Tema projekta

Projekat obrađuje temu bloga uz mogućnost unosa novih blogova i dodeljivanja kategorija, kao i pisanja samih komentara i podkomentara vezanih za blogove, i ostavljanje like-ova na blogove.

Korišćeni programski jezici i biblioteke

-C#  
-ASP .NET Core   
-FluentValidation   
-EntityFrameworkCore   
-JWT sistem autorizacije   
-JSON  
-MS SQLServer  
-Swagger

U projektu se oslanjam na višeslojnu arhitekturu. Svaki sloj komunicira sa ovim ispod. Na vrhu su klijentske aplikacije (API, MVC, Desktop). Ispod je poslovna logika. Ispod je sloj za pristup podacima (Data Access). Na dnu, kao fizičko skladište je baza podataka.

Redosled kojim implemeniram projekte je sledeći:

1. Domain
2. EFDataAccess
3. Application
4. Implementation
5. API

1. Treba imati sloj koji predstavlja centar sistema – to je njegov **Domen**. Sloj sa definicijama objekata i njihovim vezama. Postoji koncept ‚‚domain-driven design”. Izlistani su objekti koji predstavljaju tabele. On definiše osnovne klase, entitete bez kojih sistem ne može da radi. Ni u kom slučaju ne treba da zavise od framework-a ili alata. On je najnezavisniji deo sistema. (Npr. prodavnica ne može da postoji bez: kupca, prodavca, korpe, proizvoda).  
Unutar Solution-a (folder koji ima više projekata) pravim novi ‚‚Class Library .NETCore”.  
*dotnet new classlib -n Domain*Prvo pravim apstraktnu klasu ‚‚BaseEntity”. Ovo je apstraktna klasa koja sadrži osnove atribute koje će imati svaki entitet u aplikaciji. To uključuje ID entiteta, vreme kreiranja, vreme poslednje izmene, oznaku aktivnosti i oznaku obrisano. Svi ostali entiteti u aplikaciji će naslediti ovu klasu.

using System;  
using System.Collections.Generic;  
using System.Linq;  
using System.Threading.Tasks;

namespace Domain

{

  public abstract class BaseEntity

  {

    public int Id { get; set; }

    public DateTime CreatedAt { get; set; }

    public DateTime? ModifiedAt { get; set; }

    public bool IsActive { get; set; }

    public bool IsDeleted { get; set; }

    public DateTime? DeletedAt { get; set; }

  }

}

Zatim dodajem sve klase koje aplikacija treba da ima.

U C# i ASP.NET Core, korišćenje virtual i ICollection<T> zajedno sa HashSet<T> ima svoje prednosti u kontekstu ORM (Object-Relational Mapping) sistema kao što je Entity Framework Core, koji se često koristi za rad sa bazama podataka.

Sa **virtual** modifikatorom, Entity Framework Core može koristiti lazy loading. Ovo znači da se podaci iz kolekcije neće učitati sve dok se ne zahteva pristup tim podacima.

**ICollection<T>** je generički interfejs koji predstavlja kolekciju objekata.  
**HashSet<T>** je implementacija ICollection<T> interfejsa koja garantuje jedinstvenost elemenata u kolekciji. To znači da svaki element u kolekciji mora biti jedinstven.  
Kod kolekcije stavljam new HashSet<T> da izbegnem null ref exception.

using System;

using System.Collections.Generic;

using System.Linq;

using System.Threading.Tasks;

namespace Domain

{

    public class Category : BaseEntity

    {

        public string Name { get; set; }

        public virtual ICollection<BlogCategory> CategoryBlogs { get; set; } = new HashSet<BlogCategory>();

    }

}

using System;

using System.Collections.Generic;

using System.Linq;

using System.Threading.Tasks;

namespace Domain

{

    public class Image : BaseEntity

    {

        public string ImagePath { get; set; }

        public virtual ICollection<Blog> Blogs { get; set; } = new HashSet<Blog>();

    }

}

using System;

using System.Collections.Generic;

using System.Linq;

using System.Threading.Tasks;

namespace Domain

{

    public class User : BaseEntity

    {

        public string FirstName { get; set; }

        public string LastName { get; set; }

        public string Username { get; set; }

        public string Password { get; set; }

        public string Email { get; set; }

        public virtual ICollection<Blog> Blogs { get; set; } = new HashSet<Blog>();

        public virtual ICollection<Comment> Comments { get; set; } = new HashSet<Comment>();

        public virtual ICollection<Like> Likes { get; set; } = new HashSet<Like>();

        public virtual ICollection<UserUseCase> UserUseCases { get; set; } = new HashSet<UserUseCase>();

    }

}

using System;

using System.Collections.Generic;

using System.Linq;

using System.Threading.Tasks;

namespace Domain

{

    public class Blog : BaseEntity

    {

        public string Title { get; set; }

        public string Content { get; set; }

        public int IdUser { get; set; }

        public int IdImage { get; set; }

        public virtual User User { get; set; }

        public virtual Image Image { get; set; }

        public virtual ICollection<BlogCategory> BlogCategories { get; set; } = new HashSet<BlogCategory>();

        public virtual ICollection<Comment> Comments { get; set; } = new HashSet<Comment>();

        public virtual ICollection<Like> Likes { get; set; } = new HashSet<Like>();

    }

}

using System;

using System.Collections.Generic;

using System.Linq;

using System.Threading.Tasks;

namespace Domain

{

    public class BlogCategory

    {

        public int IdBlog { get; set; }

        public int IdCategory { get; set; }

        public virtual Blog Blog { get; set; }

        public virtual Category Category { get; set; }

        public bool IsDeleted { get; set; }

        public DateTime? DeletedAt { get; set; }

        public bool IsActive { get; set; }

    }

}

BlogsCategories tabelu ne izvodim iz BaseEntity jer ima kompozitni ključ.  
Ima reference ka blogu i kategoriji.  
*public Blog Blog {get; set;}  
public Category Category {get;set;}*Ovo je 1:više sa dve strane, pravim po kolekciju sa obe strane   
(u veznoj i category / u veznoj i blog).

using System;

using System.Collections.Generic;

using System.Linq;

using System.Threading.Tasks;

namespace Domain

{

    public class Like : BaseEntity

    {

        public int IdUser { get; set; }

        public int IdBlog { get; set; }

        public LikeStatus Status { get; set; }

        public virtual User User { get; set; }

        public virtual Blog Blog { get; set; }

    }

    public enum LikeStatus

    {

        Liked = 1,

        Disliked = 2,

        Null = 3

    }

}

using System;

using System.Collections.Generic;

using System.Linq;

using System.Threading.Tasks;

namespace Domain

{

    public class Comment : BaseEntity

    {

        public string CommentText { get; set; }

        public int IdUser { get; set; }

        public int IdBlog { get; set; }

        public int? IdParent { get; set; }

        public virtual User User { get; set; }

        public virtual Blog Blog { get; set; }

        public virtual Comment ParentComment { get; set; }

        public virtual ICollection<Comment> ChildrenComments { get; set; } = new HashSet<Comment>();

    }

}

**IdParent**: Ovo je opcioni ID roditeljskog komentara. Ako je komentar odgovor na neki drugi komentar (podkomentar), ovde će biti upisan ID roditeljskog komentara.  
  
**ParentComment**: Ovo je navigaciono svojstvo koje predstavlja roditeljski komentar (ako postoji). Ako je komentar podkomentar, ovo će biti referenca na roditeljski komentar.  
**ChildrenComments**: Ovo je kolekcija podkomentara koji su vezani za ovaj komentar. Ako ovaj komentar ima podkomentare, oni će biti dostupni putem ove kolekcije.

Na primer, ako korisnik napravi komentar direktno ispod blog članka, IdParent bi bio NULL jer taj komentar nema roditeljski komentar. Ako neko odgovori na taj komentar, stvarajući podkomentar, IdParent bi sadržao ID originalnog komentara, a ParentComment bi bio referenca na taj originalni komentar.

Ovo omogućava organizaciju komentara u hijerarhijsku strukturu, gde jedan komentar može imati više odgovora (podkomentara), kao što se može videti u mnogim blog platformama i društvenim mrežama.

Komentar 1

Podkomentar 1.1

Podkomentar 1.1.1

Podkomentar 1.2

Komentar 2

Komentar 3

using System;

using System.Collections.Generic;

using System.Linq;

using System.Threading.Tasks;

namespace Domain

{

    public class UseCaseLog

    {

        public int Id { get; set; }

        public DateTime Date { get; set; }

        public string UseCaseName { get; set; }

        public string Data { get; set; }

        public string Actor { get; set; }

    }

}

Ovaj entitet predstavlja logovanje za use case-ove u aplikaciji.

Id: Identifikator loga.  
Date: Datum i vreme kada se dogodio use case.  
UseCaseName: Naziv upotrebljenog use case-a.  
Data: Podaci koji se odnose na use case.  
Actor: Korisnik koji je izvršio use case.

using System;

using System.Collections.Generic;

using System.Linq;

using System.Threading.Tasks;

namespace Domain

{

    public class UserUseCase : BaseEntity

    {

        public int IdUser { get; set; }

        public int IdUseCase { get; set; }

        public virtual User User { get; set; }

    }

}

Ovaj entitet predstavlja vezu između korisnika i use case-a. To se koristi za kontrolu pristupa ili autorizaciju, gde se beleži koji korisnik ima pristup određenim use case-ovima.

2. **EFDataAccess** je projekat koji se koristi za pristup podacima kroz Entity Framework Core. Ovde se definiše **DbContext** klasa, koja je glavna klasa za interakciju sa bazom podataka kroz Entity Framework Core. Ova klasa sadrži svojstva DbSet-a za svaki entitet, koja predstavljaju tabele u bazi podataka. Odvaja pristup podacima od ostatka aplikacije.

Koristi se **‚‚Code first”** pristup – Kreće se od koda, naprave se sve potrebne klase i njihove međusobne konekcije i jednom kada napravimo te klase pustićemo da nam alat od koda napravi celu bazu podataka.

*dotnet new classlib -n EFDataAccess*EFDataAccess.csproj

<Project Sdk="Microsoft.NET.Sdk">

  <PropertyGroup>

    <TargetFramework>net8.0</TargetFramework>

    <ImplicitUsings>enable</ImplicitUsings>

    <Nullable>enable</Nullable>

  </PropertyGroup>

  <ItemGroup>

    <PackageReference Include="Microsoft.EntityFrameworkCore" Version="8.0.2" />

    <PackageReference Include="Microsoft.EntityFrameworkCore.Design" Version="8.0.2">

      <IncludeAssets>runtime; build; native; contentfiles; analyzers; buildtransitive</IncludeAssets>

      <PrivateAssets>all</PrivateAssets>

    </PackageReference>

    <PackageReference Include="Microsoft.EntityFrameworkCore.SqlServer" Version="8.0.2" />

    <PackageReference Include="Microsoft.EntityFrameworkCore.Tools" Version="8.0.2">

      <IncludeAssets>runtime; build; native; contentfiles; analyzers; buildtransitive</IncludeAssets>

      <PrivateAssets>all</PrivateAssets>

    </PackageReference>

  </ItemGroup>

  <ItemGroup>

    <ProjectReference Include="..\Domain\Domain.csproj" />

  </ItemGroup>

</Project>

XML konfiguracija za projekat EFDataAccess.csproj koji definiše zavisnosti i podešavanja za projekat.  
  
ItemGroup sa PackageReference:  
Ovde se definišu paketi koji su potrebni za rad projekta. Svaki PackageReference definiše jedan NuGet paket koji je potreban.

ItemGroup sa ProjectReference:  
Ovde se definišu reference na druge projekte u rešenju. U ovom slučaju, postoji referenca na Domain projekat, što znači da EFDataAccess projekat zavisi od Domain projekta i može koristiti njegove entitete i druge resurse.

Pisanje **Configuration** foldera u Entity Framework Core omogućava mi da konfigurišem kako će se entiteti mapirati na tabele u bazi podataka. Ove konfiguracije se koriste za definisanje ključeva, ograničenja, veza između entiteta i drugih detalja vezanih za mapiranje entiteta u bazu podataka.

U **Configure()** metodi se definišu različita svojstva entiteta, kao što su ograničenja dužine **HasMaxLength()**.

**IsRequired()** govori Entity Framework-u da je to svojstvo obavezno, odnosno, ne sme biti NULL u bazi podataka.  
**IsUnique()** dodaje ograničenje na indeks kako bi se osigurala jedinstvenost vrednosti u tom polju.  
**.OnDelete()  
DeleteBehavior** enumeracija definiše mogućnosti ponašanja prilikom brisanja povezanog entiteta. **NoAction** znači da se ne preduzima nikakva akcija prilikom brisanja roditeljskog entiteta. To znači da neće biti automatskog brisanja ili kaskadnog brisanja povezanih entiteta kada se obriše entitet.  
**DeleteBehavior.Restrict** zabranjuje brisanje roditeljskog entiteta ako postoje povezani podaci, dok **DeleteBehavior.Cascade** omogućava kaskadno brisanje povezanih podataka prilikom brisanja roditeljskog entite.

Takođe se definišu veze koristeći metode **HasMany()** i **WithOne()** kako bi se konfigurisali odgovarajući ključevi i veze.

Ove konfiguracije se koriste prilikom izgradnje modela u DbContext-u, gde se primenjuju na entitete kako bi se odredile njihove karakteristike u bazi podataka. Korišćenje ovih konfiguracija omogućava vam preciznu kontrolu nad mapiranjem entiteta i definisanje složenih veza i ograničenja u bazi podataka.

using System;

using System.Collections.Generic;

using System.Linq;

using System.Threading.Tasks;

using Domain;

using Microsoft.EntityFrameworkCore;

using Microsoft.EntityFrameworkCore.Metadata.Builders;

namespace EFDataAccess.Configurations

{

    public class CategoryConfiguration : IEntityTypeConfiguration<Category>

    {

        public void Configure(EntityTypeBuilder<Category> builder)

        {

            builder.Property(x => x.Name).IsRequired().HasMaxLength(70);

            builder.HasMany(x => x.CategoryBlogs)

                   .WithOne(y => y.Category)

                   .HasForeignKey(x => x.IdCategory)

                   .OnDelete(DeleteBehavior.Restrict);

        }

    }

}

using System;

using System.Collections.Generic;

using System.Linq;

using System.Threading.Tasks;

using Domain;

using Microsoft.EntityFrameworkCore;

using Microsoft.EntityFrameworkCore.Metadata.Builders;

namespace EFDataAccess.Configurations

{

    public class UserConfiguration : IEntityTypeConfiguration<User>

    {

        public void Configure(EntityTypeBuilder<User> builder)

        {

            builder.Property(x => x.FirstName).IsRequired().HasMaxLength(100);

            builder.Property(x => x.LastName).IsRequired().HasMaxLength(100);

            builder.Property(x => x.Username).IsRequired().HasMaxLength(40);

            builder.Property(x => x.Password).IsRequired();

            builder.Property(x => x.Email).IsRequired().HasMaxLength(100);

            builder.HasIndex(x => x.Email).IsUnique();

            builder.HasIndex(x => x.Username).IsUnique();

            builder.HasMany(x => x.Blogs)

                   .WithOne(y => y.User)

                   .HasForeignKey(x => x.IdUser)

                   .OnDelete(DeleteBehavior.NoAction);

            builder.HasMany(x => x.Comments)

                .WithOne(y => y.User)

                .HasForeignKey(x => x.IdUser);

            builder.HasMany(x => x.Likes)

                .WithOne(y => y.User)

                .HasForeignKey(x => x.IdUser);

            builder.HasMany(x => x.UserUseCases)

                .WithOne(y => y.User)

                .HasForeignKey(x => x.IdUser);

        }

    }

}

using System;

using System.Collections.Generic;

using System.Linq;

using System.Threading.Tasks;

using Domain;

using Microsoft.EntityFrameworkCore;

using Microsoft.EntityFrameworkCore.Metadata.Builders;

namespace EFDataAccess.Configurations

{

    public class ImageConfiguration : IEntityTypeConfiguration<Image>

    {

        public void Configure(EntityTypeBuilder<Image> builder)

        {

            builder.Property(x => x.ImagePath).IsRequired();

            builder.HasMany(x => x.Blogs)

                   .WithOne(y => y.Image)

                   .HasForeignKey(x => x.IdImage)

                   .OnDelete(DeleteBehavior.Restrict);

        }

    }

}

using System;

using System.Collections.Generic;

using System.Linq;

using System.Threading.Tasks;

using Domain;

using Microsoft.EntityFrameworkCore;

using Microsoft.EntityFrameworkCore.Metadata.Builders;

namespace EFDataAccess.Configurations

{

    public class CommentConfiguration : IEntityTypeConfiguration<Comment>

    {

        public void Configure(EntityTypeBuilder<Comment> builder)

        {

            builder.Property(x => x.CommentText).IsRequired();

            builder.HasMany(x => x.ChildrenComments)

                .WithOne(x => x.ParentComment)

                .HasForeignKey(x => x.IdParent)

                .OnDelete(DeleteBehavior.NoAction);

        }

    }

}

using System;

using System.Collections.Generic;

using System.Text;

using Domain;

using Microsoft.EntityFrameworkCore;

using Microsoft.EntityFrameworkCore.Metadata.Builders;

namespace EFDataAccess.Configurations

{

    public class LikeConfiguration : IEntityTypeConfiguration<Like>

    {

        public void Configure(EntityTypeBuilder<Like> builder)

        {

            builder.Property(x => x.IdUser).IsRequired();

            builder.Property(x => x.IdBlog).IsRequired();

            builder.Property(x => x.Status).HasColumnType("bigint");

        }

    }

}

using System;

using System.Collections.Generic;

using System.Linq;

using System.Threading.Tasks;

using Domain;

using Microsoft.EntityFrameworkCore;

using Microsoft.EntityFrameworkCore.Metadata.Builders;

namespace EFDataAccess.Configurations

{

    public class BlogConfiguration : IEntityTypeConfiguration<Blog>

    {

        public void Configure(EntityTypeBuilder<Blog> builder)

        {

            builder.Property(x => x.Title).IsRequired().HasMaxLength(70);

            builder.Property(x => x.Content).IsRequired();

            builder.HasMany(x => x.BlogCategories)

                .WithOne(y => y.Blog)

                .HasForeignKey(x => x.IdBlog)

                .OnDelete(DeleteBehavior.Restrict);

            builder.HasMany(x => x.Comments)

                .WithOne(y => y.Blog)

                .HasForeignKey(x => x.IdBlog);

            builder.HasMany(x => x.Likes)

                .WithOne(y => y.Blog)

                .HasForeignKey(x => x.IdBlog);

        }

    }

}

using System;

using System.Collections.Generic;

using System.Linq;

using System.Threading.Tasks;

using Domain;

using Microsoft.EntityFrameworkCore;

using Microsoft.EntityFrameworkCore.Metadata.Builders;

namespace EFDataAccess.Configurations

{

    public class BlogCategoryConfiguration : IEntityTypeConfiguration<BlogCategory>

    {

        public void Configure(EntityTypeBuilder<BlogCategory> builder)

        {

            builder.HasKey(x => new { x.IdBlog, x.IdCategory });

        }

    }

}

Ova klasa implementira IEntityTypeConfiguration<BlogCategory> interfejs kako bi se konfigurisao entitet BlogCategory.

