Dokumentacija Blog projekta

🏁 Uvod

Blog platforma je razvijena sa ciljem da korisnicima omogući kreiranje, uređivanje i praćenje blog postova uz napredne funkcionalnosti kao što su komentarisanje, lajkovanje i sistem notifikacija u realnom vremenu.

Projekat pruža interaktivno korisničko iskustvo, omogućavajući korisnicima da prate omiljene autore i primaju obaveštenja o njihovim aktivnostima.

Sistem je izgrađen korišćenjem **ASP.NET Core 8.0** za backend, dok je frontend razvijen u **React.js**. Baza podataka koristi **SQL Server**, a podacima se upravlja pomoću **Entity Framework Core ORM-a**.

CQRS (Command Query Responsibility Segregation) arhitektura omogućava odvajanje upita i komandi, što poboljšava performanse i održavanje koda.

🔑 Ključne funkcionalnosti

✓ **Pisanje i uređivanje postova** – Korisnici mogu kreirati i uređivati svoje blog postove.
✓ **Komentarisanje i lajkovanje** – Mogućnost interakcije sa sadržajem putem komentara i lajkova.
✓ **Praćenje autora** – Korisnici mogu pratiti omiljene autore i biti obavešteni o njihovim aktivnostima.
✓ **Notifikacije u realnom vremenu** – Koristi se **SignalR** za slanje obaveštenja kada dođe do interakcija (komentari, lajkovi, novi postovi).
✓ **Napredna autentifikacija i autorizacija** – JWT i Firebase omogućavaju bezbedan pristup sistemu.
✓ **Filtriranje i paginacija** – Korisnici mogu lako pretraživati i sortirati sadržaj po različitim kriterijumima.

🎯 Ciljevi projekta

✓Omogućiti korisnicima lako kreiranje, uređivanje i pregledanje blog postova.
✓Implementirati sistem praćenja autora i notifikacija u realnom vremenu.
✓Osigurati siguran sistem autentifikacije i autorizacije pomoću JWT i Firebase logovanja.
✓Omogućiti napredne opcije filtriranja, sortiranja i pretrage sadržaja.
✓Obezbediti modularnu i skalabilnu arhitekturu, primenjujući SOLID principe.

☎ Arhitektura

Aplikacija je dizajnirana kao višeslojni sistem, pri čemu svaki sloj ima jasno definisanu odgovornost:

1. Domain sloj – Definiše poslovne entitete i pravila aplikacije. Ovaj sloj je nezavistan od tehničkih implementacija i sadrži samo poslovnu logiku.

2. EFDataAccess sloj – Upravlja bazom podataka pomoću Entity Framework Core. Ovaj sloj sadrži **DbContext**, konfiguracije entiteta i mehanizme za migracije baze podataka.
Takođe, koristi **soft delete** za brisanje podataka i podržava globalne upitne filtere.

3. Application sloj – Implementira poslovnu logiku kroz **CQRS** pristup.
Definiše interfejse za komande (**commands**) i upite (**queries**), kao i **DTO**-ove za prenos podataka.
Ovaj sloj takođe sadrži validaciju podataka i logovanje use-case operacija.

4. Implementation sloj – Sprovodi specifične use-case-ove i povezuje aplikaciju sa bazom.
Ovaj sloj sadrži implementacije servisa, repozitorijuma i specijalizovane funkcionalnosti kao što su **notifikacije (SignalR), slanje emailova, upravljanje korisnicima** i slično.

5. API sloj – Omogućava **HTTP** komunikaciju između frontenda i backenda.
Izgrađen u **ASP.NET Core Web API**, pruža **RESTful** endpoint-e za **CRUD** operacije, autentifikaciju i druge funkcionalnosti.

6. Client sloj – Predstavlja frontend aplikacije razvijen u **React.js**.
Komunicira sa API slojem pomoću HTTP zahteva (**Fetch API**).
Upravljanje stanjem aplikacije vrši se pomoću **Redux Toolkit-a**, dok se **Tailwind CSS** koristi za stilizaciju interfejsa.
Klijent sadrži mehanizme za prikaz paginiranih podataka, interakciju sa korisnicima i ažuriranje UI-a u realnom vremenu koristeći **SignalR**.

🔐 Bezbednost

Autentifikacija i **autorizacija** su osigurani pomoću **JWT** i **Firebase** autentifikacije.
Podaci su validirani pomoću **FluentValidation**, dok **globalni exception handler** omogućava centralizovano rukovanje greškama.

🚀 Performanse i skalabilnost

Kako bi aplikacija bila efikasna i skalabilna, primenjene su sledeće tehnike:

✓**Paginacija, filtriranje i sortiranje –** Omogućavaju brzo pretraživanje sadržaja.
✓**Asinhrone operacije i transakcije –** Obrađuju podatke bez blokiranja aplikacije.
✓**BulkExtensions –** Optimizuju rad sa velikim količinama podataka.

Tema Projekta

Blog platforma je razvijena kako bi korisnicima omogućila kreiranje, praćenje i interakciju sa blog postovima, uz podršku za notifikacije u realnom vremenu, naprednu autentifikaciju i opcije pretrage. Cilj projekta je da pruži intuitivno korisničko iskustvo kroz funkcionalnosti kao što su komentarisanje, lajkovanje i sistem praćenja omiljenih autora.

🚀 Ključne funkcionalnosti

✓ **Pisanje i uređivanje postova** – Autori mogu kreirati blog postove, dodavati slike i uređivati sadržaj.
✓ **Kategorije postova** – Organizacija sadržaja kroz sistem kategorija omogućava lakše pretraživanje.
✓ **Komentarisanje i lajkovanje** – Korisnici mogu ostavljati komentare i reagovati na sadržaj pomoću lajkova.
✓ **Sistem praćenja autora** – Mogućnost da korisnici prate autore i dobijaju obaveštenja o njihovim novim objavama.
✓ **Notifikacije u realnom vremenu** – SignalR omogućava korisnicima da dobijaju obaveštenja kada neko lajkuje, komentariše ili napiše novi post.
✓ **Napredna autentifikacija i autorizacija** – JWT i Firebase autentifikacija omogućavaju siguran pristup sistemu.
✓ **Sortiranje, filtriranje i paginacija** – Korisnici mogu lako pronaći relevantan sadržaj kroz razne filtere i pretragu.
✓ **Use Case pristup** – Sistem koristi **CQRS arhitekturu**, što omogućava preciznu organizaciju poslovne logike i logovanje korisničkih akcija.
✓ **Asinhrone operacije i transakcije** – Obrađuju se velike količine podataka bez blokiranja aplikacije.
✓ **Globalni Exception Handling** – Centralizovano rukovanje greškama i logovanje izuzetaka.
✓ **Brisanje i uređivanje komentara/postova** – Korisnici mogu uređivati i brisati svoje postove i komentare, dok administratori mogu upravljati celim sadržajem.
✓ **Responsive dizajn –** Aplikacija je prilagođena za korišćenje na desktop i mobilnim

uređajima.
✓ **Zahtevi za autore** – Korisnici mogu podneti zahtev da postanu autori, a administratori ih mogu odobriti ili odbiti.
✓ **Obeležavanje pročitanih notifikacija** – Korisnici mogu označiti notifikacije kao pročitane.
✓ **Moderacija sadržaja** – Administrator može upravljati korisnicima, uklanjati postove i komentare koji krše pravila.
✓ **Prikaz aktivnosti korisnika** – Log sistema omogućava administratorima da prate koje akcije su korisnici izvršili (logovanje use-case-ova).
✓ **Korisnički profili** – Korisnici mogu da uređuju svoj profil.

Korišćene Tehnologije

Backend:

- **ASP.NET Core 8.0** - Web API
- **C#** - Glavni programski jezik
- **Entity Framework Core** - ORM za rad sa bazom podataka
- **SignalR** - Real-time komunikacija (notifikacije)
- **JWT (JSON Web Token)** - Autentifikacija i autorizacija
- **CQRS pristup** - Razdvajanje upita i komandi za efikasnije upravljanje podacima
- **Sentry** – Logovanje grešaka i praćenje performansi

Frontend:

- **React.js** - Klijentska aplikacija
- **Redux Toolkit** - Upravljanje globalnim stanjem aplikacije
- **Tailwind CSS, Flowbite React** - Stilizacija korisničkog interfejsa
- **Fetch API** - HTTP komunikacija sa backendom

Baza Podataka:

- **SQL Server** - Relaciona baza podataka
- **EF Core Migrations** - Upravljanje migracijama baze

Redosled kojim implementiram projekte je sledeći:

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

Kako komuniciraju slojevi?

U višeslojnim arhitekturama, komunikacija ide od najvišeg sloja (npr. API) prema najnižem (npr. EFDataAccess), što znači da API poziva Application sloj, koji zatim komunicira sa Domain slojem, ili eventualno poziva direktno neke servise iz Implementation ili pristupa podacima.
Međutim, ovo ne znači nužno da svaki sloj striktno komunicira samo sa onim ispod njega. Evo nekoliko ključnih tačaka:

Usmereni tok zavisnosti

Idealno, zavisnosti bi trebalo da teku "prema unutra". U Clean Architecture ili sličnim pristupima, unutrašnji slojevi (Domain i Application) su potpuno nezavisni od spoljašnjih implementacija (kao što su EFDataAccess, API, Client).
To se postiže korišćenjem **Dependency Inversion** principa – višim slojevima se ubrizgavaju interfejsi koji su definisani u unutrašnjim slojevima, a konkretne implementacije dolaze iz spoljašnjih slojeva (kao što je Implementation ili EFDataAccess).
Na primer, Application sloj može definisati interfejs za repozitorijum, dok je njegova konkretna implementacija smeštena u Implementation sloju.

Preskakanje slojeva

U praksi, ponekad je potrebno da neki viši sloj direktno koristi entitete iz Domain sloja, bez prolaska kroz Application sloj.
Na primer, API kontroleri mogu direktno koristiti DTO (Data Transfer Objects) koje su bazirane na Domain modelima, ali to treba raditi pažljivo kako ne bi došlo do nepotrebne zavisnosti.

Primena principa separacije odgovornosti

Svaki sloj ima jasnu ulogu.
API sloj ne bi trebao da sadrži poslovnu logiku – on samo preuzima zahteve, validira ih na nivou transporta i prosleđuje Application sloju.
Isto tako, Application sloj se ne bavi tehničkim detaljima pristupa bazi, što je zadatak EFDataAccess sloja.

Reference među projektima

Iz mojih konfiguracija se može videti kako su postavljene reference među projektima:

API Projekat

Referencira Domain, EFDataAccess, Application, i Implementation projekte.
To znači da API ima pristup svim ključnim delovima sistema, ali to je uobičajeno zato što je API ulazna tačka aplikacije.

Ovakva referenca omogućava API-ju da orkestrira operacije pozivajući servise definisane u Application sloju, a koji su implementirani koristeći logiku iz Domain i EFDataAccess slojeva.

Ostali projekti

Svaki projekat ima svoje specifične reference koje odražavaju njihovu ulogu.
Na primer, projekat koji sadrži samo **Domain** model ima referencu samo ka Domain projektu, dok **EFDataAccess** i **Application** projekti imaju reference prema Domainu, a **Implementation** dodatno prema EFDataAccess i Application slojevima.

Najbolja praksa

- Decoupling i Dependency Injection: Koristi DI kako bi se osigurala niska povezanost između slojeva.
- Definisanje jasnih interfejsa: Time se omogućava zamena konkretnih implementacija bez uticaja na više slojeve.
- Separation of concerns: Svaki sloj treba da bude odgovoran samo za svoju oblast, čime se olakšava testiranje i održavanje koda.

Domain sloj

Komanda za kreiranje projekta: *dotnet new classlib -n Domain*

Sadrži sve osnovne entitete i domenske modele koji opisuju poslovnu logiku i pravila aplikacije.
Tu se nalaze klase koje definišu poslovne koncepte i njihove međusobne odnose.
To je sloj sa definicijama objekata i njihovim vezama.
Ti izlistani objekti predstavljaju tabele.
On je najnezavisniji deo sistema. (Npr. prodavnica ne može da postoji bez: kupca, prodavca, korpe, proizvoda).

Cilj domenskog sloja

je da enkapsulira poslovnu logiku i pravila aplikacije.
On predstavlja srž sistema i treba da bude izolovan od tehničkih detalja (npr. pristup bazi podataka).
Domenski sloj sadrži entitete koji modeluju osnovne koncepte – korisnike, postove, komentare, kategorije, veze između njih i druge bitne aspekte sistema.

Korišćenje nasleđivanja

Domain/BaseEntity.cs
Obezbeđuje zajednička svojstva (kao što su identifikator, datumi kreiranja/modifikacije/brisanja, status aktivnosti/brisanja), što smanjuje dupliranje koda.
Podržava mehanizme za soft delete i praćenje vremena kreiranja/modifikacije.
Centralizuje zajednička svojstva svih entiteta, što je odlična praksa za **DRY** (Don't Repeat Yourself).

Navigaciona svojstva i kolekcije: Omogućavaju EF Core-u da kreira relacije između entiteta, olakšavajući rad sa podacima i podršku za **Lazy Loading** (ako je konfigurisan).

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.

Koristim ***virtual*** u navigacionim svojstvima kako bih omogućio lazy loading i podržao testiranje putem **mock objekata**.
Ako lazy loading nije potreban, mogu ga izostaviti i koristiti ***Include()*** za eksplicitno učitavanje povezanih entiteta **(Eager loading)**.

Trenutno koristim CQRS i već ručno kontrolišem upite, tako da virtual nije sada obavezan. **Lazy loading je podrazumevano isključen u EF Core** i mora se eksplicitno uključiti pomoću *Microsoft.EntityFrameworkCore.Proxies* paketa.

Domain/Category.cs
Jasno modeluje kategorije, pri čemu se koristi kolekcija PostCategory za definisanje više prema više relacije sa postovima.
Kolekcija je inicijalizovana kako bi se izbegli ***null reference* izuzeci.**

Domain/Role.cs
Definiše ulogu korisnika i omogućava dodelu više korisnika jednoj ulozi. Svaki korisnik ima određenu ulogu (npr. administrator, običan korisnik, autor), a uloga može imati više korisnika.

Domain/User.cs
Modeluje korisnika sa svim bitnim informacijama, uključujući lične podatke, ulogu, profilnu sliku i povezane kolekcije (postovi, komentari, lajkovi, use case-ove koji određuju prava korisnika u sistemu, lista korisnika koji prate ovog korisnika, lista korisnika koje ovaj korisnik prati).

Domain/Like.cs
Omogućava modelovanje lajkova sa podrškom i za komentare i za postove.
Korišćenje **enum**-a LikeStatus omogućava fleksibilnost u definisanju stanja:
1. Liked (1)
2. Disliked (2)
3. Null (3)

Domain/Image.cs
Centralizuje podatke o slici i povezuje ih sa postovima.

Domain/UseCaseLog.cs
Omogućava praćenje izvršenih use case-ova, beležeći ključne informacije o svakoj akciji korisnika.
Model sadrži:
1. Id zapisa jer ne nasleđuje BaseEntity,
2. Date za datum i vreme izvršenja akcije,
3. UseCaseName naziv use case-a koji je pokrenut,
4. Data za podatke koji su poslati u okviru zahteva,
5. Actor za aktera koji je izvršio akciju.
Model služi za analizu aktivnosti u sistemu, omogućavajući pregled istorije izvršenih operacija.

Domain/UserUseCase.cs

Predstavlja više prema više vezu između korisnika i use-case-ova koje mogu da izvršavaju.
Ovaj model se koristi za autorizaciju korisnika, omogućava dodelu specifičnih dozvola korisnicima i određuje koje su im akcije dozvoljene da preduzimaju u sistemu.
Polja modela:
1. IdUser identifikator korisnika kojem je dodeljen određeni use-case
2. IdUseCase identifikator use-case-a koji korisnik može da izvršava
3. User navigaciono svojstvo koje povezuje korisnika sa njegovim dodeljenim use-case-ovima

Domain/Comment.cs

Predstavlja sistem komentarisanja i omogućava korisnicima da ostavljaju komentare na postove, kao i da odgovaraju na druge komentare.
Funkcionalnosti:
1. Hijerarhijska struktura komentara (svaki komentar može imati podkomentare, omogućavajući „threaded“ komentarisanje),
2. Povezanost sa korisnicima i postovima (komentari su direktno povezani sa korisnicima i postovima),
3. Sistem lajkovanja komentara (komentari mogu dobijati lajkove putem „Likes“ kolekcije),
4. Nullable roditeljski ID (omogućava razlikovanje osnovnih i komentara i odgovora na komentare)

Domain/Follower.cs

Predstavlja sistem praćenja korisnika u blog aplikaciji, omogućavajući korisnicima da prate druge korisnike i budu obavešteni o njihovim aktivnostima.
Polja modela:
1. IdFollower – Identifikator korisnika koji prati drugog korisnika
2. IdFollowing – Identifikator korisnika koji je praćen
3. FollowedAt – Datum i vreme kada je praćenje započeto (ne nasleđuje BaseEntity.cs)
4. FollowerUser – Veza ka korisniku koji prati drugog korisnika
5. FollowingUser – Veza ka korisniku koji je praćen

Jedan-na-više relacija između korisnika – Jedan korisnik može pratiti više drugih korisnika, a svaki korisnik može biti praćen od strane više korisnika.

Prirodna jedinstvenost: Composite ključevi, koji se sastoje od ključeva entiteta koje povezuju (npr. IdFollower i IdFollowing), prirodno osiguravaju jedinstvenost veze. Na taj način se sprečava dupliranje istih veza bez potrebe za dodatnim primarnim ključem.

Ovim modelom omogućava se implementacija funkcionalnosti poput:
1. Pregleda liste korisnika koje jedan korisnik prati
2. Pregleda liste pratilaca jednog korisnika
3. Implementacije real-time obaveštenja kada neko počne da prati drugog korisnika

Domain/Notification.cs
Predstavlja sistem obaveštavanja korisnika u blog aplikaciji.
Omogućava slanje notifikacija kada dođe do značajnih interakcija između korisnika, kao što su novi postovi, komentari, lajkovi ili praćenja.
Polja modela:
1. IdUser – Identifikator korisnika koji prima notifikaciju
2. FromIdUser – Identifikator korisnika koji je generisao notifikaciju
3. Content – Tekstualni sadržaj notifikacije
4. Link – URL koji vodi korisnika do relevantnog sadržaja (npr. post, komentar)
5. IsRead – Označava da li je notifikacija pročitana
6. NotificationType enumeracija omogućava lako proširivanje sistema notifikacija i jasnu kategorizaciju notifikacija:
Post (1) – Notifikacija kada korisnik objavi novi post
Comment (2) – Notifikacija kada neko komentariše post
Like (3) – Notifikacija kada neko lajkuje post ili komentar
Follow (4) – Notifikacija kada korisnik počne da prati drugog korisnika

Funkcionalnosti i upotreba:
1. Personalizovano obaveštavanje korisnika – Svaka notifikacija je povezana sa pošiljaocem i primaocem
2. Praćenje statusa notifikacija (IsRead) – Korisnik može videti koje notifikacije su nove, a koje već pregledane.
3. Navigacija ka relevantnom sadržaju (Link) – Omogućava brzo otvaranje povezanog posta, komentara ili profila korisnika
4. Efikasno proširivanje – Novi tipovi notifikacija mogu se lako dodati proširenjem NotificationType enumeracije

Primer korišćenja:
- Kada korisnik **objavi novi post**, pratioci dobijaju notifikaciju (Post).
- Kada neko **komentariše post**, autor posta dobija notifikaciju (Comment).
- Kada neko **lajkuje post**, autor posta dobija notifikaciju (Like).
- Kada neko **zaprati drugog korisnika**, korisnik dobija notifikaciju (Follow).

Domain/PostCategory.cs
Model predstavlja više-prema-više vezu između postova i kategorija, omogućavajući da jedan post pripa u više kategorija, kao i da jedna kategorija sadrži više postova.

Ovaj model igra ključnu ulogu u organizaciji sadržaja blog aplikacije, omogućavajući fleksibilno povezivanje postova i kategorija.

Domain/Post.cs
Predstavlja objavu u aplikaciji.
Svaka objava može imati naslov, sadržaj, autora, sliku, i može biti povezana sa komentarima, lajkovima i kategorijama.
Funkcionalnost i upotreba:
1. Kreiranje i prikaz postova – Omogućava korisnicima da pišu i čitaju objave
2. Organizacija postova po kategorijama – Kroz vezu PostCategories omogućava da post pripada više kategorija
3. Interakcija korisnika – Post može primati komentare i lajkove
4. Povezanost sa autorom – Svaki post pripada tačno jednom korisniku
5. Mogućnost dodavanja slike – Post može imati sliku vezanu preko Image entiteta

Domain/AuthorRequest.cs
Model predstavlja zahtev korisnika da postane autor na platformi.
Korisnici mogu podneti zahtev sa objašnjenjem (razlogom), a administratori mogu odlučiti da ga prihvate ili odbiju.

