MODELIRANJE I IZGRADNJA BAZE PODATAKA ZA WEB APLIKACIJU

Licul, Masimo

VELEUČILIŠTE U RIJECI

Masimo Licul

MODELIRANJE I IZGRADNJA BAZE PODATAKA ZA WEB APLIKACIJU

Završni rad

Rijeka, 2022.

VELEUČILIŠTE U RIJECI

Poslovni odjel

Preddiplomski stručni studij Informatika

MODELIRANJE I IZGRADNJA BAZE PODATAKA ZA WEB APLIKACIJU

Završni rad

MENTOR

Dr. sc. Ida Panev, v. pred.

STUDENT

Masimo Licul

MBS: 2422000054/19

Rijeka, svibanj 2022.

SAŽETAK

Cilj prvog dijela rada je predstaviti ključne aspekte projektiranja relacijske baze podataka. Također objasniti .NET razvojni okvir kao moderno okruženje za razvijanje raznolikih vrsta aplikacija, te njegovih komponenti i strukture programiranja. Cilj je i prikazati način na koji API zaprima i odgovara na različite korisničke zahtjeve. Biti će objašnjeno i funkcioniranje HTTP protokola kao jednog od temelja weba. U drugom dijelu rada biti će prikazana implementacija tih tehnologija na primjeru izrade API-a za praćenje radnih sati zaposlenika.

Ključne riječi: API, .NET, baza podataka, web, HTTP

SADRŽAJ:

1. Uvod

Svrha ovog rada je izrada baze podataka i prikaz mogućnosti .NET razvojnog okvira u kombinaciji sa ostalim tehnologijama kao što su EF Core i raznim programskim strukturama. Cilj je da se prikaže rješenje za jedan poslovni problem i jednostavnost korištenja web aplikacije kako bi se taj poslovni problem riješio te povećala efikasnost poslovnog okruženja.

U radu će biti prikazana izrada API-a, za aplikaciju koja će unutar poduzeća bilježiti radne sate zaposlenika. API (enl. Application Programming Interface) je sučelje za računala ili aplikacije koje omogućava međusobnu komunikaciju, odnosno razmjenu podataka i funkcionalnosti. Pokazati će se i izrada relacijske baze podataka za API „code first[1]“ pristupom. API je pisan u slojevima (eng. layers) kako bi se postigla veća skalabilnost sustava i olakšalo njegovo održavanje. Koristit će se i Mediator[2] programska arhitektura. API će implementirati CRUD funkcionalnosti (engl. Create, Read, Update, Delete) koje će omogućavati upisivanje, čitanje, ažuriranje i brisanje radnih sati nakon što se korisnik autentificira (Hoffman, 2021.).

ASP.NET je jedna od komponenti platforme .NET 6 koja donosi razvojne okvire za različita područja. Konkretno ASP.NET koristi se za web razvoj. Alat je razvijen od strane Microsoft-a i sastavni je dio svakog Windows operacijskog sustava.

Rad je podijeljen u četiri dijela. U prvom dijelu će se proći kroz osnove projektiranja relacijske baze podataka te alata koji su korišteni u tu svrhu. U drugome dijelu detaljno će se prikazati mogućnosti ASP.NET alata te dodatne komponente koje su potrebne u izradi API-a. U trećem dijelu biti će prikazane funkcionalnosti REST-a i HTTP-a koje su ključne kako bi cijeli sustav funkcionirao. U posljednjem, četvrtom dijelu bit će opisan sam proces izrade baze podataka za web aplikaciju koja će imati svrhu bilježenja radnih sati zaposlenika.

[1] Code First – Pristup u kojemu se relacije i konfiguracije zapisuju prvo u kodu te onda preslikavaju unutar baze podataka.
[2] Mediator – programska struktura, služi za posrednu komunikaciju među objektima.

2. Projektiranje baze podataka

Relacijska baza podataka je skup podataka koji su organizirani u međusobno povezane tablice. U suvremenim sustavima, baza podataka je centralni dio oko kojeg se dalje izgrađuje i na čemu se temelji neki sustav, stoga je ključno njezino kvalitetno projektiranje. Baza podataka omogućuje pristup podacima s jednog ili više računala koji ovisno o dopuštenjima mogu čitati, brisati, izmjenjivati i upisivati podatke. Ako se radi o web okruženju, može se reći da svako od tih računala komunicira s bazom preko nekog API-a, dok sustav koji se brine o samim podacima i njihovom posluživanju nazivamo DBMS (engl. Database, Managment, System). Kako bi DBMS mogao komunicirati sa samom bazom podataka koristi se SQL (engl. Structured Query Language) koji je standardni programski jezik za relacijske baze podataka. Pomoću SQL naredbi kreiraju se CRUD operacije (Magner, 2010.).

2.1. Relacijska baza podataka

Postoji velik broj različitih tipova baza podataka, no u ovom konkretnom slučaju koristit će se relacijska baza podataka zbog svoje jednostavnosti, fleksibilnosti i mogućnosti normalizacije koja smanjuje redundanciju podataka. Takva vrsta baze podataka temelji se na četiri glavna načela koja se kratko nazivaju ACID. A predstavlja „Atomicitet“ (engl. Atomicity), što znači da svaka operacija nad podacima može imati samo dva ishoda: ili uspjeh ili grešku. C predstavlja „Konzistentnost“ (eng. Consistency). Ako se izvršavaju operacije nad podacima, mora se sačuvati njihovo stanje prije i poslije operacije. I predstavlja „Izolaciju“ (engl. Isolation). Ako postoje dvije paralelne istovremene radnje nad podacima, one ne bi trebale biti vidljive jedna drugoj. D predstavlja „Izdržljivost“ (engl. Durability). Podrazumijeva da nakon odrađene operacije nad bazom promjene ostanu trajne (Ian, 2016.).

Unutar relacijske baze podataka podaci se grupiraju u tablice. Tablice unutar relacijske baze podataka nazivamo relacijama. Kako bi izbjegli informacijske anomalije i dupliciranje, odnosno redundanciju podataka, ali i ubrzali rad same baze podataka, potrebno je voditi brigu o normalizaciji relacijskog modela. Normalizacija je u prvom redu potrebna zato jer se njome izbjegavaju teškoće koje bi nastupile kad bismo radili s nenormaliziranim podacima. Normalizacija je korisna i zato jer se njome naknadno otkrivaju i ispravljaju pogreške u

oblikovanju entiteta, veza i atributa. Od normalizacije se može odustati samo u nekim rijetkim situacijama (Magner, 2010.).

Normalizacija je proces organiziranja podataka unutar baze podataka. Taj proces uključuje uspostavljanje veza među relacijama poštujući određena pravila, s ciljem da baza podataka postane više fleksibilna, te sa manje nekonzistentnih i redundantnih podataka (Microsoft docs, 2022.).

Relacija unutar relacijske baze podataka sastoji se od stupaca koji predstavljaju atribute. Atribut sadrži opisne karakteristike retka, te ima svoje ime po kojem ga razlikujemo od ostalih u istoj relaciji. Dvije relacije mogu imati atribute s istim imenom, no tada se podrazumijeva da su to zapravo atributi s istim značenjem. Atribut unutar relacije koji je od veće važnosti nazivamo primarni ključ, takvi atributi služe za identifikaciju zapisa. Vanjski ključevi su primarni ključevi koji se ne nalaze u svojoj izvornoj relaciji, te oni služe za povezivanje dviju tablica. Vrijednosti jednog atributa su podaci iste vrste. Definirana vrsta dopuštenih vrijednosti za atribut zove se domena atributa. Vrijednost atributa može se ponavljati (osim ako atribut nije u svojstvu primarnog ključa čija vrijednost mora biti jedinstvena) i mora biti jednostavna. U rijetkim situacijama moguće je da vrijednost atributa ne mora biti upisana. Jedan redak relacije predstavlja jedan primjerak entiteta. Svaka relacija unutar relacijske baze podataka mora imati svoje posebno ime kako bi je razlikovali od ostalih (Magner, 2010.)**.**

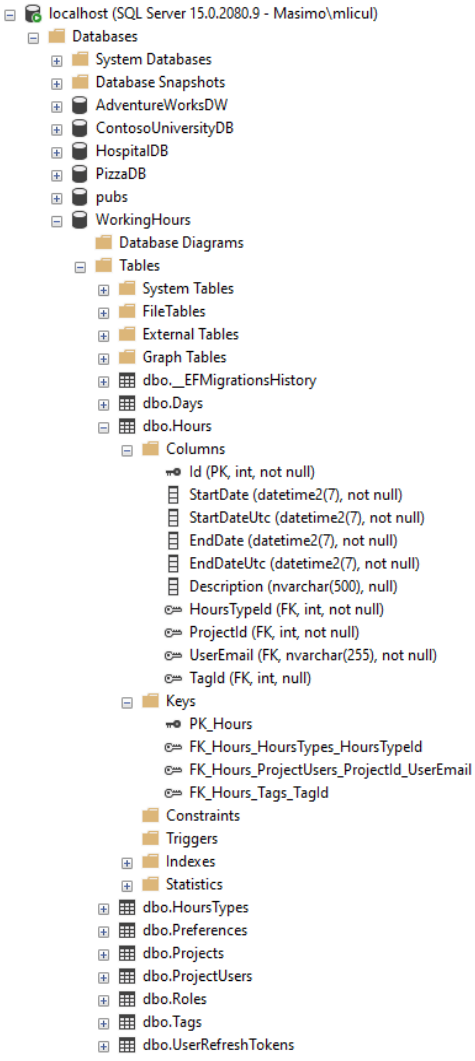
Veze među relacijama su ključne kod određivanja kako će određeni dijelovi aplikacije funkcionirati, odnosno kakvu će interakciju imati međusobno povezane relacije. Postoje tri kardinalnosti veza: jedan-prema-jedan povezuje zapis u tablici s nula ili jednim zapisom u drugoj tablici, jedan-prema-više povezuje jedan zapis u tablici s jednim ili više zapisa u drugoj tablici te veza više-prema-više povezuje više zapisa u tablici s jednim ili više zapisa u drugoj tablici.

2.2. Microsoft SQL Server Management Studio 18

API koji je izrađen u sklopu ovog rada koristiti će se lokalnom bazom podataka. Sustav za upravljanje bazom podataka koji će se koristiti je Microsoft SQL Server Management Studio 18. On predstavlja sustav koji omogućuje korisnicima ne samo prikaz baze podataka, već i

njeno upravljanje i podešavanje konfiguracija. Prva verzija izdana je 2005. godine s namjerom da se korisnicima Windows platforme olakša korištenje baze podataka (Microsoft docs, 2022.).

Slika 1 - Struktura WorkingHours baze podataka

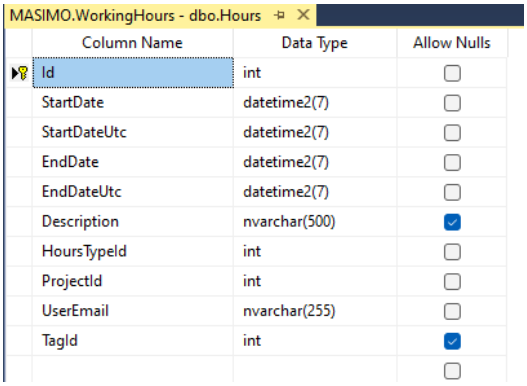


Izvor: Autor

Na slici 1 prikazana je struktura baze podataka WorkingHours koja je izrađena u sklopu ovog rada. Prikazani alat omogućava jednostavan pregled svih detalja vezanih uz samu bazu podataka, za svaku tablicu mogu se vidjeti njena svojstva kao što su imena tablice, ključevi, restrikcije, okidači, indeksi i statistika. Alat ima mogućnosti promjene postavki za svaki segment baze podataka. Na slici 2 prikazana je mogućnost promjene svojstva tablice. Pokreće

se funkcija Design te zatim se za svaki atribut unutar tablice može promijeniti njegovo ime, njegova vrsta podataka i dali ima dopuštenje da bude prazan.

Slika 2 - Promjena svojstva tablice



Izvor: Autor

3. Razvojni okvir ASP.NET Core

ASP.NET je programski okvir razvijen od strane Microsoft-a koji omogućava kreiranje web aplikacija i web API-a na .NET platformi. ASP.NET temelji se na c# programskom jeziku, no može se koristiti i Visual Basic programski jezik. .NET platforma omogućuje razvoj aplikacija za široki spektar namjena. Na slici 3 prikazana su područja u kojima se mogu razvijati .NET aplikacije. Važno je primijetiti kako sva ta područja dijele zajedničku infrastrukturu, razvojna okruženja i usluge. (Chand, M., 2022.).

Slika 3 - .NET 6 infrastruktura

Izvor: https://www.c-sharpcorner.com/article/what-is-new-in-net-6-0/

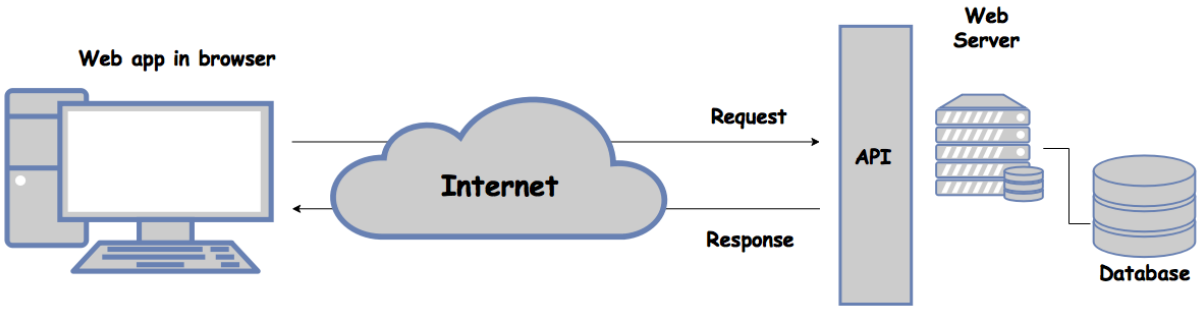
(3.5.2022)

Trenutna verzija .NET platforme je verzija 6 koja je postala dostupna javnosti u studenom 2021. godine. .NET 6 ne samo da je poboljšana verzija .NET 5 nego dodaje i mnogo novih funkcionalnosti koje pomažu programerima u razvoju i olakšavaju sam razvoj software-a. Prije verzije .NET 6 postojale su dvije verzije, .NET Framework i .NET Core Framework. Razlika je u tome što je .NET Core Framework bio otvoreni kod i funkcionirao je na platformama Windows, Linux i Mac. Sada sve to spada pod jednu platformu sa nazivom .NET. Neke od novih funkcionalnosti uključuju mogućnost Vrućeg ponovnog pokretanja (engl. Hot Reload) koja omogućuje programerima da izmjenjuju izvorni kod dok se aplikacija izvodi, bez potrebe da se ona ispočetka pokreće (Chand, 2022.).

3.1. Web API

Aplikacijsko programsko sučelje (engl. Application Programming Interface) je programski koncept koji je posrednik između klijentskih zahtjeva i samog web servera. API zaprima HTTP zahtjeve putem krajnjih točaka sustava (engl. Endpoints) i ispunjava ih putem kontrolera te vraća odgovor klijentu najčešće u JSON ili XML formatu. Na slici 4 može se vidjeti takav proces, gdje klijentsko računalo putem interneta pošalje zahtjev na neki API gdje se zahtjev obrađuje te vraća odgovor klijentu putem interneta (Hoffman, 2021.).

Slika 4 - Web API



Izvor: https://tremitch504.medium.com/rest-api-vs-soap-vs-web-api-4f8b535c01a0 (3.5.2022.)

ASP.NET Web API, kao što je prikazano u ovome radu, predstavlja jedan vrlo fleksibilan način gradnje servisa baziranih na HTTP tehnologiji, prvenstveno zbog toga što se tom sučelju može pristupiti sa svih platformi koje imaju pristup internetu. Ključna razlika između API-a i web aplikacije je ta da web aplikacija tipično vraća html prikaz, dok API šalje samo podatke. API je zato pogodan za operacije kao što su AJAX [3]što omogućava stvaranje veoma dinamične i responzivne aplikacije koja nema potrebe za stalnim osvježavanjem stranice, već se izmjenjuju samo pojedini dijelovi DOM[4]-a. (Hoffman, 2021.)