U Configure metodi se definiše primarni ključ (**HasKey**) za entitet BlogCategory, koristeći kombinaciju svojstava IdBlog i IdCategory.

using System;

using Microsoft.EntityFrameworkCore;

using Domain;

using EFDataAccess.Configurations;

namespace EFDataAccess

{

  public class BlogContext : DbContext

  {

    protected override void OnModelCreating(ModelBuilder modelBuilder)

    {

      modelBuilder.ApplyConfiguration(new BlogConfiguration());

      modelBuilder.ApplyConfiguration(new CategoryConfiguration());

      modelBuilder.ApplyConfiguration(new BlogCategoryConfiguration());

      modelBuilder.ApplyConfiguration(new ImageConfiguration());

      modelBuilder.ApplyConfiguration(new UserConfiguration());

      modelBuilder.ApplyConfiguration(new CommentConfiguration());

      modelBuilder.ApplyConfiguration(new LikeConfiguration());

      modelBuilder.Entity<Blog>().HasQueryFilter(x => !x.IsDeleted);

      modelBuilder.Entity<Category>().HasQueryFilter(x => !x.IsDeleted);

      modelBuilder.Entity<BlogCategory>().HasQueryFilter(x => !x.IsDeleted);

      modelBuilder.Entity<Image>().HasQueryFilter(x => !x.IsDeleted);

      modelBuilder.Entity<User>().HasQueryFilter(x => !x.IsDeleted);

      modelBuilder.Entity<Comment>().HasQueryFilter(x => !x.IsDeleted);

      modelBuilder.Entity<Like>().HasQueryFilter(x => !x.IsDeleted);

      modelBuilder.Entity<UserUseCase>().HasQueryFilter(x => !x.IsDeleted);

    }

    protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)

    {

      optionsBuilder.UseSqlServer(@"Data Source=DESKTOP-V80TVTT\SQLEXPRESS;Initial Catalog=blog;Integrated Security=True;Connect Timeout=60;Encrypt=False;TrustServerCertificate=False;ApplicationIntent=ReadWrite;MultiSubnetFailover=False");

    }

    public override int SaveChanges()

    {

      foreach (var entry in ChangeTracker.Entries())

      {

        if (entry.Entity is BaseEntity e)

        {

          switch (entry.State)

          {

            case EntityState.Added:

              e.CreatedAt = DateTime.Now;

              e.IsActive = true;

              e.IsDeleted = false;

              e.DeletedAt = null;

              e.ModifiedAt = null;

              break;

            case EntityState.Modified:

              e.ModifiedAt = DateTime.Now;

              break;

          }

        }

      }

      return base.SaveChanges();

    }

    public DbSet<Blog> Blogs { get; set; }

    public DbSet<Category> Categories { get; set; }

    public DbSet<BlogCategory> BlogCategories { get; set; }

    public DbSet<Image> Images { get; set; }

    public DbSet<User> Users { get; set; }

    public DbSet<Comment> Comments { get; set; }

    public DbSet<Like> Likes { get; set; }

    public DbSet<UserUseCase> UserUseCases { get; set; }

    public DbSet<UseCaseLog> UseCaseLogs { get; set; }

  }

}

Seeding podataka – **OnModelCreating(ModelBuilder modelBuilder)** je virtualna metoda gde dodajem incijalne podatke, bazni početni podaci kao preduslov funkcionisanja aplikacije.  
**ApplyConfiguration** - primenjivanje konfiguracije na bazu.

*modelBuilder.Entity<Blog>().HasQueryFilter(x => !x.IsDeleted);* Ova linija definiše globalni filter upita za entitet Blog u DbContext-  
Globalni filter upita je mehanizam koji omogućava da se automatski primeni filter prilikom izvršavanja upita nad određenim entitetom, čime se ograničava skup podataka koji se vraća iz baze podataka. Konkretno, izraz *x => !x.IsDeleted* definiše uslov za filter. U ovom slučaju, uslov kaže da će se samo neizbrisani postovi uzimati u obzir prilikom izvršavanja upita.  
Ovo je korisno za implementaciju ‘soft delete‘ u bazi podataka.

**OnConfiguring(DbContextOptionsBuilder optionsBuilder)** - override znači da je već definisan u klasi DbContext, ali mi redefinišemo njegovo ponšanje.   
Da bi mogao da se odradi override, on je označen kao virtual što znači da može da radi bez redefinisanja, a ako je abstract ne može da radi bez redefinisanja.

Prvo ulazim u SQL managment studio, konektujem se, kopiram server name u server expoler-u za dodavanje konekcije i izaberem Blog bazu podataka i kopiram connection string. Dodajem @ ispred stringa.

Dao sam mu instrukciju koji je tip baze podataka (koji SQL provider) i prosledio sam connection string.

Kako da napravim mehanizam da ne ponavljam kod za CRUD operacije?  
Na osnovu čega EntityFramework, kada pozovem **SaveChanges()**, zna da li treba da radi Insert, Update ili Delete?  
Na koji način DbContext zna koji objekat ide na koju operaciju?  
ChangeTracker - prati stanje objekta.

Redefinisaću metod SaveChanges().

Prilikom poziva SaveChanges() pristupa se svakom pozadinskom objektu i ako je taj objekat spreman da ide na insert postavljam njegove početne vrednosti.  
Na primer, postavlja se CreatedAt atribut na trenutno vreme kada se entitet doda, i ModifiedAt atribut na trenutno vreme kada se entitet izmeni.

Da bi klasa bila tabela treba da napravimo polje specifičnog tipa, a taj tip je **DbSet<>,** između treba da bude ime klase koja će predstavljati našu tabelu.

*public DbSet<Blog> Blogs { get; set; }* // tabela će se zvati Blogs

Svako DbSet svojstvo predstavlja entitet koji je definisan u domenskom sloju.

Ovaj kod definiše kako će se entiteti mapirati na tabele u bazi podataka, kako će se konfigurisati veze između njih i kako će se vršiti automatsko postavljanje određenih atributa entiteta prilikom dodavanja ili izmene. Ovo omogućava lak rad sa bazom podataka kroz Entity Framework Core.

Kada je sve spremno treba da primenim migracije.  
*dotnet ef migrations add initial migration*Analizirati migraciju, ako je sve dobro onda radim: *dotnet ef database update*

3. **Application** sloj je mesto gde postavljam logiku aplikacije i poslovna pravila. Definiše šta je naša aplikacija sposobna da uradi, ali ne i način (definiše apstrakcije, interfejse koje će ostatak aplikacije implementirati).  
Application sloj zavisi od Domain projekta i može koristiti njegove entitete i druge resurse. Application sloj ne treba da ima referencu ka DataAccess-u. Ponašanje sistema ne zavisi od eksterne biblioteke.  
Interfejsom definišem metod, kažem šta može da uradi, ali ne na koji način, to ostavljam njegovim podklasama i ovaj deo zovemo Application.  
Ovde delim sistem na 2 vrste akcija. Korisnici sistema će pokušati da promene stanje stistema (commands) ili tražiti nešto od našeg sistema (queries).  
Ovaj princip se zove **CQS (Command Query Separation)**. I komanda i query su use case-ovi. Cilj je da imam zajednički nadtip za sve komande i zajednički nadtip za sve query-e.

using System;

using System.Collections.Generic;

using System.Linq;

using System.Threading.Tasks;

namespace Application

{

    public interface IUseCase

    {

        int Id { get; }

        string Name { get; }

    }

*// Ovaj generički parametar definiše šta je od podataka neophodno da bi mogli da izvršimo komandu. Ukazuje na klasu ili tip podatka. Uvek nešto menja u sistemu, nikad ne vraćamo ništa – void.*

    public interface ICommand<TRequest> : IUseCase

    {

        void Execute(TRequest request);

    }

*// Upit može da ima ulazne parametre, a izlaz svakako ima. Imaće 2 generička tipa (1 objekat za potencijalnu pretragu).Prvi generički tip ukazuje šta je rezultat pretrage, a drugi koji je ulazni parametar. Ne menja ništa u sistemu, vraćamo klijentu rezultat kroz sistem.*

    public interface IQuery<TResponse, TSearch> : IUseCase

    {

        TResponse Execute(TSearch search);

    }

}

Kad organizujemo sistem, treba nam da možemo da logujemo svaki query koji je iko ikada pokušao da uradi. Ali ne tražimo svaku liniju u kontroleru.   
Definišemo interfejs **IUseCase**. Svaki slučaj korišćenja će imati svoj Id i Name. Upotrebom ovog interfejsa na nivou interakcije sa korisnikom pravim automatizam da zabranim izvršavanje nekog use case-a. Napraviću tabelu UserUseCases koja će imati IdUser u sebi i onda ću po broju znati doslovno svaki use case koji korisnik sme da izvrši.  
Obezbeđujem autorizaciju gde jednim ID-em na jednom mestu za svaku moguću aplikaciju u sistemu proveravamo da li trenutno ulogovani korisnik sme ili ne sme da izvrši komandu.

using System;

using System.Collections.Generic;

using System.Linq;

using System.Threading.Tasks;

namespace Application

{

    public interface IApplicationActor

    {

        int Id { get; }

        string Identity { get; }

        IEnumerable<int> AllowedUseCases { get; }

    }

}

Interfejs **IapplicationActor** će biti korisnik koga ćemo kasnije implementirati upotrebom sesija ili JWT-a. Da bismo nekoga smatrali actor-om mora imati Id i Identity(neka svaka aplikacija kaže da li to treba da bude ime ili nešto drugo).  
  
Ključno da bismo napravili automatizam u smislu izvršavanja use case-ova jeste da imamo 1 niz integer-a **AllowedUseCases** (niz brojeva koji označavaju šta trenutni korisnik u aplikaciji sme da izvrši).   
Primer: Korisnik sa Id-em 2 sme da izvrši use case 4(kreiranje komentara), a ako pokuša da izvrši use case 7(kreiranje korisnika) onda sistem treba da ga odbije.

using System;

using System.Collections.Generic;

using System.Linq;

using System.Threading.Tasks;

namespace Application

{

    public interface IUseCaseLogger

    {

        void Log(IUseCase useCase, IApplicationActor actor, object data);

    }

}

Interfejs **IuseCaseLogger** definiše metod Log koji služi za beleženje informacija u tabelu UseCaseLogs o izvršenim slučajevima korićenja (use case) u aplikaciji.

using System;

using System.Collections.Generic;

using System.Linq;

using System.Threading.Tasks;

using Application.Exceptions;

namespace Application

{

    public class UseCaseExecutor

    {

        private readonly IApplicationActor \_actor;

        private readonly IUseCaseLogger \_logger;

        public UseCaseExecutor(IApplicationActor actor, IUseCaseLogger logger)

        {

            \_actor = actor;

            \_logger = logger;

        }

        public void ExecuteCommand<TRequest>(ICommand<TRequest> command, TRequest data)

        {

            \_logger.Log(command, \_actor, data);

            if (!\_actor.AllowedUseCases.Contains(command.Id))

            {

                throw new UnauthorizedUseCaseException(command, \_actor);

            }

            command.Execute(data);

        }

        public TResponse ExecuteQuery<TResponse, TSearch>(IQuery<TResponse, TSearch> query, TSearch data)

        {

            \_logger.Log(query, \_actor, data);

            if (!\_actor.AllowedUseCases.Contains(query.Id))

            {

                throw new UnauthorizedUseCaseException(query, \_actor);

            }

            return query.Execute(data);

        }

    }

}

Ovaj objekat koji izvršava za nas query-e u stvari prenosi rezultat tako da njega koristimo kao Middleware između API-a i pozadinske poslovne logike.

**ExecuteCommand<TRequest>:** metod za izvršavanje komandi u aplikaciji. Prima generički parmetar <TRequest> koji predstavlja tip podataka koji se očekuje kao ulazni parametar komande.  
**ICommand<TRequest>:** Ovo je interfejs koji predstavlja komandu koja se izvršava. Komanda treba da bude implementacija interfejsa ICommand<TRequest>.  
Metod prvo loguje izvrašvanje komande, zatim proverava da li aktor ima dozvolu za izvršavanje date komande. Ako aktor nema dozvolu baca se izuzetak UnauthorizedUseCaseException. Ako korisnik ima dozvolu, izvršava se komanda pozivom metoda **Execute()** prosleđujući joj odgovarajuće podatke.

**ExecuteQuery<TResponse, TSearch>:** metod za izvršavanje upita u aplikaciji. Prima dva generička parametra TResponse, koji predstavlja tip podatka koji se očekuje kao rezultat upita, i TSearch, koji predstavlja tip podatka koji se očekuje kao ulazni parametar upita. IQuery<TResponse, TSearch> query: Interfejs koji predstavlja upit koji se izvršava. Upit treba da bude implementacija interfejsa IQuery<TResponse, TSearch>.  
**TSearch data**: Ovo su konkretne informacije(podaci) koji se prosleđuju kao ulazni parametar za izvršenje. Metod takođe loguje izvršavanje upita, proverava autorizaciju za izvršenje upita I na kraju vraća rezultat upita pozivom metoda Execute().

using System;

using System.Collections.Generic;

using System.Linq;

using System.Threading.Tasks;

namespace Application.Exceptions

{

    public class UnauthorizedUseCaseException : Exception

    {

*// base() - Poziv konstruktora roditeljske klase Exception*

        public UnauthorizedUseCaseException(IUseCase useCase, IApplicationActor actor)

            : base($"Actor with an ID of {actor.Id} - {actor.Identity} tried to execute {useCase.Name}")

        {

        }

    }

}

Pravim foldere **Commands, Queries, Searches i DataTransfer.**Pravljenje **DTO**-ova (Data Transfer Objects) u Application sloju ima nekoliko koristi i ciljeva.  
1. Prenos podataka izmeđuz različitih slojeva čime se izbegava direktno korišćenje domenskih modela ili entiteta iz baze.  
2. Omogućavaju očuvanje granularnosti i abstrakcije u aplikaciji.   
3. Aplikacija postaje fleksibilnija i manje zavisna od implementacija u drugim slojevima.

using System;

using System.Collections.Generic;

using System.Linq;

using System.Threading.Tasks;

namespace Application.DataTransfer

{

    public class CategoryDto

    {

        public int Id { get; set; }

        public string Name { get; set; }

    }

}

using System;

using System.Collections.Generic;

using System.Linq;

using System.Threading.Tasks;

using Microsoft.AspNetCore.Http;

namespace Application.DataTransfer

{

    public class UploadImageDto

    {

        public IFormFile Image { get; set; }

    }

}

using System;

using System.Collections.Generic;

using System.Linq;

using System.Threading.Tasks;

namespace Application.DataTransfer

{

    public class UserUseCaseDto

    {

        public int IdUseCase { get; set; }

    }

}

using System;

using System.Collections.Generic;

using System.Linq;

using System.Threading.Tasks;

namespace Application.DataTransfer

{

    public class UseCaseLogDto

    {

        public int Id { get; set; }

        public DateTime Date { get; set; }

        public string UseCaseName { get; set; }

        public string Data { get; set; }

        public string Actor { get; set; }

    }

}

using System;

using System.Collections.Generic;

using System.Linq;

using System.Threading.Tasks;

namespace Application.DataTransfer

{

    public class InsertUserDto

    {

        public string FirstName { get; set; }

        public string LastName { get; set; }

        public string Username { get; set; }

        public string Email { get; set; }

        public string Password { get; set; }

        public IEnumerable<UserUseCaseDto> UserUseCases { get; set; }

    }

}

using System;

using System.Collections.Generic;

using System.Linq;

using System.Threading.Tasks;

namespace Application.DataTransfer

{

    public class UpdateUserDto

    {

        public int Id { get; set; }

        public string FirstName { get; set; }

        public string LastName { get; set; }

        public string Username { get; set; }

        public string Password { get; set; }

        public List<int> UserUseCases { get; set; }

    }

}

using System;

using System.Collections.Generic;

using System.Linq;

using System.Threading.Tasks;

namespace Application.DataTransfer

{

    public class SingleUserDto

    {

        public int Id { get; set; }

        public string FirstName { get; set; }

        public string LastName { get; set; }

        public string Username { get; set; }

        public string Email { get; set; }

        public string Password { get; set; }

        public IEnumerable<UserUseCaseDto> UserUseCases { get; set; }

    }

}

using System;

using System.Collections.Generic;

using System.Linq;

using System.Threading.Tasks;

namespace Application.DataTransfer

{

    public class RegisterUserDto

    {

        public string FirstName { get; set; }

        public string LastName { get; set; }

        public string Username { get; set; }

        public string Email { get; set; }

        public string Password { get; set; }

    }

}

using System;

using System.Collections.Generic;

using System.Linq;

using System.Threading.Tasks;

namespace Application.DataTransfer

{

    public class LoginDto

    {

        public string Username { get; set; }

        public string Password { get; set; }

    }

}

using System;

using System.Collections.Generic;

using System.Linq;

using System.Threading.Tasks;

namespace Application.DataTransfer

{

    public class BlogCategoryDto

    {

        public int IdBlog { get; set; }

        public int IdCategory { get; set; }

    }

}

using System;

using System.Collections.Generic;

using System.Linq;

using System.Threading.Tasks;

namespace Application.DataTransfer

{

    public class InsertBlogDto

    {

        public string Title { get; set; }

        public string Content { get; set; }

        public int IdImage { get; set; }

        public IEnumerable<BlogCategoryDto> BlogCategories { get; set; }

    }

}

using System;

using System.Collections.Generic;

using System.Linq;

using System.Threading.Tasks;

namespace Application.DataTransfer

{

    public class UpdateBlogDto

    {

        public int Id { get; set; }

        public string Title { get; set; }

        public string Content { get; set; }

        public int IdImage { get; set; }

        public int IdUser { get; set; }

        public ICollection<int> BlogCategories { get; set; } = new List<int>();