Enumeracija *RequestStatus*
1. Pending (1) – Zahtev je na čekanju i administratori ga još nisu pregledali
2. Accepted (2) – Zahtev je odobren, korisnik postaje autor
3. Rejected (3) – Zahtev je odbijen

Primer upotrebe:
- Korisnik podnosi zahtev sa objašnjenjem → { IdUser = 5, Reason = "Želim da pišem članke.", Status = Pending }
- Administrator odobrava zahtev → { IdUser = 5, Status = Accepted }
- Korisnik sada može objavljivati postove

U projektu sam se odlučio za pristup gde nisu svi entiteti nasleđeni iz BaseEntity.
Za entitete kao što su Follower i UserUseCase primenjen je pristup composite ključeva, jer ovi entiteti primarno služe za modelovanje many-to-many veza bez potrebe za dodatnim **audit informacijama** (omogućavaju praćenje i istoriju promena u entitetima).
Ovaj pristup omogućava jasniju i jednostavniju strukturu baze podataka, smanjujući broj nepotrebnih kolona i potencijalnu kompleksnost modela.

Za entitete koji sadrže kompleksniju poslovnu logiku i gde je važno pratiti audit podatke, koristim BaseEntity.cs kako bih osigurao konzistentan pristup praćenja promena, kreiranja i eventualnog soft delete-a.
Primeri ovakvih entiteta su Post, Comment i drugi centralni entiteti aplikacije.

Jasna separacija poslovne logike:
Kroz ovaj folder se jasno odvaja logika koja je specifična za problematiku aplikacije od drugih tehničkih aspekata, što omogućava da se promene u poslovnim pravilima implementiraju bez uticaja na ostatak sistema.

EFDataAccess

Predstavlja most izmedju domenske logike (modela) i baze podataka.
Ovaj sloj je odgovoran za pristup podacima.
Koristi Entity Framework Core za komunikaciju sa bazom podataka (MS SQL Server).
Tu se nalaze **DbContext.cs** (glavna klasa za interakciju sa bazom podataka kroz EF), konfiguracije entiteta, migracije i sve ostalo vezano za perzistenciju podataka.

Izolacija pristupa podacima

Omogućava se promena implementacije pristupa podacima bez uticaja na ostatak aplikacije.
Na primer, ako se u budućnosti odluči za drugi ORM ili čak drugačiju bazu podataka, izmene se fokusiraju samo ovde.

Centralizovana konfiguracija

Svi aspekti koji se tiču povezivanja sa bazom i konfiguracije entiteta su svedeni na jedno mesto, što olakšava održavanje i debugging.

Podrška za migracije

Korišćenje EF Core migracija olakšava upravljanje verzijama baze podataka, što je posebno važno kod kontinuiranog razvoja i unapređenja.

„Code first”

Pristup pri kome se kreće 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.

EFDataAccess/EFDataAccess.csproj

XML konfiguracija koja 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.

EFDataAccess/BlogContext.cs

Klasa BlogContext.cs je **Entity Framework Core (EF Core) kontekst**, koji predstavlja most između aplikacije i baze podataka.
To je odgovorna klasa za rad sa podacima, omogućavajući **CRUD** operacije nad entitetima koji su definisani u aplikaciji.

Namena klase:
1. Definiše entitete (tj. tabele u bazi podataka)
2. Konfiguriše mapiranje entiteta pomoću **Fluent API**-ja
3. Automatizuje postavljanje podataka prilikom dodavanja ili ažuriranja
4. Filtrira podatke (npr. skriva logički obrisane zapise)
5. Postavlja konekciju sa bazom podataka

protecred override void OnModelCreating(ModelBuilder modelBuilder)
Ova metoda koristi ***ModelBuilder*** objekat za primenu konfiguracija entiteta i podešavanje pravila.

override - Znači da je već definisan metod u klasi DbContext, ali mi redefinišemo njegovo ponašanje.

Ovo je virtualna metoda gde dodajem incijalne podatke, to su bazni, početni podaci kao preduslov funkcionisanja aplikacije.
Definišem kako će entiteti biti mapirani u tabele u bazi podatka.

ApplyConfiguration pozivima koristim Fluent API konfiguracije koje sam definisao u zasebnim klasama (npr. PostConfiguration.cs, CategoryConfiguration.cs, itd.).
To omogućava da sve konfiguracije ostanu modularne i odvojene od same DbContext.cs klase.
U tim konfiguracijama postavljam detalje poput primarnih ključeva, relacija, ograničenja, imena kolona i slično.

Fluent API je alternativa ili dopuna atributima ([Required], [MaxLength], itd.) za konfiguraciju entiteta u EF Core.
Koristi se za definisanje odnosa, primarnih ključeva, ograničenja i drugih pravila mapiranja.

modelBuilder.Entity<Post>().HasQueryFilter(x => !x.IsDeleted);
Ova linija definiše globalni filter upita za entitet Post u Context-u.
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.

protecred override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)
Ova metoda konfiguriše konekciju sa SQL Server bazom (UseSqlServer()).
Ime baze je "blog" (Initial Catalog=blog)

Prvo otvaram SQL Management Studio, konektujem se, kopiram Server name u Server Explorer-u za dodavanje konekcije i izaberem 'blog' bazu podataka, pa kopiram **connection string**.

public override int SaveChanges()
Ova metoda se poziva svaki put kada se promena upisuje u bazu.
Prolazi kroz sve entitete koji su promenjeni (***ChangeTracker.Entries()***).
U mom slučaju, override-ujem ovu metodu kako bi dodao dodatnu logiku pre nego što se promene upišu u bazu.
Ako entitet nasleđuje BaseEntity, postavlja mu odgovarajuće vrednosti:
(Added)
CreatedAt = DateTime.Now
IsActive = true
IsDeleted = false
Resetuje vrednosti DeletedAt i ModifiedAt
(Modified)
ModifiedAt = DateTime.Now

Poziv osnovne implementacije: Nakon što sam izvršio ove pripremne radnje, pozivam ***base.SaveChanges()*** da bi se promene zaista sačuvale u bazi.

Da bi klasa bila tabela treba da napravim polje specifičnog tipa, a taj tip je ***DbSet<>***
Između **<>** treba da bude ime klase koja će predstavljati tabelu.
Svako DbSet svojstvo predstavlja entitet koji je definisan u domenskom sloju.

Pravljenje „Configurations" foldera u EF 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.

Metode za definisanje relacija između entiteta:
1. HasMany – Koristi se kada jedan entitet (npr. User) ima više povezanih entiteta (npr. Posts, Comments, itd.)
2. WithOne – Nakon definisanja kolekcije sa HasMany, WithOne se koristi da se navede referenca unutar povezanog entiteta koja pokazuje na “jedan” stranu relacije.
3. HasOne – Koristi se kada entitet ima jedan povezani entitet. Definiše se na strani entiteta koji poseduje strani ključ.
4. WithMany – Koristi se kada entitet ima mnogo povezanih entiteta. Definiše se na strani entiteta koji se referencira.
5. HasForeignKey – Definiše koji će strani ključ biti korišćen u tabeli povezane strane.

Metode za definisanje ponašanja prilikom brisanja (**OnDelete**):
1. DeleteBehavior.NoAction – Ova opcija znači da se pri brisanju roditeljskog entiteta neće izvršiti nikakva automatska akcija nad povezanim entitetima.
2. DeleteBehavior.Cascade – Automtaski briše sve povezane entitete.
3. DeleteBehavior.SetNull – Postavlja vrednost stranog ključa na *null* u povezanim entitetima, ali to zahteva da taj strani ključ bude *nullable.*
4. DeleteBehavior.Restrict – Slično kao NoAction, sprečava brisanje ako postoje povezani entiteti.

Metode za definisanje svojstava kolona i njihovih ograničenja u bazi podataka:
1. HasIndex – Koristi se da definiše indeks nad jednom ili više kolona, što može poboljšati performanse upita.
2. IsUnique – Postavlja jedinstvenost indeksa, što znači da vrednost u toj koloni mora biti jedinstvena u celoj tabeli.
3. IsRequired – Definiše da je kolona obavezna, odnosno da ne može imati *null* vrednost.
4. HasMaxLength – Postavlja maksimalnu dozvoljenu dužinu string vrednosti.
5. HasKey – Koristi se za definisanje primarnog ključa entiteta. Definiše jednu ili više kolona koje čine primarni ključ. Koristi se za kompozitni ključ.

Šta radi ***IEntityTypeConfiguration<T>?***
Ovaj interfejs zahteva implmenetaciju metode ***Configure***, u kojoj se prosleđuje *EntityTypeBuilder<T>.*
Unutar te metode se definišu pravila za mapiranje entiteta (svojsvta, indeksi, relacije, ograničenja, itd.)
Ovo omogućava da se konfiguracija entiteta održi odvojeno od samog DbContext-a, što poboljšava čitljivost i održavanje koda.

Kroz ove konfiguracije, Entity Framework Core tačno zna kako da mapira domenske klase na odgovarajuće tabele u bazi, kako da osigura integritet podataka i kako da optimizuje upite.

Configurations

EFDataAcces/Configurations/FollowerConfiguration.cs

HasKey:

Postavlja složeni primarni ključ sastavljen od IdFollower i IdFollowing.
Ovaj ključ osigurava da ne može postojati više redova sa istim kombinacijama pratilac-praćeni.

Pratilac:

HasOne(x => x.FollowerUser) → Označava da svaki zapis u Follower tabeli referencira jednog korisnika kao pratioca.
WithMany(x => x.Followers) → Korisnik može imati više zapisa u listi svojih pratilaca.
HasForeignKey(x => x.IdFollower) → IdFollower kolona je strani ključ koji upućuje na korisnika koji je pratilac.
OnDelete(DeleteBehavior.Restrict) → Sprečava brisanje korisnika ako postoje zapisi u Follower tabeli koji ga referenciraju.

Praćeni korisnik:

HasOne(x => x.FollowingUser) → Označava da svaki zapis u Follower tabeli referencira jednog korisnika kao praćenog.
WithMany(x => x.Followings) → Korisnik može imati više zapisa u listi korisnika koje prati.
HasForeignKey(x => x.IdFollowing) → IdFollowing je strani ključ koji upućuje na korisnika koji je praćen.
OnDelete(DeleteBehavior.Restrict) → Sprečava brisanje korisnika koji je praćen ako postoji zapis koji ga referencira.

EFDataAcces/Configurations/CommentConfiguration.cs

Svaki komentar može imati više podkomentara (odgovora).
HasMany(x => x.ChildrenComments) → Jedan komentar može imati više odgovora (dece).
WithOne(x => x.ParentComment) → Svaki odgovor ima jedan roditeljski komentar.
HasForeignKey(x => x.IdParent) → IdParent je strani ključ koji pokazuje na roditeljski komentar.
OnDelete(DeleteBehavior.NoAction) → Ako se roditeljski komentar obriše, podkomentari neće biti automatski obrisani.

HasMany(x => x.Likes) → Svaki komentar može imati više lajkova.
WithOne(y => y.Comment) → Svaki lajk pripada tačno jednom komentaru.
HasForeignKey(x => x.IdComment) → IdComment u tabeli Like je strani ključ koji referencira Comment.

EFDataAcces/Configurations/LikeConfiguration.cs
IdUser je obavezan → Svaki lajk mora pripadati korisniku.
IdPost je obavezan → Lajk mora biti vezan za post.
IdComment nije obavezan **(IsRequired(false))** → Lajk može biti vezan za post ili komentar, ali nije obavezno.
Status → Čuva se kao **tinyint**.

EFDataAcces/Configurations/AuthorRequestConfiguration.cs
IdUser**,** Reason su obavezna polja.
Status se čuva kao tinyint.

EFDataAcces/Configurations/CategoryConfiguration.cs
Name je obavezno polje sa maksimalno 70 karaktera.
Definiše jedan-na-više (1:N) vezu između Category i Post (CategoryPosts).
Strani ključ IdCategory u Post entitetu ima pravilo Restrict.
Ako se pokuša brisanje kategorije koja ima postove, dobiće se greška (ne dozvoljava brisanje).

EFDataAcces/Configurations/ImageConfiguration.cs
ImagePath je obavezno polje koje sadrži putanju do slike.
Definiše jedan-na-više (1:N) vezu između Image i Post.
Strani ključ idImage u Post entitetu ima pravilo Restrict.
Ako se pokuša brisanje slike koju post koristi, brisanje neće biti dozvoljeno.

EFDataAcces/Configurations/NotificationConfiguration.cs
Content i Link su obavezni sa maksimalno 250 karaktera.
Type je obavezno polje (tipa *int)* koje predstavlja tip notifikacije.
IsRead je obavezno i podrazumevano *false*.
UserReceiver (korisnik koji prima obaveštenja): ako se korisnik obriše, brišu se i njegova obaveštenja (Cascade).
FromUser (korisnik koji šalje obaveštenje): ako se korisnik obriše, njegova obaveštenja ostaju (**NoAction**).

EFDataAcces/Configurations/UserConfiguration.cs
FirstName, LastName, Username, Password, Email su obavezni (IsRequired()).
Sprečava se dupliranje emailova i korisničkih imena u bazi (**IsUnique()**).
User → **Posts** (korisnik može imati više postova).
User → **Comments** (korisnik može ostavljati komentare).
User → **Likes** (korisnik može lajkovati postove).
User → **Role** (korisnik pripada nekoj ulozi).
User → **UserUseCases** (korisnik može imati više use-case-ova).

EFDataAcces/Configurations/PostCategoryConfiguration.cs
PostCategory je tabela više-prema-više između Post i Category.
Ova tabela ne mora imati dodatni Id, jer su IdPost i IdCategory zajedno primarni ključ.

EFDataAcces/Configurations/PostConfiguration.cs
Title je obavezan (IsRequired()) i ima maksimalno 70 karaktera.
Content je obavezan (IsRequired()), ali nema ograničenje dužine.
Post može imati više kategorija(PostCategory), komentara(Comments) i lajkova(Likes).

Kada se obriše post, automatski će se obrisati i svi zapisi iz vezne tabele PostCategories koji su vezani za taj post. Takođe, automatski se brišu komentari i lajkovi tog posta.

Migracije su mehanizam za upravljanje bazom podataka koji omogućava praćenje promena u njenoj strukturi tokom razvoja aplikacije.
Kada je sve spremno, primenjujem migracije ***„dotnet ef migrations add initial migration„***
Analiziram migraciju, ako je sve dobro onda komanda: ***“dotnet ef database update”***

Application sloj

Orkestrira operacije između Domain sloja i ostalih tehničkih slojeva.
Sadrži servisne interfejse, logiku poslovnih operacija, komandno-upitne (CQRS) modele, i slično.
On definiše interfejse koje će ostali slojevi (kao Implementation) implementirati.

Definiše šta je naša aplikacija sposobna da uradi, ali ne i način.

Application sloj zavisi od Domain projekta i može koristiti njegove entitete i druge resurse.
Application sloj ne treba da ima referencu ka EFDataAccess-u.

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 CQRS (Command Query Responsibility Segregation).

I komanda i query su use-case-ovi.
Cilj je da imam zajednički **nadtip** za sve komande i zajednički nadtip za sve query-e.

Kad organizujem sistem, treba da logujem svaki query koji je iko ikada pokušao da uradi, ali ne tražim svaku liniju u kontroleru, već želim centralizovano rešenje.

Application/IUseCase.cs
Definiše osnovne karakteristike bilo kog use-case-a.

Svaki use-case ima jedinstveni Id i Name.
Ovo omogućava da svaki komandni ili upitni handler zna koji use-case je pokrenut i može se lakše logovati.
Ovaj interfejs je baza koju ostali interfejsi koriste, pa je svaki use-case (bilo da je komanda ili upit) obavezno mora implementirati.

ICommand<TRequest> Je interfejs za komande, akcije koje menjaju podatke, stanje sistema (npr. kreiranje, ažuriranje, brisanje).
Komande obično ne vraćaju nikakav rezultat osim potvrde da su izvršene, zato ***void***.
Metoda ***Execute*** prima **request** kao argument koji je potreban za izvršenje operacije.
Korišćenjem **generičkog tipa *TRequest*** omogućava se fleksibilnost, jer svaka konkretna komanda može zahtevati drugačiji skup podataka (DTO).

IQuery<TResponse, TSearch> Je interfejs za query-e, akcije koje se koriste za čitanje podataka i koje vraćaju rezultat.
Upit može da ima ulazne parametre, a izlaz svakako ima.
Ima 2 generička tipa (1 objekat za potencijalnu pretragu).
Prvi generički tip ukazuje šta je rezultat pretrage, a drugi koji je ulazni parametar.
TResponse: Ovo je tip podataka koji se vraća kao rezultat upita. Na primer, može biti lista korisnika, detalji o nekom objektu ili zbirni podaci.
TSearch: Ovaj generički tip predstavlja ulazne parametre za pretragu. Na primer, za upit koji traži korisnike sa određenim imenom, TSearch može biti objekat sa parametrima kao što su "ime" i sl.
Execute: Metoda Execute prima TSearch kao argument (koji sadrži filtere ili parametre) i vraća rezultat tipa TResponse.

IAsyncCommand<TRequest> Je interfejs koji koristi asinhronu metodu koja vraća ***Task***, što znači da se može čekati **(*await*)** i ne blokira **glavni thread**.

Application/IUseCaseLogger.cs
Ovaj interfejs definiše logovanje svih izvršenih use-case-ova.
Omogućava praćenje svih korisničkih akcija u sistemu.

useCase → Koji use-case je pokrenut?
actor → Koji korisnik ga je pokrenuo?
data → Koji su podaci prosleđeni?

Application/UseCaseExecutor.cs
Ovo je klasa koja pomaže u upravljanju komandama i upitima tako da svaki korisnik može raditi samo ono što mu je dozvoljeno.

Ovo je odličan primer primene principa separation of concerns, jer eliminiše potrebu da svaki use case pojedinačno implementira logiku autorizacije i logovanja.

Ima 3 glavne metode:

1*. ExecuteCommand<TRequest>(ICommand<TRequest> command, TRequest data)*

command – Komanda koju želim da izvršim (npr. dodavanje korisnika).
data – Podaci potrebni za komandu (npr. ime, email, lozinka).
Prvo beležim ko je pokušao da izvrši komandu i sa kojim podacima.
Zatim proveravam da li korisnik ima dozvolu.
Ako korisnik ima dozvolu, izvršavam komandu. Ako nema dozvolu, bacam izuzetak.

2. *ExecuteQuery<TResponse, TSearch>(IQuery<TResponse, TSearch> query, TSearch data)*

query – Predstavlja upit, odnosno objekat koji se koristi za pretragu ili dobijanje podatak iz baze.
data – Podaci koje prosleđujem upitu. To su parametri pretrage koji određuju šta želim da dobijem iz baze.
Beleži ko traži podatke I šta tačno traži.
Proverava da li korisnik ima pravo da izvrši upt.
Ako ima dozvolu, poziva **query.Execute(data)** i vraća rezultat.

3. ***ExecuteCommandAsync<TRequest>(IAsyncCommand<TRequest> command, TRequest data)***

Zašto je ovo korisno?
Centralizovana kontrola pristupa – Ne moram u svakoj metodi ručno proveravati dozvole.
Bolja organizacija koda – UseCaseExecutor odvaja poslovnu logiku od kontrolera.

Instanca klase UseCaseExecutor implemenitra logiku koja orkestrira izvršenje use-case-ova (komandi i upita), ona:
1. Preuzima zahteve (bilo da je reč o komandi ili query-ju) iz API sloja.
2. Izvršava logovanje tako što beleži informacije o use case-u, aktoru koji ga izvršava i podacima koji se prosleđuju.
3. Proverava autorizaciju – da li aktor ima dozvolu da izvrši taj konkretan use-case.
4. Delegira izvršenje stvarnom use-case-u (pozivajući metodu Execute na tom objektu).

Dakle, taj "objekat" (instanca UseCaseExecutor-a) se ponaša kao **middleware** koji se nalazi između API-ja i poslovne logike, osiguravajući da se pre izvršenja zahteva obave neophodni koraci kao što su logovanje i autorizacija.

Ova arhitektura se često koristi u čistoj arhitekturi (**Clean Architecture**) i CQRS (Command Query Responsibility Segregation) pristupima, gde se jasno razdvajaju operacije koje menjaju stanje sistema od onih koje samo čitaju podatke, a dodatno se uvodi mehanizam za logovanje i autorizaciju kao **cross-cutting concern**.

Application/IApplicationActor.cs
Ovaj interfejs definiše korinsika sistema i njegove use-case-ove u aplikaciji.

Id (int) – Predstavlja jedinstveni identifikator korisnika.
Ako je korisnik prijavljen, ovo će biti njegov ID iz baze podataka.
Ako je korisnik anoniman, može biti 0 (**AnonymousActor**).

Identity (string) – Predstavlja identitet korisnika (npr. korisničko ime, email ili opis).
U JWTActor, ovo je email ili korisničko ime.
U AnonymousActor, ovo je "Unauthorized user".

AllowedUseCases (IEnumerable<int>) – Lista ID-eva dozvoljenih use case-ova (akcija koje korisnik može izvesti).
Ako je korisnik prijavljen, ova lista dolazi iz njegove uloge i dozvola.
Ako je korisnik anoniman, ima samo ograničene dozvole (AnonymousActor).