[3] AJAX – (engl. Asynchronous JavaScript And XML) skup razvojnih tehnika, omogućavaju asinkrono ažuriranje web stranica.
[4] DOM – (engl. Document Object Model) struktura HTML dokumenta.

U određenoj aplikaciji moguće je imati više API-a koji dohvaćaju resurse, odnosno podatke sa različitih web servera. Na primjer, ako se želi koristiti meteorološke funkcionalnosti sa druge web stranice, pomoću vlastite web stranice potrebno je uspostaviti komunikaciju s tim serverom putem HTTP protokola te zaprimljene podatke obraditi i poslužiti.

3.1.1. Krajnje točke

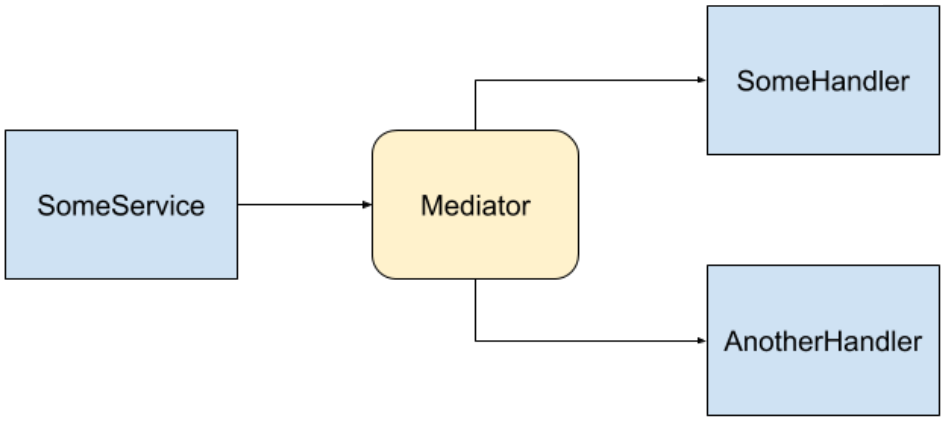
Krajnje točke (engl. Endpoints) definiraju koje resurse je moguće dohvatiti preko kakvog HTTP zahtjeva i kakve odgovore očekivati. Krajnje točke sadrže poveznicu resursa na nekom serveru. Kada se neka krajnja točka aktivira, pošalje se zahtjev na taj resurs. Zahtjev sa sobom nosi opis samog zahtjeva najčešće u XML ili JSON formatu. API može biti javna ili zaštićena sa tokenom pristupa[5] (engl. Access token). S obzirom da krajnje točke moraju biti statične, ali i dalje biti dostupne za ažuriranja, potrebno je uz zahtjev definirati na koju verziju API-a ciljamo kako bi izbjegli greške. (Smartbear, 2021.)

3.2. Mediatr programska struktura

Mediatr programska struktura unutar .NET Core API-a služi se „Mediator“ programskom strukturom i on inkapsulira objekt i način na koji on komunicira sa okolinom. Takva struktura ima mogućnost izmjene redoslijeda izvršavanja programa. „Mediator“ programska struktura djeluje tako da, umjesto da dva objekta ovise jedan o drugome, oni komuniciraju pomoću „Mediatora“. Kada objekt treba komunicirati s drugim objektom on prenosi poruku posredniku „Mediatoru“. Posrednik zatim prenosi poruku svakom primatelju u obliku koji mu je razumljiv kao što je prikazano na slici 5.

[5] Token pristupa – token koji korisnik dobiva nakon uspješne autorizacije, koristi se kada korisnik želi dohvatiti zaštićeni resurs

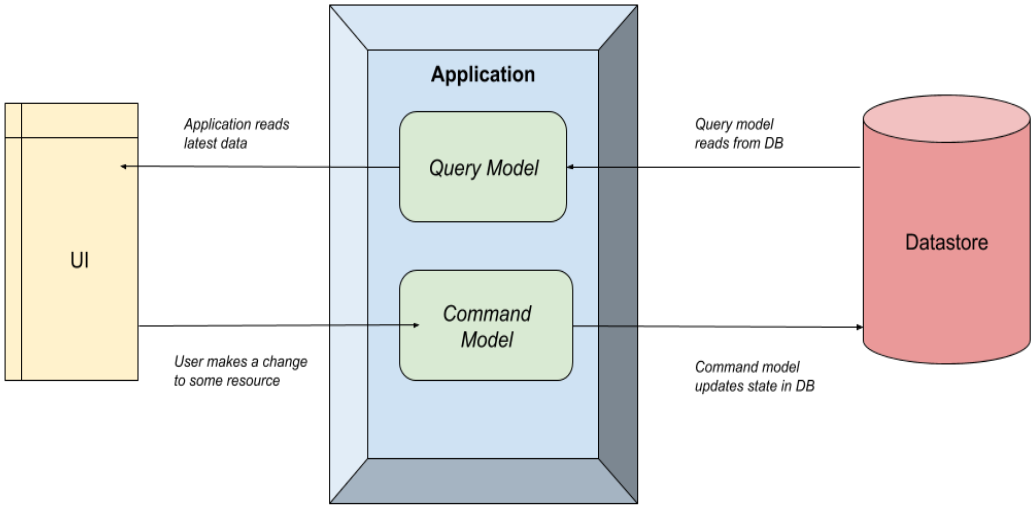
Slika 5 - Mediator programska struktura



Izvor: https://code-maze.com/cqrs-mediatr-in-aspnet-core/ (3.5.2022.)

Mediatr se oslanja i na CQRS (engl. Command Query Responsibility Segregation). To je struktura koja ima cilj da nadolazeće HTTP zahtjeve razdvoji na „Query“ one što samo čitaju iz baze podataka bez da vrše promjene, i na „Command“ koji mijenjaju podatke u bazi. Na slici 6. prikazana je CQRS programska struktura aplikacije unutr koje se zahtjevi dijele na Command i Query te se dalje obrađuju prema zadanom modelu (CQRS and MediatR in ASP.NET Core, 2021.).

Slika 6 - CQRS programska struktura



Izvor : https://code-maze.com/wp-content/uploads/2020/09/CQRS-Diagram.png (3.5.2022.)

Dakle kombinacijom ove dvije strukture dobiva se jedan moćan alat koji služi za razdvajanje procesa. Implementacija takve strukture donosi velike prednosti i gotovo je nužna za izgradnju većih poslovnih API-a. API koji je razvijan na taj način je jednostavniji za održavanje i nadograđivanje, a može se reći i da je lakše razumljiv s obzirom da su dijelovi razdvojeni na manje cjeline. (CQRS and MediatR in ASP.NET Core, 2021.)

Kako bi se demonstrirala funkcionalnost, za primjer će se kreirati GET HTTP zahtjev GetAllProjects. Dakle, kada zahtjev dođe do kontrolera, on se tu ne izvršava, nego preusmjeruje na klasu GetAllProjectsQuery, kao što je prikazano na slici 7.

Slika 7 - ProjectController klasa

```
namespace WorkingHours.API.Controllers
{
    [Route("api/[controller]")]
    [ApiController]
    1 reference
    public class ProjectController : ControllerBase
    {
        private readonly IMediator _mediator;
        0 references
        public ProjectController(IMediator mediator) => _mediator = mediator;

        // GET: api/<ProjectController>
        [HttpGet]
        [Route("all")]
        0 references
        public async Task<IActionResult> GetAllProjects() => Ok(await _mediator.Send(new GetAllProjectQuery()));
```

Izvor: Autor

Unutar GetAllProjectsQuery stvara se zahtjev u 6. liniji koda pomoću „Irequest“ te se definira što se u konačnici želi vratiti iz baze podataka. U ovome slučaju je to lista „ViewModela“ entiteta „Project“. GetAllProjectsQuery klasa prikazana je na slici 8.

Slika 8 - GetAllProjectQuery

GetAllProjectQuery.cs

WorkingHours.Query — WorkingHours.Query.Projec

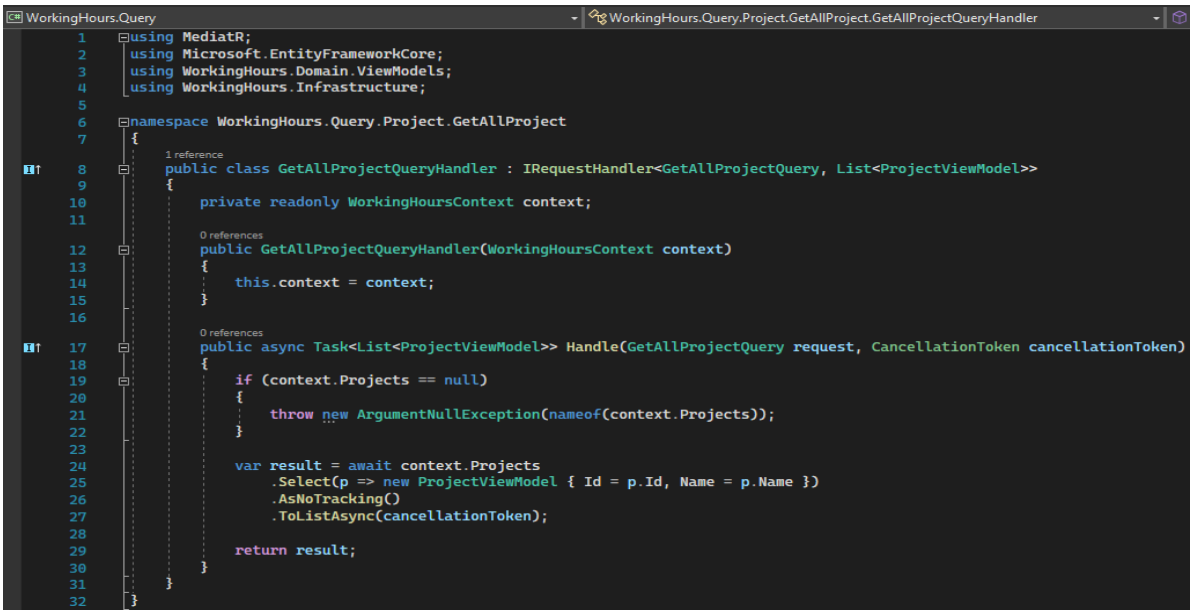
```
1   using MediatR;
2   using WorkingHours.Domain.ViewModels;
3
4   namespace WorkingHours.Query.Project.GetAllProject
5   {
        3 references
6       public class GetAllProjectQuery : IRequest<List<ProjectViewModel>>
7       {
8       }
9   }
10
```

Izvor: Autor

Posljednji korak je dakle izvršiti sam zahtjev. To se odrađuje u klasi GetAllProjectQueryHandler kao što je prikazano na slici 9. Tu je sada „IRequestHandler“ u koji je potrebno definirati izvorni zahtjev koji se pošalje kontrolerom i opet vrstu podatka koju

se očekuje. Implementira se asinkrono sučelje koje izvodi pretragu nad bazom i vraća podatke u obliku liste entiteta.

Slika 9 - GetAllProjectQueryHandler klasa

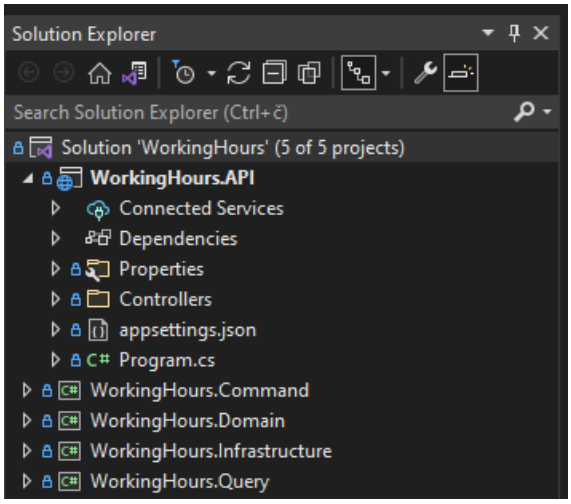


Izvor: Autor

3.3. Slojevita arhitektura

Kako bi se izbjeglo gomilanje podataka i funkcionalnosti u jednome dijelu aplikacije, jedno od rješenja je podjela u slojeve. Može se koristiti proizvoljan broj slojeva. Web API izrađen u sklopu ovog rada podijeljen je u četiri sloja, a to su: „Domain“, „Infrastructure“, „Query“ i „Command“, kao što je prikazano na slici 10. Cilj svakog sloja je da ima točnu svrhu i funkcionalnost. Pošto jedan sloj s drugim ne može direktno komunicirati, između njih je potrebno imati poveznice (engl. Reference). Unutar sloja Domain bilježe se entiteti koji će biti zapisani unutar baze podataka kao i razni modeli koji će formatirati odgovore koje zaprima korisnik. Unutar sloja Infrastructure nalaze se konfiguracije veza među entitetima, migracije, context klasa i servisi. Unutar slojeva Command i Query nalaze se metode koje obrađuju i odgovaraju na korisničke zahtjeve.

Slika 10 - Struktura API-a



Izvor: Autor

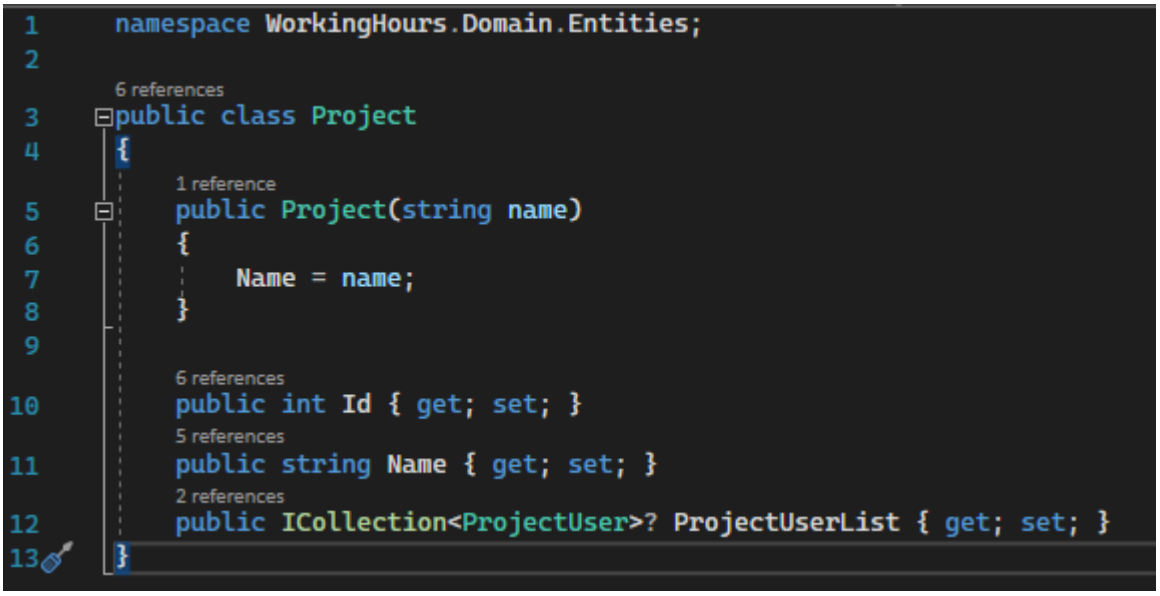
3.4. Razvojni okvir EF Core

Razvojni okvir EF Core je dio Microsoft-ovog ADO.NET paketa i služi za objektno relacijsko mapiranje (ORM). Alat je početno izdan 2016. godine s namjerom da olakša i ubrza razvoj softvera, na način da služi kao posrednik između objekata i softverskog koda. Naziv je dobio slično kao i .NET Core zato jer su imali istu namjeru: da ti alati funkcioniraju na različitim platformama kao što su Windows, MacOs i Linux. EF Core ORM služi za interakciju softverskog koda s nekom bazom podataka ili lokalnom pohranom. Može se koristiti na dva načina. Prvi način je da iz koda generira bazu podataka (engl. Code First), dok je drugi način da iz baze podataka generira modele koji će biti potrebni za manipulaciju podacima (engl. Data First). (Microsoft docs, 2021.)