    }

}

using System;

using System.Collections.Generic;

using System.Linq;

using System.Threading.Tasks;

namespace Application.DataTransfer

{

    public class SendEmailDto

    {

        public string Content { get; set; }

        public string Subject { get; set; }

        public string SendTo { get; set; }

    }

}

using System;

using System.Collections.Generic;

using System.Linq;

using System.Threading.Tasks;

namespace Application.DataTransfer

{

    public class InsertCommentDto

    {

        public int Id { get; set; }

        public string Text { get; set; }

        public int IdBlog { get; set; }

        public int? IdParent { get; set; }

        public int IdUser { get; set; }

    }

}

using System;

using System.Collections.Generic;

using System.Linq;

using System.Threading.Tasks;

namespace Application.DataTransfer

{

    public class SingleCommentDto

    {

        public int Id { get; set; }

        public string Text { get; set; }

        public int? IdParent { get; set; }

        public string Username { get; set; }

        public DateTime CreatedAt { get; set; }

        public IEnumerable<SingleCommentDto> Children { get; set; } = new List<SingleCommentDto>();

    }

}

using System;

using System.Collections.Generic;

using System.Linq;

using System.Threading.Tasks;

namespace Application.DataTransfer

{

    public class CommentDto

    {

        public int Id { get; set; }

        public string Comment { get; set; }

        public DateTime CreatedAt { get; set; }

        public string Username { get; set; }

        public int? IdParent { get; set; }

    }

}

using System;

using System.Collections.Generic;

using System.Linq;

using System.Threading.Tasks;

using Domain;

namespace Application.DataTransfer

{

    public class LikeDto

    {

        public int IdUser { get; set; }

        public int IdBlog { get; set; }

        public LikeStatus Status { get; set; }

    }

}

using System;

using System.Collections.Generic;

using System.Linq;

using System.Threading.Tasks;

using Domain;

namespace Application.DataTransfer

{

    public class LikeBlogDto

    {

        public int Id { get; set; }

        public LikeStatus Status { get; set; }

        public string Username { get; set; }

    }

}

using System;

using System.Collections.Generic;

using System.Linq;

using System.Threading.Tasks;

namespace Application.DataTransfer

{

    public class GetBlogDto

    {

        public int Id { get; set; }

        public string Title { get; set; }

        public string Content { get; set; }

        public DateTime DateCreated { get; set; }

        public string ImageName { get; set; }

        public int IdImage { get; set; }

        public string Username { get; set; }

        public IEnumerable<CategoryDto> Categories { get; set; } = new List<CategoryDto>();

        public IEnumerable<LikeBlogDto> Likes { get; set; } = new List<LikeBlogDto>();

        public IEnumerable<SingleCommentDto> Comments { get; set; } = new List<SingleCommentDto>();

    }

}

using System;

using System.Collections.Generic;

using System.Linq;

using System.Threading.Tasks;

using Application.DataTransfer;

namespace Application.Commands.Blog

{

    public interface ICreateBlogCommand : ICommand<InsertBlogDto>

    {

    }

}

using System;

using System.Collections.Generic;

using System.Linq;

using System.Threading.Tasks;

namespace Application.Commands.Blog

{

    public interface IDeleteBlogCommand : ICommand<int>

    {

    }

}

using System;

using System.Collections.Generic;

using System.Linq;

using System.Threading.Tasks;

namespace Application.Commands.Blog

{

    public interface IDeletePersonalBlogCommand : ICommand<int>

    {

    }

}

using System;

using System.Collections.Generic;

using System.Linq;

using System.Threading.Tasks;

using Application.DataTransfer;

namespace Application.Commands.Blog

{

    public interface IUpdateBlogCommand : ICommand<UpdateBlogDto>

    {

    }

}

using System;

using System.Collections.Generic;

using System.Linq;

using System.Threading.Tasks;

using Application.DataTransfer;

namespace Application.Commands.Blog

{

    public interface IUpdatePersonalBlogCommand : ICommand<UpdateBlogDto>

    {

    }

}

using System;

using System.Collections.Generic;

using System.Linq;

using System.Threading.Tasks;

using Application.DataTransfer;

namespace Application.Commands.Category

{

    public interface ICreateCategoryCommand : ICommand<CategoryDto>

    {

    }

}

using System;

using System.Collections.Generic;

using System.Linq;

using System.Threading.Tasks;

namespace Application.Commands.Category

{

    public interface IDeleteCategoryCommand : ICommand<int>

    {

    }

}

using System;

using System.Collections.Generic;

using System.Linq;

using System.Threading.Tasks;

using Application.DataTransfer;

namespace Application.Commands.Category

{

    public interface IUpdateCategoryCommand : ICommand<CategoryDto>

    {

    }

}

using System;

using System.Collections.Generic;

using System.Linq;

using System.Threading.Tasks;

using Application.DataTransfer;

namespace Application.Commands.Comment

{

    public interface ICreateCommentCommand : ICommand<InsertCommentDto>

    {

    }

}

using System;

using System.Collections.Generic;

using System.Linq;

using System.Threading.Tasks;

namespace Application.Commands.Comment

{

    public interface IDeleteCommentCommand : ICommand<int>

    {

    }

}

using System;

using System.Collections.Generic;

using System.Linq;

using System.Threading.Tasks;

namespace Application.Commands.Comment

{

    public interface IDeletePersonalCommentCommand : ICommand<int>

    {

    }

}

using System;

using System.Collections.Generic;

using System.Linq;

using System.Threading.Tasks;

using Application.DataTransfer;

namespace Application.Commands.Comment

{

    public interface IUpdatePersonalCommentCommand : ICommand<InsertCommentDto>

    {

    }

}

using System;

using System.Collections.Generic;

using System.Linq;

using System.Threading.Tasks;

using Application.DataTransfer;

namespace Application.Commands.Email

{

    public interface IEmailSender

    {

        void Send(SendEmailDto dto);

    }

}

using System;

using System.Collections.Generic;

using System.Linq;

using System.Threading.Tasks;

using Application.DataTransfer;

namespace Application.Commands.Like

{

    public interface ILikeBlogCommand : ICommand<LikeDto>

    {

    }

}

using System;

using System.Collections.Generic;

using System.Linq;

using System.Threading.Tasks;

using Application.DataTransfer;

namespace Application.Commands.User

{

    public interface ICreateUserCommand : ICommand<InsertUserDto>

    {

    }

}

using System;

using System.Collections.Generic;

using System.Linq;

using System.Threading.Tasks;

namespace Application.Commands.User

{

    public interface IDeleteUserCommand : ICommand<int>

    {

    }

}

using System;

using System.Collections.Generic;

using System.Linq;

using System.Threading.Tasks;

using Application.DataTransfer;

namespace Application.Commands.User

{

    public interface IRegisterUserCommand : ICommand<RegisterUserDto>

    {

    }

}

using System;

using System.Collections.Generic;

using System.Linq;

using System.Threading.Tasks;

using Application.DataTransfer;

namespace Application.Commands.User

{

    public interface IUpdateUserCommand : ICommand<UpdateUserDto>

    {

    }

}

using System;

using System.Collections.Generic;

using System.Linq;

using System.Threading.Tasks;

using Application.DataTransfer;

namespace Application.Queries.Blog

{

    public interface IGetBlogQuery : IQuery<GetBlogDto, int>

    {

    }

}

using System;

using System.Collections.Generic;

using System.Linq;

using System.Threading.Tasks;

using Application.DataTransfer;

using Application.Searches;

namespace Application.Queries.Blog

{

    public interface IGetBlogsQuery : IQuery<PagedResponse<GetBlogDto>, BlogSearch>

    {

    }

}

using System;

using System.Collections.Generic;

using System.Linq;

using System.Threading.Tasks;

using Application.DataTransfer;

using Application.Searches;

namespace Application.Queries.Category

{

    public interface IGetCategoriesQuery : IQuery<PagedResponse<CategoryDto>, CategorySearch>

    {

    }

}

using System;

using System.Collections.Generic;

using System.Linq;

using System.Threading.Tasks;

using Application.DataTransfer;

namespace Application.Queries.Category

{

    public interface IGetCategoryQuery : IQuery<CategoryDto, int>

    {

    }

}

using System;

using System.Collections.Generic;

using System.Linq;

using System.Threading.Tasks;

using Application.DataTransfer;

namespace Application.Queries.Comment

{

    public interface IGetCommentQuery : IQuery<CommentDto, int>

    {

    }

}

using System;

using System.Collections.Generic;

using System.Linq;

using System.Threading.Tasks;

using Application.DataTransfer;

namespace Application.Queries.User

{

    public interface IGetUserQuery : IQuery<SingleUserDto, int>

    {

    }

}

using System;

using System.Collections.Generic;

using System.Linq;

using System.Threading.Tasks;

using Application.DataTransfer;

using Application.Searches;

namespace Application.Queries.User

{

    public interface IGetUsersQuery : IQuery<PagedResponse<UserDto>, UserSearch>

    {

    }

}

using System;

using System.Collections.Generic;

using System.Linq;

using System.Threading.Tasks;

using Application.DataTransfer;

using Application.Searches;

namespace Application.Queries

{

    public interface IGetUseCaseLogsQuery : IQuery<PagedResponse<UseCaseLogDto>, UseCaseLogSearch>

    {

    }

}

using System;

using System.Collections.Generic;

using System.Linq;

using System.Threading.Tasks;

namespace Application.Queries

{

*// Parametar `T` predstavlja tip objekta koji će biti vraćen kao rezultat paginirane pretrage. `where T : class` specifira da mora biti referentni tip, odnosno klasa, a ne vrednosni tip(strukture)*

    public class PagedResponse<T> where T : class

    {

        public int TotalCount { get; set; } *// Nije ukupan broj zapisa koji vraćamo nazad, već je to ukupan broj u bazi koji odgovara kriterijumu za pretragu*

        public int ItemsPerPage { get; set; }

        public int CurrentPage { get; set; }

        public int PageCount => (int)Math.Ceiling((float)TotalCount / ItemsPerPage);

        public IEnumerable<T> Items { get; set; } = new List<T>

    }

}

**Searches** klase definišu kriterijume pretrage koje korisnik može da koristi kako bi filtrirao elemente.

using System;

using System.Collections.Generic;

using System.Linq;

using System.Threading.Tasks;

namespace Application.Searches

{

    public class BlogSearch : PagedSearch

    {

        public string Title { get; set; }

        public string Content { get; set; }

        public DateTime DateFrom { get; set; }

        public DateTime DateTo { get; set; }

        public int? IdCategory { get; set; }

    }

}

using System;

using System.Collections.Generic;

using System.Linq;

using System.Threading.Tasks;

namespace Application.Searches

{

    public class CategorySearch : PagedSearch

    {

        public string Name { get; set; }

    }

}

using System;

using System.Collections.Generic;

using System.Linq;

using System.Threading.Tasks;

namespace Application.Searches

{

    public abstract class PagedSearch

    {

        public int PerPage { get; set; } = 5;

        public int Page { get; set; } = 1;

    }

}

using System;

using System.Collections.Generic;

using System.Linq;

using System.Threading.Tasks;

namespace Application.Searches

{

    public class UseCaseLogSearch : PagedSearch

    {

        public string Actor { get; set; }

        public string UseCaseName { get; set; }

        public DateTime DateFrom { get; set; }

        public DateTime DateTo { get; set; }

    }

}

using System;

using System.Collections.Generic;

using System.Linq;

using System.Threading.Tasks;

namespace Application.Searches

{

    public class UserSearch : PagedSearch

    {

        public string Username { get; set; }

        public string Email { get; set; }

    }

}

4. **Implementation** sloj se sada dodaje (class library). Ideja je da kaže na koji način treba da se uradi ono što smo definisali u Application sloju. Treba mu referenca ka Application sloju i Data Access-u. Radim implementaciju use case-ova, odnosno interfejsa definisanih u Application sloju, poput komandi i upita. Ovde se nalazi i poslovna logika koja nije direktno vezana za pristup podacima, već za obradu tih podataka. Na primer, validacija unosa ili transofrmacija podataka pre nego što budu vraćeni korisniku.

Upravljanje izuzecima uvek radi sloj iznad (znači API ili Desktop).

using System;

using System.Collections.Generic;

using System.Linq;

using System.Threading.Tasks;

using Newtonsoft.Json;

using Application;

using Domain;

using EFDataAccess;

namespace Implementation.Logging

{

    public class EFDatabaseLogger : IUseCaseLogger

    {

        private readonly BlogContext \_context;

*// Konstruktor prima BlogContext kao zavisnost, što omogućava pristup bazi podataka preko Entity Framework-a  
        // Ova zavisnost se ne instancira unutar same klase, već se obezbeđuje spolja, što omogućava bolju modularnost i testiranje (DI – Klasa kojoj su potrebni neki drugi objekti neće praviti njihove objekte u okviru sebe, već će ih dobijati kroz konstruktor)*

*// Single Responsibility Principle SRP – Klasa mora da ima 1 razlog što se menja*

        public EFDatabaseLogger(BlogContext context)

        {

            \_context = context;

        }

        public void Log(IUseCase useCase, IApplicationActor actor, object data)

        {

            \_context.UseCaseLogs.Add(new UseCaseLog

            {

                Date = DateTime.UtcNow,

                Actor = actor.Identity,

                Data = JsonConvert.SerializeObject(data), *// Dodatni podaci koji se beleže uz use case, serijalizovani u JSON format koristeći Newtonsoft.Json*

                UseCaseName = useCase.Name.ToString()

            });

            \_context.SaveChanges();

        }

    }

}

Nakon što je entitet **UseCaseLog** kreiran i popunjen sa relevantnim informacijama, poziva se metoda SaveChanges() nad BlogContext-om, što rezultuje upisivanjem novog reda u tabelu UseCaseLogs u bazi podataka.

**Validatori** su klase koje se koriste za proveru i validaciju podataka pre nego što budu obrađeni ili upisani u bazu podataka. Koristi se **FluentValidation** biblioteka.

using System;

using System.Collections.Generic;

using System.Linq;

using System.Threading.Tasks;

using Application.DataTransfer;

using EFDataAccess;

using FluentValidation;

namespace Implementation.Validators.Blog

{

    public class CreateBlogValidator : AbstractValidator<InsertBlogDto>

    {

        private readonly BlogContext \_context;

        public CreateBlogValidator(BlogContext context)

        {

            \_context = context;

            RuleFor(x => x.Title).NotEmpty()

                .WithMessage("Title is required.")

                .MinimumLength(3)

                .WithMessage("Title must containt more than 3 letters.");

            RuleFor(x => x.Content).NotEmpty()

                .WithMessage("Content is required.")

                .MinimumLength(5)

                .WithMessage("Content must contain more than 5 letters.");

            RuleFor(x => x.IdImage).Must(ImageExists)

                .WithMessage("Image with provided ID doesn't exists.");

            RuleForEach(x => x.BlogCategories).ChildRules(categories =>

            {

                categories.RuleFor(x => x.IdCategory).Must(CategoryExists)

                    .WithMessage("Category with provied ID doesn't exists");

            });

            RuleFor(x => x.BlogCategories).Must(x => x.Select(y => y.IdCategory).Distinct().Count() == x.Count())

                .WithMessage("Duplicate categories not allowed.");

        }

        private bool ImageExists(int id)

        {

            return \_context.Images.Any(x => x.Id == id);

        }

        private bool CategoryExists(int id)

        {

            return \_context.Categories.Any(x => x.Id == id);

        }

    }

}

CreateBlogValidator klasa nasleđuje apstraktnu klasu **AbstractValidator<T>** gde je T tip podatka koji se validira, u ovom slučaju InsertBlogDto.  
Konstruktor validatora prihvata parametar BlogContext koji se koristi za proveru postojanja slike i kategorije.

Metoda **RuleFor** očekuje lambda izraz, definiše pravila za validaciju određenih polja ili svojstava objekata.

using System;

using System.Collections.Generic;

using System.Linq;

using System.Threading.Tasks;

using FluentValidation;

namespace Implementation.Validators.Blog

{

    public class DeleteBlogValidator : AbstractValidator<int>

    {

        public DeleteBlogValidator()

        {

            RuleFor(x => x).NotEmpty();

        }

    }

}

using System;

using System.Collections.Generic;

using System.Linq;

using System.Threading.Tasks;

using Application.DataTransfer;

using EFDataAccess;

using FluentValidation;

namespace Implementation.Validators.Blog

{

    public class UpdateBlogValidator : AbstractValidator<UpdateBlogDto>

    {

        private readonly BlogContext \_context;

        public UpdateBlogValidator(BlogContext context)

        {

            \_context = context;

            RuleFor(x => x.Title).NotEmpty()

                .WithMessage("Title is required.")

                .MinimumLength(3)

                .WithMessage("Title must containt more than 3 letters.");

            RuleFor(x => x.Content).NotEmpty()

                .WithMessage("Content is required.")