Prednosti ovog pristupa**:**
Omogućava rad sa prijavljenim i neprijavljenim korisnicima kroz isti interfejs.
Omogućava lakšu promenu sistema autentifikacije (JWT, OAuth, itd.) bez menjanja komandnog sistema.
Omogućava jednostavno upravljanje dozvolama korisnika.

Data Transfer Objects

Pravljenje DTO-ova (**Data Transfer Objects**) u Application sloju ima nekoliko koristi i ciljeva.

1. Prenos podataka između 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.
4. Jasnoća i konzistentnost: DTO-ovi omogućavaju da jasno definišeš šta se prenosi između slojeva. Time postaje jasna struktura podataka koja se očekuje na ulazu i izlazu, što olakšava održavanje koda.

5. Fleksibilnost u razvoju: Kada se DTO-ovi koriste, lako je dodavati ili uklanjati polja u komunikaciji između slojeva, bez potrebe za direktnom modifikacijom domen modela. Ovo omogućava evoluciju API-ja bez potrebe da se utiče na poslovnu logiku.

DTO-ovi koji se koriste u sistemu:
Application/DataTransfer/Auth/LoginUserDto.cs
Application/DataTransfer/Auth/OAuthUserDto.cs
Application/DataTransfer/Auth/RegisterUserDto.cs

Application/DataTransfer/AuthorRequests/GetAuthorRequestsDto.cs
Application/DataTransfer/AuthorRequests/UpsertAuthorRequestsDto.cs

Application/DataTransfer/Categories/GetCategoriesDto.cs
Application/DataTransfer/Categories/UpsertCategoryDto.cs

Application/DataTransfer/Comments/GetCommentDto.cs
Application/DataTransfer/Comments/GetCommentLikesDto.cs
Application/DataTransfer/Comments/GetCommentsDto.cs
Application/DataTransfer/Comments/GetUserCommentsDto.cs
Application/DataTransfer/Comments/UpsertCommentDto.cs

Application/DataTransfer/Emails/SendEmailDto.cs

Application/DataTransfer/Followers/GetFollowsDto.cs
Application/DataTransfer/Followers/InsertFollowDto.cs

Application/DataTransfer/Images/ImageProxyDto.cs
Application/DataTransfer/Images/UploadImageDto.cs

Application/DataTransfer/Likes/LikeDto.cs

Application/DataTransfer/Notifications/GetNotificationDto.cs
Application/DataTransfer/Notifications/InsertNotificationDto.cs

Application/DataTransfer/Posts/GetPostCategoriesDto.cs
Application/DataTransfer/Posts/GetPostDetailsDto.cs
Application/DataTransfer/Posts/GetPostInCategoryDto.cs

Application/DataTransfer/Posts/GetPostLikesDto.cs
Application/DataTransfer/Posts/GetPostsDto.cs
Application/DataTransfer/Posts/GetUserPostsDto.cs
Application/DataTransfer/Posts/UpsertPostDto.cs

Application/DataTransfer/UseCases/GetUseCaseLogDto.cs
Application/DataTransfer/UseCases/GetUserUseCaseDto.cs

Application/DataTransfer/Users/GetUserDto.cs
Application/DataTransfer/Users/GetUsersDto.cs
Application/DataTransfer/Users/UpsertUserDto.cs

Exceptions

U .NET svetu, **Exception.cs** je osnovna (base) klasa za sve greške i izuzetke.
Kada se dogodi greška ili nevažeća operacija, uobičajeno je baciti (**throw**) instancu klase koja nasleđuje Exception.
Ovo omogućava da se greške propagiraju kroz aplikaciju, da se loguju, hendluju ili proslede dalje (npr. korisničkom interfejsu) na centralizovan način.

Poruka greške: Svaki Exception nosi poruku koja opisuje grešku.

Stack Trace: Omogućava praćenje toka izvršavanja koji je doveo do greške.

Nasleđivanje: Koristim sopstvene (custom) izuzetke tako što nasleđujem Exception klasu.

Application sloj baca prilagođene izuzetke kada se detektuje specifična greška (npr. entitet ne postoji, neovlašćeni pristup).

API sloj ih hvata pomoću GlobalExceptionHandler.cs i prevodi u odgovarajući HTTP statusni kod.

Ovaj pristup omogućava centralizovano rukovanje greškama, što je dobro za održavanje.

Application/Exceptions/AlreadyAddedException.cs
Baca se kada korisnik pokuša da doda nešto što je već dodato (npr. već poslat zahtev za odobrenje).

Application/Exceptions/AlreadyDeletedException.cs
Baca se kada korisnik pokuša da obriše entitet koji je već obrisan.

Application/Exceptions/ConflictException.cs
Baca se kada postoji sukob podataka, npr. pokušaj kreiranja entiteta koji već postoji.

Application/Exceptions/EntityNotFoundException.cs
Baca se kada entitet sa datim ID-jem ne postoji u sistemu.

Application/Exceptions/UnauthorizedUseCaseException.cs
Pokriva generalnu autorizaciju za izvršavanje use-case-a.
Proverava da li korisnik uopšte ima dozvolu da koristi određen use-case.

Application/Exceptions/UnauthorizedUserAccessException.cs
Služi za specifične situacije unutar samog use-case-a gde korisnik možda ima pravo da izvrši komandu, ali nema pravo da menja određeni entitet (npr. može da briše komentare, ali ne i komentare drugih korisnika).

Commands

U application sloju, komandni interfejsi predstavljaju deo CQRS (Command Query Responsibility Segregation) arhitekture.
U ovom kontekstu, komande (Commands) su odgovorne za menjanje stanja sistema (kreiranje, ažuriranje i brisanje podataka).

Svaka komanda ima jasno definisan cilj:
Create - kreira novi entitet.
Update - ažurira postojeći entitet.
Delete - briše entitet (najčešće prima samo id).

Koriste DTO-ove umesto modela:
DTO (Data Transfer Object) sprečava direktno manipulisanje entitetima baze i olakšava validaciju podataka.

IAsyncCommand<T> koristi se kada je neophodno izvršiti operacije koje mogu trajati duže (npr. zapis u bazu, komunikacija sa spoljnim servisima).

Application sloj ne sadrži direktne pristupe bazi, već definiše komande koje će handleri izvršiti.

Korišćenjem CQRS-a i ICommand<T>, svaka komanda može imati svoj **Handler**, što omogućava lakše testiranje i razdvajanje poslovne logike.

Razdvajanje odgovornosti (Separation of Concerns): Svaka komanda obavlja jednu specifičnu radnju, što poboljšava čitljivost i održivost koda.

Application/Commands/AuthorRequest/ICreateAuthorRequestCommand.cs
Implementira interfejs ICommand<UpsertAuthorRequestDto>.

Koristi UpsertAuthorRequestDto, što znači da prima DTO koji sadrži podatke potrebne za kreiranje zahteva autora.

Application/Commands/AuthorRequest/IUpdateAuthorRequestCommand.cs
Takođe koristi UpsertAuthorRequestDto, ali služi za ažuriranje postojećeg zahteva.

Application/Commands/Category/ICreateCategoryCommand.cs
Implementira ICommand<UpsertCategoryDto>.
Koristi UpsertCategoryDto za kreiranje nove kategorije.

Application/Commands/Category/IDeleteCategoryCommand.cs
Implementira ICommand<int>, što znači da prima samo id kategorije koju treba obrisati.

Application/Commands/Category/IUpdateCategoryCommand.cs
Koristi isti DTO kao create komanda, ali za ažuriranje postojećih podataka o kategoriji.

Application/Commands/Comment/ICreateCommentCommand.cs
Implementira IAsyncCommand<UpsertCommentDto>, što znači da se izvršava asinhrono.
Koristi UpsertCommentDto za kreiranje komentara.

Application/Commands/Comment/IDeleteCommentCommand.cs
Implementira ICommand<int> što znači da zahteva samo id komentara za brisanje.

Application/Commands/Comment/IDeletePersonalCommentCommand.cs
Takođe implementira ICommand<int>, ali je specijalizovana za brisanje komentara koje je korisnik sam napisao.

Application/Commands/Comment/IUpdatePersonalCommentCommand.cs
Implementira ICommand<UpsertCommentDto>, ali je namenjena za ažuriranje komentara koje je korisnik napisao.

Application/Commands/Email/IEmailSender.cs
Interfejs IEmailSender.cs omogućava slanje e-mailova koristeći DTO (SendEmailDto). Ovaj interfejs bi mogao biti implementiran pomoću servisa kao što su SMTP, SendGrid, Mailgun itd.

Application/Commands/Follow/IFollowCommand.cs
Omogućava dodavanje novog praćenja između korisnika.
Nasleđuje IAsyncCommand<InsertFollowDto>, što znači da se ova komanda izvršava asinhrono.

Application/Commands/Follow/IUnfollowCommand.cs
Omogućava prekid praćenja i koristi *int* kao parametar, što znači da se koristi ID korisnika kojeg treba otpratiti.

Application/Commands/Like/ILikePostCommand.cs
Omogućava korisnicima da lajkuju objave.
Nasleđuje IAsyncCommand<LikeDto> i asinhrono se izvršava.

Application/Commands/Like/ILikeCommentCommand.cs
Omogućava korisnicima da lajkuju komentare.
Nasleđuje IAsyncCommand<LikeDto> i asinhrono se izvršava.

Application/Commands/Like/IUnlikePostCommand.cs
Omogućava korisnicima da uklone, obrišu prethodni lajk sa posta.
Nasleđuje IAsyncCommand<LikeDto> i asinhrono se izvršava.

Application/Commands/Like/IUnlikeCommentCommand.cs
Omogućava korisnicima da uklone, obrišu prethodni lajk sa komentara.
Nasleđuje IAsyncCommand<LikeDto> i asinhrono se izvršava.

Application/Commands/Notification/ICreateNotificationCommand.cs
Koristi se za kreiranje novih notifikacija u sistemu.
Implementacija bi mogla slati obaveštenja putem **WebSockets-a (SignalR)**, e-maila, ili push notifikacija.
Koristi InsertNotificationDto za prenos podatak o notifikaciji.

Application/Commands/Notification/IMarkAllNotificationsAsReadCommand.cs
Omogućava korisniku da označi sve notifikacije kao pročitane.
Koristi *int* kao parametar, što je ID korisnika koji obeležava notifikacije kao pročitane.

Application/Commands/Post/ICreatePostCommand.cs
Omogućava kreiranje nove objave.
Koristi UpsertPostDto.
Nasleđuje IAsyncCommand<UpsertPostDto> što znači da se izvodi asinhrono.

Application/Commands/Post/IUpdatePostCommand.cs
Omogućava administratorima da ažurirraju bilo koju objavu.

Application/Commands/Post/IUpdatePersonalPostCommand.cs
Omogućava korisniku da ažurira svoju objavu.
Koristi UpsertPostDto, što znači da će se podaci objave prosleđivati putem DTO-a.

Nasleđuje ICommand<UpsertPostDto>, što ukazuje da je operacija sinhrona.
Implementacija treba da proveri da li korisnik ima prava na uređivanje objave.

Application/Commands/Post/IDeletePostCommand.cs
Omogućava brisanje bilo koje objave, od strane administratora.
Nasleđuje ICommand<int> i prima ID objave koju treba obrisati.

Application/Commands/Post/IDeletePersonalPostCommand.cs
Omogućava korisniku da obriše svoju sopstvenu objavu.
Nasleđuje ICommand<int>, što znači da se poziva sa ID-jem objave i izvršava sinhrono.

Application/Commands/User/IRegisterUserCommand.cs
Omogućava registraciju novih korisnika.
Koristi RegisterUserDto.
Nasleđuje ICommand<RegisterUserDto>, što znači da je operacija sinhrona.

Application/Commands/User/IDeleteUserCommand.cs
Omogućava brisanje korisnika iz sistema pomoću ID-ja.
Nasleđuje ICommand<int>, što znači da je operacija sinhrona.

Application/Commands/User/IUpdateUserCommand.cs
Omogućava izmenu korisničkih podataka (ime, e-mail, lozinka, uloge...).
Koristi UpsertUserDto za prenos podataka.
Nasleđuje IAsyncCommand<UpsertUserDto>, što ukazuje da je operacija asinhrona.

Repository

Repository Pattern je dizajn obrazac koji služi kao sloj između aplikacije i baze podataka. Umesto da direktno komuniciram sa ORM-om (npr. Entity Framework-om), koristim **Repository sloj** koji **enkapsulira** logiku pristupa podacima.

☞ **Cilj ovog paterna je:**
Razdvajanje poslovne logike od logike pristupa podacima.
Lakše testiranje (mockujem repozitorijum umesto da radimo sa stvarnom bazom).
Manja zavisnost od ORM-a (ako jednog dana promenim ORM, neću menjati kod u celom projektu).

Umesto da LikeService direktno koristi BlogContext, napraviću poseban repozitorijum LikeRepository.cs koji će sadržati metode za rad sa bazom.

Application/Repositories/ILikeRepository.cs
Interfejs ILikeRepository.cs definiše ugovor koji konkretna implementacija repozitorijuma

(LikeRepository.cs) mora da ispuni.
Cilj interfejsa je omogućiti labavo povezivanje (**loose coupling**) između servisa (LikeService.cs) i sloja za pristup podacima (LikeRepository.cs).
To znači da LikeService.cs ne zavisi direktno od implementacije baze podataka, već samo od ugovora (ILikeRepository.cs), što olakšava testiranje i kasnije promene u implementaciji repozitorijuma.

Task<Like> GetLike(int idUser, int idPost, int? idComment)
Ova metoda asinhrono pretražuje bazu podataka i vraća Like entitet koji odgovara zadatom korisniku, postu i komentaru.

Task AddLike(Like like)
Asinhrono dodaje novi Like entitet u bazu podataka.
Ne izvršava odmah SaveChanges, što omogućava **batched operacije** (više dodavanja pre nego što se sačuva u bazu).

Task SaveChangesAsync()
Asinhrono čuva promene u bazi podataka.
Odvaja operaciju dodavanja od njenog trajnog zapisivanja, što omogućava bolju kontrolu nad transakcijama.

Services

Interfejsi INotificationService.cs i INotificationHubService.cs definišu dva odvojena aspekta obrade notifikacija u aplikaciji.

Application/Services/InotificationService.cs

Ovo je servisni sloj koji se bavi kreiranjem notifikacija i njihovim čuvanjem u bazi podataka.
Ima jednu metodu *CreateNotification(InsertNotificationDto dto),* koja prima DTO objekat sa podacima o notifikaciji.

Implementira ga NotificationService.cs, koji:

- Generiše odgovarajući link za notifikaciju.
- Kreira instancu Notification i dodaje je u bazu podataka.
- Poziva INotificationHubService.cs kako bi poslao notifikaciju u realnom vremenu korisniku.

Application/Services/INotificationHubService.cs

Ovo je servisni sloj koji omogućava slanje notifikacija korisnicima putem SignalR-a.
Ima metodu *SendNotificationToUser(int idUser, object notification),* koja omogućava slanje notifikacije određenom korisniku.

Implementira ga SignalRNotificationHub.cs, koji koristi *IHubContext<NotificationHub>* za slanje poruka korisnicima putem WebSocket konekcije.

Decoupling (razdvajanje): Korišćenje posebnog servisa za hub komunikaciju pomaže u odvojenju poslovne logike (kreiranje notifikacija) od transportnog mehanizma (slanje notifikacija), što olakšava održavanje i eventualno testiranje.

INotificationService.cs je fokusiran na logiku kreiranja i čuvanja notifikacija, dok INotificationHubService.cs upravlja slanjem tih notifikacija ili poruka u realnom vremenu.

Application/Services/IImageService.cs
Definiše interfejs IImageService.cs, koji opisuje metode koje implementacija za rad sa slikama treba da pruži.
Ovaj interfejs se koristi kao apstrakcija u aplikaciji, što omogućava lako menjanje implementacije servisa bez modifikacije kontrolera ili drugih delova koda koji ga koriste.

Interfejs sadrži 2 metode:
1. *GetImage**(string folderName, string imageName) : byte[]***
Ova metoda prihvata dva parametra:
folderName – naziv foldera u kojem se slika nalazi (npr. "Images", "UserImages")
imageName – naziv slike (npr. "profile.jpg")

Vraća niz bajtova (byte[]), što predstavlja sadržaj slike.

Ako slika ne postoji, metoda može vratiti null (kako je definisano u implementaciji ImageService.cs).

2. *GetMimeType**(string filePath) : string***
Ova metoda vraća **MIME** tip fajla na osnovu njegove ekstenzije (npr. .jpg, .png).

Ovo omogućava da se pravilno postavi **Content-Type zaglavlje** kada se slika vraća preko API-ja.

Settings

Application/Settings/EmailSettings.cs
Ova klasa služi za čuvanje **SMTP** podešavanja koja su potrebna za slanje emailova putem SMTP servera.
Klasa koristi **.env promenljive** kako bi preuzela vrednosti, čime se izbegava hardkodovanje osetljivih podataka u kodu.
Vrednosti se preuzimaju iz okruženja pomoću *Environment.GetEnvironmentVariable().*
Ako promenljiva ne postoji, baca se izuzetak ArgumentNullException da bi se osiguralo da aplikacija ne nastavi bez potrebnih parametara.

Searches

Searches (ili Search DTO-ovi) su objekti koji se koriste za prenos kriterijuma pretrage i filtriranja podataka.

Koriste se za pretragu – kada korisnik unese ključne reči, npr. Username, Title, Email, Content itd.
Koriste se za filtriranje – kada korisnik odabere filtere, npr. CategoryIds, DateFrom, DateTo, Type, SortOrder itd.

Application/Searches/PagedSearch.cs
Ova klasa je apstraktna klasa koja služi kao bazna klasa za sve pretrage (search DTO-ove) koje uključuju paginaciju.
Ona definiše osnovne parametre za paginaciju.
Umesto da svaki Search DTO implementira svoje paginacione parametre, PagedSearch.cs omogućava da se ti parametri definišu jednom, a zatim nasleđuju u svim specifičnim pretragama.
Na primer, klase kao što su AuthorRequestSearch.cs ili CommentSearch.cs nasleđuju PagedSearch i automatski imaju PerPage i Page parametre.

- **DRY princip (Don't Repeat Yourself):** Smanjujem dupliranje koda, jer ne moram svaki put da definišem ove osnovne parametre.
- **Konzistentnost:** Svi upiti koji koriste paginaciju imaju isti početni podrazumevani broj stavki po stranici i inicijalnu stranicu.

Application/Searches/AuthorRequestSearch.cs
Sadrži samo osnovne parametre za paginaciju iz PagedSearch.cs

Application/Searches/CategorySearch.cs
Omogućava filtriranje/pretragu kategorija:
1. Po ID-u (Id)
2. Po nazivu (Name)
3. Po opciji da se prikažu sve kategorije (GetAll)

Application/Searches/CommentSearch.cs
Sadrži samo osnovne parametre za paginaciju iz PagedSearch.cs

Application/Searches/FollowSearch.cs
Sadrži ID korisnika za kojeg se pretražuju podaci o praćenju.
Predstavlja korisnika za kojeg želim da pronađem ili pratioce (followers) ili korisnike koje on prati (following).

Application/Searches/NotificationsSearch.cs
Omogućava filtriranje i pretragu notifikacija po:
1. ID-u korisnika (IdUser) – Notifikacije za određenog korisnika
2. Tipu notifikacije (Type):
1 => Post
2 => Comment
3 => Like
4 => Follow

Application/Searches/UseCaseLogSearch.cs
Omogućava pretragu i filtriranje logova korišćenja (za audit log):
1. Po glumcu (*Actor* = ko je izvršio radnju)
2. Po nazivu Use-Case-a (*UseCaseName*)
3. Po datumu (*DateFrom*, *DateTo*)
4. Po redu sortiranja (*SortOrder* – podrazumevano "desc")

Application/Searches/PostSearch.cs
Omogućava pretragu i filtriranje postova na osnovu:
1. Naslova (Title)
2. Sadržaja (Content)
3. Datuma (DateFrom, DateTo)
4. Kategorija (CategoryIds)
5. Sortiranja (SortOrder)

Application/Searches/UserSearch.cs
Omogućava pretragu korisnika na osnovu:
1. Korisničkog imena (Username)
2. Email-a (Email)
3. Da li su autori (OnlyAuthors)

Queries

U Application sloju su upiti koji omogućavaju dohvatanje podataka iz baze bez menjanja stanja sistema.
Svaki upit nasleđuje IQuery<TResult, TSearch> interfejs koji definiše:
1. TResult - tip podataka koji se vraća (npr. PagedResponse<GetCommentsDto>).
2. TSearch - parametri pretrage (npr. CommentSearch).

Ova struktura omogućava da se jasno odvoji logika pretrage i filtriranja od načina na koji se podaci vraćaju korisniku.

Application/Queries/PagedResponse.cs
Ovo je generička klasa koja služi za vraćanje podataka u paginiranom formatu.
Ova klasa omogućava da API vrati podatke sa dodatnim informacijama o paginaciji, što olakšava klijentskoj aplikaciji da zna koliko ukupno ima podataka i kako da ih prikazuje.