3.4.1. Kreiranje entiteta

Entiteti se kreiraju na način da se stvori nova javna klasa i unutar nje definiraju atributi koje bi ta klasa trebala sadržavati. Može se reći da svaki entitet zapravo predstavlja jednu relaciju unutar baze podataka, dok atributi predstavljaju stupce unutar tih relacija.

Slika 11 - Project klasa



Izvor: Autor

Na slici 11 prikazan je primjer jednostavne klase Project u kojoj će se spremati samo imena projekata. Tu je i atribut ProjectUserList koji služi za uspostavljanje veze između entiteta. Pri kreiranju relacija unutar baze podataka kao ime relacije uzima se ime klase određenog entiteta, dok ostali članovi te klase predstavljaju atribute koji imaju točno određen tip podatka.

3.4.2. Konfiguracija veza među entitetima

Za postavljanje veza između entiteta koristit će se Fluent API koji je dio EF Core razvojnog okvira. Fluent API bazira se na „Fluent Interface“ programskoj arhitekturi koja omogućuje da se konfiguracije pišu konkatenacijom metoda. Pomoću Fluent API moguće je postaviti konfiguraciju modela, entiteta i imovine. (Entity Framework Tutorial, 2020.)

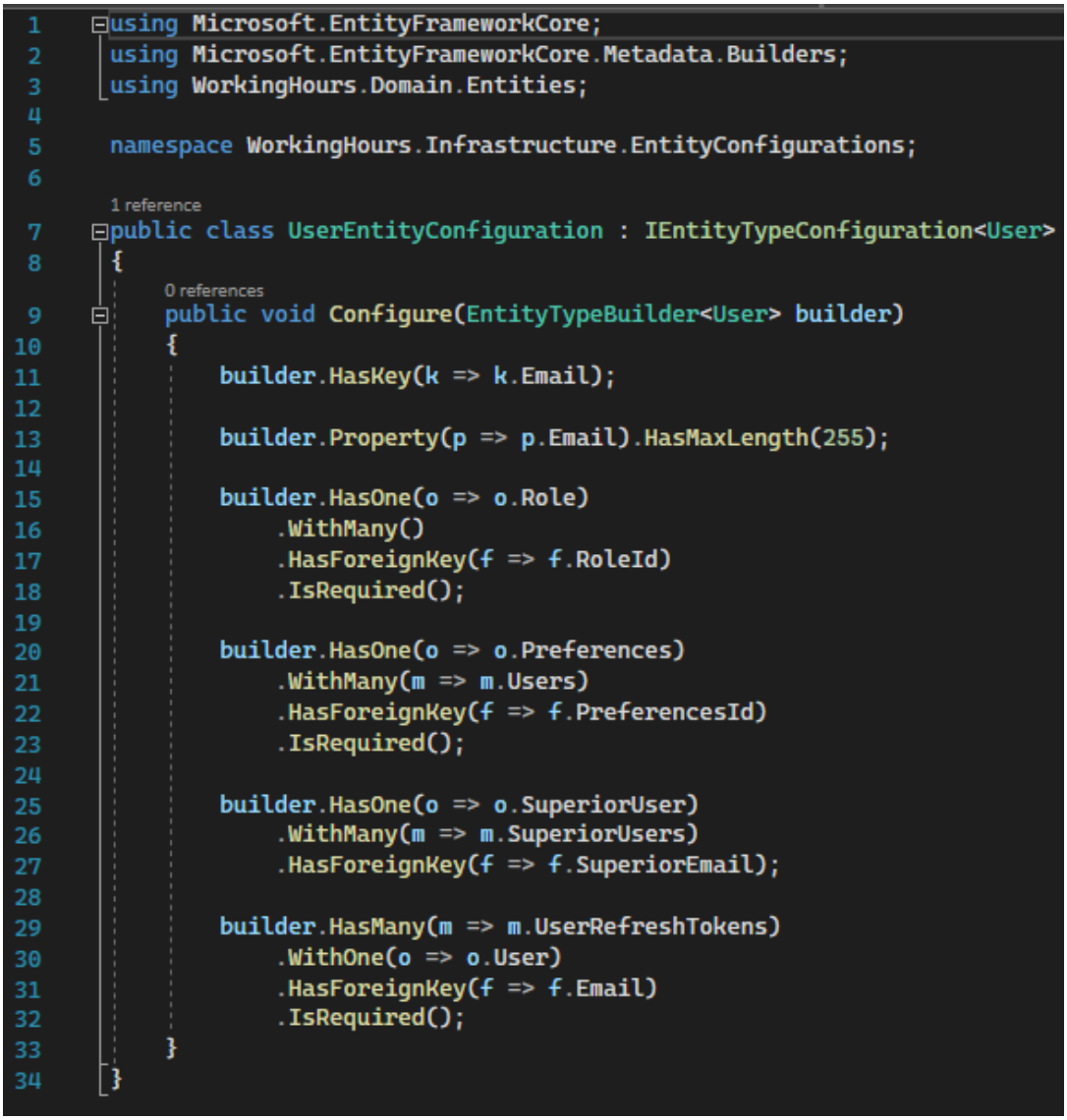
Konfiguracija modela postavlja zadanu shemu, razne funkcije nad bazom podataka te dodatne anotacije nad atributima unutar entiteta koji mogu primjerice definirati da je neki atribut striktno email.

Konfiguracija entiteta definira entitet u odnosu na ostale entitete, što znači da ona definira primarne i alternativne ključeve, vrste veza i ime tablice koje će entitet imati unutar baze podataka.

Konfiguracija imovine može se koristiti za postavljanje imena stupaca unutar tablice podataka, vanjskih ključeva, zadanih vrijednosti, tipa podataka te ispitivanja može li neki objekt poprimiti tip null.

Kako bi znali postaviti veze među entitetima dobro je unaprijed znati kakve brojnosti se točno traže za koji entitet. U slučaju da su brojnosti veza poznate dovoljno je prevesti te brojnosti u kod pomoću Fluent API dokumentacije. U sljedećem dijelu rada prikazan je primjer postavljanja konfiguracije za entitet.

Slika 12 - UserEntityConfiguration klasa



```
using Microsoft.EntityFrameworkCore;
using Microsoft.EntityFrameworkCore.Metadata.Builders;
using WorkingHours.Domain.Entities;

namespace WorkingHours.Infrastructure.EntityConfigurations;

1 reference
public class UserEntityConfiguration : IEntityTypeConfiguration<User>
{
    0 references
    public void Configure(EntityTypeBuilder<User> builder)
    {
        builder.HasKey(k => k.Email);

        builder.Property(p => p.Email).HasMaxLength(255);

        builder.HasOne(o => o.Role)
            .WithMany()
            .HasForeignKey(f => f.RoleId)
            .IsRequired();

        builder.HasOne(o => o.Preferences)
            .WithMany(m => m.Users)
            .HasForeignKey(f => f.PreferencesId)
            .IsRequired();

        builder.HasOne(o => o.SuperiorUser)
            .WithMany(m => m.SuperiorUsers)
            .HasForeignKey(f => f.SuperiorEmail);

        builder.HasMany(m => m.UserRefreshTokens)
            .WithOne(o => o.User)
            .HasForeignKey(f => f.Email)
            .IsRequired();
    }
}
```

Izvor: Autor

Slika 12 prikazuje konfiguracije modela, entiteta i imovine za entitet User. Prvo što je potrebno napraviti unutar konfiguracije je to da se definira primarni ključ entiteta pomoću HasKey metode, osim u slučajevima kada se primarni ključ zove Id - tada ga Fluent API sam dodjeli kao primarni ključ. Nakon definiranja primarnog ključa moguće je dodati neke konfiguracije imovine kao što je u ovom slučaju HasMaxLenght metoda. Najbitniji dio, kreiranje veza, definira se pomoću HasMany, HasOne, WithMany, WithOne metoda. Te

metode definiraju brojnost veze među entitetima. Koristeći metodu HasForeignKey dodjeljuje se vanjski ključ nekom atributu.

3.4.3. Klasa WorkingHoursContext

Klasa WorkingHoursContext je klasa koja predstavlja bazu. Unutar nje je potrebno nazvati sve modele koji će činiti bazu podataka. Unutar te klase moguće je i definirati veze među entitetima, no u ovome slučaju konfiguracije veza pisat će se u zasebnom dijelu aplikacije nazvanom EntityConfigurations kako bi konfiguracije bile odvojene i lakše za održavanje.

Slika 13 - WorkingHoursContext klasa

```
using Microsoft.EntityFrameworkCore;
using WorkingHours.Domain;
using WorkingHours.Domain.Entities;
using WorkingHours.Domain.Enums;
using WorkingHours.Infrastructure.EntityConfigurations;

namespace WorkingHours.Infrastructure;

80 references
public class WorkingHoursContext : DbContext
{
    0 references
    public WorkingHoursContext()
    {
    }

    0 references
    public WorkingHoursContext(DbContextOptions<WorkingHoursContext> options) : base(options)
    {
    }

    13 references
    public DbSet<User>? Users { get; set; }
    5 references
    public DbSet<Role>? Roles { get; set; }
    11 references
    public DbSet<Preferences>? Preferences { get; set; }
    12 references
    public DbSet<ProjectUser>? ProjectUsers { get; set; }
    12 references
    public DbSet<Project>? Projects { get; set; }
    14 references
    public DbSet<Hours>? Hours { get; set; }
    5 references
    public DbSet<HoursType>? HoursTypes { get; set; }
    9 references
    public DbSet<Tag>? Tags { get; set; }
    5 references
    public DbSet<Days>? Days { get; set; }
    6 references
    public DbSet<UserRefreshToken>? UserRefreshTokens { get; set; }

    0 references
    protected override void OnModelCreating(ModelBuilder modelBuilder)
    {
        base.OnModelCreating(modelBuilder);
        modelBuilder.ApplyConfigurationsFromAssembly(typeof(UserEntityConfiguration).Assembly);

        SeedEnums(modelBuilder);
    }

    1 reference
    private void SeedEnums(ModelBuilder builder)
    {
        builder.Entity<Days>().HasData(Enumeration.GetAll<Days>()
            .Select(x => new Days(x.Id, x.Name)).ToArray());

        builder.Entity<HoursType>().HasData(Enumeration.GetAll<HoursType>()
            .Select(x => new HoursType(x.Id, x.Name)).ToArray());

        builder.Entity<Role>().HasData(Enumeration.GetAll<Role>()
            .Select(x => new Role(x.Id, x.Name)).ToArray());
    }
}
```

Izvor: Autor

Kao što vidimo u slici 13, prvo je potrebno definirati da Context klasa nasljeđuje generalnu DbContext klasu kako bi posjedovala sve potrebne funkcionalnosti. Nakon toga potrebno je definirati koji će se entiteti zapisati u bazu podatka. Zatim je potrebno iz Assembly-a uzeti sve konfiguracije veza. U ovom slučaju nije potrebno navoditi sve konfiguracije posebno, već je dovoljno uzeti jednu instancu. I posljednje što je potrebno je zapisati Enum[6] vrijednosti u pripadajuće tablice.

Kada je klasa WorkingHoursContext izrađena, preostaje još samo je pozvati pri pokretanju web API-a kako bi se ona koristila. Potrebno je i dodati ConnectionString zbog toga da API zna s kojom bazom da se uspostavi veza. Poziva se kao na slici 14

Slika 14 - DbContext unutar Program.cs

```
44    builder.Services.AddDbContext<WorkingHoursContext>(options => options
45        .UseSqlServer(builder.Configuration.GetConnectionString("DefaultConnection")));
```

Izvor: Autor

U ovom slučaju DefaultConnection zapisan je u JSON formatu unutar appsettings i izgleda kao u sljedećem kodu.

Slika 15 - DefaultConnection unutar appsettings.json

```
8     "AllowedHosts": "*",
9     "ConnectionStrings": {
10      "DefaultConnection": "Server=localhost;Database=WorkingHours;Trusted_Connection=True;User ID=test;Password=1234"
11    },
```

Izvor: Autor

[6] Enum – tip podatka u koji se može upisati više elemenata od kojih svaki ima svoju brojčanu vrijednost.

3.4.4. Migracija

Kako bi stvorili relacijsku bazu podataka, potrebno je napraviti migraciju koja će uzeti sve modele, konfiguracije i DbContext te zapisati ih unutar jedne zasebne klase, nakon koje će biti potrebno ažurirati bazu podataka s novom migracijom. Migracija se kreira na način da se unutar naredbenog retka (engl. Command Prompt) ili PowerShela upiše naredba. Bitno je pozicionirati se unutar željenog projekta i sloja u kojem se nalazi klasa DbContext kako bi EF Core znao gdje i kako napraviti migraciju. Zatim treba pokrenuti sljedeću CLI naredbu „*dotnet ef migrations add init*“.

Nakon izvođenja te naredbe, generira se kod unutar mape Migrations u sloju Infrastructure koji opisuje cijelu bazu podatka. Na slici 16 prikazan je odsječak koda vezan uz entitet Hours.

Slika 16 - Isječak migracije vezan uz Hours entitet

```
184             migrationBuilder.CreateTable(
185                 name: "Hours",
186                 columns: table => new
187                 {
188                     Id = table.Column<int>(type: "int", nullable: false)
189                         .Annotation("SqlServer:Identity", "1, 1"),
190                     StartDate = table.Column<DateTime>(type: "datetime2", nullable: false),
191                     StartDateUtc = table.Column<DateTime>(type: "datetime2", nullable: false),
192                     EndDate = table.Column<DateTime>(type: "datetime2", nullable: false),
193                     EndDateUtc = table.Column<DateTime>(type: "datetime2", nullable: false),
194                     Description = table.Column<string>(type: "nvarchar(500)", maxLength: 500, nullable: true),
195                     HoursTypeId = table.Column<int>(type: "int", nullable: false),
196                     ProjectId = table.Column<int>(type: "int", nullable: false),
197                     UserEmail = table.Column<string>(type: "nvarchar(255)", nullable: false),
198                     TagId = table.Column<int>(type: "int", nullable: true)
199                 },
200                 constraints: table =>
201                 {
202                     table.PrimaryKey("PK_Hours", x => x.Id);
203                     table.ForeignKey(
204                         name: "FK_Hours_HoursTypes_HoursTypeId",
205                         column: x => x.HoursTypeId,
206                         principalTable: "HoursTypes",
207                         principalColumn: "Id",
208                         onDelete: ReferentialAction.Cascade);
209                     table.ForeignKey(
210                         name: "FK_Hours_ProjectUsers_ProjectId_UserEmail",
211                         columns: x => new { x.ProjectId, x.UserEmail },
212                         principalTable: "ProjectUsers",
213                         principalColumns: new[] { "ProjectId", "UserEmail" },
214                         onDelete: ReferentialAction.Cascade);
215                     table.ForeignKey(
216                         name: "FK_Hours_Tags_TagId",
217                         column: x => x.TagId,
218                         principalTable: "Tags",
219                         principalColumn: "Id");
220                 });
```

Izvor: Autor

Kako je već navedeno, potrebno je još samo ažurirati bazu podataka s novom migracijom. Bitno je pozicionirati se unutar željenog projekta te onda će EF Core sam prepoznati koju bazu podataka treba ažurirati s najnovijom migracijom. Ažuriranje se pokreće pomoću CLI naredbe „*dotnet ef database update"*.

