlja objekt ocjene.

```
1  public class Rating
2  {
3      public int RatingId { get; set; }
4
5      public int RecipeId { get; set; }
6
7      public int UserId { get; set; }
8
9      public int Grade { get; set; }
10
11     public virtual Recipe Recipe { get; set; }
12
13     public virtual User User { get; set; }
14 }
```

Isječak koda 2.1: Definicija klase Rating

U odsječku koda 2.1 vidimo primjer klase Rating. Jedna instanca klase predstavlja jednu ocjenu. U njoj će biti spremljeni podaci o id-u ocjene, id-u recepta, id-u korisnika koji je kreirao recept i ocjena kojom je korisnik ocijenio recept. No, ovdje vidimo još dva objekta. Jedan tipa Recipe i jedan tipa User. To su druga dva entiteta. Trenutno ne definiramo na koji način su ti entiteti povezani. To ćemo kasnije definirati u konfiguraciji.

Promotrimo još jedan zanimljiv entitet. Ovaj entitet predstavljat će nam veznu tablicu između entiteta Recipe i entiteta Ingredient. Razlog tomu je to što želimo omogućiti da jedan recept sadrži više namirnica, ali i jedna namirnica može biti u više recepata. Zbog toga ćemo na bazi imati Many-To-Many vezu.

```
public class RecipeIngredient
{
    public int RecipeIngredientId { get; set; }

    public int RecipeId { get; set; }

    public int IngredientId { get; set; }

    public decimal Amount { get; set; }

    public virtual Recipe Recipe { get; set; }

    public virtual Ingredient Ingredient { get; set; }
}
```

Isječak koda 2.2: Definicija klase RecipeIngredient

2.3 Konfiguracija entiteta

U prvom koraku definirali smo od čega će se naši entiteti sastojati. Vidjeli smo na primjeru da pojedini entiteti mogu u sebi sadržavati i druge entitete. Sljedeći korak je dodatno definirati svaki pojedini entitet. Tu ćemo prvi put koristiti usluge EF Core-a.

```
public class RecipeIngredientConfiguration
    : IEntityTypeConfiguration<RecipeIngredient>
{
    public void Configure(
        EntityTypeBuilder<RecipeIngredient> builder)
    {
        builder.HasKey(b => b.RecipeIngredientId);
        builder.HasIndex(b => new { b.RecipeId, b.IngredientId }).
    IsUnique();
        builder.HasOne(b => b.Recipe).WithMany(b => b.RecipeIngredients)
               .HasForeignKey(b => b.RecipeId)
               .OnDelete(DeleteBehavior.ClientCascade);
        builder.HasOne(b => b.Ingredient).WithMany(b => b.
    RecipeIngredients)
               .HasForeignKey(b => b.IngredientId)
               .OnDelete(DeleteBehavior.ClientCascade);
        builder.Property(b => b.Amount).IsRequired().HasColumnType("
    decimal(5,2)");
        builder.Property(b => b.RecipeId).IsRequired();
        builder.Property(b => b.IngredientId).IsRequired();
```

```
18      }
19  }
```

Isječak koda 2.3: Konfiguracijska klasa za entitet RecipeIngredient

U isječku koda 2.3 vidimo primjer jedne takve konfiguracije. Prođimo kroz neke bitne retke i objasnimo u kojem se što radi. U redu broj 7, pomoću metode HasKey, definiramo što će nam služiti kao primarni ključ u tablicama. Mi smo odabrali da nam stupac koji predstavlja RecipeIngredientId bude primarni ključ. U retku broj 8 definiramo da nam kombinacija retka i sastojka bude jedinstvena.

U retku 9 definiramo vezu između dva entiteta. Konkretno, definirali smo da za jedan recept imamo listu entiteta RecipeIngredient, kod koje je svaki član entiteta povezan s originalnim receptom pomoću RecipeId-a. Dodatno, ovdje je definirano ponašanje prilikom brisanja entiteta. Definirano je na način da prilikom brisanja jedne instance entiteta koji na sebi sadrži instancu nekog drugog entiteta dobivamo upozorenje da početnu instancu nije moguće izbrisati jer se na nju veže druga instanca entiteta. Tada imamo dvije opcije: ili prvo brišemo vezanu instancu pa tek onda početnu ili prilikom brisanja eksplicitno navedemo da želimo obrisati instancu entiteta i sve instance drugih entiteta koji se vežu na njega. Detaljnije o tome ćemo pričat kasnije u radu.

2.4 Klasa EntityContext

Za sada smo definirali entitete i njima smo pridružili određene konfiguracije. Ono što nam slijedi je definiranje klase EntityContext. To je klasa koja će predstavljati našu bazu. U njoj ćemo navesti koje entiteti će biti uključeni u našu bazu te koje konfiguracije će biti upotrebljene kod kreiranja baze. Moguće je ovdje definirati dodatne postavke u vezi same baze, ali mi ćemo taj dio trenutno zanemariti.

Promotrimo primjer klase koju koristimo u našem demo projektu.

```
1  public class EntityContext : DbContext
2  {
3      public EntityContext(DbContextOptions<EntityContext> options)
4          : base(options)
5      {
6      }
7
8      public virtual DbSet<Category> Categories { get; set; }
9      public virtual DbSet<Ingredient> Ingredients { get; set; }
10     public virtual DbSet<Rating> Ratings { get; set; }
11     public virtual DbSet<Recipe> Recipes { get; set; }
12     public virtual DbSet<User> Users { get; set; }
```

```
13      public virtual DbSet<RecipeIngredient> RecipeIngredient { get; set;
        }
14
15
16      protected override void OnModelCreating(ModelBuilder modelBuilder)
17      {
18          base.OnModelCreating(modelBuilder);
19          modelBuilder.ApplyConfiguration(new RecipeConfiguration());
20          modelBuilder.ApplyConfiguration(new UserConfiguration());
21          modelBuilder.ApplyConfiguration(new RatingConfiguration());
22          modelBuilder.ApplyConfiguration(new IngredientConfiguration());
23          modelBuilder.ApplyConfiguration(new CategoryConfiguration());
24          modelBuilder.ApplyConfiguration(new
        RecipeIngredientConfiguration());
25      }
26  }
```

Isječak koda 2.4: Definicija klase EntityContext

Kod definiranja klase, prvo što je potrebno dodati jest da naša klasa nasljeđuje DbContext klasu, koja se nalazi u paketu unutar EF Core alata. Sljedeće definiramo konstruktor. Nakon konstruktora, definiramo koje entitete će EF Core preslikati u relacije. Na posljetku, unutar metode OnModelCreating, definiramo koje sve konfiguracije EF Core treba uzeti da bi pravilno radio s entitetima.

Posljednji korak je najaviti da će se ta klasa koristiti u našem programu. To se događa pri pokretanju aplikacije kod registriranja servisa. Metodom AddDbContext, kojoj proslijedimo string pomoću kojeg se povezujemo na bazu, najavimo da je EntityContext klasa koju želimo da nam predstavlja bazu podataka. Za primjer, za string ćemo uzet vrijednost DbConncetionString.

```
1  services.AddDbContext<EntityContext>(opt
2      => opt.UseSqlServer("DbConnectionString"));
```

Isječak koda 2.5: Najava EntityContext klase unutar Startup klase

2.5 Migracije

Trenutno smo u kodu definirali kako bismo željeli da naša baza izgleda. No, još uvijek nismo kreirali niti jednu tablicu. Tu prvi puta vidimo moć EF-a. Pomoću samo jedne naredbe izgenerirat će se migracijske skripte pomoću kojih se kreiraju sve tablice u bazi. Nakon toga na nama je samo da pokrenemo te skripte i one će se izvršiti na našoj bazi.

Krenimo s prvom naredbom. Za upisivanje naredbi koristiti ćemo Windows PowerShell

kojeg ćemo pokrenuti kao administrator. Pozicionirat ćemo se u direktorij u kojem se nalazi naš projekt. U tom direktoriju izvršit ćemo sljedeću naredbu:

```
dotnet ef migrations add Initial
```

Isječak koda 2.6: Powershell naredba za pokretanje migracija

Gornja naredba je znak EF-u da kreira novu skriptu. Pošto je ovo prva migracijska skripta u projektu, najčešće joj se dodjeljuje naziv Initial. Nakon što smo izvršili gornju naredbu, EF će pokušati izgenerirati skripte. Ukoliko su se skripte uspješno izvrtjele, u našem projektu trebali bismo vidjeti direktorij Migrations.