Tip T predstavlja podatke koji će biti vraćeni na jednoj stranici.
Ograničenje **where T : class** znači da T mora biti **referentni tip (klasa)**, što sprečava upotrebu **primitivnih tipova (int, bool, itd.)**.

public int PageCount: Ovo je izračunato svojstvo koje pokazuje ukupan broj stranica.
Formula koristi **Math.Ceiling()** da bi se zaokružilo na sledeći ceo broj, što je važno kada broj zapisa nije deljiv sa ItemsPerPage.
Na primer, ako imamo 101 zapis sa 10 elemenata po stranici, PageCount će biti 11, jer će poslednja stranica imati samo 1 element.

Application/Queries/CategoryPostsResponse.cs
Ovu klasu sam koristio umesto PagedResponse.cs zato što sam želeo da proširim standardni PagedResponse.cs sa dodatnim informacijama o samoj kategoriji.
Kada dohvatam postove iz određene kategorije, pored paginiranih postova potrebne su mi i informacije o kategoriji kojoj pripadaju.
Zato sam u ovoj klasi dodao CategoryId I CategoryName.
Ovo omogućava da API vrati podatke o samoj kategoriji zajedno sa njenim postovima.

Svaki interfejs u Query-ima nasleđuje generički interfejs IQuery <TResponse, TSearch>, što znači da svaki upit ima:
TResponse – tip podatka koji vraća upit.
TSearch – tip podatka koji služi kao parametar pretrage.

Application/Queries/AuthorRequest/IGetAuthorRequest.cs
Ovaj interfejs definiše upit koji vraća paginirane zahteve autora (PagedResponse<GetAuthorRequestsDto>).
Koristi AuthorRequestSearch.cs za filtriranje/pretragu zahteva.

Application/Queries/Category/IGetCategoriesQuery.cs
Vraća paginiranu listu kategorija (PagedResponse<GetCategoriesDto>).
Koristi CategorySearch.cs za pretragu kategorija.

Application/Queries/Category/IGetCategoryQuery.cs

Vraća postove u jednoj kategoriji (CategoryPostsResponse.cs).
Koristi CategorySearch.cs za pretragu postova u okviru određene kategorije.

Application/Queries/Comment/IGetCommentQuery.cs
Dohvata pojedinačni komentar (GetCommentDto).
Koristi int kao ID komentara koji se traži.

Application/Queries/Comment/IGetCommentsQuery.cs
Vraća paginiranu listu komentara (PagedResponse<GetCommentsDto>).
Koristi CommentSearch.cs za filtriranje komentara.

Application/Queries/Follow/ICheckFollowStatusQuery.cs
Proverava da li korisnik prati nekog drugog (bool kao odgovor).
Koristi int, ID korisnika čiji status praćenja proveravam.

Application/Queries/Follow/IGetFollowersQuery.cs
Vraća paginiranu listu korisnika koji prate trenutnog korisnika
(PagedResponse<GetFollowsDto>).
Koristi FollowSearch.cs.

Application/Queries/Follow/IGetFollowingQuery.cs
Vraća paginiranu listu korisnika koje trenutni korisnik prati
(PagedResponse<GetFollowsDto>).
Koristi FollowSearch.cs.

Application/Queries/Notification/IGetNotificationsQuery.cs
Vraća paginiranu listu notifikacija (PagedResponse<GetNotificationDto>).
Koristi NotificationsSearch.cs za filtriranje notifikacija.

Application/Queries/Post/IGetPostsQuery.cs
Vraća paginiranu listu postova (PagedResponse<GetPostsDto>).
Koristi PostSearch.cs za filtriranje i pretragu postova.

Application/Queries/Post/IGetPostQuery.cs
Dohvata detalje jednog posta (GetPostDetailsDto).
Koristi int kao ID posta koji se traži.

Application/Queries/UseCaseLogs/IGetUseCaseLogsQuery.cs
Vraća paginiranu listu logova korišćenja sistema
(PagedResponse<GetUseCaseLogDto>).
Koristi UseCaseLogSearch.cs za pretragu logova.

Application/Queries/User/IGetUsersQuery.cs
Vraća paginiranu listu korisnika (PagedResponse<GetUsersDto>).
Koristi UserSearch.cs za paginaciju korisnika.

Application/Queries/User/IGetUserQuery.cs
Dohvata detalje jednog korisnika (GetUserDto).
Koristi int kao ID korisnika koji se traži.

Implementation sloj

Ovaj sloj pruža konkretne implementacije servisa ili interfejsa definisanih u Application sloju.
Ovde se nalaze specifične poslovne logike, integracije sa eksternim servisima, validacije (npr. FluentValidation) i ostali detalji koji su specifični za datu implementaciju.

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

Implementation/UseCaseEnum.cs
Ova klasa definiše enumeraciju UseCaseEnum, koja predstavlja sve moguće "use case" operacije (tj. poslovne akcije) koje se mogu izvršiti u sistemu.
Ova enumeracija se koristi za logovanje radnji, kontrolu pristupa ili validaciju dozvoljenih operacija za korisnike.
Omogućava centralizovano upravljanje use-case-ovima, što olakšava održavanje i kasnije dodavanje novih operacija u sistem.

Logging

Implementation/Logging/EFDatabaseLogger.cs
Ovo je klasa koja implementira interfejs IUseCaseLogger.cs
Njen osnovni zadatak je da evidentira (loguje) izvršavanje određenih use case-ova u aplikaciji tako što upisuje odgovarajuće podatke u bazu podataka pomoću Entity Framework-a (EF).
Ovakav pristup omogućava centralizovanu evidenciju aktivnosti u sistemu.

Klasa koristi **dependency injection (DI)** tako što kroz konstruktor prima instancu BlogContext.
Na taj način, umesto da sama kreira kontekst baze podataka, dobija ga od DI kontejnera, što olakšava testiranje i unapređuje fleksibilnost koda.

Metoda *Log()* omogućava evidentiranje svakog izvršenog use case-a. Prihvata tri parametra:

- **IUseCase useCase** – podatke o samom use case-u,

- **IApplicationActor actor** – informacije o korisniku ili sistemskom akteru koji ga je pokrenuo,
- **object data** – dodatne informacije koje se beleže u JSON formatu.

Na osnovu ovih podataka kreira se objekat tipa UseCaseLog, koji se dodaje u kolekciju UseCaseLogs unutar konteksta baze podataka.
Nakon toga, pozivom *_context.SaveChanges()* podaci se trajno upisuju u bazu.

Extensions

Implementation/Extensions/UseCaseExtension.cs
Ova **statička klasa** definiše proširenu metodu (**extension method**) za klasu User, koja omogućava ažuriranje korisničkih dozvola (use cases) na osnovu uloge korisnika (IdRole).

Metoda *UpdateUseCasesForRole()* prima korisnika (User) i kontekst baze podataka (BlogContext) kako bi ažurirala tabelu UserUseCases, osiguravajući da korisnik ima samo one dozvole koje mu pripadaju na osnovu uloge.

Statistička klasa: Definisana je kao statička, što znači da se svi članovi (u ovom slučaju metoda) ne vezuju za instancu klase.

Ekstenziona metoda: Metod *UpdateUseCasesForRole()* je ekstenziona metoda za klasu User.
To znači da se može pozvati direktno na instanci tipa User kao da je njen član (npr. *user.UpdateUseCasesForRole(context)*).
Prvi parametar metode (sa ključnom reči **this**) označava na koji tip se ekstenzija vezuje.

Dohvatanje trenutnih korisničkih dozvola**:**
var currentUseCaseIds
- Filtrira sve unose u tabeli UserUseCases za korisnika, na osnovu IdUser.
- Uzima samo ID-eve (IdUseCase) i konvertuje ih u HashSet<int> radi efikasnog pretraživanja.

Definisanje novih dozvola na osnovu uloge korisnika:
var newUseCases
Koristi **switch** izraz da **mapira** IdRole na odgovarajući skup dozvola (UseCaseEnum).
Svaka uloga (Admin, Author, User) ima specifičan skup operacija koje sme da obavlja.

Pregled uloga i njihovih dozvola:

1. **Admin**
    - Ima **potpunu kontrolu** nad postovima, komentarima, korisnicima, kategorijama i logovima.

- Može upravljati zahtevima za autore i pregledati logove aktivnosti.

2. **Author**
   - Može kreirati i upravljati svojim postovima i komentarima.
   - Ima pristup notifikacijama i sistemu praćenja.
3. **User**
   - Može samo pregledati postove, komentarisati i lajkovati.
   - Može podneti zahtev da postane autor.
   - Ima pristup praćenju i notifikacijama.

Idenftifikacija novih dozvola koje treba dodati**:**
var useCasesToAdd
Pronalazi nove permisije koje korisnik nema trenutno.
Kreira listu novih instanci UserUseCase za svaku novu permisiju.
Primer: Ako korisnik trenutno ima *UseCaseEnum.EFGetOnePostQuery*, ali mu sada treba *UseCaseEnum.EFCreatePostCommand*, dodaje se novi unos.

Identifikacija dozvola koje treba ukloniti:
var useCasesToRemove
Traži sve dozvole koje korisnik trenutno ima, ali ih nema u novom skupu *newUseCases*.
Ove dozvole su suvišne i treba ih ukloniti.
Primer: Ako korisnik pređe iz User u Author ulogu, treba mu obrisati nepotrebne dozvole.

Ažuriranje baze podataka:
Ako postoje nove dozvole (*useCasesToAdd*), dodaju se u bazu.
Ako postoje suvišne dozvole (*useCasesToRemove*), brišu se iz baze.
Na kraju *SaveChanges()* čuva promene.

✅ Automatsko ažuriranje permisija → Kada se promeni uloga korisnika, dozvole se automatski ažuriraju.
✅ Efikasnost → HashSet<int> omogućava brze pretrage i upite.
✅ Čista arhitektura → Održava logiku dozvola unutar jedne metode.

Implementation/Extensions/FileExtension.cs
Ova klasa je ekstenzija metode (extension method) za IFormFile.cs, koja omogućava jednostavno uploadovanje slika na server. Evo kako funkcioniše:
1. Brisanje starog fajla (ako postoji)
2. Generisanje novog naziva fajla

3. Definisanje putanje za upload
4. Kopiranje fajla u odgovarajući folder
5. Vraćanje imena novog fajla koji može biti sačuvan u bazi

Metoda *UploadProfileImage()* se koristi u EFUpdateUserCommand.cs.

Services

Implementation/Services/ImageService.cs
Ova klasa služi za dohvatanje i obradu slika iz servera (wwwroot foldera). Ima 2 metode:
1.*GetImage(string folderName, string imageName)* -> byte[]?
2.*GetMimeType(string filePath)* -> string

Ovaj ImageService.cs se koristi za dohvatanje korisničke profilne slike prilikom ažuriranja korisničkog profila, kao i za dohvatanje slike objave prilikom prikaza sadržaja na platformi, omogućavajući centralizovano upravljanje slikama u oba scenarija. Ovakav pristup olakšava održavanje, jer ImageService.cs sada rukuje svim operacijama nad slikama.

Implementation/Services/NotificationService.cs
Ovo je servis za upravljanje notifikacijama.
Implementira INotificationService.cs i koristi EF za rad sa bazom podataka.
_notificationHubService → Koristi SignalR ili neki drugi real-time sistem za slanje notifikacija korisnicima u realnom vremenu.

Generisanje linka za notifikaciju:
switch statement određuje koji link će se generisati za određenu vrstu notifikacije.
1. Ako je notifikacija tipa *Like*, vodi do posta (/post/{dto.IdPost}).
2. Ako je *Comment*, vodi direktno na komentar (/comment/{dto.IdComment}).
3. Ako je *Follow*, vodi na dashboard sa prikazom novih pratioca.

Kreiranje i čuvanje notifikacija:
1. Generiše se link za notifikaciju.
2. Kreira se nova notifikacija na osnovu prosleđenih podataka.
3. Dodaje se u bazu preko *_context.Notifications.AddAsync(notification)*.
4. Čuva se u bazi pozivom *await _context.SaveChangesAsync().*
5. Šalje se real-time SignalR notifikacija korisniku pomoću *_notificationHubService.SendNotificationToUser(dto.IdUser, dto);*

📌 Zašto se koristi interfejs INotificationHubService?
Zato što želim da logika kreiranja notifikacije i slanja preko SignalR-a budu razdvojene, što olakšava testiranje.

Implementation/Services/LikeService.cs
LikeService.cs je servisna klasa koja se bavi operacijama vezanim za "lajkove" u aplikaciji.
Konkretno, nudi funkcionalnost za uključivanje i isključivanje lajka (*ToggleLike*()).
Koristi ILikeRepository.cs interfejs za komunikaciju sa bazom podataka.

Analiza ToggleLike() metode:
1. Pronalazak postojećeg lajka
Proverava da li korisnik već ima lajk na postu ili komentaru.
Poziva metodu *GetLike()* iz LikeRepository.cs da pronađe lajk u bazi.
2. Ažuriranje postojećeg lajka ili kreiranje novog
Ako lajk već postoji, menja mu status.
Ako lajk ne postoji, kreira novi objekat Like i dodaje ga u bazu.

Repository

Implementation/Repositories/LikeRepository.cs
LikeService.cs je odvojio poslovnu logiku od EFLikePostCommand, EFUnlikePostCommand, EFLikeCommentCommand i EFUnlikeCommentCommand.

LikeRepository.cs omogućava čist rad sa bazom bez direktnog DbContext pristupa u komandama.
ToggleLike() metoda se može koristiti u različitim komandama (LikePost i LikeComment).

Analiza **ToggleLike()** metode:
1. Provera postojećeg lajka
Proverava da li korisnik već ima lajk na postu ili komentaru.
Koristi GetLike() metodu iz LikeRepository.cs da pronađe lajk u bazi.

2. Ažuriranje ili uklanjanje lajka
Ako lajk već postoji i korisnik pokuša ponovo da lajkuje sa istim statusom, lajk se briše.
Ako korisnik menja glas (like → dislike ili obrnuto), ažurira se samo status postojećeg lajka.

3. Kreiranje novog lajka
Ako lajk ne postoji, kreira se novi objekat Like sa svim potrebnim podacima i dodaje se u bazu.

4. Čuvanje promena u bazi
Nakon svake operacije (brisanje, ažuriranje ili dodavanje), poziva se SaveChangesAsync() kako bi se promene sačuvale.

RemoveLike() metoda:
Pronalazi postojeći lajk korisnika za dati post ili komentar.
Ako postoji, uklanja ga iz baze i čuva promene.

Validators

Validatori su u sloju Implementation kreirani pomoću **FluentValidation** biblioteke.
Validatori su ključni jer se koriste za proveru validnosti podataka koji dolaze u aplikaciju, što pomaže u održavanju konzistentnosti i integriteta podataka pre nego što se izvrše bilo kakve poslovne operacije ili upisi u bazu podataka.

Koristi se nasledjivanje iz **AbstractValidator<T>**.

Ovaj pristup omogućava ponovnu upotrebu validacije na više mesta u aplikaciji, čime se smanjuje dupliranje koda.

U konstruktoru validatora definišemo niz pravila.

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

Implementation/Validators/User/RegisterUserValidator.cs
Ova klasa osigurava da podaci za registraciju budu validni.

1. Ime (FirstName) i prezime (LastName) su obavezni i moraju imati bar 2 karaktera.
2. Korisničko ime (Username) je obavezno, mora biti unikatno i imati najmanje 2 karaktera.
3. Email (Email) je obavezan, mora biti u validnom email formatu i mora biti jedinstven.

Ovdje se koristi *DoesNotExistUsername(string username),* koji vraća true ako korisničko ime ne postoji u bazi.

Implementation/Validators/User/UpdateUserValidator.cs
Ova klasa osigurava da podaci koje korisnik unosi prilikom ažuriranja naloga budu ispravni.

1. Ime (FirstName) i prezime (LastName) ne smeju biti prazni.
2. Email (Email) ne sme biti prazan.

3. Korisničko ime (Username) ne sme biti prazno i mora biti jedinstveno.
→ Proverava se da li postoji korisnik sa istim imenom, ali izuzima trenutnog korisnika (da ne blokira vlastitu promenu).
4. Lozinka (Password) treba da sadrži minimum 3 karaktera i da se sastoji samo od slova i brojeva.
→ Ako korisnik ne menja lozinku, validacija se preskače.
5. Slika (Image) ako postoji, mora biti u jednom od sledećih formata: .jpg, .jpeg, .png, .gif

Implementation/Validators/Post/CreatePostValidator.cs
Ova klasa validira podatke pre nego što se kreira nova objava.

1. Naslov (Title) ne sme biti prazan.
2. Kategorije (CategoryIds)

- Korisnik mora odabrati bar jednu kategoriju.
- Kategorije ne smeju imati duplikate.
- Svaka kategorija mora postojati u bazi (*CategoryExists(int id)* metoda).

3. Slika (IdImage) ne sme biti prazna.
4. Sadržaj (Content) ne sme biti prazan.

Implementation/Validators/Post/UpdatePostValidator.cs
Ova klasa osigurava da podaci budu ispravni prilikom ažuriranja postojeće objave.

1. Naslov (Title) ne sme biti prazan i mora imati bar 3 karaktera.
2. Sadržaj (Content) ne sme biti prazan I mora imati bar 5 karaktera.
3. Slika (IdImage) ako se koristi mora postojati u bazi (*ImageExists(int id)* metoda)
4. Kategorije (CategoryIds) ne smeju imati duplikate i provrava se da li svaka postoji u bazi podataka

Implementation/Validators/Post/DeletePostValidator.cs
Ovo je najjednostavnija validacija.
Očekuje int vrednost (ID objave) i proverava samo da nije null ili 0.

Implementation/Validators/Like/LikeCommentValidator.cs
Ova klasa osigurava da podaci budu ispravni kada korisnik lajkuje komentar.

1. IdPost ne sme biti prazan.
2. IdUser ne sme biti prazan.
3. IdComment ne sme biti prazan.
4. Status ne sme biti prazan i mora biti validna vrednost LikeStatus enuma.

Koristi se **Must()** metoda sa **lambda funkcijom** koja proverava da li je vrednost Status definisana u enumeraciji LikeStatus.
To je dobar primer validacije gde se koristi logika da se ograniči unos samo na određene vrednosti, što dodatno sprečava greške.

Implementation/Validators/Like/LikePostValidator.cs
Ova klasa validira podatke kada korisnik lajkuje post.

1. IdPost ne sme biti prazan.
2. IdUser ne sme biti prazan.
3. Status ne sme biti prazan i mora biti validna vrednost LikeStatus enuma.

Implementation/Validators/Follow/FollowUserValidator.cs
Ova klasa osigurava ispravnost podataka prilikom dodavanja novog praćenja između korisnika.
Validacija se vrši nad DTO objektom InsertFollowDto, koji sadrži sledeća polja:
1.IdUser – ID korisnika koji želi da započne praćenje.
2.IdFollowing – ID korisnika koji će biti praćen.
3.FollowedAt – Datum i vreme kada je praćenje započeto.

Validator proverava:
1. Da li IdUser (korisnik koji prati) postoji.
2. Da li IdFollowing (korisnik koji se prati) postoji i nije isti kao IdUser.
3. Datum praćenja, ne sme biti prazan i ne može biti u budućnosti.
4. Da korisnik ne prati istog korisnika više puta.

Pomoćne metode se koristi za proveru da li korisnik postoji u bazi i da li korisnik već prati drugog korisnika.
Ako postoji vraća true i ako nađe zapis vraća true, što znači da je korisnik već zapratio tog korisnika.

Implementation/Validators/Comment/CreateCommentValidator.cs
Ova klasa proverava da li su podaci ispravni kada korisnik dodaje novi komentar na post.
1. Komentar ne sme biti prazan – Tekst komentara (CommentText) je obavezan.
2. Post mora postojati – Proverava se da li post sa tim ID-em postoji u bazi pomoću *PostExists(int id).*
3. Ako komentar ima roditelja (**IdParent**), proverava se da li roditeljski komentar postoji. Ova provera se primenjuje samo ako korisnik unese IdParent.

Implementation/Validators/Comment/UpdateCommentValidator.cs

Proverava da li je tekst komentara popunjen.

Implementation/Validators/Comment/DeleteCommentValidator.cs
Proverava samo da ID komentara nije prazan.

Implementation/Validators/Category/CreateCategoryValidator.cs
Ova klasa osigurava da nova kategorija ima ispravno ime i da je jedinstvena u bazi podataka.

Implementation/Validators/Category/UpdateCategoryValidator.cs
Ova klasa osigurava da ažuriranje kategorije ne naruši jedinstvenost naziva i da naziv nije prazan.

Metoda Must() prima lambda funkciju sa dva parametra:
dto: Ovo je cela instanca DTO-a (UpsertCategoryDto) koja se trenutno validira.
name: Ovo je vrednost svojstva Name iz DTO-a. tj. Ime kategorije koje korisnik unosi.