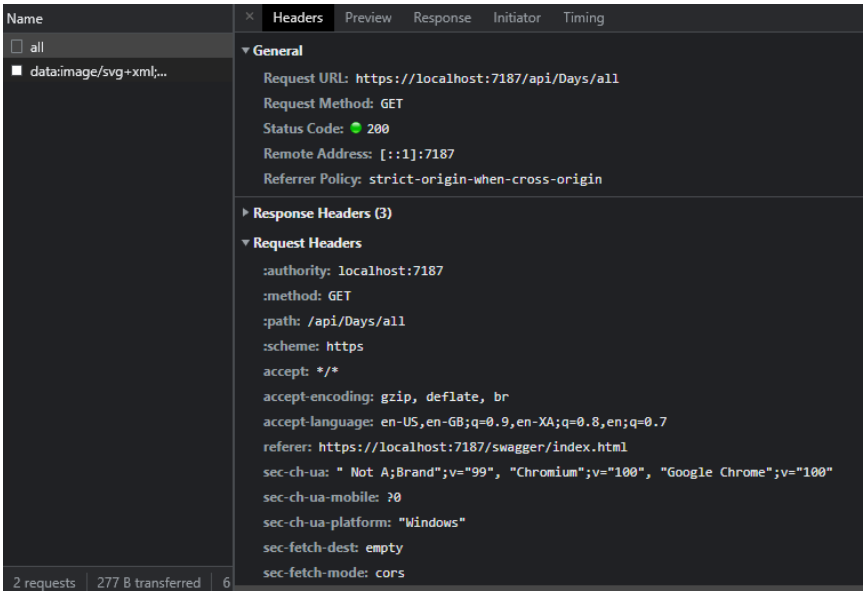
4. HTTP

HTTP (engl. Hypertext Transfer Protocol) je temeljni protokol weba. Dizajniran je 1990-ih godina. Koristi se za razmjenu informacija putem umreženih računala, kao što su HTML dokumenti. Temelji se na konceptu zahtjev – odgovor, npr. neko klijentsko računalo pošalje zahtjev nekom serveru te zatim čeka za odgovor. Bitno je naglasiti kako HTTP nema stanja, što znači da dva zahtjeva koja su neposredno poslana „ne znaju" jedan za drugog. To donosi probleme ako je potrebno npr. zapamtiti stanje košarice, u tom slučaju potrebno je implementirati kolačiće (engl. Cookies). Kolačići funkcioniraju na način da se sa svakim zahtjevom šalje dodatan dio koji omogućava „pamćenje" stanja. (Mozilla docs, 2022.)

Tipičan HTTP zahtjev sadrži sljedeće elemente: HTTP verziju, poveznicu (engl. URL), vrstu HTTP metode, HTTP zaglavlje zahtjeva. HTTP verzija definira formu zahtjeva. Poveznica definira odredište odnosno krajnju točku. Vrsta metode definira što se želi postići. Četiri osnovne metode su GET[7], POST[8], PUT[9], DELETE[10]. HTTP zaglavlje definira određene pojedinosti o zahtjevu, na primjer vrstu HTTP metode, naziv platforme i pretraživača s kojeg je poslan zahtjev. Na slici 17. prikazan je sadržaj GET zahtjeva koji kao odgovor očekuje listu svih dana u tjednu.

[7] GET – HTTP metoda koja služi za dohvaćanje podataka
[8] POST – HTTP metoda koja služi za prijenos strukturiranih podataka s ciljem stvaranja resursa
[9] PUT – HTTP metoda koja služi za prijenos strukturiranih podataka sa ciljem ažuriranja postojećeg resursa
[10] DELETE – HTTP metoda koja služi za brisanje određenog resursa

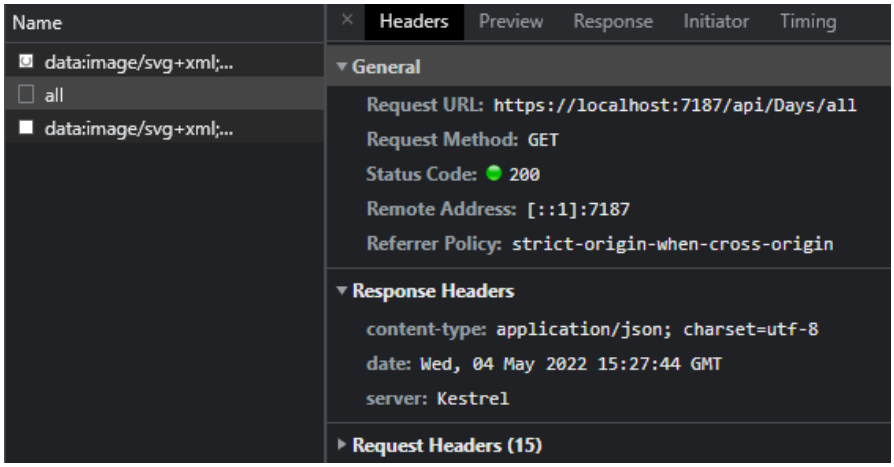
Slika 17 - Primjer GET zahtjeva



Izvor: Autor

Odgovor na HTTP zahtjev sadrži dva elementa: status odgovora i HTTP zaglavlje odgovora. Status odgovora je krucijalan iz razloga što on govori je li metoda uspjela. Status je prikazan u obliku broja s vrijednošću od 100 do 599. Statusni brojevi od 100 do 199 predstavljaju neku informaciju, ali ne govore o tome je li zahtjev uspio ili ne. Statusni brojevi od 200 do 299 predstavljaju uspješno izveden zahtjev s dodatnim opisom. Statusni brojevi od 300 do 399 predstavljaju problem s URL. Statusni brojevi od 400 do 499 predstavljaju grešku na klijentskoj strani, a statusni brojevi od 500 do 599 predstavljaju greške na poslužiteljskoj strani. Unutar zaglavlja HTTP odgovora definiran je format dospjelih podataka i datum izvršenja zahtjeva. Na slici 18. prikazan je uspješan GET odgovor (Mozilla docs 2022.).

Slika 18 - Primjer GET odgovora

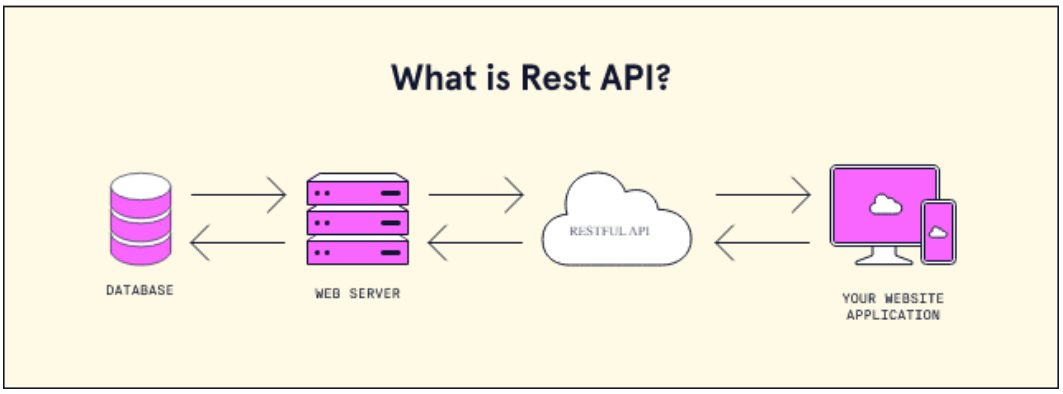


Izvor: Autor

4.1. REST

REST (engl. Representational state transfer) se temelji na HTTP protokolu i on je strukturni stil programiranja koji služi za standardizaciju komunikacije lokalnih računalnih sustava i web-a, s ciljem da sustavi s različitih platformi brže i efikasnije komuniciraju međusobno. Sustave koji to omogućavaju nazivamo RESTful. Na slici 19. prikazana je shema RESTful API-a.

Slika 19 - RESTfull API



Izvor: https://www.codecademy.com/article/what-is-rest (10.5.2022.)

Korištenjem takve strukture implementacija klijentske strane je potpuno neovisna o implementaciji serverske strane. To znači da kod sa klijentske strane može biti u bilo kojem trenutku promijenjen bez posljedica na serverskoj strani. Jedina bitna stvar je da klijentska i serverska strana znaju format komunikacije kako bi se sporazumjele. Korištenjem REST sučelja različiti klijenti izvode iste operacije koje rezultiraju istim povratnim informacijama (What is REST, 2022.).

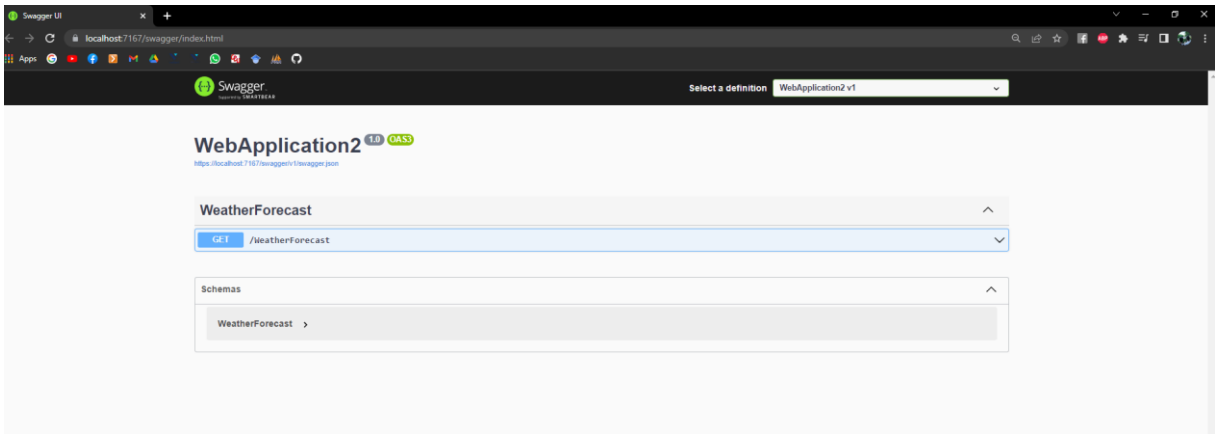
Kada se razvija neki REST API potrebno je voditi računa o sljedećim REST načelima. Jedinstveno formirano sučelje osigurava da svi zahtjevi za isti resurs budu identični, bez obzira na to od kuda ti zahtjevi dolaze. Odvojenost klijenta i poslužitelja sprječava ovisnost između klijentske i poslužiteljske strane. REST API je bez stanja što znači da svaki zahtjev mora sadržavati sve potrebne informacije kako bi se obradio. Privremenim skladištenjem podataka poboljšavaju se performanse klijentske strane. REST API ima sistemsku arhitekturu u slojevima, što znači da je potrebno voditi računa o tome da zahtjevi i odgovori prelaze kroz različite slojeve kako bi se obradili (IBM Cloud Education, 2021.).

4.2. Swagger

Swagger je alat otvorenog koda koji služi razvojnim inženjerima za testiranje i razvijanje API-a. Zadan je unutar Program klase i izvodi se zajedno s glavnim tokom programa.

Swagger pregledava kontrolere unutar .NET Core API-a i poslužuje web stranicu sa svim krajnjim točkama kako bi se mogle testirati funkcionalnosti samog API-a. Prikaz Swagger-a je na slici 20. U sredini se nalaze kontroleri sa krajnjim točkama. U ovome primjeru postoji samo jedan kontroler WeatherForecast s jednom krajnjom točkom WeatherForecast koja je tipa GET. U gornjem desnom uglu može se odabrati verzija API-a. (Swagger, 2019.)

Slika 20 - Swagger



Izvor: Autor

5. Razvoj API-a za praćenje radnih sati zaposlenika

Kao primjer implementacije svih prije navedenih tehnologija izrađen je API poslovne (eng. Enterprise) razine sa svrhom praćenja broja radnih sati zaposlenika. API će se koristiti u interne svrhe poduzeća. Daljnji plan razvoja je pridružiti API s klijentskim dijelom pomoću razvojnog okvira Angular. U nastavku će biti objašnjen sam razvoj baze podataka, zadovoljavanje traženih funkcionalnosti sam razvoj API-a. Za razvoj poslužiteljskog dijela koristiti će se razvojno okruženje Visual Studio 2022 te Microsoft SQL Server Management Studio 18 za vizualizaciju i podešavanje baze podataka.

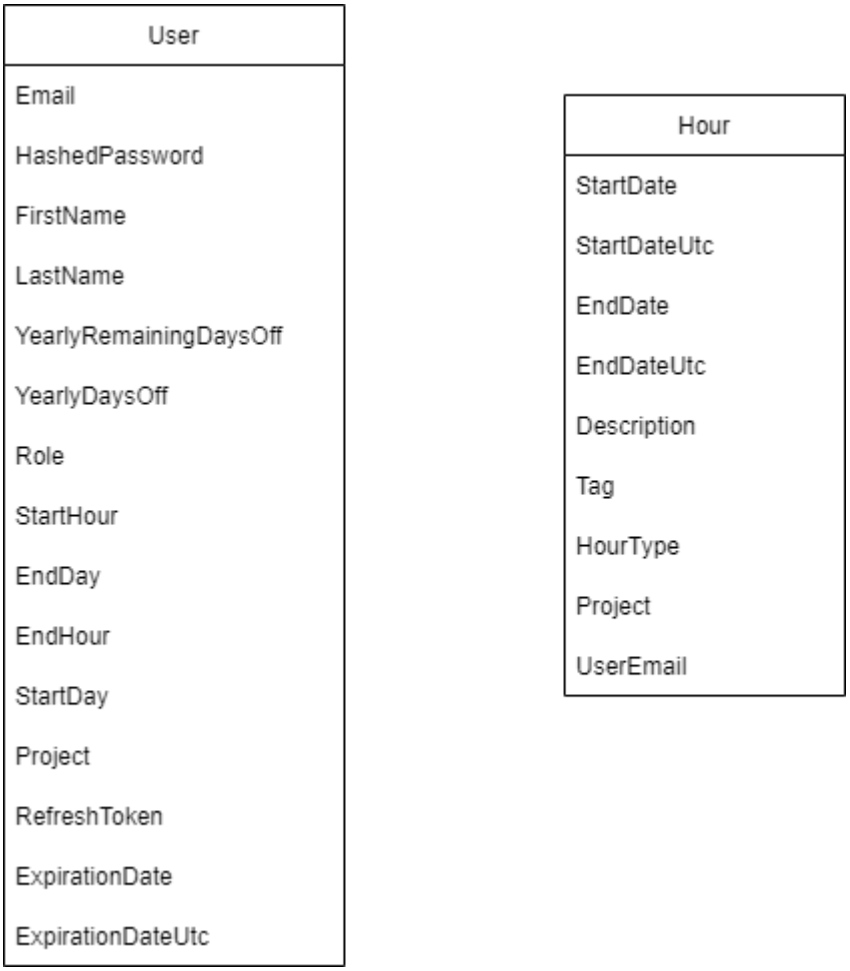
5.1. Razvoj baze podataka

U samom početku ove izrade baze podataka definirat će se kako bi sustav trebao funkcionirati. Svrha ovog informacijskog sustava je pojednostaviti bilježenje radnih sati zaposlenika unutar poduzeća. Na početku je potrebno da se svaki korisnik autentificira u sustav. Svaki korisnik može imati jednu od tri uloga: Developer, Team lead i Admin. Uloga Developer omogućava unos i pregled vlastitih sati na različitim projektima u kojima sudjeluje te izmjenu vlastitih postavki o načinu izvođenja radnih sati. Uloga Team lead omogućava stvaranje novih Developera i njihovo dodjeljivanje na određeni projekt te pregled radnih sati Developera koji su na istom projektu. Uloga Admin omogućava stvaranje i praćenje svih korisnika neovisno o projektu. Radni sati za običan osmerosatni radni dan moraju se automatski upisivati, što znači da se ručno upisuju samo sati van tog intervala. Kada se korisnik autentificira vidljiv je kalendar sa svim zapisanim radnim satima i projektima na kojima korisnik sudjeluje. Sati se bilježe na način da se za odabrani projekt odabere količina sati, vrsta sata, opis i oznaka. Vrsta sata može biti: prekovremeni, bez zadatka, godišnji odmor, blagdan, bolovanje.

5.2. Kreiranje entiteta i konfiguracija

Kako bi ostvarili prethodno definirane funkcionalnosti potrebno je kreirati entitete unutar samog API-a pomoću EF Core razvojnog okvira i postaviti njihove konfiguracije. Entiteti će nakon migracije unutar baze podataka predstavljati relacije, dok će konfiguracije služiti kao veze između relacija. Entiteti se stvaraju na način da se iz potrebitih funkcionalnosti prepoznaju „spremnici podataka“, odnosno relacije. Na primjer ako postoji funkcionalnost autentificiranja korisnika može se zaključiti da će biti potreban entitet u kojemu će se zapisivati podaci o korisnicima.