                .MinimumLength(5)

                .WithMessage("Content must contain more than 5 letters.");

            RuleFor(x => x.IdImage).Must(ImageExists)

                .WithMessage("Image with provided ID doesn't exists.");

            RuleForEach(x => x.BlogCategories).ChildRules(categories =>

            {

                categories.RuleFor(x => x).Must(CategoryExists)

                    .WithMessage("Category with provied ID doesn't exists");

            });

*// x => x.BlogCategories označava svojsvto BlogCategories u UpdateBlogDto klasi.*

*// Metoda Must se koristi za definisanje pravila koje mora biti ispunjeno. U ovom slučaju prosleđujemo lambda izraz koji proverava da li su svi elementi kolekcije jedinstveni*

*// x predstavlja kolekciju BlogCategories. Prvo primenjujemo Select metodu da izvučemo sve elemente kolekcije. Zatim primenjujemo Distinct metodu koja uklanja duplikate. Na kraju primenjujemo Count metodu kako bismo dobili broj jedinstvenih elemenata. Ukoliko je broj jedinstvenih elemenata jednak broju elemenata u kolekciji, to znači da nema duplikata.*

*// Ako postoje duplikati u kolekciji, biće generisana greška sa porukom.*

            RuleFor(x => x.BlogCategories).Must(x => x.Select(y => y).Distinct().Count() == x.Count())

                .WithMessage("Duplicate categories are not allowed.");

        }

        private bool ImageExists(int id)

        {

            return \_context.Images.Any(x => x.Id == id);

        }

        private bool CategoryExists(int id)

        {

            return \_context.Categories.Any(x => x.Id == id);

        }

    }

}

using System;

using System.Collections.Generic;

using System.Linq;

using System.Threading.Tasks;

using Application.DataTransfer;

using EFDataAccess;

using FluentValidation;

namespace Implementation.Validators.Category

{

    public class CreateCategoryValidator : AbstractValidator<CategoryDto>

    {

        public CreateCategoryValidator(BlogContext context)

        {

            RuleFor(x => x.Name).NotEmpty()

                .WithMessage("Category name is required.")

                .Must(name => !context.Categories.Any(c => c.Name == name))

                .WithMessage("Category name must be unique.");

        }

    }

}

using System;

using System.Collections.Generic;

using System.Linq;

using System.Threading.Tasks;

using FluentValidation;

namespace Implementation.Validators.Category

{

    public class DeleteCategoryValidator : AbstractValidator<int>

    {

        public DeleteCategoryValidator()

        {

            RuleFor(x => x).NotEmpty();

        }

    }

}

using System;

using System.Collections.Generic;

using System.Linq;

using System.Threading.Tasks;

using Application.DataTransfer;

using EFDataAccess;

using FluentValidation;

namespace Implementation.Validators.Category

{

    public class UpdateCategoryValidator : AbstractValidator<CategoryDto>

    {

        public UpdateCategoryValidator(BlogContext context)

        {

            RuleFor(x => x.Name).NotEmpty()

               .WithMessage("Name cannot be empty.");

            RuleFor(x => x.Name).Must((dto, name) => !context.Categories.Any(x => x.Name == name && x.Id != dto.Id))

                .WithMessage(p => $"Category with the name of {p.Name} already exists in database.");

        }

    }

}

using System;

using System.Collections.Generic;

using System.Linq;

using System.Threading.Tasks;

using Application.DataTransfer;

using EFDataAccess;

using FluentValidation;

namespace Implementation.Validators.Comment

{

    public class CreateCommentValidator : AbstractValidator<InsertCommentDto>

    {

        private readonly BlogContext \_context;

        public CreateCommentValidator(BlogContext context)

        {

            \_context = context;

            RuleFor(x => x.Text).NotEmpty()

                .WithMessage("Text is required.");

            RuleFor(x => x.IdBlog).NotEmpty()

                .WithMessage("Blog is required.")

                .Must(BlogExists)

                .WithMessage(y => $"Blog with ID {y.Id} doesn't exists.");

            RuleFor(x => x.IdParent).Must(IdComment => context.Comments.Any(y => y.Id == IdComment))

                .When(request => request.IdParent != null)

                .WithMessage("Parent comment doesn't exists in system.");

        }

        private bool BlogExists(int id)

        {

            return \_context.Blogs.Any(x => x.Id == id);

        }

    }

}

using System;

using System.Collections.Generic;

using System.Linq;

using System.Threading.Tasks;

using FluentValidation;

namespace Implementation.Validators.Comment

{

    public class DeleteCommentValidator : AbstractValidator<int>

    {

        public DeleteCommentValidator()

        {

            RuleFor(x => x).NotEmpty();

        }

    }

}

using Application.DataTransfer;

using FluentValidation;

using System;

using System.Collections.Generic;

using System.Text;

namespace Implementation.Validators.Comment

{

    public class UpdateCommentValidator : AbstractValidator<InsertCommentDto>

    {

        public UpdateCommentValidator()

        {

            RuleFor(x => x.Text).NotEmpty();

        }

    }

}

using System;

using System.Collections.Generic;

using System.Linq;

using System.Threading.Tasks;

using Application.DataTransfer;

using Domain;

using FluentValidation;

namespace Implementation.Validators.Like

{

    public class LikeValidator : AbstractValidator<LikeDto>

    {

        public LikeValidator()

        {

            RuleFor(x => x.IdBlog).NotEmpty();

            RuleFor(x => x.IdUser).NotEmpty();

            RuleFor(x => x.Status).NotEmpty()

                .Must(y => Enum.IsDefined(typeof(LikeStatus), y))

                .WithMessage("Status can only be 'Like' or 'Dislike'.");

        }

    }

}

using System;

using System.Collections.Generic;

using System.Linq;

using System.Threading.Tasks;

using Application.DataTransfer;

using EFDataAccess;

using FluentValidation;

namespace Implementation.Validators.User

{

    public class CreateUserValidator : AbstractValidator<InsertUserDto>

    {

        private readonly BlogContext \_context;

        public CreateUserValidator(BlogContext context)

        {

            \_context = context;

            RuleFor(x => x.FirstName).NotEmpty();

            RuleFor(x => x.LastName).NotEmpty();

            RuleFor(x => x.Username).NotEmpty()

                .Must(x => !context.Users.Any(user => user.Username == x))

                .WithMessage("Username is already taken.");

            RuleFor(x => x.Password).NotEmpty().MinimumLength(3);

            RuleFor(x => x.Email).NotEmpty()

                .EmailAddress()

                .Must(x => !context.Users.Any(user => user.Email == x))

                .WithMessage("Email is already taken.");

            RuleFor(x => x.UserUseCases).Must(c => c.Select(v => v.IdUseCase).Distinct().Count() == c.Count())

                .WithMessage("Duplicate UseCaseIds are not allowed.");

            RuleFor(x => x.UserUseCases.Count()).GreaterThan(0)

                .WithMessage("User must contain more than 0 UseCases");

            RuleForEach(x => x.UserUseCases).ChildRules(n =>

            {

                n.RuleFor(x => x.IdUseCase)

                    .Must(UseCaseExists)

                    .WithMessage("{PropertyValue} usecase doesn't exists.");

            });

        }

        private bool UseCaseExists(int id)

        {

            return Enum.IsDefined(typeof(UseCaseEnum), id);

        }

    }

}

using System;

using System.Collections.Generic;

using System.Linq;

using System.Threading.Tasks;

using Application.DataTransfer;

using EFDataAccess;

using FluentValidation;

using Microsoft.EntityFrameworkCore;

namespace Implementation.Validators.User

{

    public class RegisterUserValidator : AbstractValidator<RegisterUserDto>

    {

        private readonly BlogContext \_context;

        public RegisterUserValidator(BlogContext context)

        {

            \_context = context;

            RuleFor(x => x.FirstName).NotEmpty()

                .WithMessage("First name is required.");

            RuleFor(x => x.LastName).NotEmpty()

                .WithMessage("Last name is required.");

            RuleFor(x => x.Password).NotEmpty()

                .MinimumLength(3)

                .WithMessage("Password must be with at least 3 characters and not empty.");

            RuleFor(x => x.Username).NotEmpty()

                .WithMessage("Username is required.")

                .MinimumLength(4)

                .WithMessage("Minimum length for username is 4 characters.")

                .Must(DoesNotExistUsername)

                .WithMessage("Username must be unique.");

            RuleFor(x => x.Email).NotEmpty()

                .WithMessage("Email is required.")

                .EmailAddress()

                .WithMessage("Not a valid email format.")

                .Must(x => !context.Users.Any(user => user.Email == x))

                .WithMessage("Email adress must be unique.");

        }

        private bool DoesNotExistUsername(string username)

        {

            var usernames = \_context.Users.IgnoreQueryFilters().Select(x => x.Username);

            return !usernames.Contains(username);

        }

    }

}

using System;

using System.Collections.Generic;

using System.Linq;

using System.Threading.Tasks;

using Application.DataTransfer;

using EFDataAccess;

using FluentValidation;

namespace Implementation.Validators.User

{

    public class UpdateUserValidator : AbstractValidator<UpdateUserDto>

    {

        private readonly BlogContext \_context;

        public UpdateUserValidator(BlogContext context)

        {

            \_context = context;

            RuleFor(x => x.FirstName).NotEmpty();

            RuleFor(x => x.LastName).NotEmpty();

            RuleFor(x => x.Username).NotEmpty()

                .Must((dto, name) => !context.Users.Any(g => g.Username == name && g.Id != dto.Id))

                .WithMessage("Username is already taken.");

            RuleFor(x => x.Password).NotEmpty()

                .MinimumLength(3);

            RuleForEach(x => x.UserUseCases).Must(UseCaseExist)

                .WithMessage("{PropertyValue} UseCase doesn't exist");

            RuleFor(x => x.UserUseCases).Must(c => c.Select(v => v).Distinct().Count() == c.Count())

                .WithMessage("Duplicates are not allowed.");

        }

        private bool UseCaseExist(int id)

        {

            return Enum.IsDefined(typeof(UseCaseEnum), id);

        }

    }

}

using System;

namespace Implementation

{

    public enum UseCaseEnum

    {

        EFRegisterUserCommand = 1,

        EFCreateBlogCommand = 2,

        EFUpdateBlogCommand = 3,

        EFDeleteBlogCommand = 4,

        EFGetOneBlogQuery = 5,

        EFGetBlogsQuery = 6,

        EFUpdatePersonalBlogCommand = 7,

        EFDeletePersonalBlogCommand = 8,

        EFCreateCommentCommand = 9,

        EFUpdateCommentCommand = 10,

        EFDeleteCommentCommand = 11,

        EFGetOneCommentQuery = 12,

        EFGetCommentsQuery = 13,

        EFUpdatePersonalCommentCommand = 14,

        EFDeletePersonalCommentCommand = 15,

        EFLikeBlog = 16,

        EFGetUseCaseLogQuery = 17,

        EFCreateUserCommand = 18,

        EFUpdateUserCommand = 19,

        EFDeleteUserCommand = 20,

        EFGetOneUserQuery = 21,

        EFGetUsersQuery = 22,

        EFCreateCategoryCommand = 23,

        EFUpdateCategoryCommand = 24,

        EFDeleteCategoryCommand = 25,

        EFGetOneCategoryQuery = 26,

        EFGetCategoriesQuery = 27,

        EFGetImagesQuery = 28

    }

}

using System;

using System.Collections.Generic;

using System.Linq;

using System.Threading.Tasks;

using Application;

using Application.Commands.Blog;

using Application.DataTransfer;

using Domain;

using EFDataAccess;

using FluentValidation;

using Implementation.Validators.Blog;

namespace Implementation.Commands.Blog

{

    public class EFCreateBlogCommand : ICreateBlogCommand

    {

        private readonly BlogContext \_context;

        private readonly IApplicationActor \_actor;

        private readonly CreateBlogValidator \_validator;

        public EFCreateBlogCommand(CreateBlogValidator validator, IApplicationActor actor, BlogContext context)

        {

            \_validator = validator;

            \_actor = actor;

            \_context = context;

        }

        public int Id => (int)UseCaseEnum.EFCreateBlogCommand;

        public string Name => UseCaseEnum.EFCreateBlogCommand.ToString();

        public void Execute(InsertBlogDto request)

        {

            \_validator.ValidateAndThrow(request);

            var blog = new Domain.Blog

            {

                Title = request.Title,

                Content = request.Content,

                IdImage = request.IdImage,

                IdUser = \_actor.Id

            };

            foreach (var category in request.BlogCategories)

            {

                blog.BlogCategories.Add(new BlogCategory

                {

                    IdCategory = category.IdCategory

                });

            }

            \_context.Blogs.Add(blog);

            \_context.SaveChanges();

        }

    }

}

BlogContext \_context: Predstavlja instancu konteksta baze podataka koji omogućava pristup podacima.  
IApplicationActor \_actor: Predstavlja trenutnog korisnika koji izvršava komandu.  
CreateBlogValidator \_validator: Predstavlja validator koji se koristi za validaciju podataka koji se šalju za kreiranje bloga.

Konstruktor prima sve ove zavisnosti kao argumente i ih injektuje u klasu.

Id i Name svojstva su definisana iz enumeracije UseCaseEnum i vraćaju identifikator i naziv ove komande.

Execute metoda je odgovorna za izvršavanje komande za kreiranje bloga.

Prvo se poziva ValidateAndThrow metoda na validatoru kako bi se validirali podaci koji se šalju za kreiranje bloga. Ukoliko validacija ne prođe, biće bačen izuzetak koji sadrži poruke o greškama validacije.

Zatim se kreira nova instanca Blog entiteta.

Nakon toga, prolazi se kroz sve kategorije bloga koje su prosleđene u zahtevu, i za svaku se kreira nova BlogCategory instanca koja se dodaje u kolekciju BlogCategories na novom blogu.

Na kraju, novi blog se dodaje u kontekst baze podataka \_context.Blogs.Add(blog) i izvršava se čuvanje promena u bazi podataka \_context.SaveChanges().

using System;

using System.Collections.Generic;

using System.Linq;

using System.Threading.Tasks;

using Application.Commands.Blog;

using Application.Exceptions;

using EFDataAccess;

using FluentValidation;

using Implementation.Validators.Blog;

namespace Implementation.Commands.Blog

{

    public class EFDeleteBlogCommand : IDeleteBlogCommand

    {

        private readonly BlogContext \_context;

        private readonly DeleteBlogValidator \_validator;

        public EFDeleteBlogCommand(BlogContext context, DeleteBlogValidator validator)

        {

            \_context = context;

            \_validator = validator;

        }

        public int Id => (int)UseCaseEnum.EFDeleteBlogCommand;

        public string Name => UseCaseEnum.EFDeleteBlogCommand.ToString();

        public void Execute(int request)

        {

            \_validator.ValidateAndThrow(request);

            var blog = \_context.Blogs.Find(request);

            if (blog == null)

            {

                throw new EntityNotFoundException(request, typeof(Domain.Blog));

            }

            if (blog.IsDeleted)

            {

                throw new AlreadyDeletedException(request, typeof(Domain.Blog));

            }

            blog.DeletedAt = DateTime.Now;

            blog.IsActive = false;

            blog.IsDeleted = true;

            \_context.SaveChanges();

        }

    }

}

**Find()** ne prima integer već prima params object[] što znači može 1 ili viđe objekata (možemo da prosledimo vrednost kompozitnog ključa).

using System;

using System.Collections.Generic;

using System.Linq;

using System.Threading.Tasks;

namespace Application.Exceptions

{

    public class EntityNotFoundException : Exception

    {

        public EntityNotFoundException(int id, Type type)

            : base($"Entity of type {type.Name} with an ID of {id} was not found.")

        {

        }

    }

}

using System;

using System.Collections.Generic;

using System.Linq;

using System.Threading.Tasks;

namespace Application.Exceptions

{

    public class AlreadyDeletedException : Exception

    {

        public AlreadyDeletedException(int id, Type type)

            : base($"Entity of type {type.Name} with and ID of {id} was already deleted.")