Uslov: *x.Id != dto.Id*
Ovo je veoma važno u kontekstu ažuriranja.
Kada se ažurira postojeća kategorija, u DTO-u se nalazi i Id te kategorije.
Na ovaj način, uslov osigurava da se ne uzima u obzir trenutna kategorija koja se ažurira, već se traže samo druge kategorije sa istim imenom.

Implementation/Validators/Category/DeleteCategoryValidator.cs
Ova klasa osigurava da ID kategorije koji se briše nije prazan.

Implementation/Validators/AuthorRequest/AuthorRequestValidator.cs
Ova klasa služi za validaciju zahteva autora.
Sprečava slanje zahteva bez korisničkog ID-a.
Sprečava zahteve bez validnog razloga.
Status zahteva mora biti postavljen i proverava se da li je vrednost statusa validan deo enumeracije RequestStatus.

Commands

Komande u sistemu imaju neke zajedničke osobine:
1. Koriste BlogContext i Entity Framework
Svaka komanda koristi BlogContext za rad sa bazom podataka.

2. Koriste FluentValidation za validaciju ulaznih podataka
Pre nego što se podaci promene u bazi, poziva se
_validator.ValidateAndThrow(request).

3. Koriste IApplicationActor za autentifikaciju korisnika
Omogućava prepoznavanje trenutno ulogovanog korisnika (_actor.Id).
Koristi se za proveru prava pristupa i personalizaciju.

4. Koriste Id I Name kao jedinstvene identifikatore
Svaka komanda ima jedinstven ID, koji dolazi iz UseCaseEnum.cs
Konvertuje naziv enum vrednosti u string što se koristi za logovanje, debagovanje ili prikaz u API odgovorima.

5. Postoje dve verzije metoda
async Task ExecuteAsync()
void Execute()

6. Koristi se Dependency Injection (DI)
Odvajanje zavisnosti: Obezebđuje da komanda ne pravi svoje instance BlogContext ili validatora, već ih dobija iz DI kontejnera.

Implementation/Commands/Category/EFCreateCategoryCommand.cs
Ova klasa implementira interfejs ICreateCategoryCommand.cs, što znači da predstavlja "komandu" (command) u CQRS arhitekturi.
Cilj ove komande je kreiranje nove kategorije u bazi podataka.

Ovaj konstruktor koristi Dependency Injection da bi dobio:
1. CreateCategoryValidator – FluentValidation validator za validaciju podataka.
2. BlogContext – Entity Framework kontekst baze podataka.

Ovim pristupom, komanda ne zavisi direktno od kreiranja novih objekata, već koristi injektovane zavisnosti (što omogućava lakše testiranje i modularnost).

Kreira se novi Category objekat (entitet) u memoriji.
Njegov “Name” se postavlja na vrednost iz *request.Name*
Još nije dodat u bazu, samo postoji kao običan C# objekat u RAM-u.
Dodaje se kategorija u EF Core kontekst, ali još nije upisana u bazu.
Nakon *SaveChanges()* EF Core generiše **SQL INSERT** komandu i izvršava nad bazom.

Implementation/Commands/Category/EFUpdateCategoryCommand.cs
Ova komanda služi za ažuriranje postojeće kategorije u bazi podataka.

Koristim **Find(request.Id)** što je brži način pronalaska u odnosu na **FirstOrDefault()** ako je entitet već u EF Core **cache-u**, jer ne pravi opet **SELECT** upit ka bazi.

✅ Ako znam da tražim samo po Id i DbContext je aktivan, *Find()* je bolji jer je brži.
✅ Ako treba *Include()* ili filteri, koristim *FirstOrDefault()* jer nudi više mogućnosti.

Ako kategorija ne postoji, bacam izuzetak.

Implementation/Commands/Category/EFDeleteCategoryCommand.cs
Ova komanda je primer **logičkog brisanja**.
Umesto da se kategorija potpuno ukloni iz baze, ona se samo označava kao obrisana *(IsDeleted = true).*
Ovaj pristup omogućava kasniji povratak ili obnovu podataka.
Pored toga, komanda implementira niz provera kao što su validacija, proveru postojećih postova u kategoriji, proveru da li je kategorija već obrisana što omogućava veću sigurnost i preciznost prilikom brisanja.

Implementation/Commands/AuthorRequest/EFCreateAuthorRequestCommand.cs
Ovo je komanda za kreiranje zahteva za autora.
IdUser u DTO objektu se postavlja na ID trenutno prijavljenog korisnika.
Ako zahtev već postoji u bazi, baca se *AlreadyAddedException*.
Ako zahtev ne postoji, kreira se novi AuthorRequest objekat i dodaje u bazu.

Implementation/Commands/AuthorRequest/EFUpdateAuthorRequestCommand.cs
Komanda za ažuriranje zahteva za autora.
Pronalazak zahteva pomoću *Find().*
Ako zahtev ne postoji, baca se *EntityNotFoundException*.
Ažuriranje statusa na novu vrednost.
Pronalazi se korisnik kome pripada zahtev i menja mu se uloga i ažuriraju se njegovi use-case-ovi.

Implementation/Commands/Comment/EFCreateCommentCommand.cs
✅ Koristi CreateCommentValidator.cs za validaciju UpsertCommentDto pre nego što započne obradu.
✅ **Transakcija** se koristi kako bi se obezbedila konzistentnost baze podataka u slučaju greške.
✅ Kreira novi entitet Comment, dodeljuje mu korisnika koji ga je kreirao i dodaje ga u bazu podataka.
✅ Pronalazi vlasnika posta i, ako je različit od autora komentara, dodaje notifikaciju.
✅ Ako je komentar odgovor na drugi komentar, dodaje notifikaciju i vlasniku parent

komentara.
✅ Ako je sve uspešno, transakcija se potvrđuje (**CommitAsync()**).
✅ Notifikacije se šalju u **Task.Run**, što sprečava da obustave izvršenje glavne transakcije.
✅ Ako se desi greška, transakcija se vraća na prethodno stanje (**RollbackAsync()**).

Implementation/Commands/Comment/EFUpdatePersonalCommentCommand.cs
1. Proverava da li komentar postoji (*Find(request.Id)*).
2. Proverava da li trenutni korisnik (*_actor.Id*) ima pravo da ažurira komentar.
3. Koristi FluentValidation za validaciju (*_validator.ValidateAndThrow(request)*).
4. Ažurira *CommentText* i čuva promene (*_context.SaveChanges()*).

Implementation/Commands/Comment/EFDeleteCommentCommand.cs
Omogućava brisanje bilo kojeg komentara (administrator).
Ako komentar ne postoji, baca *EntityNotFoundException*.
Ako je već obrisan, baca *AlreadyDeletedException*.
Postavlja:
DeletedAt = true
IsDeleted = true
IsActive = false

Implementation/Commands/Comment/EFDeletePersonalCommentCommand.cs
Omogućava korisniku da obriše samo svoje komentare.
Koristi FluentValidation za validaciju request vrednosti.
Proverava da li trenutni korisnik (*_actor.Id*) ima pravo da briše komentar.
Ako nije vlasnik, baca *UnauthorizedUserAccessException*.
Ako komentar ne postoji, baca *EntityNotFoundException*.
Ako je već obrisan, baca *AlreadyDeletedException*.
Postavlja:
DeletedAt = true
IsDeleted = true
IsActive = false

Implementation/Commands/Email/SMTPEmailSender.cs
Ova klasa implementira interfejs IEmailSender.cs i omogućava slanje e-mail poruka putem SMTP servera.

Koristi Dependency Injection da dobije SMTP postavke koje su definisane u EmailSettings.cs klasi.

IOptions<EmailSettings> je način da pristupimo konfiguracionim podešavanjima. *emailSettings.Value* vraća konkretne vrednosti iz konfiguracije.

Kreira novi **SmtpClient objekat** i konfiguriše ga koristeći vrednosti iz EmailSettings.

MailMessage objekat predstavlja e-mail poruku.

Poziva **smtp.Send()**, što zapravo šalje e-mail preko definisanog SMTP servera.

Implementation/Commands/Follow/EFFollowCommand.cs
Sve promene (dodavanje praćenja i notifikacije) su smeštene unutar transakcije, što osigurava konzistentnost podataka.
Kada korisnik nekoga zaprati, sistem automatski kreira notifikaciju za praćenog korisnika.

Implementation/Commands/Follow/EFUnfollowCommand.cs
Omogućava korisniku da prestane da prati drugog korisnika tako što briše zapis iz tabele Followers.

Implementation/Commands/Like/EFLikeCommentCommand.cs
Ova klasa implementira ILikeCommentCommand.cs interfejs i odgovorna je za:
1. Validaciju ulaznog zahteva pomoću FluentValidation
2. Upravljanje procesom lajkovanja komentara koristeći servis ILikeService.cs
3. Kreiranje notifikacije kada korisinik lajkuje komentar pomoću INotificationService.cs
4. Rad sa bazom podatak kroz BlogContext i transakcije.

Implementation/Commands/Like/EFLikePostCommand.cs
Ova klasa implementira ILikePostCommand.cs interfejs i odgovorna je za:
1. Validaciju zahteva pomoću LikePostvalidator
2. Lajkovanje posta preko ILikeService
3. Kreiranje notifikacije ako je post uspešno lajkovan pomoću INotificationService.cs
4. Rad sa bazom podataka uz korišćenje transakcija.

Implementation/Commands/Like/EFUnlikePostCommand.cs
Ova klasa implementira interfejs IUnlikePostCommand i odgovorna je za:
1. Proveru postojanja posta - Proverava da li je ID posta prosleđen u zahtevu i da li odgovarajući post postoji u bazi podataka.
2. Upravljanje procesom uklanjanja lajka - Koristi servis ILikeService za uklanjanje lajka sa posta.
3. Rad sa bazom podataka kroz BlogContext - Nakon uklanjanja lajka, čuva promene u bazi podataka pomoću SaveChangesAsync().
4. Definisanje ID-ja i naziva use-case komande - Svaka komanda ima jedinstveni identifikator i naziv definisan pomoću UseCaseEnum.

Implementation/Commands/Like/EFUnlikeCommentCommand.cs
Ova klasa implementira interfejs IUnlikeCommentCommand i odgovorna je za:
1. Proveru postojanja komentara - Proverava da li je ID komentara prosleđen u zahtevu i da li odgovarajući komentar postoji u bazi podataka.
2. Upravljanje procesom uklanjanja lajka - Koristi servis ILikeService za uklanjanje lajka sa komentara.
3. Rad sa bazom podataka kroz BlogContext - Nakon uklanjanja lajka, čuva promene u bazi podataka pomoću SaveChangesAsync().
4. Definisanje ID-ja i naziva use-case komande - Svaka komanda ima jedinstveni identifikator i naziv definisan pomoću UseCaseEnum.

Implementation/Commands/Notification/EFCreateNotificationCommand.cs
Ovo je komanda koja kreira notifikaciju.
Koristi INotificationService.cs da delegira kreiranje notifikacije.
Automatski postavlja IdUser tako da odgovara trenutnom korisniku (_actor.Id).

Umesto direktnog upisivanja u bazu podataka, koristim servis koji sadrži logiku za dodavanje notifikacija. Na taj način imam bolju separaciju koda i testabilnost.

Implementation/Commands/Notification/EFMarkAllNotificationsAsReadCommand.cs
Ova komanda označava sve notifikacije korisnika kao pročitane.

Umesto da ažurira notifikacije jednu po jednu, koristi **EF BulkExtensions**.
Na taj način se poboljšavaju performanse kada korisnik ima veliki broj nepročitanih notifikacija.

Implementation/Commands/Post/EFCreatePostCommand.cs
Ova klasa je implementacija interfejsa ICreatePostCommand.cs i koristi se za kreiranje novog posta u bazi podataka.
Ova klasa sadrži logiku za kreiranje posta, kao i da obavi povezane operacije poput slanja notifikacija pratiocima.

Prvo se dodaje post u bazu i komituje transakcija.
Nakon uspešnog dodavanja, asinhrono se šalju notifikacije.
Osigurano je da se post uvek doda u bazu, bez obzira da li notifikacija uspe, jer se notifikacije sada šalju paralelno pomoću **Task.WhenAll().**

Implementation/Commands/Post/EFUpdatePostCommand.cs
1. Koristi validator da proveri ispravnost podataka.
2. Dohvata post iz baze.
3. Ažurira podatke o postu.
4. Ažuriranje kategorije posta:

4.1. Dobijanje trenutnih kategorija povezanih sa postom (*currentCategoryIds*).
4.2. Identifikacija kategorija koje treba ukloniti (one koje više nisu u *request.CategoryIds*).
4.3. Identifikacija kategorija koje treba dodati (nove koje nisu u trenutnim).
4.4. Uklanjanje kategorija (pronalazi odgovarajuće veze i briše ih).
4.5. Dodavanje novih kategorija (kreira nove veze sa postom).

Sve promene (ažuriranje posta i kategorija) se izvršavaju unutar transakcije.
Ako bilo šta pukne, pozivam *transaction.RollbackAsync(),* pa podaci ostaju netaknuti.

Implementation/Commands/Post/EFUpdatePersonalPostCommand.cs
Implementacija je identična običnom ažuriranju posta, ali je ova verzija namenjena autorima postova.
Ograničava ažuriranje samo na korisnike koji su vlasnici posta, dok prethodna implementacija omogućava administratorima da ažuriraju bilo koji post.

Implementation/Commands/Post/EFDeletePostCommand.cs
Ovu komandu koristi administrator koji može obrisati bilo koji post.
1. Validacija request-a (id post)
2. Proverava da li post postoji.
3. Proverava da li je već obrisan post.
Koristi se „soft“ delete.

Implementation/Commands/Post/EFDeletePersonalPostCommand.cs
Ovu komandu koriste autori, mogu samo svoje postove da brišu.
1. Pronalazi post po id-u.
2. Proverava da li korisnik ima prava.
3. Proverava da li je post već obrisan.
Koristi se “soft” delete.

Implementation/Commands/User/EFRegisterUserCommand.cs
Ova klasa implementira komandni obrazac za registraciju korisnika u ASP.NET aplikaciji sa Entity Framework-om.
Klasa ima 3 zavisnosti koje dolaze putem konstruktora:
_context – Pristup bazi podataka koristeći Entity Framework
_sender – Servis za slanje e-pošte
_validator – FluentValidation validacija korisniičkog unosa

Metoda *Execute():*
1. Proverava podatke koje unosi korisnik (RegisterUserDto).
2. Pravi novi objekat User koristeći podatke iz *requestDto*.
3. Lozinka se hešira pomoću BCrypt algoritma.

4. Korisnik dobija podrazumevanu sliku profila.
5. Automatski mu se dodeljuje User uloga.
6. Transakcija osigurava da se svi SQL upiti izvrše zajedno, ako se desi greška, svi upiti će biti poništeni (Rollback).
7. Dodaje novog korisnika u bazu i čuva promene.
8. Ažurira dozvole (use-case) u skladu sa korisničkom ulogom.
9. Poziva privatnu metodu koja šalje e-mail potvrde registracije korisniku.

Metoda *SendRegistrationMail(string mail)*:
Kreira email objekat sa naslovom, sadržajem i adresom primaoca.
Ako se e-mail ne može poslati, ispisuje se poruka o grešci u konzolu (ne prekida se registracija).

Implementation/Commands/User/EFUpdateUserCommand.cs
Ova klasa omogućava ažuriranje korisničkih podataka.
Metoda *ExecuteAsync()*:
1. Koristi asinhroni pristup (async/await) za bolje performanse.
2. Prima DTO objekat (UpsertUserDto), koji sadrži podatke za ažuriranje korisnika.
3. Pre bilo kakvih promena, uneti podaci se validiraju pomoću FluentValidation.
4. Pronalazi korisnika po Id iz baze (*FirstOrDefaultAsync()*).
5. Ažuriraju se samo prosleđeni podaci iz request-a.
6. Ako je korisnik poslao novu profilnu sliku, ona se postavlja u folder UserImages i ažurira u bazi.
7. Metoda UploadProfileImage("UserImages") sačuva sliku i vrati njen URL.
8. Lozinka se menja samo ako korisnik unese novu vrednost. Hešira se pomoću BCrypt kako bi se sigurno sačuvala u bazi.

Implementation/Commands/User/EFDeleteUserCommand.cs
Ova klasa implementira komandu za brisanje korisnika.
Koristi se “soft” delete.
Metoda *Execute()*:
1. Prima ID korisnika (request) koji treba da bude obrisan.
2. *Find(request)* traži korisnika u bazi koristeći njegov ID.
3. Ako je korisnik već obrisan (*IsDeleted == true*), baca se *AlreadyDeletedException*.

Soft delete pristup:
1. Postavlja vreme brisanja (*DeletedAt = DateTime.UtcNow*).
2. Obeležava korisnika kao obrisanog (*IsDeleted = true*).

3. Onemogućava korisnika (*IsActive = false*), što znači da više ne može da se prijavi ili koristi sistem.

Ključne prednosti soft delete pristupa:
Podaci nisu trajno obrisani – Omogućava povratak korisnika ako je potrebno.
Sigurnost – Smanjuje rizik od slučajnog brisanja važnih podataka.
Isključivanje umesto brisanja – Sprečava greške koje mogu nastati ako je korisnik povezan sa drugim entitetima (npr. komentari, postovi).

Queries

Implementation/Queries/AuthorRequest/EFGetAuthorRequestsQuery.cs
Ovaj Query implementira interfejs IGetAuthorRequestsQuery.cs i dohvata zahteve korisnika koji žele da postanu autori na blogu.
Ovaj upit omogućava filtriranje, paginaciju i selekciju iz baze podataka.

Metoda *Execute()*:
1. Uzima objekat AuthorRequestSearch, koji sadrži kriterijume za paginaicju.
2. Vraća PagedResponse<GetAuthorRequestsDto>, što znači da vraća rezultate u obliku paginirane liste DTO objekata.
3. Dohvata tabelu AuthorRequests iz baze podataka i pretvara je u **IQueryable**, što omogućava dinamičko dodavanje filtera.
4. Obezbeđuje da se prikazuju samo aktivni zahtevi (*IsActive == true*).

Izračunava koliko stavki treba preskočiti pre uzimanja sledeće stranice.
Primer: Ako je PerPage = 10, a Page = 3, treba preskočiti (10 * (3 - 1)) = 20 stavki.

.Select(x => new GetAuthorRequestsDto {...}) – mapira entitete u DTO objekte (GetAuthorRequestsDto).
.ToList() – izvršava upit i vraća listu rezultata.

Implementation/Queries/Category/EFGetCategoriesQuery.cs
Ova klasa dohvata sve kategorije bez baginacije ili sa paginacjiom.
Koristio sam ***GetAll*** kako bi omogućio fleksibilnost pri dohvatanju kategorija, tj. da mogu pozvati API i dobiti sve kategorije odjednom bez paginacije, ili da ih dobijam paginirano.

Implementation/Queries/Category/EFGetOneCategoryQuery.cs
Služi za dohvatanje jedne kategorije iz baze podataka zajedno sa njenim povezanim postovima. Takođe, podržava paginaciju kako bi se ograničio broj postova koji se vraćaju u odgovoru.

Metoda *Execute():*
1. Prima *search* objekat koji sadrži parametre (npr. Id kategorije i podatke za paginaciju).
2. Prvo se osigurava da ***search.Page*** nije manji od 1.
3. Dohvata kategoriju po ID-u. Uključuje (*Include*()) sve povezane entitete:
CategoryPosts → Post (svi postovi u kategoriji)
PostCategories → Category (sve kategorije u koje je određeni post smešten), User (korisnik koji je kreirao post)
4. Ako kategorija ne postoji, baca se *EntityNotFoundException.*
Ovo osigurava da ne dođe do rada sa null objektom.
5. Kreira upit koji selektuje samo postove iz *CategoryPosts* veze.
6. Broji ukupan broj postova za datu kategoriju.
7. Računa ukupan broj stranica (***int pageCount***).
Ako je korisnik tražio veću stranicu od maksimalne, vraćamo ga na poslednju dostupnu stranicu.
8. ***skipCount*** određuje koliko postova treba preskočiti pre nego što uzmem tražene postove.

Transformacija podataka u DTO objekte:
Skip & Take – preskače prethodne postove i uzima samo one koji pripadaju traženoj stranici.
Kreira DTO objekt **GetPostInCategoryDto** koji sadrži:

- Osnovne informacije o postu.
- Podatke o korisniku koji je kreirao post.
- Liste svih kategorija u kojima se post nalazi.

CategoryPostsResponse se koristi kao standardizovan odgovor sa podacima o kategoriji i listom postova.

Implementation/Queries/Comment/EFGetCommentsQuery.cs
Ovaj Query implementira interfejs IGetCommentsQuery.cs i služi za preuzimanje komentara iz baze podataka. Implementira paginaciju, a takođe vraća i broj komentara iz prethodnih 30 dana.

Učitava komentare iz baze zajedno sa:
1. User – korisnikom koji je ostavio komentar.
2. Likes – lajkovima na komentaru.
3. Post – postom na koji se komentar odnosi.

.AsQueryable() omogućava dodavanje dodatnih filtera i paginacije.