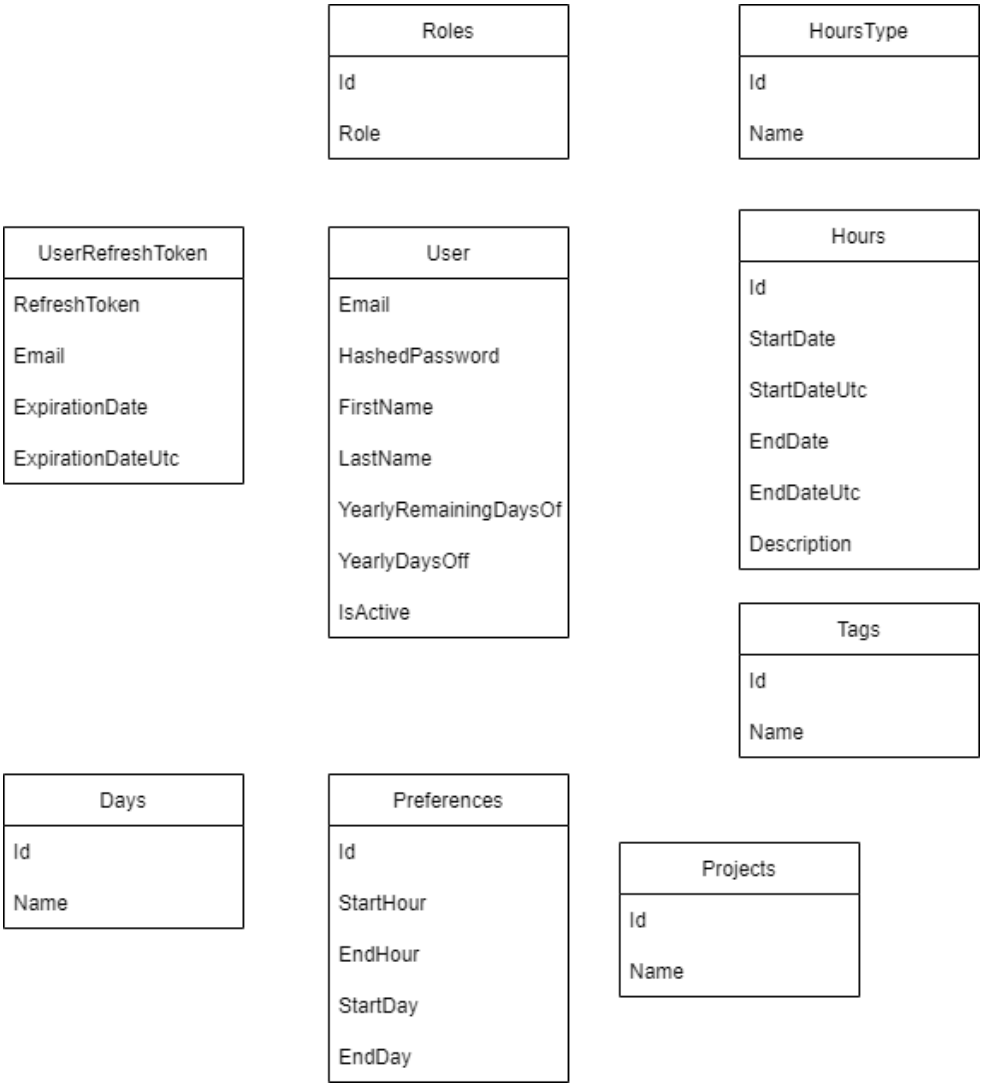
Slika 21 - Primjer osnovnih eniteta



Izvor: Autor

Na slici 21 prikazani su atributi koji su potrebni za entitete User i Hour. Nakon što se odrede atributi koje je nužno bilježiti za određeni entitet, potrebno je normalizirati entitete do treće normalne forme (3NF). Normalizacija je progresivna što znači da ako se želi doći do određene razine potrebno je zadovoljiti i prijašnje razine normalne forme. Dakle kako bi normalizirali do 3NF prijašnji model sa slike 21 potrebno je zadovoljiti uvjete prve, druge i treće normalne forme. Kako bi zadovoljili uvjete prve normalne forme potrebno je svaki atribut sadrži samo jednu vrijednost. Uvjet druge normalne forme je da svaka relacija sadrži jedan atribut koji djeluje kao jedinstveni ključ ili dva atributa koja stvaraju kompozitni ključ. Uvjet treće normalne forme je da niti jedan atribut koji je bez ključa ne smije zavisiti o nekom drugom atributu bez ključa. Rezultat normalizacije prikazan je na slici 22.

Slika 22 - Primjer entiteta u 3NF



Izvor: Autor

Sljedeći korak je dobivene entitete (slika 22) zapisati unutar .NET Core API-a. Na slici 23 prikazan je kod koji definira entitet Role. Entitet Role potreban je za spremanje Enumeration vrijednosti, odnosno vrijednosti koje se neće mijenjati. Unutar klase Role koja nasljeđuje klasu Enumeration, Enumeration vrijednosti definirane su kao objekti tipa Role te su im zadane vrijednosti Admin, Lead developer i Developer.

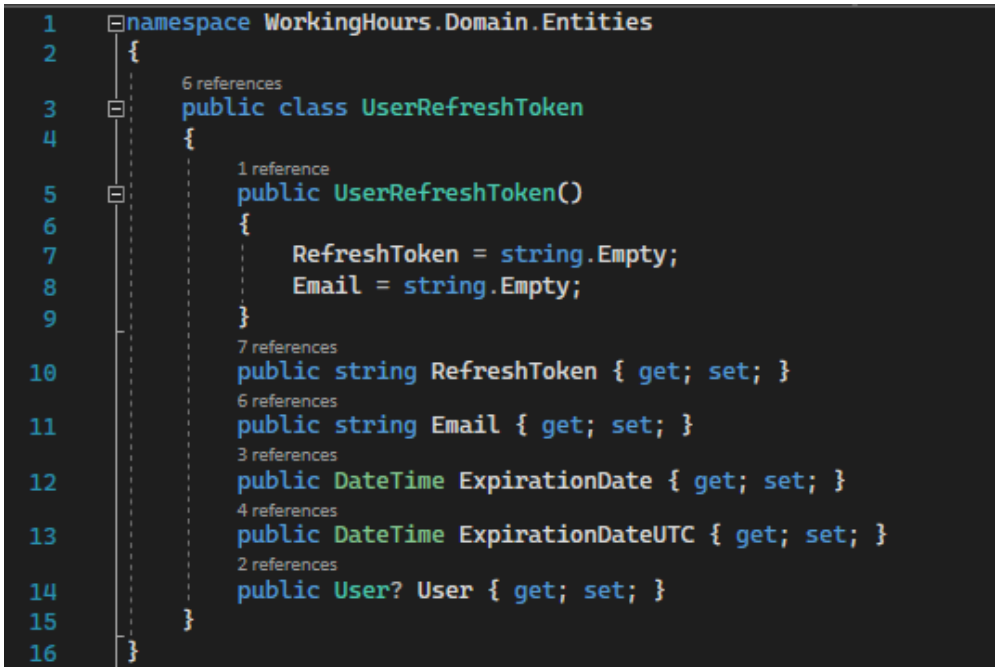
Slika 23 - Entitet Role

```
1   namespace WorkingHours.Domain.Enums;
2
    14 references
3   public class Role : Enumeration
4   {
5       public static Role Admin = new(1, "Admin");
6       public static Role LeadDeveloper = new(2, "Lead Developer");
7       public static Role Developer = new(3, "Developer");
8
        0 references
9       public Role():base()
10      {
11      }
12
        4 references
13      public Role(int id, string name) : base(id, name)
14      {
15      }
16  }
```

Izvor: Autor

Na slici 24 prikazan je entitet UserRefreshToken. Taj entitet je potreban za korištenje i implementaciju JWT tokena. Entitet UserRefreshToken sadrži atribut RefreshToken u kojem se spremaju vrijednosti tokena ali služi i kao primarni ključ relacije, atribut Email je vanjski ključ relacije User i služi za povezivanje tokena sa određenim korisnikom, atribut ExpirationDate označava datum i vrijeme kada će valjanost određenog tokena isteći po lokalnom vremenu, atribut ExpirationDateUTC ima istu svrhu no on bilježi vrijeme u UTC (engl. Universal Time Zone) formatu. Kada se korisnik autentificira, stvara se novi token koji pamti korisnikovu prijavu na određeno vrijeme. Može se reći da služi kao „Zapamti me“ gumb.

Slika 24 - Entitet UserRefreshToken



```
namespace WorkingHours.Domain.Entities
{
    6 references
    public class UserRefreshToken
    {
        1 reference
        public UserRefreshToken()
        {
            RefreshToken = string.Empty;
            Email = string.Empty;
        }
        7 references
        public string RefreshToken { get; set; }
        6 references
        public string Email { get; set; }
        3 references
        public DateTime ExpirationDate { get; set; }
        4 references
        public DateTime ExpirationDateUTC { get; set; }
        2 references
        public User? User { get; set; }
    }
}
```

Izvor: Autor

Na slici 25 prikazan je entitet Hours koji služi za pohranu svih atributa vezanih uz same radne sate. Entitet Hours sadrži atribut Id koji je njegov primarni ključ, atributi StartDate i StartDateUtc bilježe datum i vrijeme početka rada u lokalnom i UTC formatu vremena, atributi EndDate i EndDateUtc bilježe datum i vrijeme kraja rada u lokalnom i UTC formatu vremena, atribut Description služi za pohranu opisa radnih sati, atribut HoursTypeId je vanjski ključ relacije HoursType i služi za definiranje vrste radnih sati, atribut ProjectId je vanjski ključ relacije ProjectUsers i služi za dodjelu radnih sati na određeni projekt i posljednji atribut UserEmail je vanjski ključ relacije ProjectUsers koji služi za dodjelu radnih sati određenom korisniku.

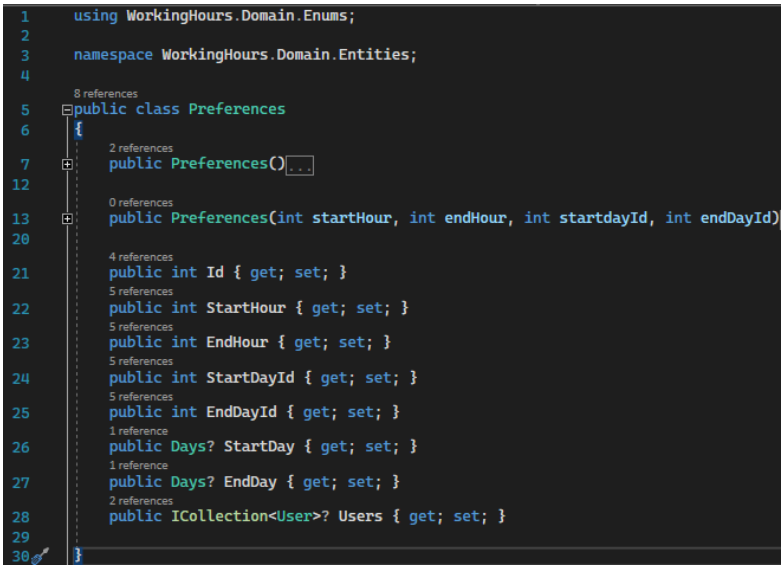
Slika 25 - Entitet Hours

```
1   using WorkingHours.Domain.Enums;
2
3   namespace WorkingHours.Domain.Entities;
4
    7 references
5   public class Hours
6   {
        1 reference
7       public Hours()
8       {
9           UserEmail = string.Empty;
10      }
11
        5 references
12      public int Id { get; set; }
        5 references
13      public DateTime StartDate { get; set; }
        5 references
14      public DateTime StartDateUtc { get; set; }
        5 references
15      public DateTime EndDate { get; set; }
        5 references
16      public DateTime EndDateUtc { get; set; }
        6 references
17      public string? Description { get; set; }
        6 references
18      public int HoursTypeId { get; set; }
        1 reference
19      public HoursType? HoursType { get; set; }
        8 references
20      public int ProjectId { get; set; }
        9 references
21      public string UserEmail { get; set; }
        2 references
22      public ProjectUser? ProjectUser { get; set;
        8 references
23      public int? TagId { get; set; }
        2 references
24      public Tag? Tag { get; set; }
25  }
```

Izvor: Autor

Na slici 26 prikazan je entitet Preferences koji služi za pohranu formata rada, odnosno omogućuje korisniku da mijenja svoje radno vrijeme. Entitet Preferences sadrži atribut Id koji je ujedno i primarni ključ, atributi StartHour i EndHour služe za postavljanje početka i kraja radnog vremena, a atributi StartDayId i EndDayId su vanjski ključevi relacije Days koji služe za postavljanje početnog i krajnjeg dana unutar radnog tjedna.

Slika 26 - Entitet Preferences

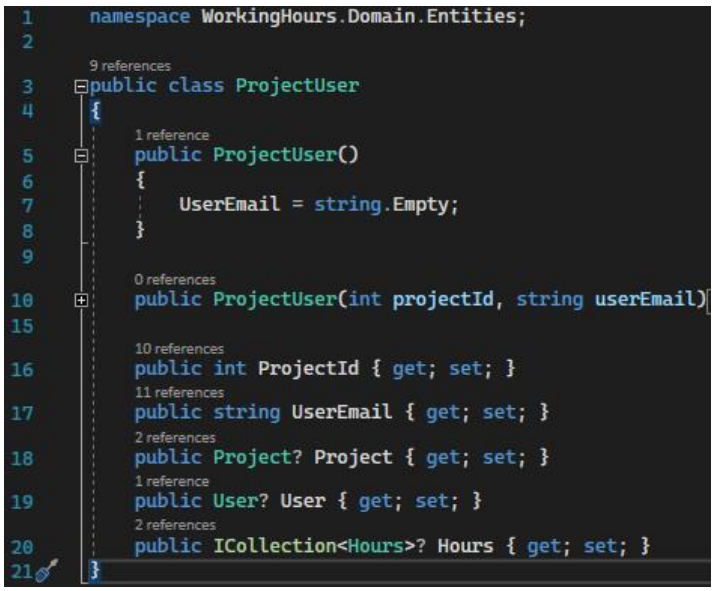


```
using WorkingHours.Domain.Enums;

namespace WorkingHours.Domain.Entities;

8 references
public class Preferences
{
    2 references
    public Preferences()...

    0 references
    public Preferences(int startHour, int endHour, int startdayId, int endDayId)

    4 references
    public int Id { get; set; }
    5 references
    public int StartHour { get; set; }
    5 references
    public int EndHour { get; set; }
    5 references
    public int StartDayId { get; set; }
    5 references
    public int EndDayId { get; set; }
    1 reference
    public Days? StartDay { get; set; }
    1 reference
    public Days? EndDay { get; set; }
    2 references
    public ICollection<User>? Users { get; set; }

}
```

Izvor: Autor

Na slici 27 prikazan je entitet ProjectUser koji služi kao agregacija između entiteta User i Project. Entitet ProjectUser sadrži atribut ProjectId koji je vanjski ključ relacije Project i atribut UserEmail koji je vanjski ključ relacije User, te zajedno čine kompozitni ključ.