Unutar tog direktorija nalaze se dvije datoteke. Jedna od njih u svom nazivu ima vremenski žig i naziv naše migracije. To je izgenerirana klasa koja predstavlja promjene koje su se dogodile u migraciji Initial. Primjetimo da unutar klase imamo izgenerirane dvije metode: Up i Down. U Up se nalaze promjene koje treba izvršiti kada želimo da promjene iz migracije imaju utjecaj na bazu, dok se u Down nalaze stvari koje treba odraditi ako želimo izbrisati promjene iz naše migracije na bazi.

```
protected override void Up(MigrationBuilder migrationBuilder)
{
    migrationBuilder.CreateTable(
        name: "Categories",
        columns: table => new
        {
            CategoryId = table.Column<int>(nullable: false)
                .Annotation("SqlServer:Identity", "1, 1"),
            CategoryName = table.Column<string>(maxLength: 50, nullable:
    false),
            CategoryDescription = table.Column<string>(maxLength: 200,
   nullable: true)
        },
        constraints: table =>
        {
            table.PrimaryKey("PK_Categories", x => x.CategoryId);
        });

    migrationBuilder.CreateTable(
            name: "RecipeIngredient",
            columns: table => new
            {
                RecipeIngredientId = table.Column<int>(nullable: false)
                    .Annotation("SqlServer:Identity", "1, 1"),
                RecipeId = table.Column<int>(nullable: false),
                IngredientId = table.Column<int>(nullable: false),
                Amount = table.Column<decimal>(type: "decimal(5,2)",
   nullable: false)
```

```
26              },
27              constraints: table =>
28              {
29                  table.PrimaryKey("PK_RecipeIngredient", x => x.
    RecipeIngredientId);
30                  table.ForeignKey(
31                      name: "FK_RecipeIngredient_Ingredients_IngredientId"
    ,
32                      column: x => x.IngredientId,
33                      principalTable: "Ingredients",
34                      principalColumn: "IngredientId",
35                      onDelete: ReferentialAction.Restrict);
36                  table.ForeignKey(
37                      name: "FK_RecipeIngredient_Recipes_RecipeId",
38                      column: x => x.RecipeId,
39                      principalTable: "Recipes",
40                      principalColumn: "RecipeId",
41                      onDelete: ReferentialAction.Restrict);
42              });
43
44          migrationBuilder.CreateIndex(
45          name: "IX_RecipeIngredient_IngredientId",
46          table: "RecipeIngredient",
47          column: "IngredientId");
48
49          migrationBuilder.CreateIndex(
50              name: "IX_RecipeIngredient_RecipeId_IngredientId",
51              table: "RecipeIngredient",
52              columns: new[] { "RecipeId", "IngredientId" },
53              unique: true);
54
55          migrationBuilder.CreateIndex(
56          name: "IX_Ratings_UserId_RecipeId",
57          table: "Ratings",
58          columns: new[] { "UserId", "RecipeId" },
59          unique: true);
60  }
```

Isječak koda 2.7: Primjer metode Up

U isječku koda 2.7 vidimo dio metode Up iz naše demo aplikacije. U toj metodi se nalaze skripte koje predstavljaju promjene u sklopu migracije Initial. U liniji broj 3 nalazi se metoda za kreiranji nove tablice. U parametrima metode navedeno je ime tablice, naziv i konfiguracija stupaca u tablici te je naveden primarni ključ te tablice. Ove informacije odgovaraju entitetima i njihovim konfiguracijama.

U liniji 17 vidimo još jednu metodu za kreiranje tablice. Razlika u odnosu na prethodnu

je to što ovdje osim primarnih ključeva imamo i strane ključeve. Npr. vidimo da tablica RecipeIngredient ima dva strana ključa. Jedan se nalazi u stupcu IngredientId i odgovara podatku koji se nalazi u tablici Ingredients, u stupcu IngredientId. Analogno za drugi strani ključ, RecipeId, koji se nalazi u tablici Recipes, u stupcu RecipeId.

Osim metoda za kreiranje tablice, u klasi vidimo još i metodu CreateIndex. Ta metoda služi za kreiranje složenih indexa. Mi to koristimo npr. u tablici Rating, gdje pomoću indeksa osiguravamo da nam kombinacija UserId i RecipeId bude jedinstvena. Time sprječavamo da isti korisnik više puta ocijeni isti recept.

Provjerimo sada što se nalazi u metodi Down.

```
protected override void Down(MigrationBuilder migrationBuilder)
{
    migrationBuilder.DropTable(
        name: "Ratings");

    migrationBuilder.DropTable(
        name: "RecipeIngredient");

    migrationBuilder.DropTable(
        name: "Ingredients");

    migrationBuilder.DropTable(
        name: "Recipes");

    migrationBuilder.DropTable(
        name: "Categories");

    migrationBuilder.DropTable(
        name: "Users");
}
```

Isječak koda 2.8: Primjer metode Down

U isječku koda 2.8 možemo primjetiti da se jedino spominje metoda DropTable. Ona služi da bismo obrisali tablicu. To je logično zato jer smo u metodi Up kreirali tablice, pa ih u metodi Down moramo izbrisati. Općenito, u metodi down trebale bi se nalaziti promjene koje poništavaju sve promjene iz metode Up.

Izmjenimo sada jednu konfiguracijsku klasu, npr. u entitetu Category definirali smo da nam kolona CategoryName prihvaća najviše 50 znakova. Promjenimo taj broj na 100 i pokrenimo novu migraciju koju ćemo nazvati ChangedCategoryName. Nakon što je migracija uspješno prošla, kreirala se nova migracijska skripta.

```
public partial class ChangedCategoryName : Migration
{
    protected override void Up(MigrationBuilder migrationBuilder)
    {
        migrationBuilder.AlterColumn<string>(
            name: "CategoryName",
            table: "Categories",
            maxLength: 100,
            nullable: false,
            oldClrType: typeof(string),
            oldType: "nvarchar(50)",
            oldMaxLength: 50);
    }

    protected override void Down(MigrationBuilder migrationBuilder)
    {
        migrationBuilder.AlterColumn<string>(
            name: "CategoryName",
            table: "Categories",
            type: "nvarchar(50)",
            maxLength: 50,
            nullable: false,
            oldClrType: typeof(string),
            oldMaxLength: 100);
    }
}
```

Isječak koda 2.9: Migracijska skripta za drugu migraciju

U isječku koda 2.9 primjećujemo da je migracijska skripta vrlo jednostavna. I u metodi Up i u metodi Down koristi se AlterColumn naredba koja mijenja veličinu stupca u tablici.

Migracijske skripte će se na bazi izvršavati jedna po jedna. Ako bi imali više skripta bilo bi nam teško pratiti kakvo bi bilo stanje baze da se sve one izvrše. Zbog toga EF Core generira još jednu klasu, EntityContextModelSnapshot, u kojoj možemo vidjeti kako bi baza izgledala ako se pokrenu sve izgenerirane skripte.

```
protected override void BuildModel(ModelBuilder modelBuilder)
{
    modelBuilder.Entity("Web.Kuharica.Domain.Models.Entities.Recipe", b
   =>
    {
        b.Property<int>("RecipeId")
            .ValueGeneratedOnAdd()
            .HasColumnType("int")
            .HasAnnotation("SqlServer:ValueGenerationStrategy",
   SqlServerValueGenerationStrategy.IdentityColumn);

```

```
10          b.Property<int>("CategoryId")
11              .HasColumnType("int");
12
13          b.Property<string>("Description")
14              .IsRequired()
15              .HasColumnType("nvarchar(max)");
16
17          b.Property<string>("Title")
18              .IsRequired()
19              .HasColumnType("nvarchar(50)")
20              .HasMaxLength(50);
21
22          b.Property<int>("UserId")
23              .HasColumnType("int");
24
25          b.HasKey("RecipeId");
26
27          b.HasIndex("CategoryId");
28
29          b.HasIndex("UserId");
30
31          b.ToTable("Recipes");
32      });
33
34      modelBuilder.Entity("Web.Kuharica.Domain.Models.Entities.Recipe", b
    =>
35      {
36          b.HasOne("Web.Kuharica.Domain.Models.Entities.Category", "
    Category")
37              .WithMany("Recipes")
38              .HasForeignKey("CategoryId")
39              .OnDelete(DeleteBehavior.ClientCascade)
40              .IsRequired();
41
42          b.HasOne("Web.Kuharica.Domain.Models.Entities.User", "User")
43              .WithMany("Recipes")
44              .HasForeignKey("UserId")
45              .OnDelete(DeleteBehavior.ClientCascade)
46              .IsRequired();
47      });
48  }
```

Isječak koda 2.10: Primjer klase EntityContextModelSnapshot

Isječak koda 2.10 prestavlja klasu EntityContextModelSnapshot. U njoj za svaki entitet iz našeg koda vidimo koja mu tablica odgovara. Tu također možemo pronaći informaciju o stupcima unutar tablica te primarnim i stranim ključevima. Kada smo se uvjerili da su nam skripte izgenerirane pravilno, možemo pokrenuti sljedeću naredbu:

```
dotnet ef database update
```