Mapira entitete u GetCommentsDto koji sadrži:
1. Osnovne podatke o komentaru (tekst, datum kreiranja, parent ID itd.).
2. Informacije o korisniku (Username).
3. Naslov posta na koji se komentar odnosi (PostTitle).
4. Broj lajkova (LikesCount).
5. Detalje o lajkovima (Likes kao lista GetCommentLikesDto).

Implementation/Queries/Comment/EFGetOneCommentQuery.cs
Implementira interfejs IGetCommentQuery.cs i služi za dohvatanje jednog komentara iz baze podataka na osnovu njegovog ID-a.
Uključuje povezane entitete (User, Post, Likes).

Vraća DTO (GetCommentDto), koji sadrži:
1. Osnovne informacije o komentaru (tekst, datum kreiranja, da li je obrisan).
2. Informacije o korisniku (Username, FirstName, LastName).
3. Detalje o postu (PostTitle, IdPost).
4. Broj lajkova (LikesCount).

Implementation/Queries/Follow/EFGetFollowingsQuery.cs
Implementira interfejs IGetFollowingQuery.cs i služi za dohvatanje spiska korisnika koje prati određeni korisnik.
Implementira paginaciju i vraća podatke o korisnicima koje je korisnik odabrao da prati.
Filtrira sve zapise u tabeli Followers tako da se vraćaju samo oni koji se odnose na korisnika sa ID-em *search.IdUser*.
Za svakog korisnika kojeg prati, vraćaju se podaci kao što su: *Id, FirstName, LastName, Username, Email, ProfilePicture.*

Implementation/Queries/Follow/EFGetFollowersQuery.cs
Implementira interfejs IGetFollowersQuery.cs i služi za dohvatanje spiska korisnika koji prate određenog korisnika.
Kao i prethodni upit za followings, ovaj upit takođe implementira paginaciju i vraća podatke o korisnicima koji prate datog korisnika,
Filtriranje: Upit filtrira sve zapise u tabeli Followers tako da se vraćaju samo oni koji se odnose na korisnika sa ID-em iz *search.IdUser* kao korisnika kojeg prate.
Za svakog korisnika koji prati, vraćaju se podaci kao što su: *Id, FirstName, LastName, Username, Email, ProfilePicture.*

Implementation/Queries/Follow/EFCheckFollowStatusQuery.cs
Ova klasa je implementacija interfejsa ICheckFollowStatusQuery.cs koji se koristi za proveru da li korisnik prati određenog drugog korisnika.

Upit proverava da li postoji veza u tabeli Followers između trenutno prijavljenog korisnika (_actor.Id) i korisnika koji se identifikuje preko idFollowing.

Upit koristi ***Any()*** metodu koja proverava da li u tabeli Followers postoji bar jedan zapis koji zadovoljava sledeće:
1. f.IdFollower == _actor.Id (ID trenutno ulogovanog korisnika).
2. f.IdFollowing == idFollowing (ID korisnika kojeg prati).

Ovaj upit je koristan u situacijama kada je potrebno omogućiti funkcionalnosti poput:
- Prikazivanje opcije za praćenje / prestanak praćenja na korisničkom profilu.
- Prikazivanje različitih UI opcija baziranih na tome da li korisnik prati drugog korisnika.

Implementation/Queries/Notification/EFGetNotificationsQuery.cs
Ova klasa predstavlja implementaciju query-a za dobijanje obaveštenja (notifications), koristi se za filtriranje i dohvatanje obaveštenja prema zadatim parametrima pretrage.

Metoda *Execute():*
1. Prima NotificationsSearch objekat, koji sadrži parametre za pretragu obaveštenja i vraća paginirani odgovor sa listom obaveštenja.
2. Filtriranje po korisniku (IdUser): Ako je vrednost *IdUser* prisutna u *search*, koristi se za filtriranje obaveštenja prema korisniku koji je primio obaveštenje.
3. Filtriranje po tipu obaveštenja (*Type*): Ako je vrednost Type prisutna u *search*, obaveštenja se filtriraju prema vrsti obaveštenja.
Ovdje se vrsta prepoznaje kao *enum*, pa je potrebno izvršiti **kastovanje** sa *(NotificationType)search.Type.Value*.
4. Odabrana obaveštenja se mapiraju u GetNotificationDto, koji sadrži samo potrebna polja za klijentsku aplikaciju, kao što su ID, tip, sadržaj, status (pročitano/novi), link ka obaveštenju i datum kreiranja.

Implementation/Queries/Post/EFGetPostQuery.cs
Koristi se za dobijanje detalja o pojedinačnom blog postu. Koristi Entity Framework za pristup podacima i implementira IGetPostQuery.cs, što znači da implementira metod *Execute()* koji vraća GetPostDetailsDto objekat na osnovu prosleđenog ID posta.

Metoda *Execute()* je definisana u interfejsu IQuery<GetPostDetailsDto, int> i koristi ID posta (idPost) da bi dobila sve detalje o tom postu iz baze podataka.

Ovaj upit vraća DTO objekat (GetPostDetailsDto), koji predstavlja podatke koje vraćamo korisniku.

Upit ka bazi podataka:

Ovaj deo koristi Entity Framework da bi se napravio upit ka bazi podataka.
Include() se koristi da se učitaju povezani entiteti. Dakle:
1. Likes (lajkovi) su povezani sa postom.
2. User (korisnik) koji je postavio post.
3. Image (slika) vezana za post.
4. Comments (komentari) povezani sa postom, zajedno sa korisnicima koji su ih ostavili i lajkovima za svaki komentar.
5. PostCategories i njihove povezane kategorije.

Priprema komentara:
Prvo se prikupljaju komentari posta.
Ako komentari ne postoje, koristi se prazna lista.
Zatim, komentari se grupišu po IdParent (što omogućava kreiranje **hijerarhije komentara**, tj. odgovora na komentare).

Funkcija za izgradnju hijerarhije komentara:
Ova metoda rekurzivno gradi hijerarhiju komentara.
Ako postoji roditeljski komentar (IdParent), komentari se nalaze u odgovarajućoj grupi, a potom se za svaki komentar kreira DTO koji uključuje sve relevantne podatke.
Komentari mogu imati svoju "decu" (odgovore), i to se rešava rekurzivnim pozivom **BuildCommentTree**.

Ova metoda efikasno gradi stablo tako što koristi **rekurziju** i pretraživanje prethodno grupisanih komentara.
Umesto da se za svaki komentar posebno vrši upit u bazu, svi komentari su već učitani i grupovani u memoriji, što smanjuje broj **roundtrip**-ova ka bazi i poboljšava performanse.

- Prvo se učitava post sa svim svojim komentarima (*Include(com => com.Comments)*), zajedno sa korisnicima i lajkovima.
- Komentari se učitavaju kao **ravna lista**, gde svaki komentar ima *IdParent* koji pokazuje na roditeljski komentar (ako postoji).
- Grupisanje komentara po *idParent*
- Ova linija kreira rečnik (Dictionary) gde je ključ *IdParent*, a vrednost lista komentara sa tim *IdParent*.
- *IdParent ?? 0* znači da se null vrednosti (komentari bez roditelja) tretiraju kao 0.
Ovo omogućava da komentari prvog nivoa budu grupisani pod ključem 0.

Kako funkcioniše rekurzija u *BuildCommentTree*?

1. Pronalazak komentara prvog nivoa
    - Komentari koji nemaju roditelja (*IdParent* = *null*) se grupišu pod ključem 0.
    - Prvi poziv *BuildCommentTree(null)* pronalazi sve komentare sa *IdParent* = *0.*
2. Rekurzivni proces
    - Za svaki komentar prvog nivoa (npr. Id = 1), poziva se *BuildCommentTree(1),* što pronalazi njegove podkomentare (IdParent = 1).
    - Proces se ponavlja za svaki komentar dok ne budu obrađeni svi nivoi.
3. Zaustavljanje rekurzije
    - Ako za trenutni *IdParent* ne postoje komentari u *commentsLookup*, vraća se prazna lista.
    - To znači da komentar nema podkomentare i rekurzija prestaje da se širi.

Rekurzija se prirodno završava kada više nema podkomentara za obradu.

Kreiranje rezultata:
Kreira DTO (GetPostDetailsDto) sa svim podacima koji su prikupljeni iz baze:
1. Osnovni podaci o postu kao što su ID, naslov, sadržaj, datum kreiranja, itd.
2. Slika i korisnik koji je postavio post.
3. Kategorije posta.
4. Lajkovi za post.
5. Komentari, uz pomoć prethodno definisane rekurzije.

Implementation/Queries/Post/EFGetPostsQuery.cs

Ova klasa predstavlja implementaciju inerfejsa IGetPostsQuery.cs, za dobijanje blog postova sa podrškom za pretragu, filtriranje, sortiranje, paginaciju i prikupljanje informacija o kategorijama postova.

Metoda *Execute():*
Ovo je glavna metoda koja izvršava upit.
Metoda prima parametar *PostSearch*, koji sadrži kriterijume za pretragu i filtriranje postova, kao što su naslov, sadržaj, kategorije i sortiranje.

Ovde se kreira osnovni upit svih postova, uključujući povezane kategorije (preko PostCategories i Category), kako bi se kasnije filtrirali ili prikazivali podaci o kategorijama postova.

Postoje različiti uslovi za filtriranje postova na osnovu pretrage:
1. Preraga po naslovu
2. Pretraga po sadržaju
3. Filtriranje po kategorijama
4. Sortiranje po datumu kreiranja

Kreira se odgovor u formatu PagedResponse<GetPostsDto>, koji sadrži sledeće:
1. CurrentPage: Trenutna stranica.
2. ItemsPerPage: Broj postova po stranici.
3. TotalCount: Ukupan broj postova koji odgovaraju pretrazi.
4. LastMonthCount: Broj postova koji su kreirani u poslednjih 30 dana.
5. Items: Lista postova sa detaljima, uključujući ID, naslov, datum kreiranja, korisničko ime autora, putanju slike i kategorije.

Implementation/Queries/UseCaseLogs/EFGetUseCaseLogsQuery.cs
Ova klasa je odgovorna za dobavljanje logova korisničkih akcija.

Metoda Execute() izvršava logiku za dobavljanje podataka o logovima korisničkih akcija na osnovu parametara pretrage koje prima putem objekta *search* (tipa UseCaseLogSearch).

Kreira se upit za preuzimanje svih logova iz baze, zatim se primenjuju filteri ako su zadati parametri u pretrazi.

Filtriranje po:
1. akteru,
2. imenu use-case-a,
3. datumu(from i to)
4. sortiranje po datumu

Kada se svi filteri primene, potrebno je da se upit paginira.
To se postiže pomoću parametara *search.PerPage* i *search.Page*

Na kraju, PagedResponse<GetUseCaseLogDto> se kreira sa sledećim podacima:
1. CurrentPage: trenutna stranica koju korisnik vidi.
2. ItemsPerPage: broj stavki po stranici.
3. TotalCount: ukupni broj logova koji zadovoljavaju kriterijume pretrage.
4. Items: lista DTO objekata (GetUseCaseLogDto), koja se koristi za slanje rezultata.

Implementation/Queries/User/EFGetUsersQuery.cs
Ovaj query predstavlja implementaciju upita za dobijanje korisnika iz baze.
Upit se koristi za pretragu korisnika u bazi, sa mogućnošću filtriranja na osnovu različitih kriterijuma, kao što su ime korisnika, email, i tip korisnika (autor, korisnik, admin).
Takođe, implementira paginaciju i vraća podatke o korisnicima sa informacijama o ulogama i vremenu kreiranja.

Metoda *Execute()* izvršava glavni upit za dobijanje korisnika sa mogućnošću filtriranja i paginacije.
Prima parametar UserSearch, koji sadrži kriterijume za pretragu, i vraća objekat tipa PagedResponse<GetUsersDto>, koji sadrži korisničke podatke sa informacijama o stranicama.

UserSearch.cs
1. OnlyAuthors: Ako je postavljeno na true, upit filtrira korisnike koji imaju ulogu "Author".
2. Username: Ako je postavljen, pretraga filtrira korisnike čije korisničko ime sadrži zadati tekst (ignorisan je razlika između velikih i malih slova).
3. Email: Ako je postavljen, pretraga filtrira korisnike čiji email sadrži zadati tekst (opet, ignorisan je razlika između velikih i malih slova).

PagedResponse<GetUsersDto>
Kreira se objekat koji sadrži sve potrebne informacije za paginaciju, kao i listu korisnika sa svim potrebnim podacima (kao što su ID, ime, email, korisničko ime, rola, itd.).
Nakon što su svi podaci obrađeni i filtrirani, rezultati se vraćaju u odgovoru kao paginirani skup korisničkih podataka.

Implementation/Queries/User/EFGetOneUserQuery.cs
Ovaj query se koristi za dobijanje podataka o jednom korisniku iz baze podataka, uključujući korisničke podatke, postove, komentare, lajkove, kao i broj pratilaca.
Ovaj upit omogućava da se dobiju svi relevantni podaci o korisniku na temelju njegovog ID-a.

Execute(int idUser) je metoda koja prima ID korisnika i vraća podatke o tom korisniku u formatu GetUserDto.
Upit uključuje puno veza i uključivanja različitih entiteta.

Upit koristi ***AsNoTracking()*** da bi se optimizovala performansa čitanja podataka bez praćenja promena, jer se podaci samo čitaju, a ne ažuriraju.

Koristi *Include()* za uključivanje povezanih entiteta: Role, Posts, Comments, Likes, i UserUseCases.
Svaki *Include()* predstavlja vezu sa drugim entitetima (poput postova, kategorija postova, slika, komentara i sl.).

Takođe, koristi ***ThenInclude()*** da bi uključio povezane entitete unutar kolekcija (npr. uključivanje kategorija postova ili slika vezanih za postove i komentare).

Kroz Select, upit izračunava dodatne informacije o korisniku:
1. broj pratilaca
2. broj osoba koje korisnik prati
3. broj postova korisnika

Kada se korisnik pronađe, vraća se objekat tipa GetUserDto koji sadrži sve relevantne informacije o korisniku.
Svi povezani entiteti kao što su UserUseCases, UserPosts, UserComments, i CommentLikes se mapiraju u odgovarajuće DTO objekte.
1. UserUseCases: Svi slučajevi upotrebe (akcije koje korisnik može da izvrši) se mapiraju u GetUserUseCaseDto.
2. UserPosts: Svi postovi korisnika se mapiraju u GetUserPostsDto, uključujući slike i kategorije postova.
3. UserComments: Komentari korisnika su mapirani u GetUserCommentsDto, a uključuju i podatke o postu na koji je komentar ostavljen.
4. CommentLikes: Lajkovi na komentarima korisnika se mapiraju u LikeDto.

API

Predstavlja prezentacioni sloj koji izlaže RESTful endpoint-e.
Tu se nalaze kontroleri, middleware-i, rute, autentifikacija, autorizacija i ostale konfiguracije vezane za HTTP komunikaciju.

Princip rada API-a

Kada klijent pošalje HTTP zahtev:

- Za pretragu entiteta koristi query na odgovarajućem endpoint-u.
- Za kreiranje entiteta koristi command.
- API prepoznaje koji use-case treba da se izvrši na osnovu zahteva i delegira implementaciju Application sloju.

- Implementacija diktira način obrade podataka – bilo da se koristi Entity Framework (EF) ili drugi mehanizam za rad sa bazom.
- Na kraju, API koristi domenske objekte za čuvanje podataka u bazu.

Slojna arhitektura

Sistem je razvijen kroz jasno definisane slojeve koji određuju način komunikacije između komponenti.
Klijentske aplikacije ne moraju da budu svesne unutrašnjih granica sistema – one jednostavno šalju zahteve API-u.
Na primer, kada klijent iz browser-a pošalje AJAX zahtev na http://89.445.21.09:80/api/blogs sa HTTP POST metodom, on ne zna da taj zahtev dolazi do kontrolera.
Kontroler zatim delegira posao dalje kroz sistem.

Kontroler ne bi trebalo da zavisi od konkretne implementacije, već isključivo od apstrakcije.
Metode u kontroleru komuniciraju sa interfejsima, a ne direktno sa implementacijama.
To omogućava fleksibilnost i modularnost koda, bez obzira na to da li se u pozadini koristi SQL baza, MongoDB ili čak običan tekstualni fajl.

Principi dizajna

U razvoju ovog API-a primenjeni su **SOLID** principi:

- **Single Responsibility Principle (SRP)** – Svaka klasa i komponenta API-a ima jasno definisanu odgovornost.
  Na primer, kontroleri su podeljeni tako da rukovode samo jednom vrstom resursa (npr. CommentsController.cs upravlja komentarima, dok LikesController.cs rukuje lajkovima).
  Ovo olakšava održavanje, testiranje i proširivost koda.
- **Liskov Substitution Principle (LSP)** – Svaka apstrakcija (nadklasa, interfejs) mora biti kompatibilna sa bilo kojom svojom implementacijom.
- **Interface Segregation Principle (ISP)** – Klijent prima samo podatke koji su mu potrebni, bez suvišnih detalja o internoj implementaciji klase.
- **Dependency Injection (DI)** – Vezuje interfejse za implementacije, omogućavajući kontroleru da zatraži određeni interfejs, dok DI container obezbeđuje odgovarajuću implementaciju.

API/api.csproj

XML konfiguracija u ASP .NET API projektu definiše sve ključne aspekte projekta, uključujući:

1. Osnovne postavke projekta (PropertyGroup)

- Projekat koristi .NET 8
- Omogućava nullable reference tipove, što znači da će kompajler upozoravati ako postoji mogućnost null reference
- Omogućava implicitne ***using*** direktive, što znači da se automatski dodaju uobičajene using direktive.

2. Zavisnosti (NuGet paketi) (ItemGroup -> PackageReference)

BCrypt.Net-Next – Koristi se za hešovanje lozinki.
DotNetEnv – Koristi se za učitavanje promenljivih iz .env fajla, što je korisno za podešavanje API ključeva i tajnih vrednosti.
Microsoft.AspNetCore.Authentication.JwtBearer – Omogućava JWT autentifikaciju u ASP.NET aplikaciji.
Microsoft.AspNetCore.Mvc.NewtonsoftJson – Popularna biblioteka za serijalizaciju i deserializaciju JSON podataka u .NET-u.
Swagger i OpenAPI omogućavaju automatsku dokumentaciju API-ja.
SignalR.Common – Omogućava real-time komunikaciju.
Microsoft.EntityFrameworkCore.Design – Omogućava EF Core alate (npr. dotnet ef komande).
Microsoft.EntityFrameworkCore.SqlServer – Omogućava rad sa SQL Server bazom podataka.
Sentry – Koristi se za error tracking i monitoring aplikacije.
System.Diagnostics.Tools – Omogućava korišćenje alata za dijagnostiku.

3. Reference ka drugim projektima (ItemGroup -> ProjectReference)

Reference ka projektima: Domain, EFDataAccess, Application, Implementation

Ovo je **Clean Architecture** pristup, gde API sloj samo delegira zahteve kroz Application sloj.

API/Program.cs

Koristim stari pristup sa Startup.cs klasom, što je stil ranijih verzija .NET-a (5,6).

Metoda ***Main()*** je ulazna tačka aplikacije.
Poziva ***CreateHostBuilder(string[] args)***, zatim gradi ***Build()*** i pokreće ***Run()*** aplikaciju.

Host.CreateDefaultBuilder(args)
1. Automatski postavlja osnovne servise (npr. logging, konfiguraciju, dependency injection)
2. Koristi **Kestrel** kao web server
3. Učitava konfiguracije iz:
- appsettings.json
- appsettings.{Environment}.json
- Environment promenljivih (Environment Variables)

ConfigureWebHostDefaults(webBuilder => { ... })
1. Postavlja Kestrel server
2. Poziva ***webBuilder.UseStartup<Startup>()***, što znači da se sva konfiguracija aplikacije nalazi u Startup.cs
3. Poziva ***webBuilder.UseSentry(o),*** alat za prikupljanje grešaka i praćenje performansi aplikacije.
- *o.Dsn* – Postavlja **DSN (Data Source Name)**, što omogućava aplikaciji da šalje podatke ka Sentry serveru.
- *o.Debug = true* – Omogućava detaljne logove o greškama (trebalo bi da bude false u produkciji).
- *o.TracesSampleRate = 1.0* – Aktivira 100% praćenje svih transakcija (može se smanjiti na 0.1 u produkciji).

API/.env
SMTP_SERVER – Definiše server koji se koristi za slanje emailova (u ovom slučaju, Gmail SMTP).
SMTP_PORT– Koristi port 587 koji je standardan za SMTP sa TLS enkripcijom.
SENDER_EMAIL – Email adresa sa koje se šalju emailovi.
SENDER_PASSWORD – **App Password** generisan za ovu email adresu.

JWT_SECRET_KEY – Tajni ključ za potpisivanje JWT tokena.
Mora biti:
1. 32 karaktera dug (najmanje 256 bita).
2. Sigurno čuvan jer omogućava validaciju korisničkih tokena.
JWT_ISSUER – Ko izdaje token (tvoj API projekat).
JWT_AUDIENCE – Ko može koristiti token (može biti specifična aplikacija ili "Any").
JWT_EXPIRY_MINUTES – Token ističe posle **120 minuta**.