        {

        }

    }

}

using System;

using System.Collections.Generic;

using System.Linq;

using System.Threading.Tasks;

using Application;

using Application.Commands.Blog;

using Application.Exceptions;

using EFDataAccess;

namespace Implementation.Commands.Blog

{

    public class EFDeletePersonalBlogCommand : IDeletePersonalBlogCommand

    {

        private readonly BlogContext \_context;

        private readonly IApplicationActor \_actor;

        public EFDeletePersonalBlogCommand(BlogContext context, IApplicationActor actor)

        {

            \_context = context;

            \_actor = actor;

        }

        public int Id => (int)UseCaseEnum.EFDeletePersonalBlogCommand;

        public string Name => UseCaseEnum.EFDeletePersonalBlogCommand.ToString();

        public void Execute(int request)

        {

            var blog = \_context.Blogs.Find(request);

            if (blog == null)

            {

                throw new EntityNotFoundException(request, typeof(Domain.Blog));

            }

            if (\_actor.Id != blog.IdUser)

            {

                throw new UnauthorizedUserAccessException(\_actor, Name);

            }

            if (blog.IsDeleted)

            {

                throw new AlreadyDeletedException(request, typeof(Domain.Blog));

            }

            blog.DeletedAt = DateTime.Now;

            blog.IsActive = false;

            blog.IsDeleted = true;

            \_context.SaveChanges();

        }

    }

}

using System;

using System.Collections.Generic;

using System.Linq;

using System.Threading.Tasks;

namespace Application.Exceptions

{

    public class UnauthorizedUserAccessException : Exception

    {

        public UnauthorizedUserAccessException(IApplicationActor actor, string UseCaseName)

            : base($"User with identity: ${actor.Identity} with ID: {actor.Id} has tried to execute Use Case {UseCaseName}")

        {

        }

    }

}

using System;

using System.Collections.Generic;

using System.Linq;

using System.Threading.Tasks;

using Application.Commands.Blog;

using Application.DataTransfer;

using Application.Exceptions;

using EFDataAccess;

using FluentValidation;

using Implementation.Validators.Blog;

using Microsoft.EntityFrameworkCore;

namespace Implementation.Commands.Blog

{

    public class EFUpdateBlogCommand : IUpdateBlogCommand

    {

        private readonly BlogContext \_context;

        private readonly UpdateBlogValidator \_validator;

        public EFUpdateBlogCommand(BlogContext context, UpdateBlogValidator validator)

        {

            \_context = context;

            \_validator = validator;

        }

        public int Id => (int)UseCaseEnum.EFUpdateBlogCommand;

        public string Name => UseCaseEnum.EFUpdateBlogCommand.ToString();

        public void Execute(UpdateBlogDto request)

        {

            \_validator.ValidateAndThrow(request);

            var blog = \_context.Blogs.Include(x => x.BlogCategories).FirstOrDefault(x => x.Id == request.Id);

            if (blog == null)

            {

                throw new EntityNotFoundException(request.Id, typeof(Domain.Blog));

            }

            blog.Title = request.Title;

            blog.Content = request.Content;

            blog.IdImage = request.IdImage;

            blog.ModifiedAt = DateTime.Now;

*// Ako postoje kategorije koje nisu prisutne u prosleđenom UpdateBlogDto objektu, one se označavaju kao neaktivne i obrisane, a za nove kategorije se dodaju nove veze između bloga i kategorija.*

            var categoryDelete = blog.BlogCategories.Where(x => !request.BlogCategories.Contains(x.IdCategory));

            foreach (var category in categoryDelete)

            {

                category.IsActive = false;

                category.IsDeleted = true;

                category.DeletedAt = DateTime.Now;

            }

            var categoryIds = blog.BlogCategories.Select(x => x.IdCategory);

            var categoryInsert = request.BlogCategories.Where(x => !categoryIds.Contains(x));

            foreach (var IdCategory in categoryInsert)

            {

                blog.BlogCategories.Add(new Domain.BlogCategory

                {

                    IdCategory = IdCategory

                });

            }

            \_context.SaveChanges();

        }

    }

}

using System;

using System.Collections.Generic;

using System.Linq;

using System.Threading.Tasks;

using Application;

using Application.Commands.Blog;

using Application.DataTransfer;

using Application.Exceptions;

using EFDataAccess;

using FluentValidation;

using Implementation.Validators.Blog;

using Microsoft.EntityFrameworkCore;

namespace Implementation.Commands.Blog

{

    public class EFUpdatePersonalBlogCommand : IUpdatePersonalBlogCommand

    {

        private readonly BlogContext \_context;

        private readonly UpdateBlogValidator \_validator;

        private readonly IApplicationActor \_actor;

        public EFUpdatePersonalBlogCommand(BlogContext context, UpdateBlogValidator validator, IApplicationActor actor)

        {

            \_context = context;

            \_validator = validator;

            \_actor = actor;

        }

        public int Id => (int)UseCaseEnum.EFUpdatePersonalBlogCommand;

        public string Name => UseCaseEnum.EFUpdatePersonalBlogCommand.ToString();

        public void Execute(UpdateBlogDto request)

        {

            \_validator.ValidateAndThrow(request);

            var blog = \_context.Blogs.Include(x => x.BlogCategories).FirstOrDefault(x => x.Id == request.Id);

            if (blog == null)

            {

                throw new EntityNotFoundException(request.Id, typeof(Domain.Blog));

            }

            if (\_actor.Id != blog.IdUser)

            {

                throw new UnauthorizedUserAccessException(\_actor, Name);

            }

            blog.Title = request.Title;

            blog.Content = request.Content;

            blog.IdImage = request.IdImage;

            blog.ModifiedAt = DateTime.Now;

            var categoryDelete = blog.BlogCategories.Where(x => !request.BlogCategories.Contains(x.IdCategory));

            foreach (var category in categoryDelete)

            {

                category.IsActive = false;

                category.IsDeleted = true;

                category.DeletedAt = DateTime.Now;

            }

            var categoryIds = blog.BlogCategories.Select(x => x.IdCategory);

            var categoryInsert = request.BlogCategories.Where(x => !categoryIds.Contains(x));

            foreach (var IdCategory in categoryInsert)

            {

                blog.BlogCategories.Add(new Domain.BlogCategory

                {

                    IdCategory = IdCategory

                });

            }

            \_context.SaveChanges();

        }

    }

}

using System;

using System.Collections.Generic;

using System.Linq;

using System.Threading.Tasks;

using Application.Commands.Category;

using Application.DataTransfer;

using Domain;

using EFDataAccess;

using FluentValidation;

using Implementation.Validators.Category;

namespace Implementation.Commands.Category

{

    public class EFCreateCategoryCommand : ICreateCategoryCommand

    {

        private readonly BlogContext \_context;

        private readonly CreateCategoryValidator \_validator;

        public EFCreateCategoryCommand(CreateCategoryValidator validator, BlogContext context)

        {

            \_validator = validator;

            \_context = context;

        }

        public int Id => (int)UseCaseEnum.EFCreateCategoryCommand;

        public string Name => UseCaseEnum.EFCreateCategoryCommand.ToString();

        public void Execute(CategoryDto request)

        {

            \_validator.ValidateAndThrow(request);

            var category = new Domain.Category

            {

                Name = request.Name

            };

            \_context.Categories.Add(category);

            \_context.SaveChanges();

        }

    }

}

using System;

using System.Collections.Generic;

using System.Linq;

using System.Threading.Tasks;

using Application.Commands.Category;

using Application.Exceptions;

using EFDataAccess;

using FluentValidation;

using Implementation.Validators.Category;

using Microsoft.EntityFrameworkCore;

namespace Implementation.Commands.Category

{

    public class EFDeleteCategoryCommand : IDeleteCategoryCommand

    {

        private readonly BlogContext \_context;

        private readonly DeleteCategoryValidator \_validator;

        public EFDeleteCategoryCommand(BlogContext context, DeleteCategoryValidator validator)

        {

            \_context = context;

            \_validator = validator;

        }

        public int Id => (int)UseCaseEnum.EFDeleteCategoryCommand;

        public string Name => UseCaseEnum.EFDeleteCategoryCommand.ToString();

        public void Execute(int id)

        {

            \_validator.ValidateAndThrow(id);

            var category = \_context.Categories.Include(x => x.CategoryBlogs).FirstOrDefault(x => x.Id == id);

            if (category == null)

            {

                throw new EntityNotFoundException(id, typeof(Domain.Category));

            }

            if (category.CategoryBlogs.Count > 0)

            {

                throw new ConflictException("Category is not empty.");

            }

            if (category.IsDeleted == true)

            {

                throw new AlreadyDeletedException(id, typeof(Domain.Category));

            }

            category.DeletedAt = DateTime.Now;

            category.IsDeleted = true;

            category.IsActive = false;

            \_context.SaveChanges();

        }

    }

}

using System;

using System.Collections.Generic;

using System.Linq;

using System.Threading.Tasks;

namespace Application.Exceptions

{

    public class ConflictException : Exception

    {

        public ConflictException(string message)

            : base(message)

        {

        }

    }

}

using System;

using System.Collections.Generic;

using System.Linq;

using System.Threading.Tasks;

using Application.Commands.Category;

using Application.DataTransfer;

using Application.Exceptions;

using EFDataAccess;

using FluentValidation;

using Implementation.Validators.Category;

namespace Implementation.Commands.Category

{

    public class EFUpdateCategoryCommand : IUpdateCategoryCommand

    {

        private readonly BlogContext \_context;

        private readonly UpdateCategoryValidator \_validator;

        public EFUpdateCategoryCommand(UpdateCategoryValidator validator, BlogContext context)

        {

            \_validator = validator;

            \_context = context;

        }

        public int Id => (int)UseCaseEnum.EFUpdateCategoryCommand;

        public string Name => UseCaseEnum.EFUpdateCategoryCommand.ToString();

        public void Execute(CategoryDto request)

        {

            var category = \_context.Categories.Find(request.Id);

            if (request == null)

            {

                throw new EntityNotFoundException(request.Id, typeof(Domain.Category));

            }

            \_validator.ValidateAndThrow(request);

            category.Name = request.Name;

            category.ModifiedAt = DateTime.Now;

            \_context.SaveChanges();

        }

    }

}

using System;

using System.Collections.Generic;

using System.Linq;

using System.Threading.Tasks;

using Application;

using Application.Commands.Comment;

using Application.DataTransfer;

using EFDataAccess;

using FluentValidation;

using Implementation.Validators.Comment;

namespace Implementation.Commands.Comment

{

    public class EFCreateCommentCommand : ICreateCommentCommand

    {

        private readonly BlogContext \_context;

        private readonly CreateCommentValidator \_validator;

        private readonly IApplicationActor \_actor;

        public EFCreateCommentCommand(BlogContext context, CreateCommentValidator validator, IApplicationActor actor)

        {

            \_context = context;

            \_validator = validator;

            \_actor = actor;

        }

        public int Id => (int)UseCaseEnum.EFCreateCommentCommand;

        public string Name => UseCaseEnum.EFCreateCommentCommand.ToString();

        public void Execute(InsertCommentDto request)

        {

            \_validator.ValidateAndThrow(request);

            request.IdUser = \_actor.Id;

            var comment = new Domain.Comment

            {

                CommentText = request.Text,

                IdBlog = request.IdBlog,

                IdParent = request.IdParent,

                IdUser = request.IdUser

            };

            \_context.Comments.Add(comment);

            \_context.SaveChanges();

        }

    }

}

using System;

using System.Collections.Generic;

using System.Linq;

using System.Threading.Tasks;

using Application.Commands.Comment;

using Application.Exceptions;

using EFDataAccess;

namespace Implementation.Commands.Comment

{

    public class EFDeleteCommentCommand : IDeleteCommentCommand

    {

        private readonly BlogContext \_context;

        public EFDeleteCommentCommand(BlogContext context)

        {

            \_context = context;

        }

        public int Id => (int)UseCaseEnum.EFDeleteCommentCommand;

        public string Name => UseCaseEnum.EFDeleteCommentCommand.ToString();

        public void Execute(int request)

        {

            var comment = \_context.Comments.Find(request);

            if (comment == null)

            {

                throw new EntityNotFoundException(request, typeof(Domain.Comment));

            }

            if (comment.IsDeleted == true)

            {

                throw new AlreadyDeletedException(request, typeof(Domain.Comment));

            }

            comment.DeletedAt = DateTime.Now;

            comment.IsDeleted = true;

            comment.IsActive = false;

            \_context.SaveChanges();

        }

    }

}

using System;

using System.Collections.Generic;

using System.Linq;

using System.Threading.Tasks;

using Application;

using Application.Commands.Comment;

using Application.Exceptions;

using EFDataAccess;

using FluentValidation;

using Implementation.Validators.Comment;

namespace Implementation.Commands.Comment

{

    public class EFDeletePersonalCommentCommand : IDeletePersonalCommentCommand

    {

        private readonly BlogContext \_context;

        private readonly IApplicationActor \_actor;

        private readonly DeleteCommentValidator \_validator;

        public EFDeletePersonalCommentCommand(BlogContext context, IApplicationActor actor, DeleteCommentValidator validator)

        {

            \_context = context;

            \_actor = actor;

            \_validator = validator;

        }

        public int Id => (int)UseCaseEnum.EFDeletePersonalCommentCommand;

        public string Name => UseCaseEnum.EFDeletePersonalCommentCommand.ToString();

        public void Execute(int request)

        {

            \_validator.ValidateAndThrow(request);

            var comment = \_context.Comments.Find(request);

            if (comment == null)

            {

                throw new EntityNotFoundException(request, typeof(Domain.Comment));

            }

            if (\_actor.Id != comment.IdUser)

            {

                throw new UnauthorizedUserAccessException(\_actor, Name);

            }

            if (comment.IsDeleted == true)

            {

                throw new AlreadyDeletedException(request, typeof(Domain.Comment));

            }

            comment.DeletedAt = DateTime.Now;

            comment.IsDeleted = true;

            comment.IsActive = false;

            \_context.SaveChanges();

        }

    }

}

using System;

using System.Collections.Generic;

using System.Linq;

using System.Threading.Tasks;

using Application;

using Application.Commands.Comment;

using Application.DataTransfer;

using Application.Exceptions;

using EFDataAccess;

using FluentValidation;

using Implementation.Validators.Comment;

namespace Implementation.Commands.Comment

{

    public class EFUpdatePersonalCommentCommand : IUpdatePersonalCommentCommand

    {

        private readonly BlogContext \_context;

        private readonly IApplicationActor \_actor;

        private readonly UpdateCommentValidator \_validator;

        public EFUpdatePersonalCommentCommand(UpdateCommentValidator validator, IApplicationActor actor, BlogContext context)

        {

            \_validator = validator;

            \_actor = actor;

            \_context = context;

        }

        public int Id => (int)UseCaseEnum.EFUpdatePersonalCommentCommand;

        public string Name => UseCaseEnum.EFUpdatePersonalCommentCommand.ToString();

        public void Execute(InsertCommentDto request)

        {

            \_validator.ValidateAndThrow(request);

            var comment = \_context.Comments.Find(request.Id);

            if (comment == null)

            {

                throw new EntityNotFoundException(request.Id, typeof(Domain.Comment));

            }

            if (\_actor.Id != comment.IdUser)

            {

                throw new UnauthorizedUserAccessException(\_actor, Name);

            }

            var query = \_context.Comments.Where(x => x.Id == request.Id).FirstOrDefault();

            comment.CommentText = request.Text;

            \_context.SaveChanges();

        }

    }

}

using System;

using System.Collections.Generic;

using System.Linq;

using System.Threading.Tasks;

using Application.Commands.Email;

using Application.DataTransfer;

using System.Net;

using System.Net.Mail;

namespace Implementation.Commands.Email

{

    public class SMTPEmailSender : IEmailSender

    {

        public void Send(SendEmailDto dto)

        {

            var smtp = new SmtpClient

            {

                Host = "smtp.gmail.com",

                Port = 587,

                EnableSsl = true,

                DeliveryMethod = SmtpDeliveryMethod.Network,

                UseDefaultCredentials = false,

*// Allow less security apps*

                Credentials = new NetworkCredential("blogapiasp@gmail.com", "tmoe xbbp gxmn jeco")

            };

            var message = new MailMessage("blogapiasp@gmail.com", dto.SendTo);

            message.Subject = dto.Subject;

            message.Body = dto.Content;

            message.IsBodyHtml = true;

            smtp.Send(message);

        }

    }

}

using System;

using System.Collections.Generic;

using System.Linq;

using System.Threading.Tasks;

using Application.Commands.Like;

using Application.DataTransfer;

using EFDataAccess;

using FluentValidation;

using Implementation.Validators.Like;

namespace Implementation.Commands.Like

{

    public class EFLikeBlogCommand : ILikeBlogCommand

    {

        private readonly BlogContext \_context;

        private readonly LikeValidator \_validator;

        public EFLikeBlogCommand(LikeValidator validator, BlogContext context)

        {

            \_validator = validator;

            \_context = context;

        }

        public int Id => (int)UseCaseEnum.EFLikeBlog;

        public string Name => UseCaseEnum.EFLikeBlog.ToString();

        public void Execute(LikeDto request)

        {

            \_validator.ValidateAndThrow(request);

            var findLike = \_context.Likes.Where(x => x.IdBlog == request.IdBlog && x.IdUser == request.IdUser).FirstOrDefault();

            if (findLike == null)

            {

                var like = new Domain.Like

                {

                    IdUser = request.IdUser,

                    IdBlog = request.IdBlog,

                    Status = request.Status

                };

                \_context.Likes.Add(like);

                \_context.SaveChanges();

            }

            else

            {

                findLike.Status = request.Status;

                \_context.SaveChanges();

            }

        }

    }

}

using System;

using System.Collections.Generic;

using System.Linq;

using System.Threading.Tasks;

using Application.Commands.Email;

using Application.Commands.User;

using Application.DataTransfer;

using Domain;

using EFDataAccess;

using FluentValidation;

using Implementation.Validators.User;

namespace Implementation.Commands.User

{

    public class EFRegisterUserCommand : IRegisterUserCommand

    {

        private readonly BlogContext \_context;

        private readonly RegisterUserValidator \_validator;

        private readonly IEmailSender \_sender;

        public EFRegisterUserCommand(IEmailSender sender, RegisterUserValidator validator, BlogContext context)

        {

            \_sender = sender;

            \_validator = validator;

            \_context = context;

        }

        public int Id => (int)UseCaseEnum.EFRegisterUserCommand;

        public string Name => UseCaseEnum.EFRegisterUserCommand.ToString();

        public void Execute(RegisterUserDto request)

        {

            \_validator.ValidateAndThrow(request);

            var user = new Domain.User

            {

                FirstName = request.FirstName,

                LastName = request.LastName,

                Username = request.Username,

                Email = request.Email,

                Password = request.Password

            };

            user.Password = EasyEncryption.SHA.ComputeSHA256Hash(request.Password);

            \_context.Users.Add(user);

            \_context.SaveChanges();

            var userForUseCases = new List<int> { 2, 5, 6, 7, 8, 9, 12, 13, 14, 15, 16 };

            int id = user.Id;

            foreach (var uucId in userForUseCases)

            {

                user.UserUseCases.Add(new UserUseCase

                {

                    IdUser = id,

                    IdUseCase = uucId

                });

            }

            \_context.SaveChanges();

            \_sender.Send(new SendEmailDto

            {

                Subject = "Registration",

                Content = "<h2>Successfully Registered</h2>",

                SendTo = request.Email

            });

        }

    }

}

using System;

using System.Collections.Generic;

using System.Linq;

using System.Threading.Tasks;

using Microsoft.EntityFrameworkCore;

using Application.DataTransfer;

using Application.Queries;

using Application.Queries.Blog;

using Application.Searches;

using EFDataAccess;

namespace Implementation.Queries.Blog

{

    public class EFGetBlogsQuery : IGetBlogsQuery

    {

        private readonly BlogContext \_context;

        public EFGetBlogsQuery(BlogContext context)

        {

            \_context = context;

        }

        public int Id => (int)UseCaseEnum.EFGetBlogsQuery;

        public string Name => UseCaseEnum.EFGetBlogsQuery.ToString();

        public PagedResponse<GetBlogDto> Execute(BlogSearch search)

        {

            var blogs = \_context.Blogs.Include(x => x.Comments).Include(x => x.Likes).Include(x => x.BlogCategories).ThenInclude(x => x.Category).AsQueryable();

            if (!string.IsNullOrEmpty(search.Title) || !string.IsNullOrWhiteSpace(search.Title))

            {

                blogs = blogs.Where(x => x.Title.ToLower().Contains(search.Title.ToLower()));

            }

            if (!string.IsNullOrEmpty(search.Content) || !string.IsNullOrWhiteSpace(search.Content))

            {

                blogs = blogs.Where(x => x.Content.ToLower().Contains(search.Content.ToLower()));

            }

            if (search.DateFrom != null && search.DateFrom > search.DateTo)

            {

                blogs = blogs.Where(x => x.CreatedAt >= search.DateFrom);

            }

            if (search.DateTo != null && search.DateTo > search.DateFrom)

            {

                blogs = blogs.Where(x => x.CreatedAt <= search.DateTo);

            }

            if (search.IdCategory.HasValue)

            {

                blogs = blogs.Where(x => x.BlogCategories.Any(x => x.IdCategory == search.IdCategory.Value));

            }

            var skipCount = search.PerPage \* (search.Page - 1);

            var response = new PagedResponse<GetBlogDto>

            {

                CurrentPage = search.Page,

                ItemsPerPage = search.PerPage,

                TotalCount = blogs.Count(),

                Items = blogs.Skip(skipCount).Take(search.PerPage).Select(x => new GetBlogDto

                {

                    Id = x.Id,

                    Title = x.Title,

                    Content = x.Content,

                    DateCreated = x.CreatedAt,

                    Username = x.User.Username,

                    IdImage = x.IdImage,

                    ImageName = x.Image.ImagePath,

                    Categories = x.BlogCategories.Select(y => new CategoryDto

                    {

                        Id = y.Category.Id,

                        Name = y.Category.Name

                    }).ToList(),

                    Comments = x.Comments.Select(z => new SingleCommentDto

                    {

                        Id = z.Id,

                        Text = z.CommentText,

                        CreatedAt = z.CreatedAt,

                        Username = z.User.Username

                    }).ToList(),

                    Likes = x.Likes.Select(w => new LikeBlogDto

                    {

                        Id = w.Id,

                        Status = w.Status,

                        Username = w.User.Username

                    }).ToList()

                }).ToList()

            };

            return response;

        }

    }

}

\_context.Blogs: Ovo se odnosi na DbSet koji predstavlja tabelu blogova u bazi podataka. Blogs je svojevrsni interfejs između vaše aplikacije i tabele Blogs u bazi podataka. Blogs sadrži metode i svojstva koji vam omogućavaju izvršavanje upita nad bazom podataka, kao što su Include, Where, OrderBy, ToList, itd.