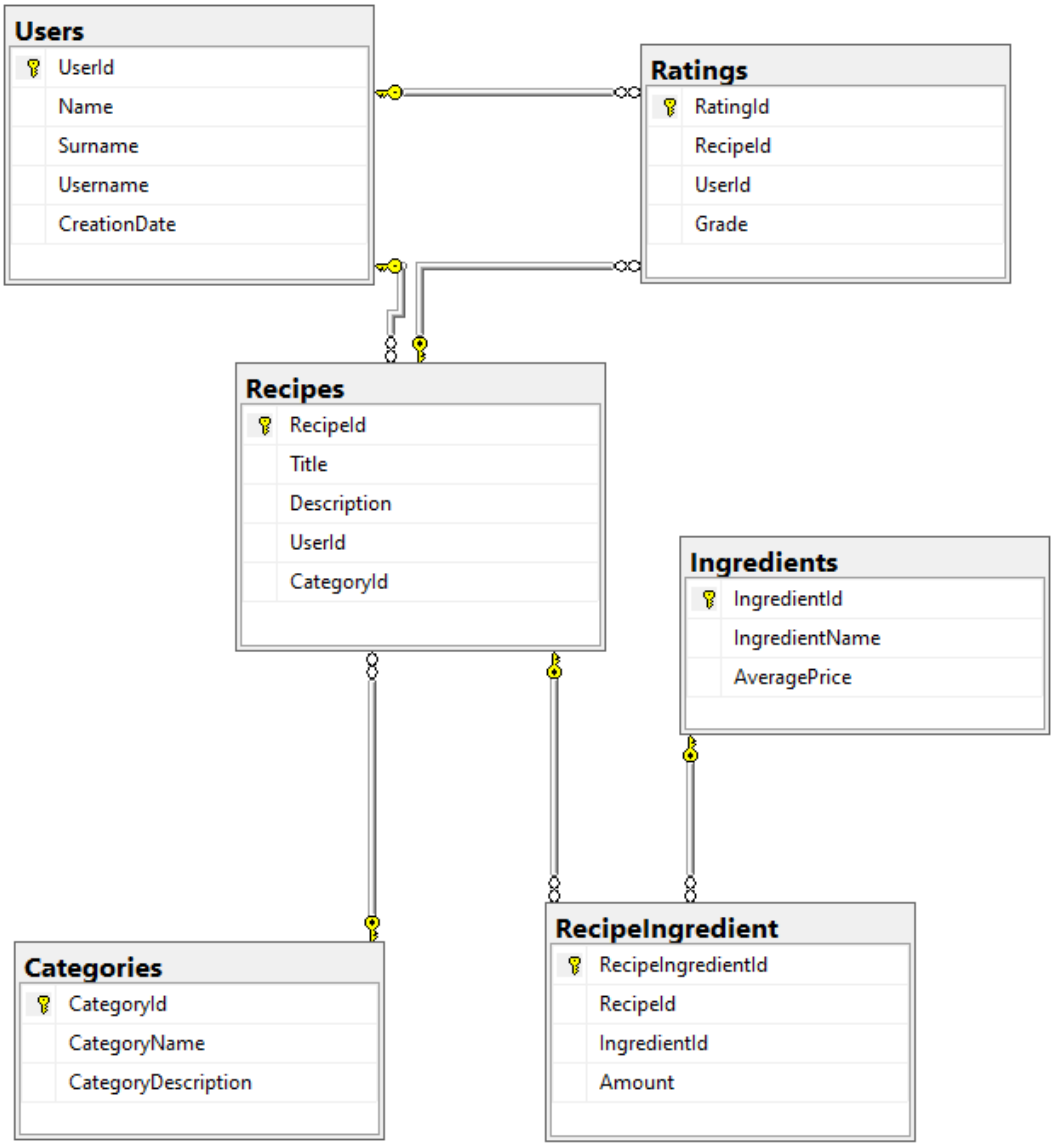
Isječak koda 2.11: Naredba za primjenjivanje skripta na bazu

Pomoću gornje naredbe primjenjujemo migracijske skripte na bazu. Ako naša naredba prođe u redu, trebali bismo vidjeti promjene na bazi. U slučaju naše demo aplikacije, na bazi vidimo da su se kreirale tablice koje odgovaraju entitetima iz koda. Osim toga, kreirana je još jedna tablica, _EFMigrationsHistory, u kojoj su spremljeni podaci o dosad provedenim migracijama.

	MigrationId	ProductVersion
1	20201219105410_Initial	3.1.9
2	20201219113013_ChangedCategoryName	3.1.9

Slika 2.1: Stanje u tablici _EFMigrationsHistory nakon 2 migracije

Za kraj ovog odjelka, pogledajmo kakav je konačan izgleda naše baze. Koristit ćemo diagram baze podataka kako bismo detaljno vidjeli sve tablice, atribute unutar tablica te veze između tablica. Primjetimo da niti u jednom trenutku nismo pisali kod koji bi bio razumljiv bazi podataka. Nismo uopće razmišljali u jeziku relacija. Jedino što smo mi kreirali bili su modeli. EF Core je obavio posao prebacivanja našeg koda u jezik razumljiv bazi. U daljnjim odmlomcima vidjet ćemo da nije kreiranje baze jedina snaga EF Core-a. Ono što nam alat omogućuje je daljnje komuniciranje s bazom. Mi i dalje jedino vodimo računa o modelima iz našeg koda. Sav posao na bazi vezan uz tablice i veze i dalje obavlja EF Core.



Slika 2.2: Stanje baze podataka nakon izvršenih migracija

2.6 Dodavanje novog podatka u bazu

Prva radnja koju ćemo opisati biti će dodavanje novog podatka u bazu. Dodavanje podataka odvija se u tri koraka. U prvom koraku napunimo željene entitete s podacima koje želimo da se zapišu na bazu. U drugom koraku svaku instancu entiteta koja se zapisuje u bazu dodajemo u kontekst koji predstavlja našu bazu. U zadnjem koraku pozovemo metodu koja ne temelju promjena u kontekstu otkriva promjene koje treba zapisati na bazu. Pogledajmo prvo jednostavan primjer dodavanja jedne vrijednosti za entitet Category u bazu.

```
1 public async Task AddCategoryAsync(AddCategoryRequest addCategoryRequest
      )
2 {
3     var category = addCategoryRequest.MapToEntity();
4     _context.Categories.Add(category);
5     await _context.SaveChangesAsync();
6 }
```

Isječak koda 2.12: Dodavanje nove kategorije u bazu

```
1 public static Category MapToEntity(this AddCategoryRequest
      addCategoryRequest)
2 {
3     return new Category
4     {
5         CategoryDescription = addCategoryRequest.CategoryDescription,
6          CategoryName = addCategoryRequest.CategoryName
7     };
8 }
```

Isječak koda 2.13: Preslikavanje podataka u entitet Category

U isječku koda 2.12 vidimo dodavanje jedne instance entiteta Category u bazu. U liniji 3 vidimo da prvo trebamo kreirati instancu entiteta i napuniti ju željenim podacima. Zatim u liniji 4 dodajemo taj entitet u kontekst koji predstavlja bazu podataka. Na kraju, u liniji 5, pozivamo metodu SaveChangesAsync(), kojom promjene u entitetu prebacujemo na bazu. Nakon što se metoda uspješno izvrši, u tablici Categories u bazi bismo trebali vidjeti novi zapis.

Trenutno smo vidjeli primjer dodavanja jednostavnog entiteta. Upotrebljvamo riječ jednostavan zato jer u sebi ne sadrži podentitete. Pogledajmo sada kako bi izgledalo dodavanje novog entiteta koje u sebi podentitet. To možemo napraviti na dva načina.

```
1 public class AddRatingRequest
2 {
3     public int UserId { get; set; }
4
```

```
5     public int RecipeId { get; set; }
6
7     public int Grade { get; set; }
8 }
```

Isječak koda 2.14: Definicija klase AddRatingRequest

```
1 public async Task AddRatingWitouthValidationAsync(AddRatingRequest
      addRatingRequest)
2 {
3     var rating = new Rating
4     {
5         Grade = addRatingRequest.Grade,
6         RecipeId = addRatingRequest.RecipeId,
7         UserId = addRatingRequest.UserId
8     };
9
10    await _context.Ratings.AddAsync(rating);
11    await _context.SaveChangesAsync();
12 }
```

Isječak koda 2.15: Dodavanje nove instance entiteta Rating bez validacije

U isječku koda 2.15 vidimo primjer metode za dodavanje nove instance entiteta Rating. U liniji 3, na temelju ulaznih parametara, kreiramo novu instancu entiteta Rating. Nakon toga, u liniji 10 dodajemo novokreirani zapis u naš kontekts. Na kraju, u liniji 11, pozovemo metodu SaveChangesAsync(). Ukoliko je sve prošlo u redu, nakon što se metoda izvrši trebali bismo vidjeti novi zapis na bazi. Iako se ova metoda čini poprilično čista, ipak moramo pripaziti na neke stvari. Primječujemo da u ovoj metodi nigdje nemamo validaciju ulaznih parametara. Zbog toga može doći do greške prilikom dodavanja zapisa u bazu. Razlog tomu je to što nam vrijednosti RecipeId i UserId predstavljaju strane ključeve. Problem bi se desio kada na bazi u tablicama User ili Recipe nebi postojao zapis s tim id-em. Ne bi se mogla stvoriti relacija između tablica te bismo od baze dobili odgovor da je došlo do greške prilikom dodavanja zapisa.

Da bismo se osigurali da ne dođe do pokušaja dodavanja nepostojećeg stranog ključa, potrebno je izvršiti validaciju ulaznih parametara. Tek ako validacije prođu, krećemo u dodavanje novog zapisa u bazu.

```
1 public async Task AddRatingAsync(AddRatingRequest addRatingRequest)
2 {
3     var user = await _context.Users.SingleOrDefaultAsync(user => user.
      UserId == addRatingRequest.UserId);
4
5     var recipe = await _context.Recipes.SingleOrDefaultAsync(recipe =>
      recipe.RecipeId == addRatingRequest.RecipeId);
```

```
6
7       if(recipe != null && user != null)
8       {
9           var rating = new Rating
10          {
11              Grade = addRatingRequest.Grade,
12              Recipe = recipe,
13              User = user
14          };
15
16          await _context.Ratings.AddAsync(rating);
17          await _context.SaveChangesAsync();
18      }
19  }
```

Isječak koda 2.16: Dodavanje nove instance entiteta Rating sa validacijom

U isječku koda 2.16 dodali smo novu instancu entiteta Rating. Pritom smo prvo izvršili validaciju ulaznih parametara. U liniji 3 izvršena je validacija ulaznog parametra UserId na način da je s baze dohvaćen zapis iz tablice User kojem odgovara UserId iz ulaznog parametra. Analogno, i liniji 5 izvršili smo validaciju za vrijednost RecipeId. U liniji 7 provjeravamo da li su obje validacije prošle uspješno. Ako su se dohvatila oba dva podatka stranih ključeva, kreće se u kreiranje nove instance entiteta Rating u liniji 9. Nakon što je instanca kreirana, u liniji 16 dodaje se u kontekst te u liniji 17 šalje na bazu.