.env fajl je postavljen u **.gitignore**!

API/Startup.cs

using **Microsoft.OpenApi.Models** – Omogućava integraciju Swagger-a za generisanje API dokumentacije.
using **DotNetEnv** – Učitava .env fajl za rad sa okruženjem.

Startup(IConfiguration configuration):
IConfiguration je ugrađeni ASP.NET Core servis koji čita konfiguracione vrednosti (appsettings.json, .env).

Env.Load();
Učitava .env fajl koristeći DotNetEnv.
Ovim se omogućava rad sa okruženjem bez hardkodovanih vrednosti.

_configuration = configuration;
Konfiguracija se prosleđuje kroz Startup.cs klasu, omogućavajući pristup u celoj aplikaciji.

ConfigureServices(IServiceCollection services):
-Kreiraju se objekti *jwtSettings* i *emailSettings* na osnovu JWTSettings.cs i EmailSettings.cs klasa.

_configuration.Bind(nameof(JWTSettings), jwtSettings);
Ovim ASP.NET Core automatski popunjava vrednosti iz konfiguracije (.env)

-Registrujemo ih kao **Singleton** servise – Ovi objekti su dostupni kroz ceo životni ciklus aplikacije i ne menjaju se nakon pokretanja.

-*services.AddControllers()* – Registruje **MVC** pattern, omogućavajući API kontrolerima da obrađuju HTTP zahteve.

-*services.AddDbContext<BlogContext>();*
Registruje Entity Framework DbContext za rad sa bazom.

-*JWTService*
Koristi **Scoped** životni vek (kreira se novi objekat po HTTP zahtevu).
Uzima JWT podešavanja iz services i kreira instancu JWTService.

-*OAuthService* se dodaje kao Scoped servis.

-*services.LoadUseCases();*
Poziva se ekstenzija nad IServiceCollection.
Registruje automatski sve use-case klase u aplikaciji.

-*services.AddScoped<IUseCaseLogger, EFDatabaseLogger>();*
Registruje EFDatabaseLogger kao implementaciju interfejsa IUseCaseLogger, ali sa Scoped životnim vekom. To znači da se kreira nova instanca EFDatabaseLogger za svaki HTTP zahtev. To je idealno za logovanje jer omogućava praćenje jednog korisničkog zahteva bez curenja podataka između različitih zahteva.

-*services.AddHttpContextAccessor();*
Ovo 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.

-*services.AddApplicationActor();*
Registruje trenutnog korisnika.
Omogućava da aplikacija zna koji korisnik je trenutno prijavljen koristeći JWT.

-*services.AddJWT(jwtSettings);*
Registruje ekstenziju koja konfiguriše JWT autentifikaciju.

-*services.AddSignalR();*
Omogućava rad sa real-time komunikacijom u aplikaciji.

-*services.AddSwaggerGen();*
Generiše automatsku dokumentaciju za API.
Definiše JWT autentifikaciju u Swagger UI, omogućavajući unos tokena u **Authorization** header-u.

-*services.AddTransient<IEmailSender, SMTPEmailSender>();*
Registruje servis za slanje e-mailova koristeći SMTP protokol.

-*services.AddCors();*
1. Ova politika dozvoljava samo http://localhost:5173
2. Dozvoljava sve HTTP metode i zaglavlja.
3. Omogućava slanje kolačića i autentifikacije (*AllowCredentials()*).

Metoda ***Configure()*** u Startup.cs klasi se koristi za konfigurisanje HTTP request **pipeline-a** u ASP.NET Core aplikaciji.

Prima dva parametra:
IApplicationBuilder app – Interfejs koji omogućava dodavanje middleware komponenti u HTTP request pipeline. Sve što se doda kroz app.UseMiddleware<T>() ili app.UseXYZ() postaje deo obrade zahteva.

IWebHostEnvironment env – Sadrži informacije o okruženju (Development, Staging, Production). Može se koristiti za prilagođavanje ponašanja aplikacije na osnovu okruženja u kojem se pokreće.

-Prvo se omogućava da u **razvojnom** okruženju prikazujem detaljne podatke o greškama kada dođe do izuzetaka u aplikaciji.

-*app.UseDefaultFiles()*
Middleware koji omogućava da se podrazumevano posluže fajlovi kao što su index.html bez eksplicitnog navođednja u URL-u.

-*app.UseStaticFiles();*
Omogućava posluživanje statičkih fajlova (CSS, JS, slike, PDF-ovi itd.) iz wwwroot foldera.

-*app.UseRouting();*
Ovaj middleware omogućava ASP.NET Core aplikaciji da obradi URL zahteve i pronađe odgovarajući kontroler ili endpoint koji će ih obraditi.

-*app.UseCors("AllowSpecificOrigin");*
Ovaj middleware omogućava **Cross-Origin Resource Sharing (CORS)**, što znači da omogućava ili ograničava koje druge aplikacije (domeni) mogu slati HTTP zahteve mojoj API aplikaciji.
Po defaultu, browser-i blokiraju zahteve koji dolaze sa drugog domena (CORS politika). Ako frontend radi na http://localhost:5173, a API na http://localhost:5207, browser će blokirati zahteve osim ako ne dozvolim CORS.

Redosled middleware-a je bitan!
- *app.UseRouting();* mora doći **pre** app.UseAuthorization(); i app.UseEndpoints();
- *app.UseCors();* mora biti **između** app.UseRouting(); i app.UseAuthorization(); kako bi se primenile CORS politike pre nego što se izvrše pravila autentifikacije i autorizacije.
- *app.UseAuthentication();* mora biti **pre** *app.UseAuthorization();* u pipeline-u, jer autorizacija zavisi od korisnika koji je već autentifikovan.

-*app.UseAuthentication();*
Ovaj middleware proverava da li zahtev sadrži validne kredencijale (npr. JWT, cookie, API ključ, itd.) i postavlja korisnika (HttpContext.User) ako je autentifikacija uspešna.

-*app.UseAuthorization();*
Ovaj middleware proverava da li korisnik ima prava pristupa određenom API resursu. Koristi **role-based** ([Authorize(Roles = "Admin")]) i **policy-based** ([Authorize(Policy = "RequireAdmin")]) autorizaciju.

Ovo sam napisao, jer je bitno da znam za ovu opciju, ali u projektu koristim CQRS pristup sa use-case-ovima za autorizaciju, što znači da logiku implementiram unutar samih use-case-ova, umesto da koristim **[Authorize]** atribute i app.UseAuthorization();.

-*app.UseSwagger();*
Registruje **Swagger** middleware, koji generiše OpenAPI specifikaciju (JSON dokument) za API.
Swagger automatski prepoznaje sve kontrolere i rute u aplikaciji i kreira dokumentaciju u JSON formatu, koja opisuje dostupne API endpointove, HTTP metode, parametre, odgovore itd.

-*app.UseSwaggerUI();*
Dodaje **Swagger UI**, vizuelni interfejs koji omogućava pregled API dokumentacije u pretraživaču i testiranje API endpoint-ova direktno kroz web aplikaciju.

-*app.UseMiddleware<GlobalExceptionHandler>();*
Registruje GlobalExceptionHandler.cs middleware, koji presreće i obrađuje sve izuzetke u aplikaciji.
Ako neka metoda baci izuzetak koji nije obrađen unutar koda (try-catch blok), ovaj middleware će ga uhvatiti i vratiti odgovarajući HTTP odgovor klijentu.

-*app.UseSentryTracing();*
Aktivira **Sentry tracing**, alat za praćenje performansi i logovanje grešaka.
Kada aplikacija naiđe na grešku ili spor odziv, Sentry prikuplja podatke i šalje ih na svoj server za analizu.

-*app.UseEndpoints(endpoints => {…})*
Ovo određuje kako će aplikacija presretati HTTP zahteve i usmeravati ih ka odgovarajućim kontrolerima i SignalR hub-u.

endpoints.MapControllers();
Mapira sve API rute koje su definisane u kontrolerima (npr. GET /api/posts će se proslediti PostsController klasi).

endpoints.MapHub<NotificationHub>("/api/notificationsHub");
Ovo omogućava WebSocket komunikaciju putem SignalR-a na ruti

/api/notificationsHub.
Klijenti mogu da se povežu na NotificationHub i primaju real-time obaveštenja.

Services

API/Services/OauthService.cs

OAuthService.cs klasa implementira autentifikaciju korisnika putem OAuth-a (Google-om).
Servis proverava da li korisnik postoji u bazi, a ako ne postoji, registruje ga automatski i generiše JWT.

Metoda *AuthenticateUser()*
1. Traži korisnika u bazi sa istim *Email* i provera da li je aktivan.
2. Ako korisnik postoji, kreira mu JWT i vraća ga.
3. Ako korisnik ne postoji, registruje ga pozivanjem *RegisterUser()* metode.

Metoda *RegisterUser()*
1. Parsira ime korisnika (prvo ime i prezime).
2. Dohvata ulogu korisnika iz baze.
3. Kreira novog korisnika, dodeljuje mu:

- Nasumično generisanu lozinku (jer OAuth ne koristi standardnu lozinku).
- Automatski generisan Username.
- Sliku profila preuzetu iz OAuth podataka.
- Ulogu dodeljenu iz baze.

4. Dodaje korisnika u bazu i poziva SaveChangesAsync().
5. Ažurira UseCases korisnika (verovatno dodaje dozvole za ulogu).
6. Generiše JWT i vraća ga.

API/Services/JWTService.cs

JWTService prima *issuer* (izdavač tokena) i *secretKey* (tajni ključ za potpisivanje).

Metoda ***GenerateClaims(User user)***
Kreira novi objekat tipa **JWTActor** i popunjava ga podacima koji dolaze iz objekat *user.*
JWTActor je klasa koja implementira IApplicationActor i koristi se da predstavi podatke o korisniku u JWT-u.

Objekat *actor* se koristi da bi se serializovao u JSON i dodao kao claim (**ActorData**) u JWT.
Ovaj claim sadrži sve podatke o korisniku, uključujući ID, ime, email, itd.

Omogućava aplikaciji da na osnovu JWT-a dobije sve informacije o korisniku.

Kreira listu **claim-ova** (atributa tokena) za korisnika.
Claim-ovi sadrže:
1. Jti (jedinstveni identifikator tokena).
2. Iss (issuer, tj. ko je izdao token).
3. Iat (vreme izdavanja tokena).
4. Korisničke podatke kao custom claim-ove, uključujući JSON-serializovan objekat *ActorData*.

Metoda ***GenerateToken(IEnumerable<Claim> claims)***
1. Koristi ***SymmetricSecurityKey*** i ***HmacSha256*** algoritam za potpisivanje.
2. Podešava:

- Izdavača (*issuer*)
- Publiku (*audience*)
- Vreme važenja tokena (120 min)

3. Kreira i vraća JWT kao string.

API Core

API/Core/AnonymousActor.cs

Ova klasa implmentira interfejs IApplicationActor.cs i predstavlja korisnika koji nije autentifikovan.
AllowedUseCases sadrži listu ID-eva dozvoljenih use-case-ova za anonimne korisnike (akcije poput pregledanja javnog sadržaja).

API/Core/APIExtension.cs

Ovaj statički fajl sadrži proširenja (extension methods) za IServiceCollection, čime se pojednostavljuje konfiguracija aplikacije. Njegova svrha je da centralizuje i olakša registraciju servisa u ASP.NET Dependency Injection (DI) kontejneru.
Ovo je korisno jer odvaja konfiguraciju od Startup.cs – Umesto da glavni fajl postane pretrpan registracijom servisa, sve je lepo organizovano u ovom fajlu.

Metoda ***LoadUseCases(this IServiceCollection services)***
Ova metoda registruje sve komande, upite, servise, repozitorijume i validatore u DI kontejner.
Ovo je korisno zbog CQRS pristupa, aplikaciju ima jasnu separaciju između komandi i upita, a ovde sve registrujemo na jednom mestu.

Metoda ***AddApplicationActor(this IServiceCollection services)***
Ova metoda registruje trenutnog korisnika koji pravi HTTP zahtev.

1. Dobavlja **IHttpContextAccessor** – Ovo omogućava pristup trenutnom HTTP zahtevu.
2. Izvlači korisničke podatke iz JWT (polje "ActorData").
3. Ako korisnik nije prijavljen, vraća anonimnog korisnika (AnonymousActor).
4. Ako jeste prijavljen, deserializuje JWT podatke u objekat JWTActor.

Omogućava aplikaciji da zna ko je korisnik koji pravi zahtev i da na osnovu njegovih privilegija odredi šta može da radi.

Metoda ***AddJWT(this IServiceCollection services, JWTSettings jwtSettings)***
Ova metoda konfiguriše JWT autentifikaciju.
Dodaje JWTManager.
Postavlja **JwtBearer** autentifikaciju:
1. Proverava issuer, audience i tajni ključ
2. Omogućava validaciju roka trajanja tokena

JWT autentifikacija omogućava sigurno upravljanje korisničkim sesijama bez potrebe za čuvanjem sesija na serveru.

API/Core/JWTActor.cs
Ova klasa predstavlja korisnika koji je autentifikovan pomoću JWT i implementira IApplicationActor.cs interfejs.
Koristi se za čuvanje podataka o trenutno prijavljenom korisniku i njegovim dozvolama unutar aplikacije.

Nakon uspešne autentifikacije, backend generiše JWT.
Tokom ovog procesa podaci iz baze se mapiraju u JWTActor objekat.

Nakon što se JWTActor kreira, on se koristi za generisanje JWT, koji se kasnije šalje klijentu (frontend-u).

Kada korisnik šalje zahtev (npr. GET /api/posts), server uzima JWT token iz Authorization headera, dešifruje ga i dobija podatke o korisniku.

1. JWTActor klasa služi za čuvanje podataka o trenutno prijavljenom korisniku.
2. Implementira IApplicationActor interfejs, što omogućava lako korišćenje u autorizaciji.
3. Nakon autentifikacije, koristi se za kreiranje JWT tokena.
4. Kada korisnik pošalje zahtev, JWT se dešifruje i vraća JWTActor, čime sistem zna ko je trenutno prijavljen i koje akcije može da izvrši.

U kontrolerima mogu pristupiti trenutnom korisniku ovako:
var actor = (JWTActor)HttpContext.Items["User"];

API/Core/JWTManager.cs

Ova klasa pruža sloj apstrakcije između LoginController.cs i JWTService.cs
Ubacuju se zavisnosti *_context* i *_jwtService* kroz konstruktor koristeći Dependency Injection.
BlogContext: Omogućava pristup podacima u bazi koristeći Entity Framework Core.
JWTService: Servis koji generiše JWT i priprema claims (tvrdnje) koje će biti uključene u token.

Metoda ***MakeToken(string username, string password)***
Ova metoda uzima korisničko ime i lozinku, poziva *FetchUser()* metodu, a zatim generiše JWT koristeći JWTService.
U suštini, metoda delegira odgovornost za validaciju korisničkih podataka i kreiranje tokena na druge objekte (FetchUser i JWTService).

Metoda ***FetchUser(string username, string password)***
Ova metoda koristi EF Core da pronađe korisnika u bazi podataka i proverava da li lozinka odgovara. Ako korisnik nije pronađen ili lozinka nije validna, baca izuzetak.

1. JWTManager je centralna klasa za autentifikaciju u aplikaciji.
2. Koristi EF Core za rad sa bazom i **BCrypt** za proveru lozinke pošto je loznika iz baze hašovana.
3. Koristi JWTService za generisanje tokena.

API/Core/JWTSettings.cs

Ova klasa sadrži konfiguracione vrednosti za rad sa JWT, koje se koriste u autentifikaciji korisnika putem tokena. Parametri koji se definišu u ovoj klasi su:

1. **JwtIssuer**: Entitet koji izdaje JWT (obično API server ili autentifikacija).
2. **JwtAudience**: Ciljana publika za koju je token namenjen (npr. klijentske aplikacije).
3. **JwtSecretKey**: Tajni ključ koji se koristi za potpisivanje i verifikaciju JWT-a. Mora biti sigurno čuvan.
4. **TokenExpiryMinutes**: Vreme trajanja tokena u minutama (koliko dugo je token validan).

Konstruktor klase čita ove vrednosti iz okruženja (environment variables). Ako neka od vrednosti nedostaje, baca se greška (ArgumentNullException).

API/Core/GlobalExceptionHandler.cs

Ovo je globalni **middleware** za hvatanje i obradu izuzetaka u ASP.NET aplikaciji.
On omogućava centralizovano rukovanje greškama umesto da se exception-i hvataju pojedinačno u svakom kontroleru ili servisu.

1. Šta radi GlobalExceptionHandler?

- Middleware se postavlja između HTTP zahteva i odgovora.
- Ako se u nekoj tački pipeline-a dogodi izuzetak (exception), ovaj middleware ga hvata i obrađuje.
- Loguje grešku ako nije eksplicitno definisan način njenog rukovanja.
- Vraća odgovarajući HTTP statusni kod i JSON odgovor sa porukom o grešci.

_next je **delegat** koji omogućava nastavak izvršenja sledećeg middleware-a u pipeline-u.
_logger je interfejs za logovanje, koji omogućava upisivanje error logova.

Metoda ***Invoke(HttpContext httpContext)***
Ova metoda obrađuje HTTP zahteve i presreće izuzetke:
1. Pokušava da obradi zahtev
2. Ako dođe do greške, postavlja *ContentType* odgovora na "application/json"
3. Postavlja *statusCode* na 500 Internal Server Error kao podrazumevanu vrednost
4. Postavlja generičku poruku greške

Hvatanje specifičnih izuzetaka
Svaka greška se mapira na odgovarajući HTTP status kod:
1. Već dodat entitet (**AlreadyAddedException**)
Ako korisnik pokuša da doda već postojeći entitet, vraća **400 Bad Request**
2. Zabranjen pristup (**UnauthorizedUseCaseException**)
Ako korisnik nema dozvolu za određen use-case (kada pokušava da obriše komentar, ali nema pravo da koristi ovu funkcionalnost, bez obzira na to čiji je komentar), vraća **403 Forbidden**
3. Neautorizovan korisnik (**AuthenticationException**)
Ako korisnik nije autentifikovan, vraća **401 Unauthorized**
4. Entitet nije pronađen (**EntityNotFoundException**)
Ako neki entitet (npr. korisnik, post, komentar) ne postoji u bazi, vraća **404 Not Found**
5. Validacione greške (**ValidationException**)
Ako model nije validan (npr. polje je prazno, lozinka prekratka), vraća **422 Unprocessable Entity**
JSON odgovor sadrži listu svih grešaka.
_validator.ValidateAndThrow(request); FluentValidation proverava podatke u request-u i generiše listu grešaka, ako podaci nisu validni.
Kada koristim ValidateAndThrow(request), FluentValidation interno proverava pravila i baca baš ValidationException ako nešto nije ispravno.
6. Konflikt (**ConflictException**)
Ako dođe do konflikta (npr. pokušaj dupliranja podataka), vraća **409 Conflict**
7. Zabranjen pristup za određenog korisnika (**UnauthorizedUserAccessException**)
Ako korisnik ima pristup use-case-u, ali nema prava nad konkretnim podacima (npr. ako

pokuša da obriše tuđi komentar, iako ima pravo da briše komentare), vraća **403 Forbidden**
8. Generički (nepoznati) izuzeci
Ako izuzetak nije predviđen, loguje ga i vraća **500 Internal Server Error**

Postavljanje odgovora:
Ako nije eksplicitno definisan response, koristi se podrazumevani
Odgovor se **serijalizuje** u JSON i šalje klijentu.

API/Core/NotificationHub.cs

Ova klasa predstavlja SignalR hub koji omogućava real-time komunikaciju između servera i klijenta. Svaki korisnik koji se poveže na ovaj hub može primati obaveštenja i izvršavati određene operacije, poput pridruživanja grupi ili označavanja obaveštenja kao pročitanih.

Metoda ***JoinGroup(string idUser)***
Metoda JoinGroup u klasi NotificationHub služi za dodavanje korisnika u SignalR grupu zasnovanu na njegovom idUser.

Kako funkcioniše?

1. Korisnik se povezuje na SignalR hub:
   Kada se korisnik prijavi i otvori aplikaciju, SignalR uspostavlja vezu (OnConnectedAsync() se poziva).

2. Klijent poziva JoinGroup(idUser):
   Nakon uspostavljanja konekcije, frontend poziva JoinGroup, šaljući korisnikov ID.

3. Dodavanje u grupu:
   Na backendu, JoinGroup poziva:
   await Groups.AddToGroupAsync(Context.ConnectionId, idUser);

   - *Context.ConnectionId* je jedinstveni ID konekcije korisnika.
   - *idUser* je jedinstveni ID korisnika koji služi kao naziv grupe.

To znači da svi korisnici sa istim idUser ID-jem (ako se prijave sa više uređaja) postaju deo iste SignalR grupe.

4. Slanje poruka grupi:
   Kasnije, kada backend želi poslati obaveštenje samo tom korisniku, može koristiti: *await Clients.Group(idUser.ToString()).SendAsync("ReceiveNotification", notificationData);*

Ovim osiguravam da samo taj korisnik dobije obaveštenje, umesto da se notifikacija šalje svima. Takođe, ako korisnik koristi više uređaja, svi uređaji će biti deo iste grupe i sihronizovani.