Include(x => x.Comments): Metoda **Include** se koristi za uključivanje povezanih entiteta u rezultujući upit. U ovom slučaju, x => x.Comments ukazuje da želimo da uključimo komentare (entitete) povezane sa svakim blogom. Ovo se **naziva "eager loading"** i osigurava da kada se dohvataju blogovi, odmah se dohvatiti i svi povezani komentari, tako da se izbegne "lazy loading", koji bi rezultirao dodatnim upitima u bazu podataka svaki put kada se pristupi komentarima.

Metoda **ThenInclude** se koristi za uključivanje drugog nivoa povezanih entiteta.

**AsQueryable():** Ova metoda pretvara DbSet u IQueryable, što omogućava dodavanje dodatnih uslova i filtriranje u upitu. Ovo je često korisno kada želite da dinamički kreirate upit u zavisnosti od različitih faktora ili kada želite da nastavite da gradite upit nakon što ste već dodali neka ograničenja.

Na primer, ako imate paginaciju sa 10 stavki po stranici i trenutno ste na drugoj stranici (search.Page = 2), formula će izračunati skipCount = 10 \* (2 - 1) = 10. To znači da treba preskočiti 10 stavki kako bi se došlo do početka druge stranice, jer je na prvoj stranici već prikazano prvih 10 stavki.

**PagedResponse**<GetBlogDto> označava da metoda Execute vraća paginiranu listu blogova gde je svaki blog predstavljen objektom klase GetBlogDto. Ovo omogućava jasno definisanje strukture i formata podataka koji se prenose između slojeva aplikacije.

Metoda **Skip()** se koristi za preskakanje određenog broja redova u rezultatu upita.

Metoda **Take()** se koristi za uzimanje određenog broja redova iz rezultata upita.

using Application;

using Application.DataTransfer;

using Application.Queries.Blog;

using EFDataAccess;

using Microsoft.EntityFrameworkCore;

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

namespace Implementation.Queries.Blog

{

    public class EFGetBlogQuery : IGetBlogQuery

    {

        private readonly BlogContext \_context;

        public EFGetBlogQuery(BlogContext context)

        {

            \_context = context;

        }

        public int Id => (int)UseCaseEnum.EFGetOneBlogQuery;

        public string Name => UseCaseEnum.EFGetOneBlogQuery.ToString();

        public ICollection<SingleCommentDto> parentComments { get; set; } = new List<SingleCommentDto>();

        GetBlogDto IQuery<GetBlogDto, int>.Execute(int search)

        {

            var blog = \_context.Blogs.Find(search);

            var query = \_context.Blogs.Include(l => l.Likes)

                .Include(i => i.Image)

                .Include(com => com.Comments)

                .Include(u => u.User)

                .Include(bc => bc.BlogCategories)

                .ThenInclude(c => c.Category)

                .FirstOrDefault(a => a.Id == search);

            var result = new GetBlogDto

            {

                Id = query.Id,

                Title = query.Title,

                Content = query.Content,

                DateCreated = query.CreatedAt,

                ImageName = query.Image.ImagePath,

                IdImage = query.IdImage,

                Username = query.User.Username,

                Categories = query.BlogCategories.Select(x => new CategoryDto

                {

                    Id = x.Category.Id,

                    Name = x.Category.Name

                }).ToList(),

                Likes = query.Likes.Select(z => new LikeBlogDto

                {

                    Id = z.Id,

                    Status = z.Status,

                    Username = z.User.Username,

                }).ToList(),

                Comments = query.Comments.Select(t => new SingleCommentDto

                {

                    Id = t.Id,

                    Text = t.CommentText,

                    CreatedAt = t.CreatedAt,

                    Children = t.ChildrenComments.Select(c => new SingleCommentDto

                    {

                        Id = c.Id,

                        IdParent = c.IdParent.Value,

                        Text = c.CommentText,

                        CreatedAt = c.CreatedAt,

                    }).ToList()

                }).ToList()

            };

            foreach (var res in result.Comments)

            {

                if (res.IdParent == null)

                {

                    parentComments.Add(res);

                }

            }

            result.Comments = parentComments;

            return result;

        }

    }

}

Metoda Execute(int search): prima int vrednost search, koja predstavlja identifikator bloga koji se traži.

Ova linija koda koristi generički interfejs IQuery<TResponse, TSearch>, gde su TResponse i TSearch generički tipovi koji se koriste za rezultat i parametre upita. U ovom slučaju, GetBlogDto je tip koji se koristi za rezultat upita (što je TResponse), a int je tip parametra pretrage (što je TSearch). Dakle, ovaj deo koda definiše implementaciju metode Execute interfejsa IQuery, koja prima identifikator bloga (int) kao parametar pretrage i vraća DTO objekat koji predstavlja informacije o tom blogu (GetBlogDto).

Prvo se koristi \_context.Blogs.Find(search) kako bi se pronašao blog sa datim identifikatorom. Zatim se izvršava upit nad bazom podataka da bi se dobio blog sa svim potrebnim informacijama, kao što su lajkovi, slike, komentari, korisnik i kategorije bloga. Dobijeni rezultat se mapira na DTO objekat GetBlogDto. U petlji se prolazi kroz komentare bloga kako bi se odvojili roditeljski komentari od dece. Nakon toga, roditeljski komentari se dodaju u listu parentComments. Na kraju se lista roditeljskih komentara dodeljuje svojstvu Comments rezultujućeg DTO objekta. Konačno, vraća se DTO objekat sa svim informacijama o blogu.

using System;

using System.Collections.Generic;

using System.Linq;

using System.Threading.Tasks;

using Application.DataTransfer;

using Application.Queries;

using Application.Queries.Category;

using Application.Searches;

using EFDataAccess;

namespace Implementation.Queries.Category

{

    public class EFGetCategoriesQuery : IGetCategoriesQuery

    {

        private readonly BlogContext \_context;

        public EFGetCategoriesQuery(BlogContext context)

        {

            \_context = context;

        }

        public int Id => (int)UseCaseEnum.EFGetCategoriesQuery;

        public string Name => UseCaseEnum.EFGetCategoriesQuery.ToString();

        public PagedResponse<CategoryDto> Execute(CategorySearch search)

        {

            var query = \_context.Categories.AsQueryable();

            if (!string.IsNullOrEmpty(search.Name) || !string.IsNullOrWhiteSpace(search.Name))

            {

                query = query.Where(x => x.Name.ToLower().Contains(search.Name.ToLower()));

            }

            query = query.Where(x => x.IsActive == true);

            var skipCount = search.PerPage \* (search.Page - 1);

            var response = new PagedResponse<CategoryDto>

            {

                CurrentPage = search.Page,

                ItemsPerPage = search.PerPage,

                TotalCount = query.Count(),

                Items = query.Skip(skipCount).Take(search.PerPage).Select(x => new CategoryDto

                {

                    Id = x.Id,

                    Name = x.Name

                }).ToList()

            };

            return response;

        }

    }

}

using System;

using System.Collections.Generic;

using System.Linq;

using System.Threading.Tasks;

using Application.DataTransfer;

using Application.Exceptions;

using Application.Queries.Category;

using EFDataAccess;

namespace Implementation.Queries.Category

{

    public class EFGetCategoryQuery : IGetCategoryQuery

    {

        private readonly BlogContext \_context;

        public EFGetCategoryQuery(BlogContext context)

        {

            \_context = context;

        }

        public int Id => (int)UseCaseEnum.EFGetOneCategoryQuery;

        public string Name => UseCaseEnum.EFGetOneCategoryQuery.ToString();

        public CategoryDto Execute(int search)

        {

            var category = \_context.Categories.Find(search);

            if (category == null)

            {

                throw new EntityNotFoundException(search, typeof(Domain.Category));

            }

            return new CategoryDto

            {

                Id = category.Id,

                Name = category.Name

            };

        }

    }

}

using System;

using System.Collections.Generic;

using System.Linq;

using System.Threading.Tasks;

using Application.DataTransfer;

using Application.Exceptions;

using Application.Queries.Comment;

using EFDataAccess;

using Microsoft.EntityFrameworkCore;

namespace Implementation.Queries.Comment

{

    public class EFGetCommentQuery : IGetCommentQuery

    {

        private readonly BlogContext \_context;

        public EFGetCommentQuery(BlogContext context)

        {

            \_context = context;

        }

        public int Id => (int)UseCaseEnum.EFGetOneUserQuery;

        public string Name => UseCaseEnum.EFGetOneUserQuery.ToString();

        public CommentDto Execute(int search)

        {

            var comment = \_context.Comments.Include(x => x.User).FirstOrDefault(y => y.Id == search);

            if (comment == null)

            {

                throw new EntityNotFoundException(search, typeof(Domain.Comment));

            }

            return new CommentDto

            {

                Id = comment.Id,

                Comment = comment.CommentText,

                Username = comment.User.Username,

                CreatedAt = comment.CreatedAt,

                IdParent = comment.IdParent

            };

        }

    }

}

using System;

using System.Collections.Generic;

using System.Linq;

using System.Threading.Tasks;

using Microsoft.EntityFrameworkCore;

using Application.DataTransfer;

using Application.Queries.User;

using EFDataAccess;

namespace Implementation.Queries.User

{

    public class EFGetUserQuery : IGetUserQuery

    {

        private readonly BlogContext \_context;

        public EFGetUserQuery(BlogContext context)

        {

            \_context = context;

        }

        public int Id => (int)UseCaseEnum.EFGetOneUserQuery;

        public string Name => UseCaseEnum.EFGetOneUserQuery.ToString();

        public SingleUserDto Execute(int search)

        {

            var user = \_context.Users.Include(x => x.UserUseCases).FirstOrDefault(x => x.Id == search);

            return new SingleUserDto

            {

                Id = user.Id,

                FirstName = user.FirstName,

                LastName = user.LastName,

                Username = user.Username,

                Email = user.Email,

                Password = user.Password,

                UserUseCases = user.UserUseCases.Select(x => new UserUseCaseDto

                {

                    IdUseCase = x.IdUseCase

                })

            };

        }

    }

}

using System;

using System.Collections.Generic;

using System.Linq;

using System.Threading.Tasks;

using EFDataAccess;

using Application.DataTransfer;

using Application.Queries;

using Application.Searches;

namespace Implementation.Queries

{

    public class EFGetUseCaseLogsQuery : IGetUseCaseLogsQuery

    {

        private readonly BlogContext \_context;

        public EFGetUseCaseLogsQuery(BlogContext context)

        {

            \_context = context;

        }

        public int Id => (int)UseCaseEnum.EFGetUseCaseLogQuery;

        public string Name => UseCaseEnum.EFGetUseCaseLogQuery.ToString();

        public PagedResponse<UseCaseLogDto> Execute(UseCaseLogSearch search)

        {

            var query = \_context.UseCaseLogs.AsQueryable();

            if (!string.IsNullOrEmpty(search.Actor) || !string.IsNullOrWhiteSpace(search.Actor))

            {

                query = query.Where(x => x.Actor.ToLower().Contains(search.Actor.ToLower()));

            }

            if (!string.IsNullOrEmpty(search.UseCaseName) || !string.IsNullOrWhiteSpace(search.UseCaseName))

            {

                query = query.Where(x => x.UseCaseName.ToLower().Contains(search.UseCaseName.ToLower()));

            }

            if (search.DateFrom != null && search.DateFrom >= search.DateTo)

            {

                query = query.Where(x => x.Date >= search.DateFrom);

            }

            if (search.DateTo != null && search.DateTo > search.DateFrom)

            {

                query = query.Where(x => x.Date <= search.DateTo);

            }

            var skipCount = search.PerPage \* (search.Page - 1);

            var response = new PagedResponse<UseCaseLogDto>

            {

                CurrentPage = search.Page,

                ItemsPerPage = search.PerPage,

                TotalCount = query.Count(),

                Items = query.Skip(skipCount).Take(search.PerPage).Select(x => new UseCaseLogDto

                {

                    Actor = x.Actor,

                    Data = x.Data,

                    Date = x.Date,

                    UseCaseName = x.UseCaseName

                }).ToList()

            };

            return response;

        }

    }

}

5.API

Ako pravim ovaj **API** kao sloj onda se ograđujem da koristim **HTTP** protokol koji će biti komunikacioni kanal između klijenta koji će biti neka Client Side Rendering aplikacija na računaru.   
Taj klijent šalje HTTP request koji dođe do sistema i ukoliko hoće da pretraži blogove to će na endpoint-u biti query. Ako hoće da da kreira blog to će biti command. A kad dođem na API – on će imati instrukciju koji je to use case koji je definisan u Application sloju na tom endpoint-u i preko koje implemntacije će se taj use case izvršiti, a onda će implemntaicja diktirati da li će to ići ka bazi podataka upotrebom EF ili na drugi način i onda će sam korak čuvanja podataka uzeti domenski objekat i smestiti ga u samu bazu podataka.

Imam sistem po slojevima gde imam jasno definisan način kako slojevi komuniciraju, ali klijentske aplikacije ne treba da budu svesne kako su granice našeg sistema definisane. Sa aspekta klijentske aplikacije - msilim na browser koji preko JavaScript-a pokušava AJAX-om da komunicira sa sistemom.

Klijent šalje zahtev na našu adresu i port 89.445.21.09:80 /api/blogs i poslao je HTTP-om POST request. On ne zna da to udara na kontroler. Naveo sam da kontroler delegira to ostalim delovima sistema.   
Kontroler ne treba da se promeni kada se promeni naš način implementacije koda.   
  
Kontroler ne treba da zavisi od implementacije nego od apstrakcije.  
Metoda u kontroleru je dobila interfejs, i kontroler treba da zna samo koji use case izvršava, a nije svestan koji je pozadinski kod koji se izvršava, da li je to na primer SQL, Mongo ili txt fajl.

API povezujem sa Implementation, Application, i DataAccess.  
  
Klijent uvek treba da dobije ili niz ili objekat od API-a, ne treba string!

JavaScript nema potrebe da vidi interne detalje klase koju smo preuzeli (nepotrebna svojstva).   
To je SOLID - **Interface Segregation Principle**

DI moze da vezuje interfejse za implementacije.

Kada kontroler zatraži određeni interfejs, posao DI container-a je da mu pošalje određenu implementaciju interfejsa. Ovime je kod modularan.

**Liskov Substitution Principle** kaže da svaki nadtip (apstraktna klasa, nadklasa, interfejs) mora da bude kompatabilna sa bilo kojom implementacijom.

Da bi aktor bio u kontroleru, komandama - Dodajem DI container  
services.AddTransient<>

**JWT**  
Autorizacija: Korisnik unese kredencijale u formu, klikom na ‚‚submit‘‘ dugme poslaće se prvi AJAX zahtev ka serveru koji će ići na /api/auth rutu. Proslediću username i password kroz body i zatim server proverava da li postoji korisnik sa tim username-om i password-om, ako postoji on ga spakuje i vrati klijentu objekat u kome je **token** {token: “snv321nv129xzfrn3AA3”}. Ukoliko ne postoji vraća statusni kod 401. Taj tekst neće klijentu značiti mnogo, ali na osnovu tog teksta server može svaki put da verifikuje klijenta. Klijent kad uzme token najčešće ga čuva u LocalStorage-u i svakim sledećim AJAX zahtevom prosleđuje ovaj token. Svakim sledećim zahtevom se dodaje i 1 header: Authorization: Bearer 8yt#&!Gbjhasbd1uvbyZX&^f#  
Na ovaj način server zna da nas prepozna.   
Klijent pored toga što dobija token i vraća svaki put serveru da bi ga ovaj validirao, mora da ima način da ima određene informacije o JWT-u. Iz tog razloga JWT je napravljen na specifičan način da ima 3 segmenta:  
1.Header – sadrži metapodatke o tokenima i tip algoritma. Ovo uključuje informacije poput tipa tokena(JWT) i tipa korišćenog algoritma(HMAC, RSA).  
2.Payload – sadrži samu korisničku informaciju ili tvrdnje(claims) koje se prenose između klijenta i servera. Kodiran je u JSON formatu.  
3.Signature – koristi se za proveru autentičnosti tokena. Potpis se generiše koristeći header i payload zajedno sa tajnim ključem koji samo server poseduje.