Primjetimo razliku između dodavanja novog podatka u isječcima 2.15 i 2.16 U 2.15 smo kod kreiranja nove instance entiteta Rating popunjavali vrijednost UserId. Dakle, dodali smo vrijednost stranog ključa. U 2.16 nismo dodali ključ, nego smo na temelju ključa dohvatili cijeli podatak o podentitetu User te smo onda kod kreiranja Rating-a dodali instancu dohvaćenog entiteta User. Dakle, moguće je dodati enitett na način da se kod kreiranja pošalje samo vrijednost stranog ključa ili se pošalje cijeli podentitet. Zapravo su oba dva načina jednaka. EF Core odradi posao umjesto nas i na temelju podatka koji mu je poslan popuni vrijednosti koje nedostaju. Ako mu pošaljemo strani ključ, on zna na koji podentitet se on odnosi i popuni vrijednost podentiteta. Ako mu pošaljemo vrijednost koja odgovara podentitetu, on prepoznaje kojom vrijednosti treba popuniti strani ključ da bi ga povezao sa navedenim podentitetom.

Promotrimo još jedan slučaj dodavanja novog zapisa u bazu. Ovaj je najsloženiji do sad, jer uključuje promjene u više tablica. Dodat ćemo novu vrijednost u tablicu Recipe. Istovremeno, za taj recept ćemo dodati sastojke. Sjetimo se da se ovdje radi o vezi Many-To-Many. Zbog toga moramo dodati vrijednosti u veznu tablicu. Pogledajmo kako rješiti taj problem.

```
1 public class IngredientWithAmount
```

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

    public decimal Amount { get; set; }
}
```

Isječak koda 2.17: Definicija klase IngredientWithAmount

```
public class AddRecipeRequest
{
    public string Title { get; set; }

    public string Description { get; set; }

    public int UserId { get; set; }

    public int CategoryId { get; set; }

    public List<IngredientWithAmount> Ingredients { get; set; }
}
```

Isječak koda 2.18: Definicija klase AddRecipeRequest

```
public async Task AddRecipeAsync(AddRecipeRequest addRecipeRequest)
{
    var recipe = addRecipeRequest.MapToEntity();
    await _context.Recipes.AddAsync(recipe);

    if(addRecipeRequest.Ingredients != null && addRecipeRequest.
    Ingredients.Any())
    {
        foreach (var ingredientsRequest in addRecipeRequest.Ingredients)
        {
            var ingredient = await _context.Ingredients.
    SingleOrDefaultAsync(ingredients => ingredients.IngredientId ==
    ingredientsRequest.IngredientId);

            if (ingredient != null)
            {
                var recipeIngredient = new RecipeIngredient
                {
                    Recipe = recipe,
                    Ingredient = ingredient,
                    Amount = ingredientsRequest.Amount
                };

                await _context.RecipeIngredient.AddAsync(
    recipeIngredient);
```

```
22                 }
23             }
24         }
25
26         await _context.SaveChangesAsync();
27     }
```

Isječak koda 2.19: Definicija klase IngredientWithAmount

U isječku koda 2.19 vidimo primjer metode za dodavanje recepta u bazu. U liniji 3 na temelju ulaznih parametara kreiramo novu instancu entiteta Recipe. Nakon toga, u liniji 4 dodajemo novokreiranu instancu u kontekst. U liniji 6 provjeravamo da li je u ulaznim parametrima poslana vrijednost sastojka koji ulaze u ovaj recept. Ako je, za svaki poslani id od namirnice u liniji 10 dohvatimo odgovarajuću namirnicu iz baze. U liniji 12 provjerimo da li namirnica s tim id-em postoji, kreiramo novi entitet koji u liniji 21 dodajemo u kontekst. On će na bazi prestavljati vrijednosti u veznoj tablici. Na kraju metode, u liniji 26, potrebno je pozvati metodu koja promjene iz konteksta šalje na bazu.

Primjetimo da smo prvo dodali sve izmjene u kontekst, a tek na kraju metode pozvali metodu za slanje promjena na bazu. Ako bi došlo do greške kod slanja podataka na bazu, ne bi se izvršila niti jedna promjena iz naše metode. To odgovara transakcijama na bazi podataka. Ako se dogodi greška u nekom koraku transakcije, sve dotadašnje promjene se poništavaju. Analogno je ponašanje i kod EF Core-a.

Trenutno smo prošli korake dohvaćanja podataka iz baze te dodavanja novih. No, ponekad ne želimo dodati potpuno novi podatak nego samo promjeniti određene vrijednosti nekog podatka. U sljedećem odlomku pokazat ćemo kako izgleda kod u slućaju mijenjanja podataka na bazi.

2.7 Dohvaćanje podataka iz baze podataka

Sljedeća radnja koju ćemo opisati biti će dohvaćanje podataka iz baze. Proučit ćemo načine na koje je moguće dohvatiti podatke koji su povezani u bazi. Spomenut ćemo i stvari na koje treba obratiti pažnju kako bi naši upiti na bazu trajali najkraće moguće.

Pogledajmo primjer klase UsersRepository. To je naša klasa koja će nam služiti za komunikaciju između programa i baze podataka. U toj klasi nalazi se metoda GetUsersAsync(), koja dohvaća popis korisnika iz tablice User.

```
1 public class UsersRepository : IUsersRepository
2 {
3     private readonly EntityContext _context;
```

```
4
5       public UsersRepository(EntityContext context)
6       {
7           _context = context;
8       }
9
10      public async Task<IEnumerable<User>> GetUsersAsync()
11      {
12          return await _context.Users.AsNoTracking().ToListAsync();
13      }
14  }
```

Isječak koda 2.20: Klasa UsersRepository

U isječku koda 2.20 primjećujemo da klasa sadrži privatnog člana _context. On je tipa EntityContext. Njega u konstruktoru klase popunimo sa kontekstom baze podataka kojeg smo najavili kod pokretanja programa u isječku koda 2.5. Varijabla _context predstavljat će nam bazu podataka u ovoj klasi. Promotrimo sada metodu GetUserAsync(). Povratni tip joj je Task koji u sebi sadrži listu kojoj su članovi tipa entiteta User. Ako pogledamo tijelo metode, u njemu vidimo naredbu *_context.Users.AsNoTracking().ToListAsync()*. Varijablu _context smo već spomenili, ona nam predstavlja našu bazu. Dodavši na nju naredbu Users rekli smo da nam se dohvati samo tablica Users. AsNoTracking je metoda kojom eksplicitno kažemo EF Core-u da ne prati promjene koje ćemo eventualno raditi na podacima. Ona se primarno koristi kod podataka koje želimo samo pročitati, a ne mjenjati. Osnovna namjena te komande je ubrzati izvršavanje našeg upita, jer se ne spremaju informacije potrebne za praćenje dohvačenih podataka. Posljednja naredba je ToListAsync(), koja asinkrono transformira dobivene podatke u listu. Ako se sve uspješno izvrtilo, metoda GetUserAsync trebala bi uspješno vratiti podatke o korisnicima.

Trenutno smo dohvatili sve podatke iz tablice Users. U praksi je vrlo čest slučaj da se ne dohvaćaju svi podaci iz tablice, nego se oni filtriraju po nekim parametrima. Pogledajmo dva primjera: jedan u kojem u odgovoru očekujemo samo jedan podatak i jedan gdje u odgovoru može biti niti jedan, jedan ili više podataka.

```
1  public async Task<User> GetUserByIdAsync(int userId)
2  {
3      return await _context.Users
4              .SingleOrDefaultAsync(user => user.UserId == userId);
5  }
6
7  public async Task<IEnumerable<User>> GetUsersByNameAsync(string name)
8  {
9      return await _context.Users
10             .Where(user => user.Name == name).ToListAsync();
```

```
11 }
```

Isječak koda 2.21: Dohvaćanje korisnika po id-u

U isječku koda 2.21 nalaze se dvije metode. Pogledajmo metodu GetUserByIdAsync(). U toj metodi, nakon što dohvatimo sve zapise iz tablice Users, pomoću metode SingleOrDefaultAsync(), provjerimo koji zapis odgovara uvjetu da ima jednak UserId kao vrijednost ulaznog parametra. Kako je UserId primarni ključ, sigurni smo da u bazi postoji najviše jedan podatak s tim id-em. Zbog toga možemo koristiti SingleOrDefault. No, s tom metodom treba biti oprezan jer ukoliko bi za više zapisa na bazi vrijedilo da zadovoljavaju uvjet u zagradama, metoda bi bacila iznimku i postojala bi opsanost da nam program prestane raditi. Zbog toga, kada postoji ta mogućnost, umjesto metode SingleOrDefault() koristimo metodu Where().

Primjer korištenja metode Where() je u metodi GetUsersByNameAsync(). To je metoda koja vraća sve korisnike koji zadovoljavaju uvjet da im je ime jednako imenu koje dobivamo kao ulazni parametar metode. Kako ime ne treba biti jedinstveno, metoda Where() može vratiti više zapisa koji zadovoljavaju uvjet. Kod nje ne treba paziti o broju elemenata koji zadovoljavaju uvjet zato jer vraćamo listu elemenata, a ta lista može biti prazna, imati samo jednog člana ili više njih.

Za sada smo vidjeli dohvaćanje podataka iz samo jedne tablice. Tablica Users nije imala niti jedan strani ključ. Provjerimo kako dohvatiti podatke o entitetu kada se ti podaci nalaze u više tablica.