Slika 27 - Entitet ProjectUser

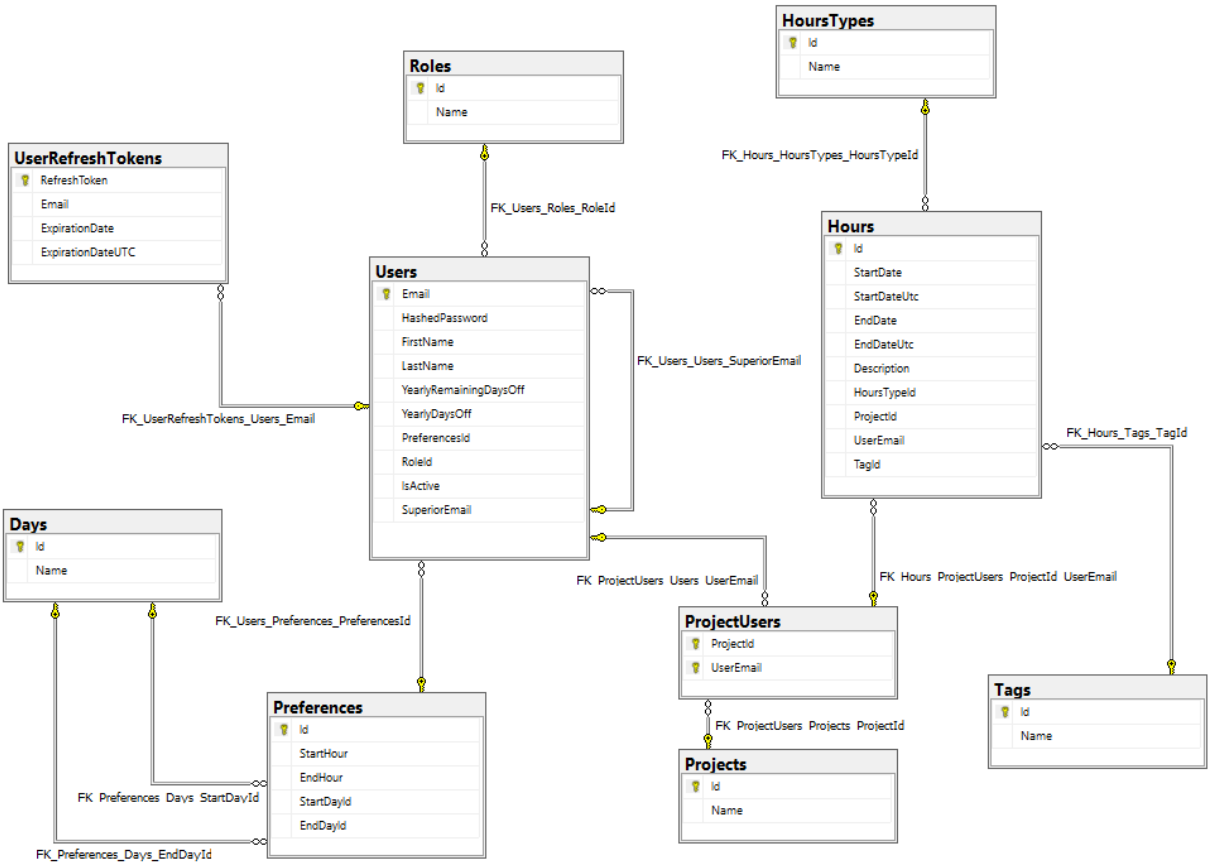


```
namespace WorkingHours.Domain.Entities;

9 references
public class ProjectUser
{
    1 reference
    public ProjectUser()
    {
        UserEmail = string.Empty;
    }

    0 references
    public ProjectUser(int projectId, string userEmail)

    10 references
    public int ProjectId { get; set; }
    11 references
    public string UserEmail { get; set; }
    2 references
    public Project? Project { get; set; }
    1 reference
    public User? User { get; set; }
    2 references
    public ICollection<Hours>? Hours { get; set; }
}
```

Izvor:Autor

Nakon izrade svih potrebnih entiteta, potrebno je dodijeliti veze među entitetima na način da se ispiše konfiguracija za svaki entitet kao što je već objašnjeno u prijašnjim dijelovima rada. Kada postoje konfiguracije i entiteti, preostaje još napraviti migraciju i ažurirati bazu podataka. Na slici 28 prikazan je ER dijagram baze podataka unutar aplikacije Microsoft SQL Server Management Studio-a.

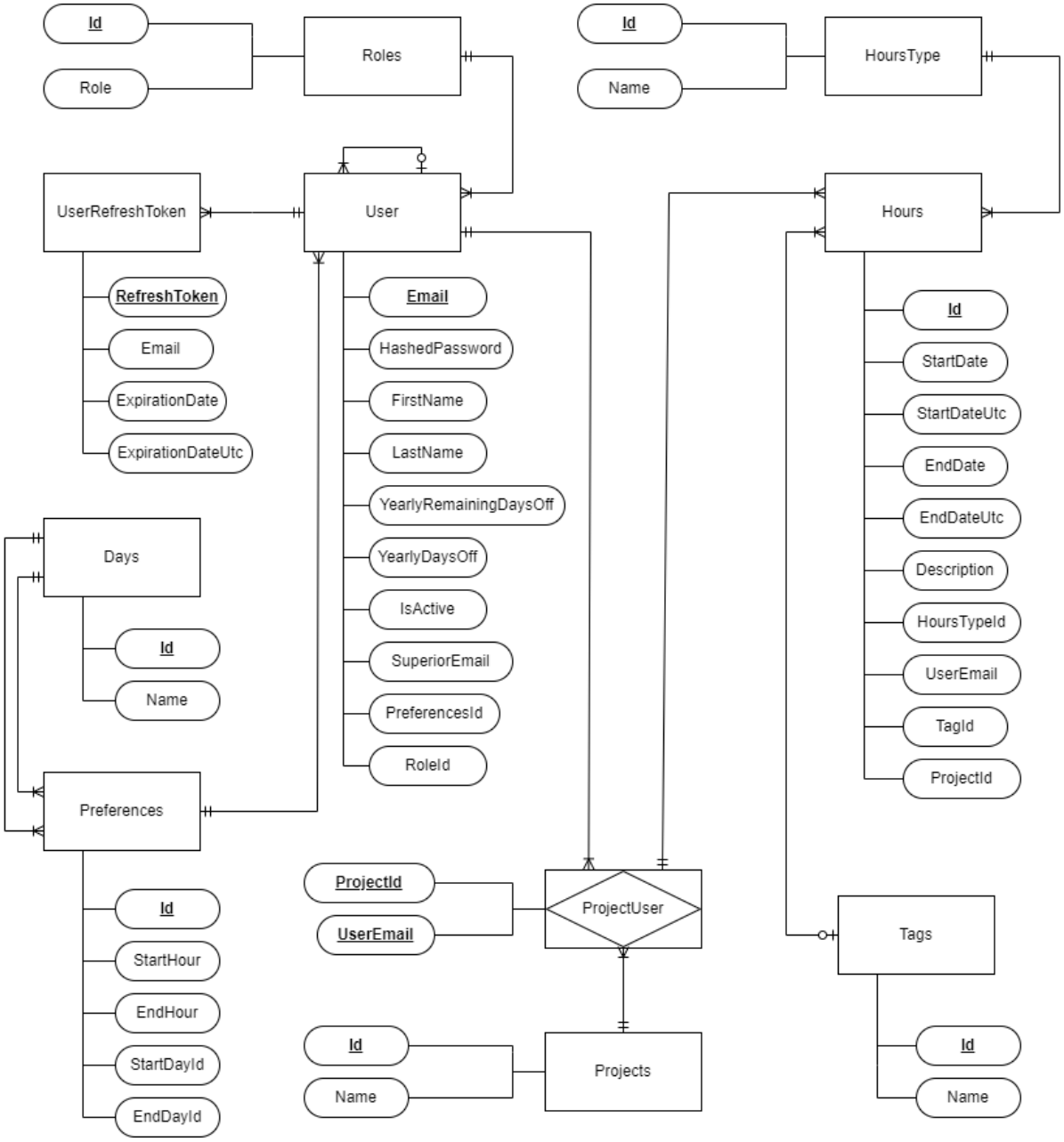
Slika 28 - MSSMS dijagram



Izvor: Autor

Na slici 29 prikazan je kompletan EVA model baze podataka sa Martinovom notacijom za prikaz brojnosti veza.

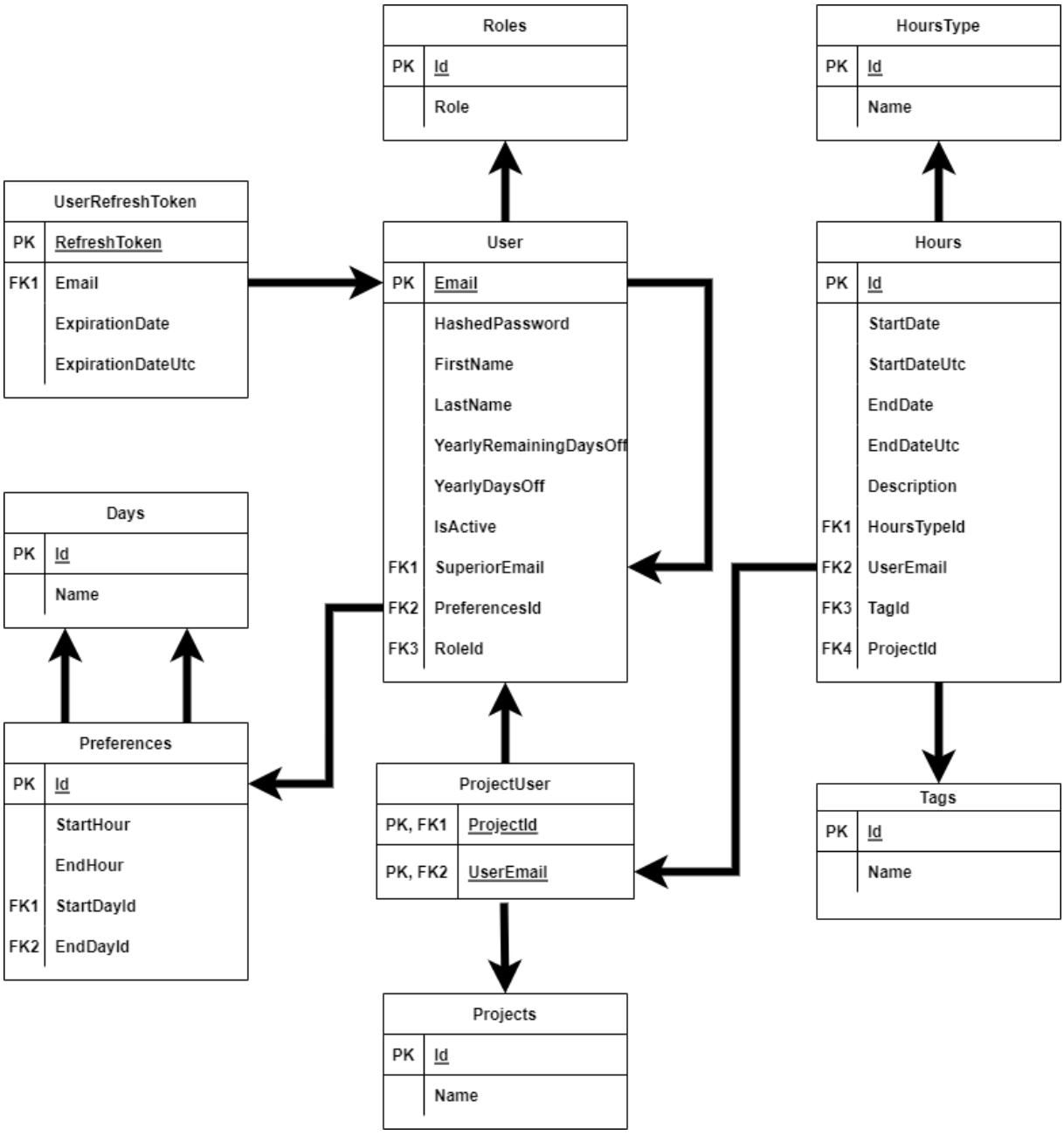
Slika 29 - EVA model



Izvor: Autor

Na slici 30 prikazan je relacijski model podataka.

Slika 30 - Relacijski model podataka



Izvor: Autor

U nastavku su zapisane relacijske sheme baze podataka:

Roles (**Id** (PK), Role)

HoursType (**Id** (PK), Name)

Tags (**Id** (PK), Name)

Projects (**Id** (PK), Name)

Days (**Id** (PK), Name)

UserRefreshToken (**RefreshToken** (PK), Email (FK1), ExpirationDate, ExpirationDateUtc)

User (**Email** (PK), HashedPassword, FirstName, LastName, YearlyRemainingDaysOff, YearlyDaysOff, IsActive, SuperiorEmail (FK1), PreferencesId (FK2), RoleId (FK2))

Preferences (**Id** (PK), StartHour, EndHour, StartDayId (FK1), EndDayId (FK2))

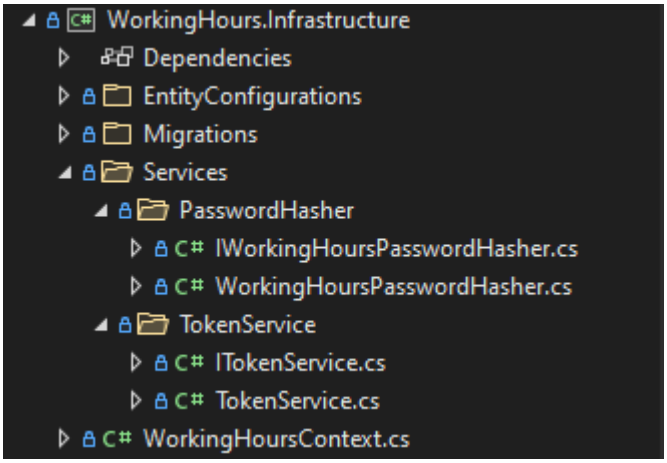
ProjectUsers (**ProjectId** (PK, FK1), **UserEmail** (PK, FK2))

Hours (**Id** (PK), Startdate, StartdateUtc, EndDate, EndDateUtc, Description, HoursTypeId (FK1), UserEmail (FK2), TagId (FK3), ProjectId (FK4))

5.3. Razvoj poslužiteljskog dijela

Za razvoj API-a koristit će se već navedeni razvojni okviri i programske strukture. Za potrebe enkripcije zaporki i generiranje JWT tokena koristit će se dodatni servisi putem Dependency Injection dizajnerskog uzorka, što znači da će određena klasa komunicirati sa samim servisom putem nekog sučelja (eng. Interface).