Instaliram novi paket: Microsoft.AspNetCore.AuthenticationJWTBearer

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

using Application;

using Application.Commands.Blog;

using Application.Commands.Category;

using Application.Commands.User;

using Application.Commands.Like;

using Application.Commands.Comment;

using Application.Queries;

using Application.Queries.Blog;

using Application.Queries.Category;

using Application.Queries.User;

using Application.Queries.Comment;

using Implementation.Validators.Blog;

using Implementation.Validators.Category;

using Implementation.Validators.User;

using Implementation.Validators.Like;

using Implementation.Validators.Comment;

using Implementation.Commands.Blog;

using Implementation.Commands.Category;

using Implementation.Commands.User;

using Implementation.Commands.Comment;

using Implementation.Commands.Like;

using Implementation.Queries;

using Implementation.Queries.Blog;

using Implementation.Queries.Category;

using Implementation.Queries.Comment;

using Implementation.Queries.User;

using Microsoft.AspNetCore.Authentication.JwtBearer;

using Microsoft.IdentityModel.Tokens;

using Newtonsoft.Json;

namespace API.Core

{

    public static class APIExtension

    {

        public static void LoadUseCases(this IServiceCollection services)

        {

            services.AddTransient<UseCaseExecutor>();

*// Commands*

            services.AddTransient<IRegisterUserCommand, EFRegisterUserCommand>();

            services.AddTransient<ICreateBlogCommand, EFCreateBlogCommand>();

            services.AddTransient<IUpdateBlogCommand, EFUpdateBlogCommand>();

            services.AddTransient<IDeleteBlogCommand, EFDeleteBlogCommand>();

            services.AddTransient<IUpdatePersonalBlogCommand, EFUpdatePersonalBlogCommand>();

            services.AddTransient<IDeletePersonalBlogCommand, EFDeletePersonalBlogCommand>();

            services.AddTransient<ICreateCategoryCommand, EFCreateCategoryCommand>();

            services.AddTransient<IUpdateCategoryCommand, EFUpdateCategoryCommand>();

            services.AddTransient<IDeleteCategoryCommand, EFDeleteCategoryCommand>();

            services.AddTransient<ICreateCommentCommand, EFCreateCommentCommand>();

            services.AddTransient<IUpdatePersonalCommentCommand, EFUpdatePersonalCommentCommand>();

            services.AddTransient<IDeleteCommentCommand, EFDeleteCommentCommand>();

            services.AddTransient<IDeletePersonalCommentCommand, EFDeletePersonalCommentCommand>();

            services.AddTransient<ICreateUserCommand, EFCreateUserCommand>();

            services.AddTransient<IUpdateUserCommand, EFUpdateUserCommand>();

            services.AddTransient<IDeleteUserCommand, EFDeleteUserCommand>();

            services.AddTransient<ILikeBlogCommand, EFLikeBlogCommand>();

*// Queries*

            services.AddTransient<IGetBlogsQuery, EFGetBlogsQuery>();

            services.AddTransient<IGetBlogQuery, EFGetBlogQuery>();

            services.AddTransient<IGetCategoriesQuery, EFGetCategoriesQuery>();

            services.AddTransient<IGetCategoryQuery, EFGetCategoryQuery>();

            services.AddTransient<IGetCommentQuery, EFGetCommentQuery>();

            services.AddTransient<IGetUserQuery, EFGetUserQuery>();

            services.AddTransient<IGetUseCaseLogsQuery, EFGetUseCaseLogsQuery>();

*// Validators*

            services.AddTransient<RegisterUserValidator>();

            services.AddTransient<CreateBlogValidator>();

            services.AddTransient<CreateCategoryValidator>();

            services.AddTransient<CreateCommentValidator>();

            services.AddTransient<CreateUserValidator>();

            services.AddTransient<LikeValidator>();

            services.AddTransient<UpdateBlogValidator>();

            services.AddTransient<UpdateCategoryValidator>();

            services.AddTransient<UpdateCommentValidator>();

            services.AddTransient<UpdateUserValidator>();

            services.AddTransient<DeleteBlogValidator>();

            services.AddTransient<DeleteCategoryValidator>();

            services.AddTransient<DeleteCommentValidator>();

            services.AddTransient<CreateUserValidator>();

            services.AddTransient<UpdateUserValidator>();

        }

        public static void AddApplicationActor(this IServiceCollection services)

        {

            services.AddTransient<IApplicationActor>(x =>

            {

                var accessor = x.GetService<IHttpContextAccessor>();

                var user = accessor.HttpContext.User;

                if (user.FindFirst("ActorData") == null)

                {

                    return new AnonymousActor();

                }

                var actorString = user.FindFirst("ActorData").Value;

                var actor = JsonConvert.DeserializeObject<JWTActor>(actorString);

                return actor;

            });

        }

        public static void AddJWT(this IServiceCollection services)

        {

            services.AddAuthentication(options =>

            {

                options.DefaultChallengeScheme = JwtBearerDefaults.AuthenticationScheme;

                options.DefaultSignInScheme = JwtBearerDefaults.AuthenticationScheme;

                options.DefaultScheme = JwtBearerDefaults.AuthenticationScheme;

            }).AddJwtBearer(cfg =>

            {

                cfg.RequireHttpsMetadata = false;

                cfg.SaveToken = true;

                cfg.TokenValidationParameters = new TokenValidationParameters

                {

                    ValidIssuer = "asp\_api",

                    ValidateIssuer = true,

                    ValidAudience = "Any",

                    IssuerSigningKey = new SymmetricSecurityKey(Encoding.UTF8.GetBytes("ThisIsMyVerySecretKey")),

                    ValidateIssuerSigningKey = true,

                    ValidateLifetime = true,

                    ClockSkew = TimeSpan.Zero

                };

            });

        }

    }

}

Sve metode u klasi **APIExtension** su **static** jer se radi o klasi koja definiše proširenja za **IServiceCollection** interfejs.

**LoadUseCases(IServiceCollection services):** Ova metoda je **extension method** koja se koristi za registrovanje svih use case-ova u aplikaciji. Use case-ovi obuhvataju komande (commands) i upite (queries) koji definišu funkcionalnosti koje aplikacija pruža. U ovoj metodi, use case-ovi se dodaju kao zavisnosti u Dependency Injection kontejner, tako da su kasnije dostupni za injektovanje u druge delove aplikacije.

**AddApplicationActor(IServiceCollection services):** Ova metoda takođe predstavlja extension method koje se koristi za registrovanje servisa koji omogućava pristup informacijama o trenutno autentifikovanom korisniku (aktoru) u aplikaciji. Kada korisnik pristupi aplikaciji, JWT token koji nosi informacije o korisniku se prosleđuje sa zahtevom. Ova metoda omogućava ekstrakciju informacija iz JWT tokena i njihovo korišćenje unutar aplikacije.

**AddJWT(IServiceCollection services):** Ova metoda se koristi za konfigurisanje autentifikacije putem JWT tokena. Ona dodaje podršku za autentifikaciju putem JWT tokena u ASP.NET Core aplikaciju. Konfiguracija obuhvata postavljanje parametara validacije tokena, kao što su validnost izdavača (issuer), provere potpisa, vremenski period validnosti tokena itd. Takođe, ova metoda omogućava registrovanje JWT token autentifikacije kao podrazumevanog šeme autentifikacije u aplikaciji.

using System;

using System.Collections.Generic;

using System.Linq;

using System.Threading.Tasks;

using Application;

namespace API.Core

{

    public class AnonymousActor : IApplicationActor

    {

        public int Id => 0;

        public string Identity => "Unauthorized user";

        public IEnumerable<int> AllowedUseCases => new List<int> { 1, 2, 3, 4, 5, 6, 7, 8, 23, 24 };

    }

}

Ovo je implementacija IApplicationActor interfejsa koja predstavlja anonimnog korisnika. Ovaj korisnik nema ID, identitet mu je "Unauthorized user", a dozvoljeni use case-ovi su navedeni u listi.

using Application;

namespace API.Core

{

    public class JWTActor : IApplicationActor

    {

        public int Id { get; set; }

        public string Identity { get; set; }

        public IEnumerable<int> AllowedUseCases { get; set; }

    }

}

Ovo je implementacija IApplicationActor interfejsa koja predstavlja korisnika autentikovanog putem JWT tokena. Ova klasa sadrži ID, identitet i dozvoljene upotrebe slučaja korisnika.

using System.IdentityModel.Tokens.Jwt;

using System.Security.Claims;

using System.Text;

using EFDataAccess;

using Microsoft.EntityFrameworkCore;

using Newtonsoft.Json;

using Microsoft.IdentityModel.Tokens;

using System.Security.Cryptography;

namespace API.Core

{

    public class JWTManager

    {

        private readonly BlogContext \_context;

        public JWTManager(BlogContext context)

        {

            \_context = context;

        }

        private static byte[] GenerateRandomBytes(int length)

        {

            byte[] randomBytes = new byte[length];

            using (var rng = new RNGCryptoServiceProvider())

            {

                rng.GetBytes(randomBytes);

            }

            return randomBytes;

        }

        public string MakeToken(string username, string password)

        {

            var user = \_context.Users.Include(u => u.UserUseCases)

                .FirstOrDefault(x => x.Username == username && x.Password == password && x.IsActive == true);

            if (user == null)

            {

                return null;

            }

            byte[] keyBytes = GenerateRandomBytes(32); *// 32 bajta = 256 bita*

            string secretKey = Convert.ToBase64String(keyBytes);

            var actor = new JWTActor

            {

                Id = user.Id,

                AllowedUseCases = user.UserUseCases.Select(x => x.IdUseCase),

                Identity = user.Username

            };

            var issuer = "asp\_api";

            var claims = new List<Claim>

            {

                new Claim(JwtRegisteredClaimNames.Jti, Guid.NewGuid().ToString(), ClaimValueTypes.String, issuer),

                new Claim(JwtRegisteredClaimNames.Iss, "asp\_api", ClaimValueTypes.String, issuer),

                new Claim(JwtRegisteredClaimNames.Iat, DateTimeOffset.UtcNow.ToUnixTimeSeconds().ToString(), ClaimValueTypes.Integer64, issuer),

                new Claim("UserId", actor.Id.ToString(), ClaimValueTypes.String, issuer),

                new Claim("ActorData", JsonConvert.SerializeObject(actor), ClaimValueTypes.String, issuer)

            };

            var key = new SymmetricSecurityKey(Encoding.UTF8.GetBytes(secretKey));

            var credentials = new SigningCredentials(key, SecurityAlgorithms.HmacSha256);

            var dateTimeNow = DateTime.UtcNow;

            var token = new JwtSecurityToken(

                issuer: issuer,

                audience: "Any",

                claims: claims,

                notBefore: dateTimeNow,

                expires: dateTimeNow.AddMinutes(120),

                signingCredentials: credentials

            );

            return new JwtSecurityTokenHandler().WriteToken(token);

        }

    }

}

Metoda **MakeToken** u klasi JWTManager koristi se za generisanje JWT tokena na osnovu korisničkog imena i lozinke koje su prosleđene kao ulazni parametri.

1.Pretraga korisnika: Metoda prvo traži korisnika u bazi podataka na osnovu korisničkog imena i lozinke koji su prosleđeni kao ulazni parametri. Ovo se obavlja preko BlogContext koji predstavlja kontekst baze podataka.

2.Provera korisnika: Nakon pretrage, vrši se provera da li je korisnik pronađen i da li je aktivan (atribut IsActive je postavljen na true). Ako korisnik nije pronađen ili nije aktivan, funkcija vraća null, što znači da nije moguće generisati JWT token za datog korisnika.

3.Generisanje ključa: Zatim se generiše slučajni niz bajtova dužine 32 bajta (256 bita) pomoću kriptografske slučajne generacije brojeva. Ovaj niz bajtova će biti korišćen **kao tajni ključ** za potpisivanje JWT tokena.

4.Priprema podataka o korisniku: Informacije o korisniku koje će biti sadržane u JWT tokenu se dodaju u obliku tvrdnji (claims). Ove tvrdnje obuhvataju identifikacioni broj korisnika (Id), dozvoljene slučajeve upotrebe koje korisnik ima pravo da izvrši (AllowedUseCases), i identitet korisnika (Identity).

5.Kreiranje JWT tokena: Nakon pripreme tvrdnji, kreira se novi JWT token pomoću JwtSecurityToken klase. U konstruktoru ove klase se postavljaju atributi tokena, kao što su izdavač (issuer), publika (audience), tvrdnje (claims), vreme kada token postaje validan (notBefore), vreme kada token ističe (expires), i potpis (signingCredentials) koji se koristi za potpisivanje tokena. Potpis se vrši pomoću tajnog ključa koji je generisan u prethodnom koraku.

6.Pisanje tokena: Na kraju, JWT token se pretvara u string pomoću JwtSecurityTokenHandler klase i vraća se kao rezultat metode.

using API.Core;

using Microsoft.AspNetCore.Mvc;

using Application.DataTransfer;

namespace API.Controllers

{

    [ApiController]

    [Route("api/[controller]")]

    public class TokenController : ControllerBase

    {

        private readonly JWTManager \_manager;

        public TokenController(JWTManager manager)

        {

            \_manager = manager;

        }

        [HttpPost]

        public IActionResult Post([FromBody] LoginDto request)

        {

            var token = \_manager.MakeToken(request.Username, request.Password);

            if (token == null)

            {

                return Unauthorized();

            }

            return Ok(new

            {

                token

            });

        }

    }

}

Ovaj kontroler predstavlja API endpoint koji omogućava generisanje JWT tokena na osnovu korisničkog imena i lozinke. Evo detaljnog objašnjenja:

**Anotacije**(oznake) kontrolera: Ova klasa je anotirana sa [ApiController], što označava da je kontroler API kontroler, i sa [Route("api/[controller]")], što definiše deo putanje URL-a na koji će zahtevi biti prosleđeni. [controller] deo putanje će biti zamenjen sa imenom kontrolera, što znači da će URL biti /api/Token za ovaj kontroler.

Konstruktor: Konstruktor klase TokenController prima JWTManager kao parametar. JWTManager je klasa koja se koristi za generisanje JWT tokena.

Metoda Post: Ova metoda je anotirana sa [HttpPost], što znači da će odgovarati HTTP POST zahtevima. Prima LoginDto objekat kao ulazni parametar, koji sadrži korisničko ime i lozinku.

Generisanje JWT tokena: Unutar metode Post, poziva se metoda MakeToken objekta \_manager (instanca JWTManager klase) sa korisničkim imenom i lozinkom koje su prosleđene kao deo tela zahteva. Ako je generisanje tokena uspešno, vraća se JWT token kao odgovor sa statusom 200 OK. U suprotnom, ako token nije generisan (npr. zbog nevalidnih korisničkih podataka), vraća se status 401 Unauthorized.

Ovaj kontroler **(Logovanje)** omogućava korisnicima da se autentifikuju na osnovu korisničkog imena i lozinke, i da zatim dobiju JWT token koji mogu koristiti za pristup zaštićenim resursima u aplikaciji.

namespace API.Core

{

    public class AppSettings

    {

        public string JwtSecretKey { get; set; }

        public string JwtIssuer { get; set; }

        public string EmailFrom { get; set; }

        public string EmailPassword { get; set; }

        public string ApplicationInstance { get; set; }

    }

}

Klasa **AppSettings** služi za čuvanje podešavanja (konfiguracija) aplikacije.

using System;

using System.Collections.Generic;

using System.Linq;

using System.Threading.Tasks;

using Application.Exceptions;

using FluentValidation;

using Newtonsoft.Json;

namespace API.Core

{

    public class GlobalExceptionHandler

    {

        private readonly RequestDelegate \_next;

        public GlobalExceptionHandler(RequestDelegate next)

        {

            \_next = next;

        }

        public async Task Invoke(HttpContext httpContext)

        {

            try

            {

                await \_next(httpContext);

            }

            catch (Exception ex)

            {

                httpContext.Response.ContentType = "application/json";

                object response = null;

                var statusCode = StatusCodes.Status500InternalServerError;

                switch (ex)

                {

                    case UnauthorizedUseCaseException \_:

                        statusCode = StatusCodes.Status403Forbidden;

                        response = new

                        {

                            message = "You are not allowed to execute this operation."

                        };

                        break;

                    case EntityNotFoundException prom:

                        statusCode = StatusCodes.Status404NotFound;

                        response = new

                        {

                            message = "Content not found"

                        };

                        break;

                    case ValidationException validationException:

                        statusCode = StatusCodes.Status422UnprocessableEntity;

                        response = new

                        {

                            message = "Failed due to validation errors.",

                            errors = validationException.Errors.Select(x => new

                            {

                                x.PropertyName,

                                x.ErrorMessage

                            })

                        };

                        break;

                    case ConflictException exception:

                        statusCode = StatusCodes.Status409Conflict;

                        response = new

                        {

                            message = exception.Message.ToString()

                        };

                        break;

                }

                httpContext.Response.StatusCode = statusCode;

                if (response != null)

                {

                    await httpContext.Response.WriteAsync(JsonConvert.SerializeObject(response));

                    return;

                }

                await Task.FromResult(httpContext.Response);

            }

        }

    }

}

**Global Exception Handler**  
Da ne bismo u kontroleru svaki put pisali try/catch.Prepoznace da se desio exception

Exception registrovati kao Middleware u Startup-u. Request će proći kroz niz middleware-a, pa i kroz ovaj koji dodajem.Dobija kao ulazni parametar delegat ka sledećem Middleware-u

Pokušava da izvrši sve što će se desiti naredno, pa i naš use case. Ako dođe do izuzetka, u catch pišem kod koji će upravljati tim izuzetkom.