```
1 public async Task<IEnumerable<Recipe>> GetRecipesAsync()
2 {
3     return await _context.Recipes
4         .Include(recipe => recipe.Category)
5         .Include(recipe => recipe.User)
6         .ToListAsync();
7 }
```

Isječak koda 2.22: Dohvaćanje svih recepata

U isječku koda 2.22 vidimo metodu za dohvat svih recepeta. Razlika od gornjih primjera je u tome što smo kod definiranja entiteta Recipe naveli da će on u sebi sadržavati druge entitete Category i User. Zbog toga moramo eksplicitno navesti da želimo da nam se u entitetu popune i ti podentiteti. To radimo pomoću metode Include(). Ako bismo razmišljali što se događa na bazi podataka, onda možemo zamisliti da pomoću metode Include() u zahtjevu tražimo da nam se osim stranog ključa dohvate i podaci iz tablice na koju strani ključ pokazuje. Ako bismo razmišljali u jeziku SQL, onda bi metoda Include() bila analogna naredbni Join. Primjetimo kako ovdje nismo koristili metodu AsNoTracking(). To znači da će nam se osim samih podataka pamtiti i podaci o izmjenama podataka. To nam

omogućuje da dohvaćene podatke kasnije mijenjamo. U sljedećem odlomku pokazat ćemo kako se mijenjaju podaci.

2.8 Uređivanje podataka na bazi

Uređivanje podataka je radnja koja u sebi uključuje dohvaćanje postojećih podataka, njihovo mijenjanje i spremanje promjena na bazi. U prvom korako se dohvaćeni podaci s baze spreme u odgovarajuće entitete. Zatim se u drugom koraku mijenjaju željeni atributi unutar instance entitneta. U trećem koraku se pozove metoda iz EF Core-a koja promjene na entitetima preslika u promjene na bazi i tamo ih izvrši.

```
1  public async Task UpdateUserAsync(UpdateUserRequest updateUserRequest,
       int userId)
2  {
3      var user = await _context.Users
4                  .SingleOrDefaultAsync(user => user.UserId == userId);
5
6      if(user != null)
7      {
8          if(!String.IsNullOrWhiteSpace(updateUserRequest.Username))
9          {
10             user.Username = updateUserRequest.Username;
11         }
12
13         if (!String.IsNullOrWhiteSpace(updateUserRequest.Name))
14         {
15             user.Name = updateUserRequest.Name;
16         }
17
18         if (!String.IsNullOrWhiteSpace(updateUserRequest.Surname))
19         {
20             user.Surname = updateUserRequest.Surname;
21         }
22
23         await _context.SaveChangesAsync();
24     };
25 }
```

Isječak koda 2.23: Izmjene na entitetu User

U isječku koda 2.23 nalazi se primjer metode za izmjene na entitetu User. Kao ulazne parametre uzima id korisnika kojeg želimo izmjeniti i varijablu tipa UpdateUserRequest, u kojoj se nalaze podaci koje na entitetu želimo izmjeniti. U liniji 3 dohvaćamo željenog korisnika s baze. Ako korisnik sa zadanim id-om ne postoji, u varijabli user bit će spremljena vrijednost null i ne ćemo uči u tijelo if naredbe. Ako pak korisnik s tim id-om postoji na

bazi, njegove vrijednosti će biti spremljene u varijabli user. Zatim ulazimo u tijelo if naredbe mijenjamo podatak po podatak ukoliko je potrebno. Na kraju if naredbe, pozovemo metode SaveChangesAsync(), koja naše promjene na entitetu šalje na bazu.

Ovo je bio primjer jednog načina kako mijenjati postojeći podatak. U ulaznim parametrima za vrijednosti koje ne želimo mijenjati pošaljemo vrijednost null. Drugi način bi bio da za podatke koje ne želimo mijenjati pošaljemo trenutne vrijednosti. Onda u metodi za ažuriranje podatka ne bismo trebali ići parmaterar po parametar i gledati koji je različit od null, nego bismo sve vrijednosti zamjenili sa onima koje su u ulaznim parametrima metode.

Primjećujemo da ovdje nismo mijenjali vrijednosti stranih ključeva. No, situacija je analogna situaciji kod dodavanja vrijednosti stranih ključeva. Možemo mijenjati na dva načina. Prvi način je da promjenimo samo vrijednost stranog ključa. Drugi način je da po vrijednosti stranog ključa dohvatimo cijeli vrijednost podentiteta i nju stavimo na mjesto vrijednosti podentiteta.

```
1  public class UpdateRatingRequest
2  {
3      public int UserId { get; set; }
4
5      public int RecipeId { get; set; }
6
7      public int Grade { get; set; }
8  }
```

Isječak koda 2.24: Definicija klase UpdateRatingRequest

```
1  public async Task UpdateRatingAsync(UpdateRatingRequest
       updateRatingRequest, int ratingId)
2  {
3      var rating = await _context.Ratings.SingleOrDefaultAsync(rating =>
       rating.RatingId == ratingId);
4
5      var recipe = await _context.Recipes.SingleOrDefaultAsync(recipe =>
       recipe.RecipeId == updateRatingRequest.RecipeId);
6
7      var user = await _context.Users.SingleOrDefaultAsync(user => user.
       UserId == updateRatingRequest.UserId);
8
9      if(rating != null && recipe != null && user != null)
10     {
11         rating.Recipe = recipe;
12         rating.User = user;
13         rating.Grade = updateRatingRequest.Grade;
14     }
15
```

```
16      await _context.SaveChangesAsync();
17  }
```

Isječak koda 2.25: Izmjene na entitetu Rating

U isječku koda 2.25 vidimo primjer izmjene vrijednosti iz tablice Rating. Primjećujemo da smo ovdje radili validaciju stranih ključeva koje dobimo kao ulazne parametre metode. Zatim smo pomoću tih parametara dobili odgovarajuće podatke s baze. Tek ako svi podaci prođu validaciju krećemo s ažuriranjem podatka na bazi. Na kraju metode, u liniji 16, zovemo metodu kojom se promjene na instanci entiteta šalju u bazu.

Trenutno smo prošli kroz tri osnovne radnje u radu s bazom podataka. To su dohvaćanje podataka, ažuriranje i zapisivanje novih vrijednosti. Preostalo nam je još vidjeti kako izbrisati podatke iz baze. Time ćemo se baviti u sljedećem poglavlju.

2.9 Brisanje podataka iz baze

Posljednja radnja koju trebamo proći jest brisanje podataka iz baze. Brisanje se izvodi u tri osnovna koraka. U prvom korako s baze dohvatimo podatak koji želimo obrisati i preslikamo ga u instancu entiteta. U drugom korako pozovemo metodu Remove(), kojoj kao parametar proslijedimo tu instancu entiteta. U trećem koraku, pomoću metode SaveChangesAsync(), promjene u kontekstu preslikamo na bazu. Provjerimo kako to izgleda na primjeru brisanja korisnika iz baze.

```
1  public async Task DeleteUserAsync(int userId)
2  {
3      var user = await _context.Users.SingleOrDefaultAsync(user => user.
       UserId == userId);
4      _context.Users.Remove(user);
5      await _context.SaveChangesAsync();
6  }
```

Isječak koda 2.26: Metoda za brisanje korisnika

U isječku koda 2.26 izbrisali smo korisnika iz baze. Id korisnika kojeg želimo izbrisati dobiven je iz ulaznog parametra metode. Iako se na prvi pogled čini kao jednostavna radnja, problem je u tome što postoji mogućnost da neka druga tablica u bazi sadrži id izbrisanog korisnika kao strani ključ. Ako neka tablica koristi izbrisani podatak kao strani ključ, prilikom poziva metode za spremanje promjena javit će nam se greška. Taj problem možemo riješiti na više načina. Jedan bi bio da prođemo po svim tablicama koje sadrže vrijednost izbrisanog podatka kao strani ključ i izbrišemo te podatke. No, sličnu stvar možemo napraviti i u kodu. Pogledajmo na primjeru brisanja recepta kako bi to izgledalo.