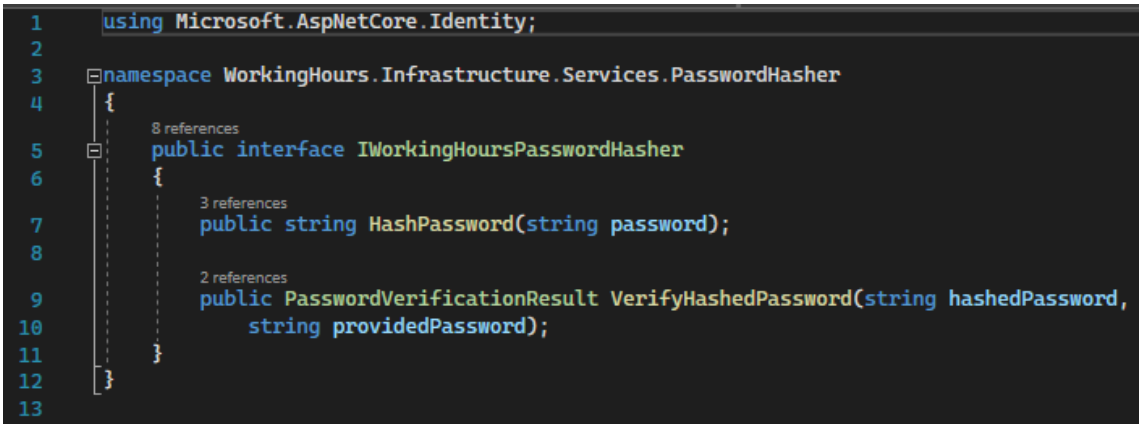
Slika 31 - Prikaz servisa



Izvor: Autor

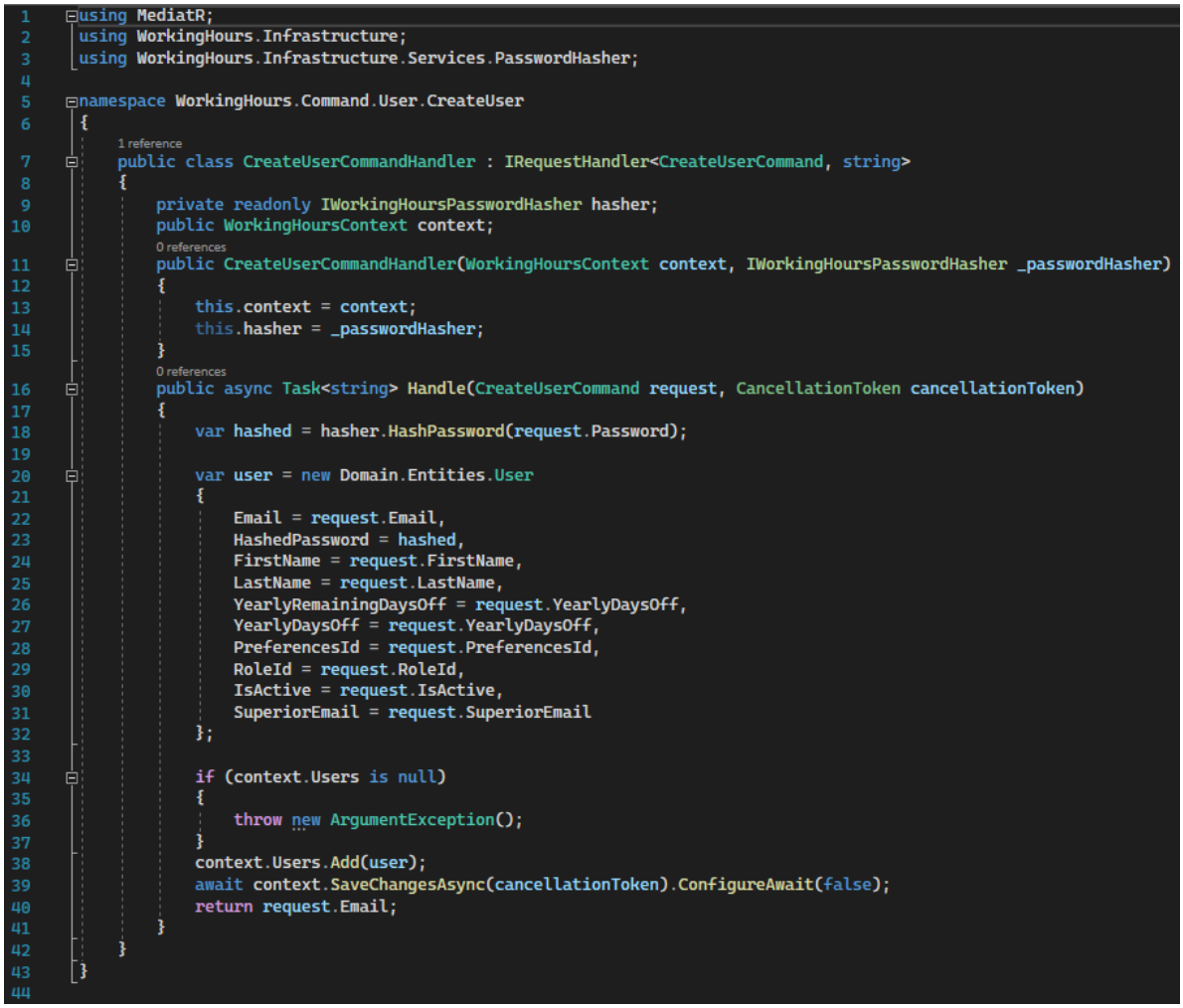
Na slici 31 prikazani su servisi PasswordHasher i TokenService sa svojim sučeljima. Sučelje IWorkingHoursPasswordHasher prikazano je na slici 31. Dakle, može se primijetiti da sučelje sadržava samo potrebna imena metoda za izvršavanje neke operacije.

Slika 32 - Sučelje IWorkingHoursPasswordHasher



Izvor: Autor

Slika 33 - Implementacija PasswordHasher servisa



```
1   using MediatR;
2   using WorkingHours.Infrastructure;
3   using WorkingHours.Infrastructure.Services.PasswordHasher;
4
5   namespace WorkingHours.Command.User.CreateUser
6   {
        1 reference
7       public class CreateUserCommandHandler : IRequestHandler<CreateUserCommand, string>
8       {
9           private readonly IWorkingHoursPasswordHasher hasher;
10          public WorkingHoursContext context;
            0 references
11          public CreateUserCommandHandler(WorkingHoursContext context, IWorkingHoursPasswordHasher _passwordHasher)
12          {
13              this.context = context;
14              this.hasher = _passwordHasher;
15          }
            0 references
16          public async Task<string> Handle(CreateUserCommand request, CancellationToken cancellationToken)
17          {
18              var hashed = hasher.HashPassword(request.Password);
19
20              var user = new Domain.Entities.User
21              {
22                  Email = request.Email,
23                  HashedPassword = hashed,
24                  FirstName = request.FirstName,
25                  LastName = request.LastName,
26                  YearlyRemainingDaysOff = request.YearlyDaysOff,
27                  YearlyDaysOff = request.YearlyDaysOff,
28                  PreferencesId = request.PreferencesId,
29                  RoleId = request.RoleId,
30                  IsActive = request.IsActive,
31                  SuperiorEmail = request.SuperiorEmail
32              };
33
34              if (context.Users is null)
35              {
36                  throw new ArgumentException();
37              }
38              context.Users.Add(user);
39              await context.SaveChangesAsync(cancellationToken).ConfigureAwait(false);
40              return request.Email;
41          }
42      }
43  }
44
```

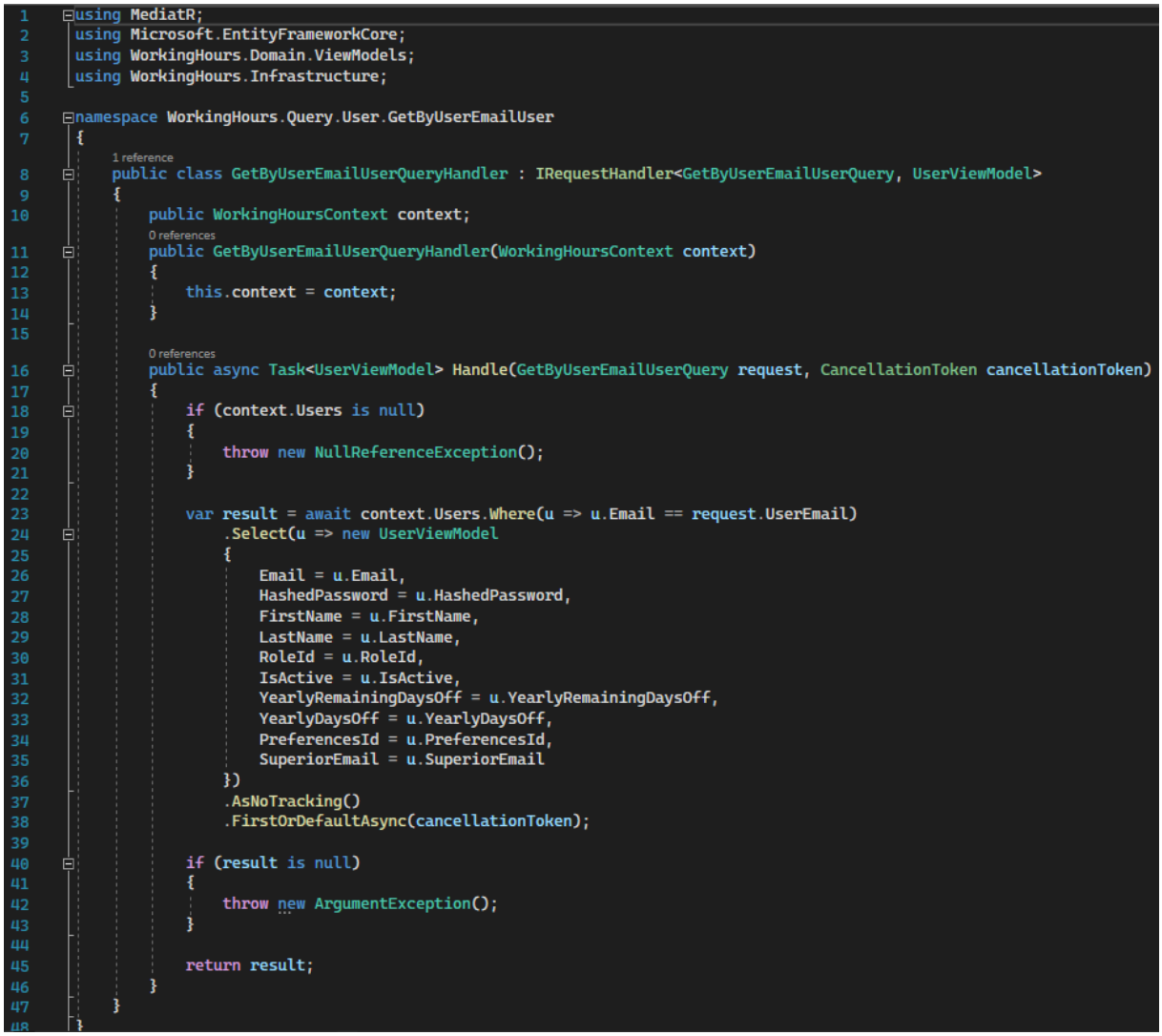
Izvor: Autor

Na slici 33 prikazana je implementacija servisa PasswordHasher unutar krajnje točke koja generira novog korisnika. Potrebno je stvoriti i inicijalizirati objekt klase IWorkingHoursPasswordHasher te se zatim preko objekta dohvaćaju potrebne metode servisa, kao što je prikazano na slici 33 unutar 18. linije koda.

Na slici 33 prikazan je i jedan primjer krajnje točke koja spada u Command kategoriju, što znači da ona mijenja stanje unutar baze podataka. U ovome slučaju upisuje novog korisnika.

Na slici 34 prikazan je i jedan primjer Query krajnje točke koja „čita“ iz baze podataka. Ta krajnja točka služi za dohvaćanje jednog korisnika putem Email atributa.

Slika 34 - Primjer Query krajnje točke



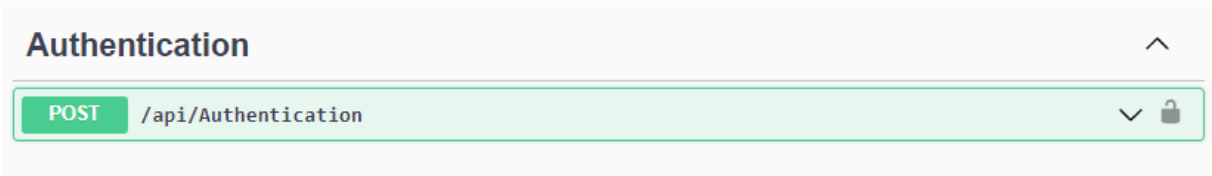
```
using MediatR;
using Microsoft.EntityFrameworkCore;
using WorkingHours.Domain.ViewModels;
using WorkingHours.Infrastructure;

namespace WorkingHours.Query.User.GetByUserEmailUser
{
    1 reference
    public class GetByUserEmailUserQueryHandler : IRequestHandler<GetByUserEmailUserQuery, UserViewModel>
    {
        public WorkingHoursContext context;
        0 references
        public GetByUserEmailUserQueryHandler(WorkingHoursContext context)
        {
            this.context = context;
        }

        0 references
        public async Task<UserViewModel> Handle(GetByUserEmailUserQuery request, CancellationToken cancellationToken)
        {
            if (context.Users is null)
            {
                throw new NullReferenceException();
            }

            var result = await context.Users.Where(u => u.Email == request.UserEmail)
                .Select(u => new UserViewModel
                {
                    Email = u.Email,
                    HashedPassword = u.HashedPassword,
                    FirstName = u.FirstName,
                    LastName = u.LastName,
                    RoleId = u.RoleId,
                    IsActive = u.IsActive,
                    YearlyRemainingDaysOff = u.YearlyRemainingDaysOff,
                    YearlyDaysOff = u.YearlyDaysOff,
                    PreferencesId = u.PreferencesId,
                    SuperiorEmail = u.SuperiorEmail
                })
                .AsNoTracking()
                .FirstOrDefaultAsync(cancellationToken);

            if (result is null)
            {
                throw new ArgumentException();
            }

            return result;
        }
    }
}
```

Izvor: Autor

5.4. Funkcionalnosti API-a

Funkcionalnosti pojedinog API-a definiraju njegove krajnje točke. U nastavku biti će prikazane sve krajnje točke za svaki entitet, odnosno cijeli API.

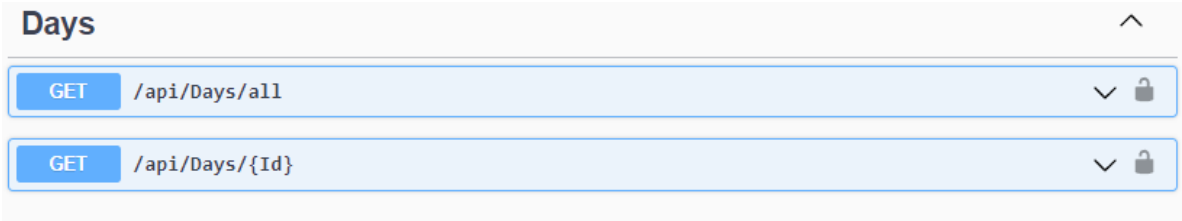
Slika 35 - Krajnja točka Authentication



Izvor: Autor

Na slici 35 prikazana je krajnja točka koja služi za autentifikaciju korisnika. Ona uzima dva parametra, korisnikov email i zaporku te vraća povratnu informaciju o uspješnosti autentifikacije.

Slika 36 - Krajnje točke za entitet Days



Izvor: Autor

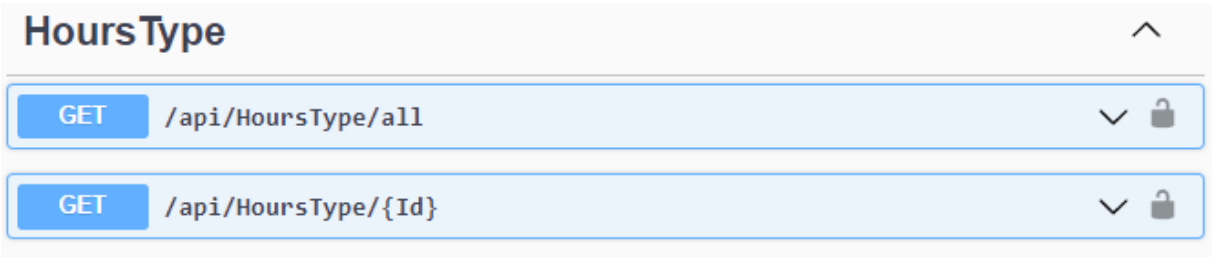
Na slici 36 prikazane su GET metode za entitet Days koje omogućuju dohvat svih zapisa spremljenih u toj relaciji ili jednog zapisa s određenim identifikatorom Id.

Slika 37 - Krajnje točke za entitet Hours

Izvor: Autor

Na slici 37 prikazane su GET metode za entitet Hours koje omogućuju dohvat svih zapisa spremljenih u toj relaciji, zapisa koji pripadaju određenom korisniku, zapisa koji su na nekom projektu i zapisa koji imaju određenu oznaku. Pomoću POST krajnje točke moguće je kreirati novi zapis radnog sata.

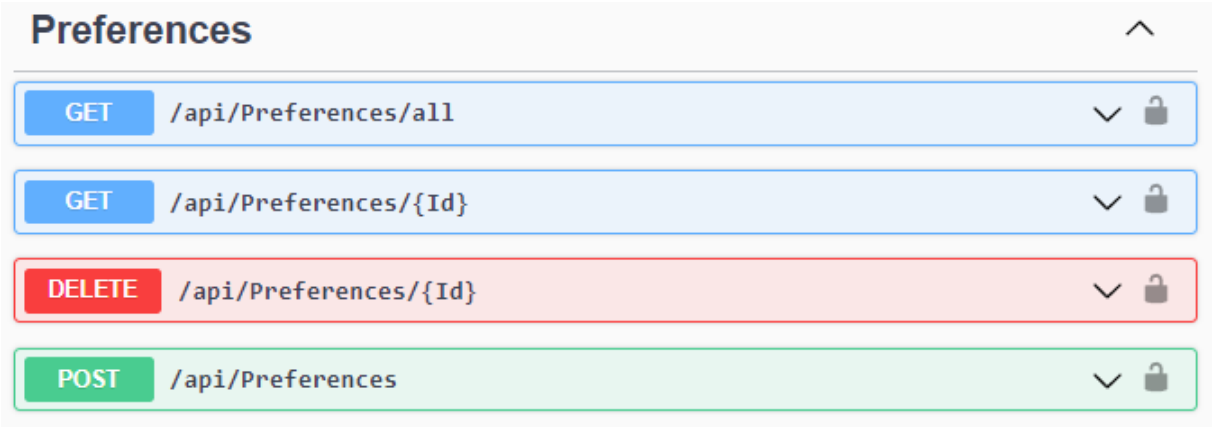
Slika 38 - Krajnje točke za entitet HoursType



Izvor: Autor

Na slici 38 prikazane su GET metode za entitet HoursType koje omogućuju dohvat svih zapisa spremljenih u toj relaciji ili jednog zapisa sa određenim identifikatorom Id.