Presreli smo izuzetak. Ono što je radio .NET ispod haube sad ručno radim.

`object response = null;` telo response-a, za početak je NULL.

U switch-u pokušavam da prepoznam izuzetak i da menjam statusni kod.  
Validation exception iz Validator-a hvatam u globalnom handler-u.

Prilikom definisanja ValidationException-a oni su definisali 1 svojstvo Errors koje je get i sadrži niz grešaka. Mi imamo pristup, svaki element dodajemo u JSON body.

Od svih grešaka koje sadrži ‚‚Errors” ja ću uz pomoć Select-a izabrati koje mi trebaju da pošaljem klijentu.

Napravljen je mehanizam da imam automatsku obradu izuzetaka.

using API.Core;

using Application;

using EFDataAccess;

using Implementation.Logging;

using Microsoft.OpenApi.Models;

using Application.Commands.Email;

using Implementation.Commands.Email;

namespace API

{

    public class Startup

    {

        public Startup(IConfiguration configuration)

        {

            Configuration = configuration;

        }

        public IConfiguration Configuration { get; }

        public void ConfigureServices(IServiceCollection services)

        {

            var appSettings = new AppSettings();

            Configuration.Bind(appSettings);

            services.AddTransient<BlogContext>();

            services.LoadUseCases();

            services.AddTransient<IUseCaseLogger, EFDatabaseLogger>();

            services.AddApplicationActor();

            services.AddHttpContextAccessor();

            services.AddJWT();

            services.AddTransient<JWTManager>();

            services.AddSwaggerGen(c =>

            {

                c.SwaggerDoc("v1", new OpenApiInfo { Title = "BlogAPI", Version = "v1" });

                c.AddSecurityDefinition("Bearer", new OpenApiSecurityScheme

                {

                    Description = @"JWT Authorization header using the Bearer scheme. \r\n\r\n

                      Enter 'Bearer' [space] and then your token in the text input below.

                      \r\n\r\nExample: 'Bearer 12345abcdef'",

                    Name = "Authorization",

                    In = ParameterLocation.Header,

                    Type = SecuritySchemeType.ApiKey,

                    Scheme = "Bearer"

                });

                c.AddSecurityRequirement(new OpenApiSecurityRequirement()

                {

                    {

                        new OpenApiSecurityScheme

                        {

                            Reference = new OpenApiReference

                            {

                                Type = ReferenceType.SecurityScheme,

                                Id = "Bearer"

                            },

                            Scheme = "oauth2",

                            Name = "Bearer",

                            In = ParameterLocation.Header

                        },

                        new List<string>()

                    }

                });

            });

            services.AddTransient<IEmailSender, SMTPEmailSender>();

            services.AddControllers();

        }

  public void Configure(IApplicationBuilder app, IWebHostEnvironment env)

        {

*//  Proverava da li je aplikacija pokrenuta u razvojnom okruženju. Ako jeste, dodaje se middleware komponenta DeveloperExceptionPage, koja prikazuje detaljne informacije o izuzecima prilikom razvoja aplikacije.*

            if (env.IsDevelopment())

            {

                app.UseDeveloperExceptionPage();

            }

*// Dodaje middleware za rutiranje, što omogućava aplikaciji da odredi koji kod će se izvršiti na osnovu dolaznog HTTP zahteva.*

            app.UseRouting();

*// Dodaje middleware koji omogućava serveru da poslužuje statičke datoteke, poput HTML, CSS, JavaScript i slika.*

            app.UseStaticFiles();

*// Dodaje middleware za autorizaciju, koji omogućava aplikaciji da proveri da li korisnik ima odgovarajuće dozvole za pristup određenom resursu.*

            app.UseAuthorization();

*// Dodaje middleware za autentifikaciju, koji omogućava aplikaciji da autentifikuje korisnike na osnovu pristiglih kredencijala ili tokena.*

            app.UseAuthentication();

*// Dodaje middleware za obrađivanje Cross-Origin Resource Sharing (CORS) zahteva. Ovaj middleware omogućava definisanje politika CORS-a koje određuju koje origin domene su dozvoljene da pristupaju resursima na serveru.*

            app.UseCors(x =>

            {

                x.AllowAnyOrigin();

                x.AllowAnyMethod();

                x.AllowAnyHeader();

            });

*// Dodaje middleware za podršku Swagger-u, koji generiše dokumentaciju API-ja na osnovu definicija ruta i kontrolera u aplikaciji.*

            app.UseSwagger();

*// Dodaje middleware koji generiše HTML interfejs za Swagger dokumentaciju, omogućavajući pregled API specifikacija putem web pregledača.*

            app.UseSwaggerUI(c =>

            {

                c.SwaggerEndpoint("/swagger/v1/swagger.json", "BlogAPI v1");

            });

*// Dodaje middleware komponentu GlobalExceptionHandler koja obrađuje sve izuzetke koji nisu već obrađeni i pruža odgovarajući odgovor korisniku ili aplikaciji.*

            app.UseMiddleware<GlobalExceptionHandler>();

*// Dodaje middleware za definisanje krajnjih tačaka (endpoints) aplikacije, tj. mapiranje HTTP zahteva na odgovarajuće akcije u kontrolerima.*

            app.UseEndpoints(endpoints =>

            {

                endpoints.MapControllers();

            });

        }

    }

}

**Startup** klasa se koristi za konfigurisanje servisa koje aplikacija koristi, kao i za konfigurisanje različitih delova aplikacije kao što su middleware-i, ruta, i sl.  
  
Konstruktor prima IConfiguration objekat koji se koristi za čitanje konfiguracionih vrednosti aplikacije.

Metod **ConfigureServices**:

Ovaj metod se koristi za konfigurisanje servisa koje aplikacija koristi. Ovde se obično konfigurišu servisi za pristup podacima, servisi za autentifikaciju i autorizaciju, kao i servisi koji su specifični za aplikaciju.

Prvo se čita konfiguracija aplikacije iz appsettings.json fajla i vezuje se za objekat AppSettings.

Zatim se konfigurišu servisi za pristup podacima (u ovom slučaju, BlogContext).

Poziva se metoda LoadUseCases() koja konfiguriše sve upotrebljavane slučajeve (use cases) u aplikaciji.

Registracija servisa za evidenciju korišćenja slučajeva (IUseCaseLogger) kao implementacija za bazu podataka (EFDatabaseLogger).

Pozivaju se ekstenzije za dodavanje podrške za glumce aplikacije (AddApplicationActor) i za JWT autentifikaciju (AddJWT).

**AddHttpContextAccessor** je ASP.NET Core ekstenzija koja se koristi za registraciju HttpContextAccessor servisa. Ovaj servis omogućava pristup trenutnom HTTP zahtevu unutar aplikacije.  
Kada se HttpContextAccessor registruje kao servis, može se ubrizgati u druge servise ili kontrolere u aplikaciji, omogućavajući im da pristupe informacijama o trenutnom HTTP zahtevu, kao što su URL, zaglavlja, sesija, identitet korisnika i druge korisne informacije.Na primer, možete koristiti HttpContextAccessor da biste pristupili informacijama o autentifikovanom korisniku ili da biste dobili trenutni URL za generisanje povratnih linkova. Ovo može biti korisno u različitim delovima aplikacije, kao što su filtri, servisi, kontroleri i tako dalje.Registracija HttpContextAccessor servisa pomaže u olakšavanju pristupa kontekstu HTTP zahteva i čini ga dostupnim unutar cele aplikacije gde god je potrebno.

Dodaje se podrška za Swagger dokumentaciju (AddSwaggerGen), gde se definiše verzija API-ja i opis njegovih funkcionalnosti. Takođe se dodaje definicija sigurnosne šeme za JWT autentifikaciju.

Konačno, dodaju se servisi za slanje e-pošte (IEmailSender) i kontroleri (AddControllers).

Metoda **Configure** je deo ASP.NET Core aplikacije i koristi se za konfigurisanje middleware komponenti koje se koriste za obradu HTTP zahteva.

Ova metoda se poziva prilikom pokretanja aplikacije i konfiguriše middleware lanac za obradu HTTP zahteva. Svaki middleware dodaje specifičnu funkcionalnost koja se izvršava u okviru tog lanca, u skladu sa redosledom u kome su dodati.

using Microsoft.AspNetCore.Mvc;

using Application;

using Application.DataTransfer;

using Application.Commands.Blog;

using Application.Queries.Blog;

using Application.Searches;

using Application.Commands.Like;

namespace API.Controllers

{

    [ApiController]

    [Route("api/[controller]")]

    public class BlogsController : ControllerBase

    {

        private readonly UseCaseExecutor \_executor;

        private readonly IApplicationActor \_actor;

        public BlogsController(UseCaseExecutor executor, IApplicationActor actor)

        {

            \_executor = executor;

            \_actor = actor;

        }

        [HttpPost]

        [Route("/api/like")]

        public IActionResult Like([FromBody] LikeDto request, [FromServices] ILikeBlogCommand command)

        {

            string json = System.Text.Json.JsonSerializer.Serialize(request);

            request.IdUser = \_actor.Id;

            \_executor.ExecuteCommand(command, request);

            return StatusCode(StatusCodes.Status201Created);

        }

        [HttpGet]

        public IActionResult Get([FromQuery] BlogSearch dto, [FromServices] IGetBlogsQuery query)

        {

            return Ok(\_executor.ExecuteQuery(query, dto));

        }

        [HttpGet("{id}", Name = "GetBlog")]

        public IActionResult Get(int id, [FromServices] IGetBlogQuery query)

        {

            return Ok(\_executor.ExecuteQuery(query, id));

        }

        [HttpPost]

        public IActionResult Post([FromBody] InsertBlogDto dto, [FromServices] ICreateBlogCommand command)

        {

            \_executor.ExecuteCommand(command, dto);

            return StatusCode(StatusCodes.Status201Created);

        }

        [HttpPut("{id}")]

        public IActionResult Put(int id, [FromBody] UpdateBlogDto dto, [FromServices] IUpdateBlogCommand command)

        {

            dto.Id = id;

            \_executor.ExecuteCommand(command, dto);

            return NoContent();

        }

        [HttpDelete("{id}")]

        public IActionResult Delete(int id, [FromServices] IDeleteBlogCommand command)

        {

            \_executor.ExecuteCommand(command, id);

            return NoContent();

        }

    }

}

using Microsoft.AspNetCore.Mvc;

using Application;

using Application.Commands.Category;

using Application.DataTransfer;

using Application.Queries.Category;

using Application.Searches;

namespace API.Controllers

{

    [ApiController]

    [Route("api/[controller]")]

    public class CategoriesController : ControllerBase

    {

        private readonly IApplicationActor \_actor;

        private readonly UseCaseExecutor \_executor;

        public CategoriesController(IApplicationActor actor, UseCaseExecutor executor)

        {

            \_actor = actor;

            \_executor = executor;

        }

        [HttpGet]

        public IActionResult Get([FromQuery] CategorySearch search, [FromServices] IGetCategoriesQuery query)

        {

            return Ok(\_executor.ExecuteQuery(query, search));

        }

        [HttpGet("{id}", Name = "GetCategory")]

        public IActionResult Get(int id, [FromServices] IGetCategoryQuery query)

        {

            return Ok(\_executor.ExecuteQuery(query, id));

        }

        [HttpPost]

        public IActionResult Post([FromBody] CategoryDto dto, [FromServices] ICreateCategoryCommand command)

        {

            \_executor.ExecuteCommand(command, dto);

            return StatusCode(StatusCodes.Status201Created);

        }

        [HttpPut("{id}")]

        public IActionResult Put(int id, [FromBody] CategoryDto dto, [FromServices] IUpdateCategoryCommand command)

        {

            dto.Id = id;

            \_executor.ExecuteCommand(command, dto);

            return NoContent();

        }

        [HttpDelete("{id}")]

        public IActionResult Delete(int id, [FromServices] IDeleteCategoryCommand command)

        {

            \_executor.ExecuteCommand(command, id);

            return NoContent();

        }

    }

}

using Application;

using Application.Commands.Comment;

using Application.DataTransfer;

using Application.Queries.Comment;

using Microsoft.AspNetCore.Mvc;

namespace API.Controllers

{

    [ApiController]

    [Route("api/[controller]")]

    public class CommentsController : ControllerBase

    {

        private readonly UseCaseExecutor \_executor;

        private readonly IApplicationActor \_actor;

        public CommentsController(UseCaseExecutor executor, IApplicationActor actor)

        {

            \_executor = executor;

            \_actor = actor;

        }

        [HttpGet("{id}", Name = "GetComment")]

        public IActionResult Get(int id, [FromQuery] CommentDto dto, [FromServices] IGetCommentQuery query)

        {

            return Ok(\_executor.ExecuteQuery(query, id));

        }

        [HttpPost]

        public IActionResult Post([FromBody] InsertCommentDto dto, [FromServices] ICreateCommentCommand command)

        {

            \_executor.ExecuteCommand(command, dto);

            return StatusCode(StatusCodes.Status201Created);

        }

        [HttpPut("{id}")]

        public IActionResult Put(int id, [FromBody] InsertCommentDto dto, [FromServices] IUpdatePersonalCommentCommand command)

        {

            dto.Id = id;

            dto.IdUser = \_actor.Id;

            \_executor.ExecuteCommand(command, dto);

            return NoContent();

        }

        [HttpDelete("{id}")]

        public IActionResult Delete(int id, [FromServices] IDeleteCommentCommand command)

        {

            \_executor.ExecuteCommand(command, id);

            return NoContent();

        }

        [HttpDelete("user/{id}")]

        public IActionResult DeletePersonalComment(int id, [FromServices] IDeletePersonalCommentCommand command)

        {

            \_executor.ExecuteCommand(command, id);

            return NoContent();

        }

    }

}

using Microsoft.AspNetCore.Mvc;

using Application;

using Application.DataTransfer;

using Domain;

using EFDataAccess;

namespace API.Controllers

{

    [ApiController]

    [Route("api/[controller]")]

    public class ImagesController : ControllerBase

    {

        private readonly IApplicationActor \_actor;

        private readonly UseCaseExecutor \_executor;

        private readonly BlogContext \_context;

        public ImagesController(BlogContext context, UseCaseExecutor executor, IApplicationActor actor)

        {

            \_context = context;

            \_executor = executor;

            \_actor = actor;

        }

        [HttpPost]

        public IActionResult Post([FromForm] UploadImageDto dto)

        {

            var guid = Guid.NewGuid();

            var extension = Path.GetExtension(dto.Image.FileName);

            var newFileName = guid + extension;

            var uploadsFolder = Path.Combine(Directory.GetCurrentDirectory(), "wwwroot", "Images");

            var filePath = Path.Combine(uploadsFolder, newFileName);

            Directory.CreateDirectory(uploadsFolder);

            using (var fileStream = new FileStream(filePath, FileMode.Create))

            {

                dto.Image.CopyTo(fileStream);

            }

            var image = new Image

            {

                ImagePath = newFileName

            };

            \_context.Images.Add(image);

            \_context.SaveChanges();

            return StatusCode(StatusCodes.Status201Created);

        }

    }

}

using Application;

using Application.Commands.User;

using Application.DataTransfer;

using Microsoft.AspNetCore.Mvc;

namespace API.Controllers

{

    [ApiController]

    [Route("api/[controller]")]

    public class RegisterController : ControllerBase

    {

        private readonly UseCaseExecutor \_executor;

        public RegisterController(UseCaseExecutor executor)

        {

            \_executor = executor;

        }

        [HttpPost]

        public IActionResult Post([FromBody] RegisterUserDto dto, [FromServices] IRegisterUserCommand command)

        {

            \_executor.ExecuteCommand(command, dto);

            return StatusCode(StatusCodes.Status201Created);

        }

    }

}

using Application;

using Application.Queries;

using Application.Searches;

using Microsoft.AspNetCore.Mvc;

namespace API.Controllers

{

    [ApiController]

    [Route("api/[controller]")]

    public class UseCaseLogsController : ControllerBase

    {

        private readonly UseCaseExecutor \_executor;

        public UseCaseLogsController(UseCaseExecutor executor)

        {

            \_executor = executor;

        }

        [HttpGet]

        public IActionResult Get([FromQuery] UseCaseLogSearch search, [FromServices] IGetUseCaseLogsQuery query)

        {

            return Ok(\_executor.ExecuteQuery(query, search));

        }

    }

}

using Microsoft.AspNetCore.Mvc;

using Application;

using Application.Commands.User;

using Application.DataTransfer;

using Application.Queries.User;

using Application.Searches;

namespace API.Controllers

{

    [ApiController]

    [Route("api/[controller]")]

    public class UsersController : ControllerBase

    {

        private readonly UseCaseExecutor \_executor;

        public UsersController(UseCaseExecutor executor)

        {

            \_executor = executor;

        }

        [HttpGet]

        public IActionResult Get([FromServices] IGetUsersQuery query, [FromQuery] UserSearch search)

        {

            return Ok(\_executor.ExecuteQuery(query, search));

        }

        [HttpGet("{id}", Name = "GetUser")]

        public IActionResult Get(int id, [FromServices] IGetUserQuery query)

        {

            return Ok(\_executor.ExecuteQuery(query, id));

        }

        [HttpPost]

        public IActionResult Post([FromBody] InsertUserDto dto, [FromServices] ICreateUserCommand command)

        {

            \_executor.ExecuteCommand(command, dto);

            return StatusCode(StatusCodes.Status201Created);

        }

        [HttpPut("{id}")]

        public IActionResult Put(int id, [FromBody] UpdateUserDto dto, [FromServices] IUpdateUserCommand command)

        {

            dto.Id = id;

            \_executor.ExecuteCommand(command, dto);

            return NoContent();

        }

        [HttpDelete("{id}")]

        public IActionResult Delete(int id, [FromServices] IDeleteUserCommand command)

        {

            \_executor.ExecuteCommand(command, id);

            return NoContent();

        }

    }

}