```
1  public async Task DeleteRecipeAsync(int recipeId)
```

```
2 {
3     var recipe = await _context.Recipes.Include(recipe => recipe.
    RecipeIngredients).Include(recipe => recipe.Ratings).
    SingleOrDefaultAsync(recipe => recipe.RecipeId == recipeId);
4
5     _context.Recipes.Remove(recipe);
6     await _context.SaveChangesAsync();
7 }
```

Isječak koda 2.27: Metoda za brisanje recepta

U isječku koda 2.27 vidimo primjer metode za brisanje recpeta. Promotrimo detaljnije liniju 3. Primjetimo da smo ovdje koristili metodu Include(). Pomoću te metode u varijablu recipe smo osim vrijednosti iz tablice Recipes dodali i vrijednosti iz vezne tablice RecipeIngredients. Na taj način ćemo pratiti i stanje u podentitetu. Sljedeće što radimo je brisanje u liniji 5. Pošto smo gore naveli da se promjene odnose i na podentitete, metoda Remove() prvo prođe po podentitetu i izbriše sve zapise koje za strani ključ imaju podatak koji želimo obrisati. Tek nakon što su svi podaci obrisani, kreće se sa brisanjem osnovnog entiteta. Na taj način osigurali smo se da nam metoda za spremanje promjena ne baci grešku. Kako entitet Recipe sadrži dva podentiteta, RecipeIngredient i Rating, kodom iz linije 3 osigurali smo se da sve vrijednosti stranih ključeva, koji se referenciraju na podatak koji želimo izbrisati, budu izbrisane također.

S ovime završavamo pregled četiri osnovne metode za upravljanje bazom podataka. Preostajem nam još testirati naš program. Za sada smo samo kreirali tablice po našim entitetima. No, te tablice su trenutno prazne. U sljedem poglavlju iskoristit ćemo metode koje smo ovdje definirali kako bismo napunili te tablice.

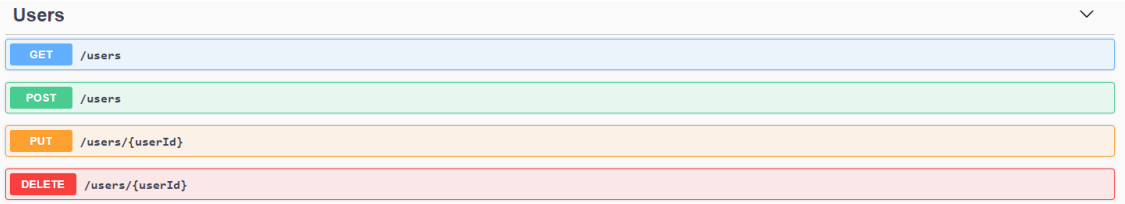
Poglavlje 3

Prikaz rada web kuharice

U drugom odjeljku prikazali smo moć alata Entity Framework Core. Prvo smo ga iskoristili kako bismo kreirali bazu podataka za našu aplikaciju. Nakon što smo kreirali tablice i veze između njih, u kodu smo definirali metode pomoću kojih ćemo upravljati bazom podataka. Još nam je preostalo pomoću tih metoda napuniti našu bazu podacima. Ići ćemo entitet po entitet i za svaki entitet demonstrirati rad naše aplikacije.

3.1 Entitet User

Prva tablica koju ćemo napuniti podacima biti će tablica koja odgovara entitetu User. U toj tablici spremat ćemo osnovne podatke o korisnicima naše aplikacije. Provjerimo koje smo metode vezane za entitet User definirali.



Slika 3.1: Prikaz metoda vezanih za entitet User

Na slici 3.1 vidimo prikaz metoda koje koristimo za upravljanje entitetom User. Get metoda koristi se za dohvat svih korisnika s baze. Pomoću metode s ključnom riječi post dodajemo novi zapis o korisniku na bazu. Put metoda koristi se kako bismo mijenjali podatke o određenom korisniku. Id korisnika kojeg ažuriramo dobijemo iz rute. Posljednja metode je metoda delete pomoću koje brišemo korisnika iz baze.

Na početku je tablica Users prazna. Koristeći metodu post dodajmo nove zapise u tablicu. Nakon što smo četiri puta pozvali metodu post, dobijemo četiri zapisa na bazi.

	UserId	Name	Surname	Username	CreationDate
1	1	Ivo	Ivić	ivo123	2020-12-28 19:01:49.0571166
2	2	Hrvoje	Horvat	hhorvat	2020-12-28 19:02:07.5751339
3	3	Ivana	Tomić	ivanaivana	2020-12-28 19:02:37.9326087
4	4	Petar	Perić	ZGperoZG	2020-12-28 19:03:20.5883642

Slika 3.2: Stanje tablice Users nakon četiri poziva metode post

Trenutno imamo četiri podatka u tablici. Recimo da jedan korisnik odluči promjeniti korisničko ime. Pozvala bi se metoda put sa novim podacima. Zatim neka drugi korisnik odluči da želi obrisati svoj račun. Tada bi pozvali metodu delete. Nakon svega toga, dolazi novi korisnik i želi se prijaviti u naš sustav. Za njega pozivamo metodu post. Nakon svih tih promjena, tablica Users bi imala sljedeći izgled:

	UserId	Name	Surname	Username	CreationDate
1	1	Ivo	Ivić	ivo123	2020-12-28 19:01:49.0571166
2	2	Hrvoje	Horvat	horvat1809	2020-12-28 19:02:07.5751339
3	3	Ivana	Tomić	ivanaivana	2020-12-28 19:02:37.9326087
4	5	Jurica	Jurić	djuro7	2020-12-28 19:12:13.7463504

Slika 3.3: Stanje tablice Users nakon gore navedenih poziva

Na slici 3.3 vidimo stanje u tablici Users nakon svih gornjih poziva. Preostaje nam još isprobati metodu get.

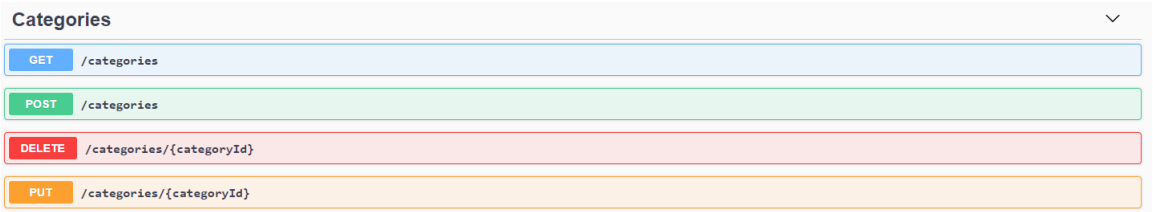
```
[
    {
        "name": "Ivo",
        "surname": "Ivić",
        "username": "ivo123",
        "creationDate": "2020-12-28T19:01:49.0571166"
    },
    {
        "name": "Hrvoje",
        "surname": "Horvat",
        "username": "horvat1809",
        "creationDate": "2020-12-28T19:02:07.5751339"
    },
    {
        "name": "Ivana",
        "surname": "Tomić",
        "username": "ivanaivana",
        "creationDate": "2020-12-28T19:02:37.9326087"
    },
    {
        "name": "Jurica",
        "surname": "Jurić",
        "username": "djuro7",
        "creationDate": "2020-12-28T19:12:13.7463504"
    }
]
```


Slika 3.4: Odgovor metode get users

Na slici 3.4 vidimo odgovor koji dobimo od metode get. Primjetimo da ovi podaci odgovaraju onima s tablice. Dakle, možemo zaključiti da sve metode vezane za entitet User rade očekivano. Sljedeći na redu nam je entitet Categories.

3.2 Entitet Category

Nakon što smo kreirali zapise u tablici Users, sljedeću punimo tablicu Categories. To je tablica u koju spremamo podatke o kategorijama recepata. Ponovno, kao i za entitet User, imamo četiri osnovne metode za upravljanje podacima. Get metoda će nam služiti za dohvaćanje svih kategorija, put za promjene na jednoj kategoriji, post za dodavanje nove kategorije i delete za brisanje određene kategorije.



Slika 3.5: Prikaz metoda vezanih za entitet Categories

Na slici 3.5 vidimo prikaz metoda koje ćemo koristiti za popunjavanje tablice Categories. Pozovimo prvo četiri puta metodu s ključnom riječi post da dobijemo četiri zapisa u bazi.

	CategoryId	CategoryName	CategoryDescription
1	1	Morska hrana	Riblja jela, jela s morskim plodovima i ostali specijaliteti.
2	2	Deserti	Slatka jela koja se poslužuju nakon glavnog.
3	3	Jela od mesa	Jela kod kojig je glavna namirnica meso.
4	4	Vegeterijanska hrana	Hrana prilagođena vegeterijanskoj kuhinji.

Slika 3.6: Stanje u tablici Categories nakon četiri poziva metode post

Isprobajmo sada i ostale metode vezane za taj entitet. To znači da ćemo jedan zapis promjeniti, jedan izbrisati i na kraju dohvatiti popis svih kategorija. Nakon što smo to napravili dobivamo sljedeće stanje u tablici:

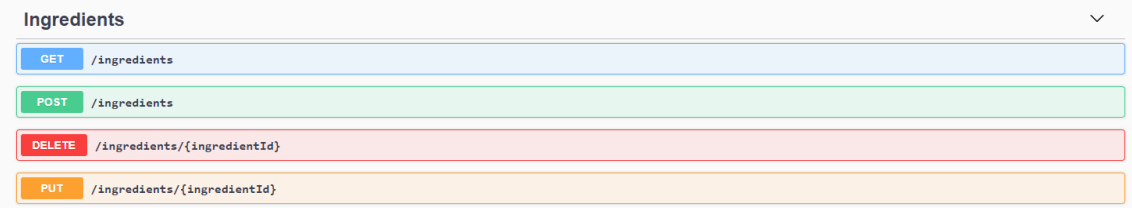
	CategoryId	CategoryName	CategoryDescription
1	1	Morska hrana	Riblja jela, jela s morskim plodovima i ostali specijaliteti.
2	2	Deserti i slastice	Slatka jela koja se poslužuju nakon glavnog.
3	4	Vegeterijanska hrana	Hrana prilagođena vegeterijanskoj kuhinji.

Slika 3.7: Stanje u tablici nakon poziva metoda delete i put

Na slici 3.7 vidimo konačno stanje u tablici. Vidimo da smo izbrisali zapis koji je imao id jednak tri, te da smo kategoriju Deserti preimenovali u Deserti i slastice. S time smo rješili tablicu Categories. Sljedeće na redu su nam namirnice.

3.3 Entitet Ingredient

Treću tablicu koju ćemo popuniti podacima bit će tablica Ingredients.U njoj ćemo zapisati informacije o namirnicama. To je ujedno i posljednja tablica koja u sebi ne sadrži strani ključ. Kao i u prethodnim slučajevima, imamo četiri osnovne metode za upravljanje tablicom Ingredients.



Slika 3.8: Prikaz metoda vezanih za entitet Ingredients

Na slici 3.8 vidimo te četiri metode. Ponovno ćemo prvo nekoliko pozvati metodu post da dodamo nove zapise u bazu.

	IngredientId	IngredientName	AveragePrice
1	1	Krumpir	6.00
2	2	Pastrva	60.00
3	3	Luk	4.50
4	4	Riza	2.24
5	5	Ulje	3.00
6	6	Maslinovo ulje	65.00

Slika 3.9: Stanje u tablici nakon šest poziva metode post

Na slici 3.9 vidimo stanje tablice nakon 6 poziva metode post. U svakom pozivu dodali smo novi zapis u tablicu. Sada koristimo metode put i delete da bismo izmjenili i obrisali podatke. Nakon toga, naša tablica ima sljedeće zapise:

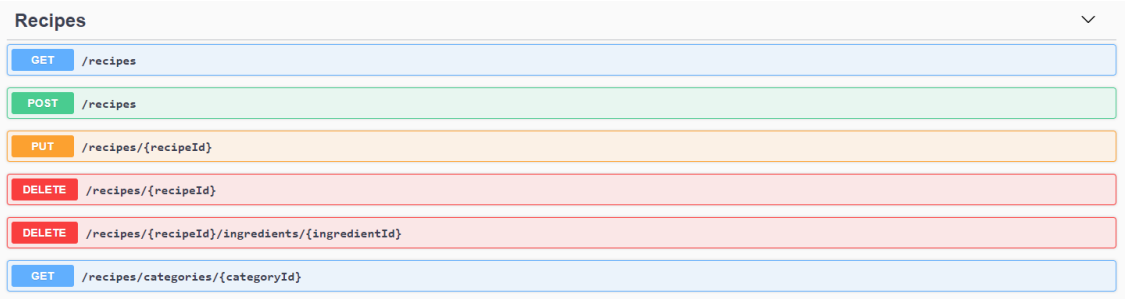
	IngredientId	IngredientName	AveragePrice
1	2	Pastrva	60.00
2	3	Luk	4.50
3	4	Riza	2.24
4	5	Ulje	3.00
5	6	Maslinovo ulje	80.00

Slika 3.10: Stanje u tablici nakon poziva metode put i delete

Na slici 3.10 primjećujemo da smo obrisali zapis s id-em jedan te maslinovom ulju povečali prosječnu cijenu na 80. Time smo završili s popunjavanjem tablice Ingredients. Sljedeći na redu je glavni entitet, entitet Recipes.

3.4 Entitet Recipe

Entitet Recipe je središnji entitet našeg programa. On sadrži podatke o receptima. Kao podentitete sadrži entitete Category i User, s kojima je povezan pomoću stranih ključeva CategoryId i UserId. Unutar ovog odjeljka obradit ćemo još jedan entitet koji je strogo vezan uz entitet Recipe. Radi se o entitet RecipeIngredient. To je pomoćni entitet koji će nam na bazi odgovarati veznoj tablici između tablice Ingredients i tablice Recipes. Razlog tomu je što između Recipe i Ingredient postoji Many-to-many veza. Zbog toga ćemo u našem programu u istom koraku odraditi kreiranje novog recepta i spajanje tog recepta sa svim potrebnim namirnicama iz tablice Ignredients.



Slika 3.11: Prikaz metoda vezanih uz entitet Recipe

Na slici 3.11 vidimo popis svih metoda vezanih za entitet Recipe. Imamo dvije get metode, jednu koja dohvaća sve recepte i jedna koja dohvaća sve recepte filtrirane po kategoriji. Zatim imamo jednu metodu s ključnom riječi post. Ta metoda služi za dodavanje

novog recepta. Metoda s ključnom riječi put služi nam kako bismo izmjenili podatke o određenom receptu. Tu se još nalaze i dvije metode s ključnom riječi delete. Jedna nam služi kako bismo obrisali jedan zapis o receptu, a druga da bismo obrisali vezu između namirnice i recepta, tj. da bismo maknuli određenu namirnicu iz recepta. Dodajmo prvo jedan zapis u tablice Recipes.