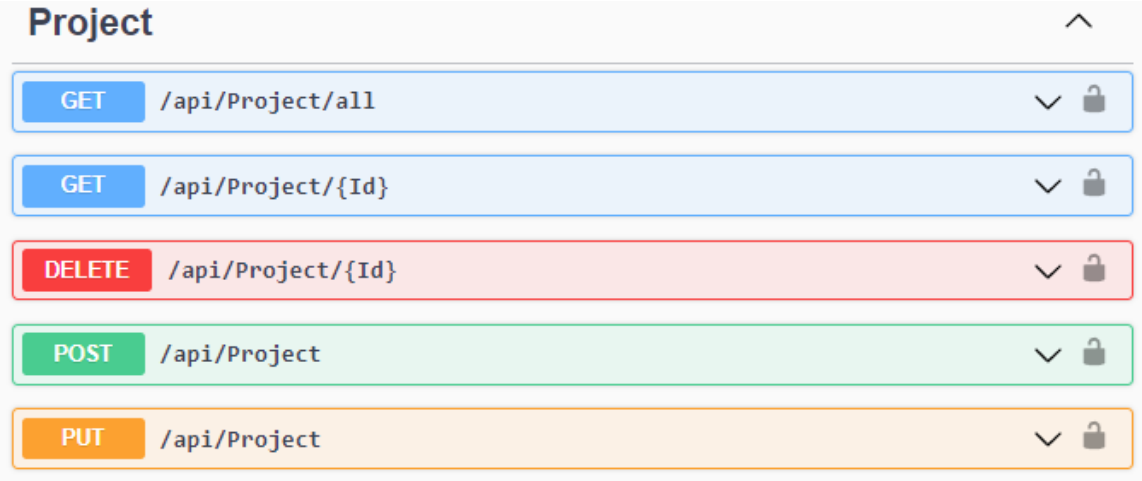
Slika 39 - Krajnje točke za entitet Preferences



Izvor: Autor

Na slici 39 prikazane su GET metode za entitet Preferences koje omogućuju dohvat svih zapisa spremljenih u toj relaciji ili jednog zapisa s određenim identifikatorom Id. Korištenjem DELETE metode moguće je izbrisati određeni zapis korištenjem identifikatora Id. Pomoću POST metode kreira se novi zapis preferenci.

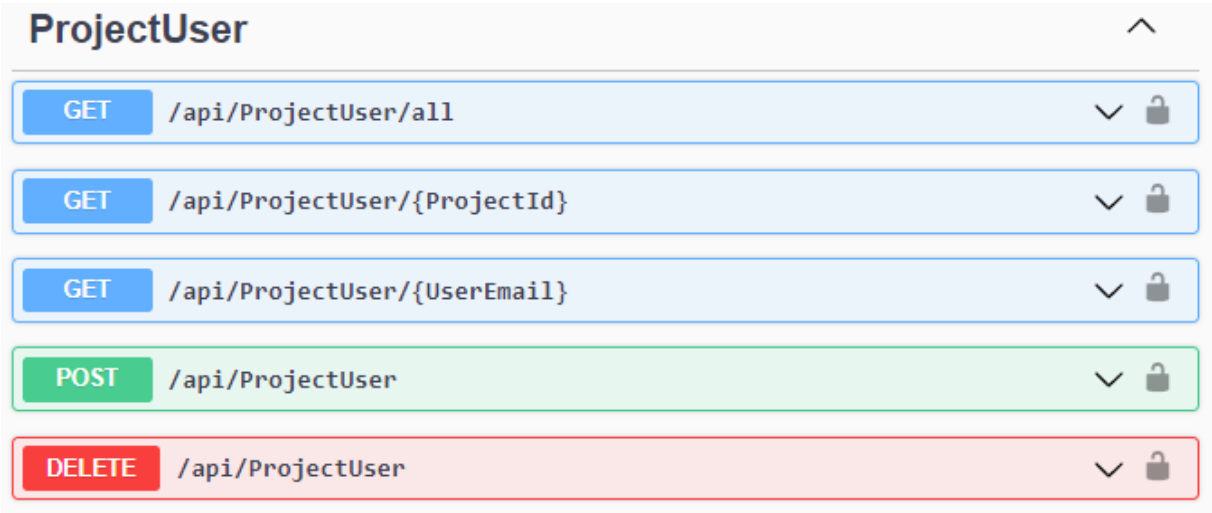
Slika 40 - Krajnje točke za entitet Project



Izvor: Autor

Na slici 40 prikazane su GET metode za entitet Project koje omogućuju dohvat svih zapisa spremljenih u toj relaciji ili jednog zapisa s određenim identifikatorom Id. Korištenjem DELETE metode moguće je izbrisati određeni zapis korištenjem identifikatora Id. Pomoću POST metode dodaje se novi projekt. Korištenjem PUT metode može se ažurirati neki zapis projekta.

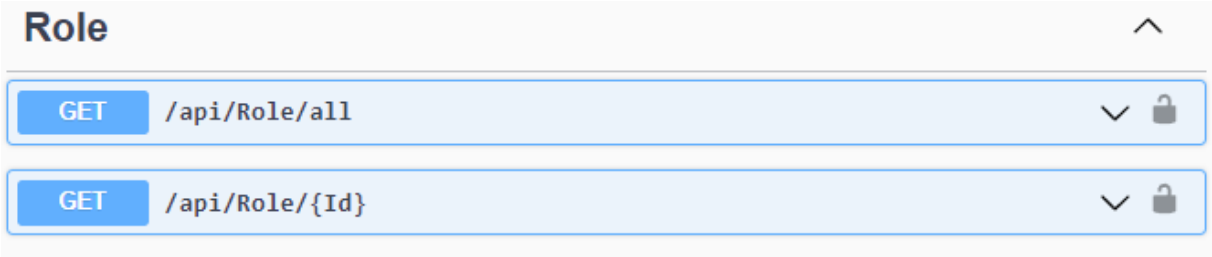
Slika 41 - Krajnje točke za entitet ProjectUser



Izvor: Autor

Na slici 41 prikazane su GET metode za entitet ProjectUser koje omogućuju dohvat svih zapisa spremljenih u toj relaciji ili više zapisa s određenim identifikatorom ProjectId ili UserEmail. Korištenjem DELETE metode moguće je izbrisati određeni zapis. Potrebno je specificirati ProjectId i UserEmail pošto se radi o agregaciji koja ima složeni ključ. Pomoću POST metode dodaje se određenog korisnik na projekt.

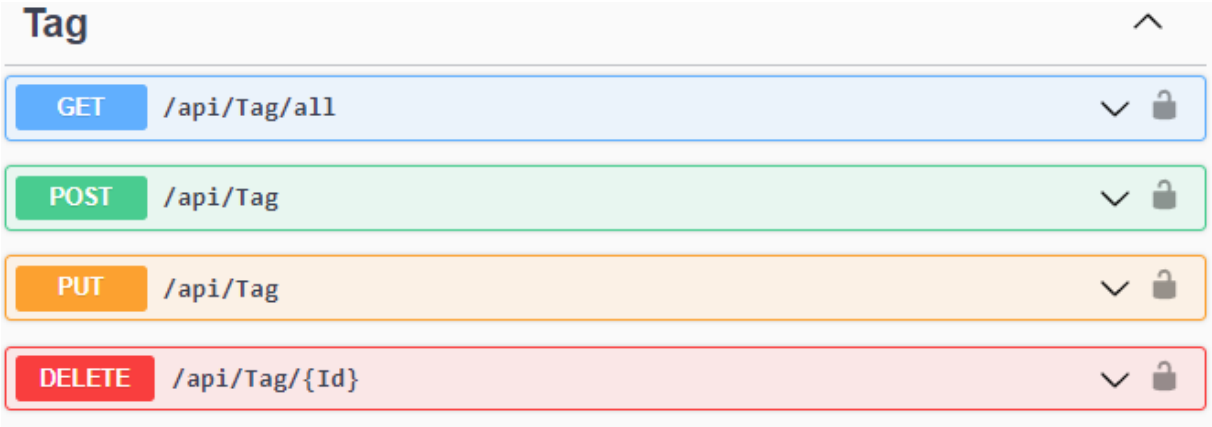
Slika 42 - Krajnje točke za entitet Role



Izvor: Autor

Na slici 42 prikazane su GET metode za entitet Role koje omogućuju dohvat svih zapisa spremljenih u toj relaciji ili jednog zapisa s određenim identifikatorom Id.

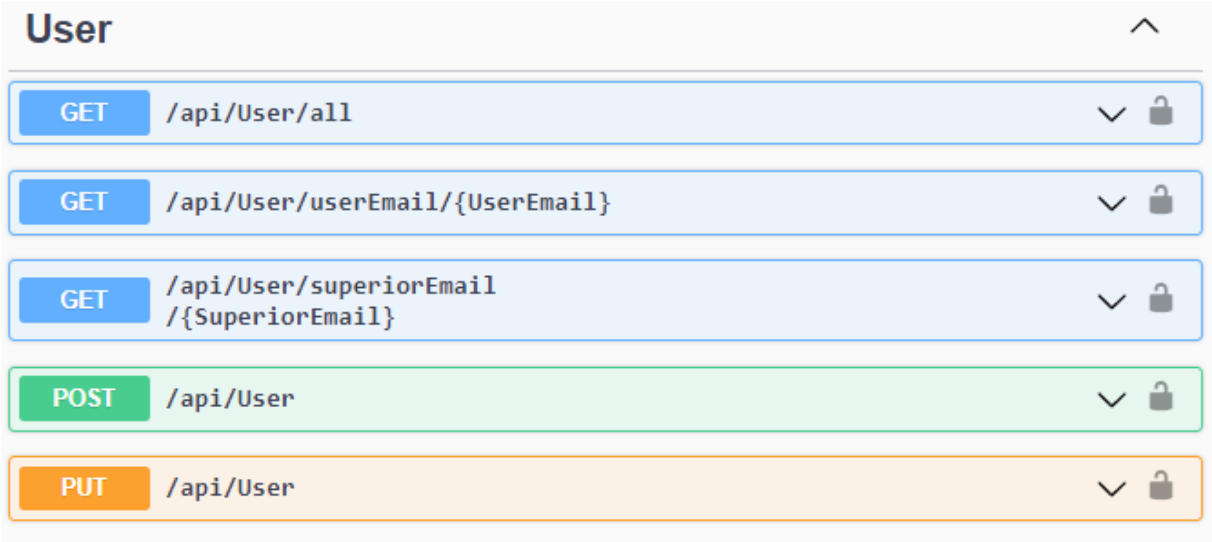
Slika 43 - Krajnje točke za entitet Tag



Izvor: Autor

Na slici 43 prikazana je GET metoda za entitet Tag koja omogućava dohvat svih zapisa spremljenih u toj relaciji. Pomoću POST metode moguće je kreirati novu oznaku. Korištenjem PUT metode može se ažurirati određenu oznaku. Putem DELETE metode moguće je izbrisati određeni zapis.

Slika 44 - Krajnje točke za entitet User



Izvor: Autor

Na slici 44 prikazane su GET metode za entitet User koje omogućavaju dohvat svih korisnika spremljenih u toj relaciji, samo određenog korisnika i svih korisnika koji imaju zajedničkog nadređenog. Pomoću POST metode moguće je kreirati novog korisnika. Korištenjem PUT metode može se ažurirati određenog korisnika.

6. Zaključak

Cilj izrade ovog rada bio je prikaz razvoja baze podataka koja će se u konačnici koristiti u poslovnim API-ima. Također, pridruženi cilj bio je i prezentacija razvojnog okvira .NET kao jednog modernog okvira na kojemu je moguće stvarati skalabilne, efikasne i robusne sustave za široku namjenu.

U današnjem svijetu podaci, odnosno informacije, su krucijalne kod obavljanja poslovnih aktivnosti. Podaci koji su zanimljivi nekom poslovnom sustavu stalno se kreiraju i bilježe. Svi ti podaci prije ili nakon obrade bit će smješteni unutar neke baze podataka. Dakle, može se zaključiti da je baza podataka jako bitan dio bilo kakvog programskog sustava. Stoga

pri projektiranju baze podataka ključan je dobar dizajn kako bi se osigurala sigurnost, efektivnost i brzina transakcija.

Svrha izrađenog API-a za praćenje radnih sati zaposlenika je njegovo postavljanje unutar domene poslovnog sustava kako bi na temelju svojih funkcionalnosti ubrzao i pojednostavnio bilježenje radnih sati zaposlenika. Sam API izrađen je u slojevima koristeći .NET 6 razvojni okvir, EF Core kao sredstvo interakcije s bazom podataka te MediatR programskom strukturom za definiranje kontrolera.

POPIS KRATICA

1. **CRUD** Create, Read, Update, Delete
2. **ASP** Microsoft® Active Server Pages
3. **EF** Entity Framework
4. **JSON** Javascript Object Notation
5. **SQL** Structured Query Language
6. **DBMS** Database Managment System
7. **API** Application Programming Interface
8. **CLI** Command Line Interface
9. **REST** Representational State Transfer
10. **HTTP** Hypertext Transfer Protocol
11. **ACID** Atomicity Consistency Isolation Durability
12. **XML** Extensible Markup Language
13. **DOM** Document Object Model
14. **ORM** Object Relational Mapper
15. **CQRS** Command Query Responsibility Segregation
16. **URL** Uniform Resource Location
17. **EVA** Entiteti Veze Atributi
18. **UTC** Universal Time Zone

POPIS SLIKA

POPIS LITERATURE

1. *What is REST*, https://www.codecademy.com/article/what-is-rest (3.5.2022.)
2. Bogard, J. (2022), *Mediatr github repository*, https://github.com/jbogard/MediatR (3.5.2022.)
3. Chand, M. (2022), *What is new in .NET 6.0*, https://www.c-sharpcorner.com/article/what-is-new-in-net-6-0/ *(3.5.2022.)*
4. Magner, R. (2010), *Osnove projektiranja baza podataka*, https://www.srce.unizg.hr/files/srce/docs/edu/osnovni-tecajevi/d310_polaznik.pdf (29.5.2022.)
5. Ian, (2016), *What does ACID mean in Database Systems*, https://database.guide/what-is-acid-in-databases/ (4.5.2022.)
6. Hoffman, C. (2021), *What is an API, and how do developers use them*, https://www.howtogeek.com/343877/what-is-an-api/ (4.5.2022.)
7. *Swagger*, (2019), https://www.techtarget.com/searchapparchitecture/definition/Swagger#:~:text=Swagger%20is%20an%20open%20source,and%20human%2Dreadable%20API%20documentation. (4.5.2022.)
8. (2021), *CQRS and MediatR in ASP.NET Core, https://code-maze.com/cqrs-mediatr-in-aspnet-core/* (4.5.2022.)
9. Microsoft docs, (2021), *Entity framework core*, https://docs.microsoft.com/en-us/ef/core/ (4.5.2022.)
10. Mozilla docs, (2022.), *An overview of HTTP*, https://developer.mozilla.org/en-US/docs/Web/HTTP/Overview (4.5.2022.)
11. Mozilla docs, (2022.), *HTTP response status codes*, https://developer.mozilla.org/en-US/docs/Web/HTTP/Status#redirection_messages
12. Microsoft docs, (2022.), *What is SQL Server Management Studio (SSMS)?* https://docs.microsoft.com/en-us/sql/ssms/sql-server-management-studio-ssms?view=sql-server-ver15 (6.5.2022.)
13. Smartbear, (2021.), *API Endpoints – What Are They? Why Do They Matter?* https://smartbear.com/learn/performance-monitoring/api-

endpoints/#:~:text=Simply%20put%2C%20an%20endpoint%20is,of%20a%20server%20or%20service (6.5.2022.)

14. Entity Framework Tutorial, (2020.), *Fluent API in Entity Framework Core* , https://www.entityframeworktutorial.net/efcore/fluent-api-in-entity-framework-core.aspx (6.5.2022.)
15. Microsoft docs, (2022.), *Description of the database normalization basics* https://docs.microsoft.com/en-us/office/troubleshoot/access/database-normalization-description (10.5.2022.)
16. IBM Cloud Education, (2021.), *REST APIs* https://www.ibm.com/cloud/learn/rest-apis (16.5.2022.)