```
{
    "Title": "Pastrva sa žara",
    "Description": "Pastrvu ispecite na žaru. Dodajte luk po želji.",
    "UserId": 1,
    "CategoryId": 1,
    "Ingredients": [
        {
            "IngredientId": 2,
            "Amount": 0.5
        },
        {
            "IngredientId": 3,
            "Amount": 0.1
        },
        {
            "IngredientId": 6,
            "Amount": 0.1
        }
    ]
}
```

Slika 3.12: Sadržaj tijela u pozivu metode post

Na slici 3.12 vidimo sadržaj tijela u pozivu metode s ključnom riječi post. U njemu šaljemo ime jela, opis kako napraviti jelo, id korisnika koji kreira recept, id kategorije kojoj recept pripada te listu koju čine parovi id-a namirnice i količina namirnice koja je potrebna za taj recept. Sadržaj iz liste zapisat će se u veznu tablicu. Nakon što se metoda post uspješno izvršila, na bazi bismo trebali vidjeti zapise o tom receptu.

	RecipeId	Title	Description	UserId	CategoryId
1	1	Pastrva sa žara	Pastrvu ispecite na žaru. Dodajte luk po želji.	1	1

Slika 3.13: Sadržaj tablice Recipes

	RecipeIngredientId	RecipeId	IngredientId	Amount
1	1	1	2	0.50
2	2	1	3	0.10
3	3	1	6	0.10

Slika 3.14: Sadržaj tablice RecipeIngredient

Na slikama 3.13 i 3.14 vidimo sadržaje u tablicama Recipes i RecipeIgnredient nakon jednog poziva metode post. Vidimo da su se podaci o receptu uspješno spremili u tablicu Recipes, dok su se podaci o namirnicama za taj recept zapisali u tablicu RecipeIngredient. Možemo još isprobati metodu delete koja briše jednu namirnicu iz recepta. Možemo se odlučiti za namirnicu s id-em dva. Tada bi u rutu umjesto recipeId stavili jedan, jer želimo promjeniti recept id-a jedan, te umjesto ingredientId broj dva, jer želimo obrisati namirnicu s id-em dva. Ako bismo to izvršilo, tablica RecipeIngredients promjenila bi svoj sadržaj.

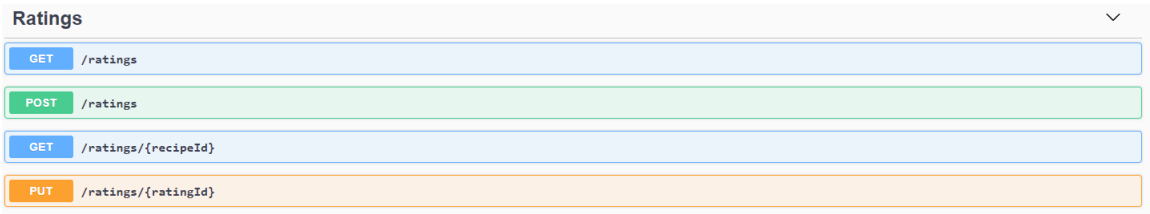
	RecipeIngredientId	RecipeId	IngredientId	Amount
1	2	1	3	0.10
2	3	1	6	0.10

Slika 3.15: Sadržaj tablice RecipeIngredient nakon brisanja jedne namirnice

Trenutno smo napunili sve tablice vezane za recepte i sadržaj recepata. Potrebno je napuniti još jednu tablicu, onu u koju će se spremati podaci o ocjenama recepata.

3.5 Entitet Rating

Posljednji za isprobati nam je ostao entitet Rating. To je entitet koji predstavlja jedan zapis o ocjeni recepta. On u sebi sprema podatke o ocjeni koju korisnik dodijeli receptu, id korisnika koji ocjenjuje i id recepta kojeg korisnik ocjenjuje. Pogledajmo koje sve metode vezane za entitet Rating imamo.



Slika 3.16: Prikaz metoda vezanih uz entitet Rating

Na slici 3.16 vidimo popis metoda vezanih uz entitet Rating. Primjetimo da imamo dvije metode get, pomoću kojih možemo dohvatiti sve ocjene ili samo jednu filtriranu po id-u. Imamo i jednu metodu s ključnom riječi put. Ona nam služi da bismo izmjenili podatke o jednoj ocjeni. Također, imamo metodu s ključnom rječi post. Ona nam služi da bismo dodali novi zapis o ocjeni.

Primjetimo da ovdje nemamo niti jednu metodu koja bi brisala ocjenu. To je zbog toga jer ne želimo da korisnik nakon što ocjeni recept tu ocjenu izbriše. No, to ne znači da ocjena ostaje zauvijek. Naime, ako bi netko obrisao recept, onda bi se obrisale i sve ocjene vezane za taj recept. Također, ako bi se obrisao određeni korisnik, tada bi se obrisale i sve ocjene koje je on dao.

Probajmo dodati nekoliko ocjena za recept s id-em jedan. To nam je trenutno jedini recept u bazi. No, imamo četiri različiti korisnika pa možemo dodati četiri različite ocijene.

	RatingId	RecipeId	UserId	Grade
1	1	1	1	4
2	2	1	2	3
3	3	1	3	1
4	4	1	5	2

Slika 3.17: Sadržaj tablice Ratings

Na slici 3.17 vidimo sadržaj tablice Ratings nakon što smo četiri puta ocijenili recept is id-em jedan. Probajmo dodati još jednu ocjenu za isti recept i istog korisnika. Ako bismo to učinili dobili bismo sljedeću poruku:

```
1 Cannot insert duplicate key row in object 'dbo.Ratings' with unique
    index 'IX_Ratings_UserId_RecipeId'. The duplicate key value is (5,
    1).
```

Isječak koda 3.1: Poruka o greški prilikom dodavanju duplog zapisa

Primjetimo da nam u poruci piše da je nemoguće dodati duplicirani zapis u tablicu Ratings. To je zbog toga što smo prilikom konfiguracije entiteta definirali da nam kominacija id-eva recepta i korisnika bude jedinstvena. To smo napravili zbog toga što ne želimo da nam isti korisnik više puta ocijeni isti recept.

S time smo prošli kroz sve funkcionalnosti našeg programa. Primjetimo da smo cijelo vrijeme upravljali bazom podataka isključivo kroz kod napisan u jeziku C#. Niti u jednom trenutku nismo pisali SQL naredbe za komunikaciju s bazom. Jedino što smo mi napravili je definirali da ćemo s bazom komunicirati preko paketa Entity Framework Core. Tako je EF Core postao posrednik između našeg programa i baze. U završnom odjeljku spomenut ćemo koje prednosti alat Entity Framework Core donosi prilikom izrade softvera i koje su mu eventualne mane. Također, spomenut ćemo i u kojem smjeru bismo mogli nastaviti sa izradom našeg softvera.

Poglavlje 4

Zaključak

U ovom odjeljku preostaje nam dati konačnu ocjenu alata, navesti u kojim segmentima nam je bio koristan, a u kojim segmentima nam je možda bio ograničavajući. Također, spomenut ćemo i moguće daljnje smjerove našeg rada.

4.1 Prednosti i mane alata Entity Framework Core

Prvo ćemo spomenuti stvari zbog kojih se sve više developera odlučuje na korištenje navedenog alata. Prva je ta što se cijeli razvoj softvera odrađuje u jeziku C#. Nije potrebno učiti jezik SQL da bismo mogli uspješno upravljati bazom podataka. Sva komunikacija s bazom ide preko alata koji se brine da kod koji mi napišemo bude razumljiv bazi podataka. Kako su svi naši upiti prema bazi unaprijed pripremljeni, mnogo je teže napraviti SQL injekciju i zbog toga je naš softver sigurniji.

Sljedeća pozitivna stvar je ta što sve modele pišemo u kodu samo jednom. To omogućuje jednostavnije i brže promjene u samim modelima, jer je promjenu potrebno odraditi samo na jednom mjestu. Pošto radimo s modelima, naš kod je objektno orijentiran. To znači da možemo koristiti sve mogućnosti objektnog programiranja, kao što je npr. nasljeđivanje modela. Također, alat umjesto nas odradi još neke stvari, kao što je npr. samo spajanje na bazu. Ne trebamo koristiti neki dodatan paket nego alatu proslijedimo potrebne parametre,a on se pobrine za uspješno spajanje.

No, prilikom rada s alatom treba obratiti pažnju na potencijalne probleme. Iako kod jednostavnih upita ne dolazi do problema s performansama, kod kompleksnih upita može doći do problema. Među njima je najpoznatiji ”N+1 problem”. Na primjeru naše baze, to bi bila tablica Ratings. U toj tablici, imamo stupac UserId, koji predstavlja strani ključ na tablicu Users. Problem bi nastao kada bi mi dohvatili svaki zapis u tablici Ratings, i onda za svaki

taj zapis tražili odgovarajućeg korisnika iz tablice Users. Tako bi za svaki zapis u tablici Ratings imali novi upit na bazu pomoću kojeg bismo dohvatili odgovarajućeg korisnika. Taj problem rješavamo sa metodom Include(), u kojoj već kod inicijalnog dohvaćanja podataka iz tablice Ratings definiramo da za svaki zapis želimo njegov odgovarajući podatak iz tablice Users. Ako bismo tako kreirali naš upit, imali bismo samo jedan upit na bazu u kojem bi bili dohvaćeni svi potrebni podaci.

Također, iako sam alat od nas ne zahtjeva znanje SQL-a, preporuča se da osoba koja koristi alat bude barem malo upućena u SQL. To posebno dolazi do izražaja kod kreiranja složenijih upita. Osoba koja dobro poznaje SQL može unaprijed predvidjeti potencijalne probleme na bazi kod kreiranja upita.

Spomenimo ovdje još jednu stvar. Primjetimo da smo u našem programu koristili code first pristup. To znači da su naše tablice kreirane pomoću alata Entity Framework Core. Nakon što smo ih kreirali, naš razvoj može ići u dva različita smjera. Jedan smjer je onaj koji smo obradili u radu, a to je da smo nakon kreiranja tablica nastavili koristiti navedeni alat. No, alat ima svoja ograničenja. Npr, nije moguće upravljati svim funkcionalnostima koje baza podržava. Primjer toga bili bi okidači (triggers). Da bismo dodali okidače, potrebno je koristiti konvencionalne metode(jezik SQL). To je drugi smjer u kojem možemo razvijati našu aplikaciju. Nakon što smo kreirali tablice pomoće Entity Frameworka, sve ostale stvari odradimo pomoću SQL-a.

Zaključak svega bi bio da je alat veoma koristan, ali treba biti oprezan kod korištenja. Iako on odradi velik dio posla za nas, moramo biti oprezni kod kreiranja upita, jer postoji mogućnost da nepažljivim korištenjem možemo ozbiljno narušiti brzinu samog programa. Stoga, prije korištenja alata, treba izdvojiti određeno vrijeme kako bi se upoznali sa mogućnostima tog alata. Također, moramo biti svjesni ograničenja koje alat ima. Ako u nekom trenutku primjetimo da trebamo koristiti neku mogućnost baze koju alat ne podržava, prebacujemo se na SQL i u jeziku SQL definiramo stvari koje su nam potrebne.

Na kraju nam je preostalo spomenuti smjer u kojem bi se ovaj rad mogao nastaviti.

4.2 Smjer daljnjeg rada

U našem radu koristili smo osnovne funkcionalnosti alata. Pokazali smo kako kreirati tablice u bazi i kako ih napuniti pomoću jednostavnih upita. Jako malo pažnje smo posvetili optimizaciji upita i usporedbi performansi upita kreiranih preko alata i odgovarajućih upita pomoću SQL-a. Jedan smjer u kojem bi se rad mogao nastaviti bilo bi kreiranje analognih upita pomoću SQL-a i usporedba brzine svakog upita. Naravno, kod jednostavnijih upita

bi razlika najvjerojatnije bila zanemariva. Zbog toga bi prije toga trebalo kreirati kompleksnije upite te na temelju tih upita i njihovih brzina usporediti brzine između dva pristupa.

Drugi potencijalni smjer nastavka rada bilo bi detaljnije razmatranje samog alata. Zanimljivo bi bilo detaljnije proučiti klasu DBContext i ostale klase koje smo koristili prilikom izrada tablica. Mi smo do sad samo koristili te klase, bez da smo detaljnije znali zbog čega ih koristimo i što se u našem programu dogodi prilikom instanciranja tih klasa. Kada bismo znali pojedinosti o svakoj klasi alata, vjerujem da bismo mogli ranije otkriti potencijalne probleme u korištenju alata.

Bibliografija

[1] *Overview of Entity Framework Core - EF Core*, https://docs.microsoft.com/en-us/ef/core/, pristupljeno prosinac, 2020.

[2] *The pros and cons of Object Relational Mapping*, https://centralblue.co.uk/blog/2019/01/the-pros-and-cons-of-object-relational-mapping-orm, pristupljeno siječanj, 2021.

[3] H. Schwichtenberg, *Modern Data Access with Entity Framework Core: Database Programming Techniques for .NET, .NET Core, UWP, and Xamarin with C*, Apress, 2018.

[4] J. Smith, *Entity Framework Core in Action*, Manning Publications Co., 2018.

Sažetak

U ovom radu bavimo se aplikacijskim okvirom Entity Framework Core. Na početku objašnjavamo osnovne mogućnosti alata. Cilj nam je kreirati nove tablice u bazi. Prvo u kodu definiramo klase koje će predstavljati tablice na bazi. Zatim dodatno konfiguriramo svaki atribut unutar klase. Nakon toga, na temelju tih klasa i konfiguracija, izgeneriramo tablice u bazi. Nakon što su tablice kreirane, definiramo četiri osnovne metode za upravljanje podacima u tablici: umetanje novog zapisa, dohvaćanje postojećih zapisa, izmjena određenog zapisa i brisanje zapisa sa baze. Nakon toga, prikazujemo funkcionalnost programa koji koristi te četiri osnovne metode. Na kraju, prolazimo kroz prednosti i mane alata Entity Framework Core i navodimo slučajeve u kojima je korisno upotrebljavati navedeni alat.

Summary

This paper is about Entity Framework Core framework. At the beginning, we are describing basic functionalities of the tool. Our goal is to create new tables in the database. First, we define classes in code which will represent tables in database. Then we additionally configure every attribute in the class. After that, based on those classes and configurations, we generate tables in database. Once the tables are created, we define four basic methods for managing data inside the table: inserting new data, getting existing data, updating the data and deleting the data from database. After that, we present the functionality of the program, which uses those four methods. At the end, we go through pros and cons of the Entity Framework Core tool and define cases, in which it is handy to use the tool.

Životopis

Rođen sam 13. ožujka 1995. godine u Zagrebu. Pohađao sam Osnovnu školu August Cesarec u Krapini, nakon koje upisujem smjer prirodoslovno-matematička gimnazija u Srednjoj školi Krapina. Godine 2013. upisujem Preddiplomski sveučilišni studij matematike na Matematičkom odsjeku Prirodoslovno-matematičkog fakulteta u Zagrebu. Preddiplomski studij završavam 2018. godine te iste godine upisujem diplomski studij Računarstvo i matematika, također na Prirodoslavno-matematičkom fakultetu u Zagrebu. 2019. godine sudjelujem na Infobipovom Summer Internship programu koji traje deset tjedana. Nakon isteka programa, zapošljavam se u Infobipu, u kojem sam trenutno zaposlen kao student, na poziciji Software Engineering Interna.