Metoda ***OnConnectedAsync()***
1. Ova metoda se poziva automatski kada se korisnik poveže na SignalR hub.
2. Pokušava da pronađe *IdUser* iz JWT => *Context.User?.FindFirst("IdUser")?.Value*
3. Štampa poruku u konzoli da prikaže korisnikov ID i njegov ***ConnectionId*** (jedinstveni identifikator konekcije).
4. Poziva *base.OnConnectedAsync();* da osigura ispravno povezivanje korisnika.

Metoda ***OnDisconnectedAsync(Exception exception)***
Ova metoda se poziva kada korisnik izgubi konekciju sa hub-om.
Proverava da li korisnik ima *IdUser* i uklanja njegovu konekciju iz odgovarajuće grupe.
Poziva *base.OnDisconnectedAsync(exception);* da osigura ispravno prekidanje veze.

Metoda ***MarkAllAsRead(int idUser)***
Ova metoda emituje event **"***NotificationsMarkedAsRead***"** svim korisnicima sa istom *idUser* grupom.
Kada se pozove, frontend reaguje tako što ažurira UI i obeležava notifikacije.

API/Core/SignalRNotificationHub.cs
Ova klasa ne nasleđuje Hub, već koristi ***IHubContext<NotificationHub>*** da bi omogućila slanje notifikacija izvan hub-a.

- SignalR hub funkcioniše samo kada je korisnik povezan.
- Ponekad API treba da pošalje notifikaciju korisniku čak i kada on nije na SignalR konekciji.
- IHubContext<NotificationHub> omogućava bilo kom servisu u aplikaciji da komunicira sa korisnicima koji su trenutno povezani na SignalR.

Metoda ***SendNotificationToUser(int idUser, object notification)***
Koristi _hubContext da pošalje poruku korisniku bez potrebe da on eksplicitno pozove SignalR hub.

_hubContext.Clients.User(idUser.ToString()).SendAsync("ReceiveNotification", notification);
Ovim se poruka direktno šalje korisniku, bez obzira da li se on trenutno nalazi na hub-u

ili ne.

NotificationHub

- Direktno komunicira sa povezanim korisnicima.
- Omogućava grupisanje korisnika po *idUser*.
- Rukuje povezivanjem i prekidanjem konekcije korisnika.

SignalRNotificationHub

- Koristi *IHubContext<NotificationHub>* da omogući slanje notifikacija van SignalR huba.
- Omogućava slanje notifikacija bez obzira da li je korisnik trenutno povezan ili ne.

Controllers

Pregled kontrolera u API-ju
Svi kontroleri u ovom API-ju prate CQRS (Command-Query Responsibility Segregation) princip, koristeći UseCaseExecutor za izvršavanje komandi (command) i upita (query).

Ključne karakteristike svih kontrolera

1. **Zavisni su od UseCaseExecutor servisa** – koristi se za pokretanje komandi i upita na osnovu Use Case interfejsa.
2. **Razdvajanje logike pomoću Use Case interfejsa** – svaki entitet (postovi, komentari itd.) ima zasebne komande za kreiranje, ažuriranje i brisanje.
3. **Podržavaju osnovne CRUD operacije**, ali i dodatne operacije specifične za pojedinačne entitete (npr. personalizovane akcije za korisnike).
4. **Korišćenje atributa [FromBody], [FromQuery] i [FromServices]** – omogućavaju pravilno mapiranje podataka iz HTTP zahteva, koristeći Dependency Injection.

Atribut	**Izvor podataka**	**Kada koristiti?**
[FromBody]	Telo HTTP zahteva (JSON, XML)	Kada radim sa složeni objektima (DTO) i šaljem podatke u POST, PUT, PATCH.

Atribut	Izvor podataka	Kada koristiti?
[FromQuery]	Query string parametri (URL)	Kada filtriram ili pretražujem podatke (GET /api/posts?title=test&page=2).
[FromServices]	Dependency Injection servis	Kada API metodi treba instanca servisa, repozitorijuma ili komande.

API/Controllers/PostsController.cs

📌 **Endpoint:** api/posts

- **Kreiranje posta** (POST)
  Prima DTO (UpsertPostDto) iz tela HTTP zahteva.
  Koristi ICreatePostCommand za izvršenje.
  Vraća 201 Created nakon uspešnog kreiranja.
- **Dohvatanje liste postova sa pretragom** (GET)
  Omogućava pretragu i filtriranje postova (PostSearch).
  Poziva IGetPostsQuery za dohvatanje podataka.
  Vraća 200 OK sa listom postova.
- **Dohvatanje pojedinačnog posta** (GET {id})
  Prima id posta iz URL-a.
  Koristi IGetPostQuery da dohvati podatke.
  Vraća 200 OK sa traženim postom.
- **Ažuriranje posta** (PUT {id})
  Postavlja id iz URL-a u DTO objekat.
  Poziva IUpdatePostCommand za ažuriranje.
  Vraća 204 No Content nakon uspešnog ažuriranja.
- **Brisanje posta** (DELETE {id})
  Briše post na osnovu id parametra.
  Koristi IDeletePostCommand.
  Vraća 204 No Content ako je uspešno.
- **Ažuriranje ličnog posta** (PUT {id}/personal)
  Posebna putanja za ažuriranje ličnih postova.
  Koristi IUpdatePersonalPostCommand.
  Vraća 204 No Content.

- **Brisanje ličnog posta** (DELETE {id}/personal)
  Briše lični post pomoću IDeletePersonalPostCommand.
  Vraća 204 No Content.

API/Controllers/CommentsController.cs

📌 **Endpoint:** api/comments

- **Kreiranje komentara** (POST)
  Podaci o komentaru UpsertCommentDto dolaze iz tela HTTP zahteva (JSON).
  Komanda za kreiranje komentara, dobavlja se iz DI kontejnera.
  Vraća 201 Created nakon uspešnog kreiranja.

- **Dohvatanje komentara** (GET)
  Omogućava pretragu parametrima iz URL-a (CommentSearch).
  Poziva IGetCommentsQuery za dohvatanje podataka iz baze.
  Vraća 200 OK sa listom komentara.

- **Dohvatanje jednog komentara** (GET {id})
  Očekuje ID komentara u URL-u
  Koristi IGetCommentQuery za dohvatanje jednog komentara.
  Vraća 200 OK sa detaljima komentara.

- **Ažuriranje komentara** (PUT {id})
  Telo zahteva sadrži nove podatke komentara (UpsertCommentDto).
  dtoRequest.Id = id; Osigurava da ID u DTO-u odgovara putanji.
  dtoRequest.IdUser = _actor.Id; – Postavlja ID trenutnog korisnika koji ažurira komentar.
  Poziva IUpdatePersonalCommentCommand za ažuriranje.
  Vraća 204 No Content nakon uspešnog ažuriranja.

- **Brisanje komentara** (DELETE {id})
  Briše komentar na osnovu ID parametra.
  Koristi IDeleteCommentCommand koja se dobavlja iz DI kontejnera.
  Vraća 204 No Content ako je uspešno brisanje.

- **Brisanje ličnog komentara** (DELETE {id}/personal)
  Briše lični komentar pomoću IDeletePersonalCommentCommand.
  Vraća 204 No Content ako je uspešno brisanje.

API/Controllers/CategoriesController.cs

📌 **Endpoint:** api/categories

- **Kreiranje kategorije** (POST)
  Podaci o kategoriji UpsertCategoryDto dolaze iz tela HTTP zahteva (JSON).
  Komanda za kreiranje kategorije, dobavlja se iz DI kontejnera.
  Vraća 201 Created nakon uspešnog kreiranja.

- **Dohvatanje kategorije** (GET)
  Omogućava pretragu parametrima iz URL-a (CategorySearch).
  Poziva IGetCategoriesQuery za dohvatanje podataka iz baze.
  Vraća 200 OK sa listom kategorija.

- **Dohvatanje jedne kategorije** (GET {id})
  Dohvata kategoriju sa opcijom paginacije.
  Očekuje ID kategorije u URL-u.
  Definisani su podrazumevani parametri za paginaciju iz URL query parametra *page = 1* i *perPage = 3.*
  Pravi novi objekat CategorySearch koji sadrži podatke o pretrazi.
  Koristi IGetCategoryQuery za dohvatanje jedne kategorije.
  Vraća 200 OK sa pronađenim podacima.

- **Ažuriranje kategorije** (PUT {id})
  Telo zahteva sadrži nove podatke kategorije (UpsertCategoryDto).
  dtoRequest.Id = id; Osigurava da ID u DTO-u odgovara putanji.
  Poziva IUpdateCategoryCommand za ažuriranje.
  Vraća 204 No Content nakon uspešnog ažuriranja.

- **Brisanje kategorije** (DELETE {id})
  Briše komentar na osnovu ID parametra.
  Koristi IDeleteCategoryCommand koji se dobavlja iz DI kontejnera.
  Vraća 204 No Content ako je uspešno brisanje.

API/Controllers/AuthorRequestsController.cs

📌 **Endpoint:** api/authorrequests

- **Podnošenje zahteva za autorstvo** (POST)
  Klijent štalje zahtev sa podacima u telu zahteva.
  Automtaski se postavlja *IdUser = _actor.Id,* što znači da korisnik podnosi zahtev za sebe.
  UseCaseExecutor izvršava ICreateAuthorRequestCommand, koji verovatno čuva zahtev u bazi.
  Vraća 201 Created nakon uspešnog kreiranja.

- **Pregled svih zahteva** (GET)
  Klijent šalje GET /api/authorrequests.

Može koristiti query parametre (status=Pending).
IGetAuthorRequestsQuery izvršava pretragu i vraća rezultate.
Vraća 200 OK sa listom zahteva.

- **Prihvatanje zahteva** (PUT accept)
  Administrator poziva PUT na /api/authorrequests/accept?id=5 i šalje telo zahteva.
  IdRole = 2 označava da korisnik sada postaje Autor.
  IUpdateAuthorRequestCommand ažurira zahtev u bazi.
  Vraća 204 No Content ako je uspešno ažuriranje.

- **Odbijanje zahteva** (PUT reject)
  Administrator poziva PUT na /api/authorrequests/reject?id=5 i šalje telo zahteva.
  IdRole = 3 označava da je zahtev odbijen.
  Vraća 204 No Content nakon uspešnog ažuriranja.

API/Controllers/FollowersController.cs

📌 **Endpoint:** api/followers

- **Dodavanje pratioca** (POST)
  Klijent (korisnik) šalje POST /api/followers sa podacima u telu zahteva.
  Automatski se postavlja *IdUser* = *_actor.Id*, što znači da korisnik započinje praćenje drugog korisnika.
  UseCaseExecutor izvršava IFollowCommand, koji dodaje vezu u bazu.
  Vraća 201 Created nakon uspešnog kreiranja.

- **Dohvatanje liste pratilaca korisnika** (GET {id}/followers)
  Klijent šalje GET /api/followers/{id}/followers gde {id} označava ID korisnika čije pratioce želimo videti.
  FollowSearch može sadržati filtere (paginaciju).
  Vraća 200 OK sa listom pratilaca

- **Dohvatanje liste korisnika koje neko prati** (GET {id}/following)
  Klijent šalje GET /api/followers/{id}/following gde {id} označava ID korisnika čiju listu praćenih korisnika želimo videti.
  Vraća 200 OK sa listom korisnika koje {id} korisnik prati.

- **Otpraćivanje korisnika** (DELETE {id}/unfollow)
  Klijent šalje DELETE /api/followers/{id}/unfollow gde {id} označava ID korisnika koga otpraćujemo.
  IUnfollowCommand briše zapis o praćenju iz baze.
  Vraća 204 No Content nakon uspešnog brisanja.

- **Provera da li korisnik prati drugog korisnika** (GET {id}/check)
  Klijent šalje GET /api/followers/{id}/check gde {id} označava ID korisnika kojeg proveravamo.
  ICheckFollowStatusQuery vraća true ili false u zavisnosti od toga da li trenutni korisnik prati {id}.
  Vraća 200 OK sa rezultatom (true ili false).

API/Controllers/LikesController.cs

📌 **Endpoint:** api/likes

- **Lajkovanje posta** (POST posts/{id})
  Klijent šalje POST /api/likes/posts/{id} gde {id} označava ID posta koji korisnik lajkuje.
  LikeDto sadrži podatke o lajkovanju (npr. korisnik koji lajkuje).
  IdUser se automatski postavlja na ID trenutno prijavljenog korisnika (_actor.Id).
  UseCaseExecutor izvršava ILikePostCommand, koji verovatno dodaje lajk u bazu.
  Vraća 200 OK sa podacima o lajkovanju.

- **Lajkovanje komentara** (POST comments/{id})
  Klijent šalje POST /api/likes/comments/{id} gde {id} označava ID komentara koji korisnik lajkuje.
  LikeDto sadrži informacije o lajku.
  UseCaseExecutor izvršava ILikeCommentCommand, koji verovatno dodaje lajk na komentar u bazu.
  Vraća 200 OK sa podacima o lajkovanju.

- **Brisanje lajka sa posta** (DELETE posts/{id})
  Prima id posta koji korisnik želi da unlajkuje.
  Kreira LikeDto sa ID-em trenutno prijavljenog korisnika (_actor.Id) i ID-em posta.
  Izvršava IUnlikePostCommand, koji briše lajk iz baze.
  Vraća 204 No Content nakon uspešnog brisanja.

- **Brisanje lajka sa komentara** (DELETE comments/{id})
  Prima id komentara koji korisnik želi da unajkuje.
  Kreira LikeDto sa ID-em trenutno prijavljenog korisnika (_actor.Id) i ID-em komentara.
  Izvršava IUnlikeCommentCommand, koji briše lajk iz baze.
  Vraća 204 No Content nakon uspešnog brisanja.

API/Controllers/LoginController.cs

📌 **Endpoint:** api/login

Ovaj kontroler omogućava korisnicima da se prijave koristeći korisničko ime i lozinku.
Ako su kredencijali ispravni, API generiše JWT.
JWTManager je servis koji generiše JWT na osnovu korisničkog imena i lozinke.
Konstruktor prima instancu JWTManager preko Dependency Injection (DI).

- **Logovanje korisnika** (POST)
  Prima LoginUserDto, koji sadrži korisničko ime i lozinku.
  Poziva _manager.MakeToken(), koji generiše JWT token.
  Ako su podaci tačni, vraća { token } kao odgovor (200 OK).
  Ako su podaci netačni, vraća 401 Unauthorized.

API/Controllers/OAuthController.cs

📌 **Endpoint:** api/auth

OAuthService je servis koji obavlja autentifikaciju korisnika putem Google OAuth-a.
Koristi Dependency Injection (DI) za instanciranje OAuthService.

- **Google logovanje** (POST)
  Prima OAuthUserDto, koji sadrži korisničke podatke.
  Poziva _authService.AuthenticateUser(), koji proverava Google token i vraća JWT ako je autentifikacija uspešna.
  Ako ne uspe, vraća 400 Bad Request ("Authentication failed").
  Ako dođe do nepredviđene greške, vraća 500 Internal Server Error.

API/Controllers/RegisterController.cs

📌 **Endpoint:** api/register

UseCaseExecutor je servis koji izvršava IRegisterUserCommand komandu (CQRS pristup).
Konstruktor koristi Dependency Injection (DI) za ubacivanje UseCaseExecutor.

- **Registracija korisnika** (POST)
  Prima RegisterUserDto, koji sadrži podatke za registraciju.
  Poziva _executor.ExecuteCommand(command, dtoRequest), koji pokreće IRegisterUserCommand za registraciju.
  Ako je registracija uspešna, vraća 201 Created.

API/Controllers/NotificationsController.cs

📌 **Endpoint:** api/notifications

- **Dodavanje nove notifikacije** (POST)
  Prima podatke (InsertNotificationDto) iz tela HTTP zahteva.
  Postavlja FromIdUser na ID trenutno prijavljenog korisnika (koji šalje notifikaciju).

Izvršava komandu ICreateNotificationCommand putem UseCaseExecutor-a.
Vraća 201 Created nakon uspešnog kreiranja.

- **Dohvatanje liste notifikacija** (GET)
  Prima parametre pretrage (NotificationsSearch) iz query stringa.
  Postavlja ID korisnika u pretragu (da korisnik može videti samo svoje notifikacije).
  Pokreće upit IGetNotificationsQuery da dobije podatke.
  Vraća listu notifikacija u 200 OK odgovoru.

- **Označavanje svih notifikacija kao pročitanih** (PATCH mark-as-read)
  Poziva komandu IMarkAllNotificationsAsReadCommand da označi sve notifikacije kao pročitane za trenutno prijavljenog korisnika (_actor.Id).
  Ne očekuje podatke u telu zahteva – samo izvršava komandu.
  Vraća 200 OK kao potvrdu.

API/Controllers/UseCaseLogsController.cs

📌 **Endpoint:** api/usecaselogs

- **Dohvatanje logova** (GET)
  Prima parametre pretrage (UseCaseLogSearch) iz query stringa.
  Pokreće upit IGetUseCaseLogsQuery da dobije podatke iz baze.
  Vraća listu logova u 200 OK odgovoru.

API/Controllers/UsersController.cs

📌 **Endpoint:** api/users

- **Dohvatanje liste korisnika** (GET)
  Prima parametre pretrage iz query stringa (UserSearch).
  Poziva IGetUsersQuery, koji vraća listu korisnika na osnovu kriterijuma pretrage.
  Rezultat vraća kao 200 OK sa listom korisnika

- **Dohvatanje jednog korisnika** (GET {id})
  Prima parametre pretrage (NotificationsSearch) iz query stringa.
  Postavlja ID korisnika u pretragu (da korisnik može videti samo svoje notifikacije).
  Pokreće upit IGetNotificationsQuery da dobije podatke.
  Vraća listu notifikacija u 200 OK odgovoru.

- **Ažuriranje korisnika** (PUT {id})
  Prima ID korisnika iz URL-a.
  Prima podatke za ažuriranje iz forme ([FromForm] UpsertUserDto).
  Poziva IUpdateUserCommand da ažurira podatke u bazi.

Nakon ažuriranja, ponovo poziva IGetUserQuery da dobije novog korisnika i vrati ga u 200 OK odgovoru.
Ako ažurirani korisnik ne postoji, vraća 404 Not Found.

- **Brisanje korisnika** (DELETE {id})
  Prima ID korisnika iz URL-a.
  Poziva IDeleteUserCommand da obriše korisnika iz baze.
  Ne vraća nikakav sadržaj (204 No Content).

API/Controllers/ImagesController.cs

📌 **Endpoint:** api/images

- **Upload slike** (POST)
  Očekuje **multipart form-data** sa fajlom (dtoRequest.Image).
  Generiše novi naziv koristeći Guid.NewGuid(), čime sprečava duplikate. Kreira direktorijum ako ne postoji (wwwroot/Images).
  Snima fajl na disk.
  Dodaje putanju slike u bazu podataka.

- **Dohvatanje slike objave** (GET {image-name})
  Koristi _imageService.GetImage() za dohvat slike iz wwwroot/Images.
  Ako slika ne postoji, vraća 404 Not Found.
  Vraća fajl sa MIME tipom (GetMimeType).

- **Dohvatanje profilne slike korisnika** (GET profile-image/{profilePicture})
  Isti princip kao GetPostImage, ali traži slike iz wwwroot/UserImages/.

- **Proxy za slike** (POST proxy)
  Prima URL slike.
  Koristi **HttpClient** da preuzme sliku sa eksternog servera.
  Vraća sliku kao FileResult.

- **Proxy sa keširanjem** (GET proxy)
  Proverava da li je slika već sačuvana (wwwroot/ProxyImages).
  Ako jeste, vraća keširanu verziju.
  Ako nije, preuzima sliku i snima u keš.
  Koristi MD5 heš URL-a da spreči konflikte imena.

Client sloj – Frontend aplikacija

Blog platforma je razvijena sa ciljem da korisnicima omogući kreiranje, uređivanje i praćenje blog postova uz napredne funkcionalnosti kao što su komentarisanje, lajkovanje i sistem notifikacija u realnom vremenu. Projekat pruža interaktivno

korisničko iskustvo, omogućavajući korisnicima da prate omiljene autore i primaju obaveštenja o njihovim aktivnostima.

Frontend koristi **Vite** kao alat za build i lokalni razvoj, što omogućava izuzetno brze reload-ove i modernu konfiguraciju.
Za **upravljanje globalnim stanjem** koristi se **Redux Toolkit**, dok je za **trajnost podataka** u lokalnom skladištu integrisan **redux-persist**.
Tailwind CSS i **Flowbite React** biblioteke korišćene su za modularnu i elegantnu stilizaciju komponenata, dok je **Firebase** implementiran radi autentifikacije i bezbedne komunikacije sa backend sistemom.

Komunikacija sa API slojem obavlja se putem **HTTP REST zahtevâ** koristeći **Fetch API**, a za real-time funkcionalnosti (poput notifikacija) koristi se **SignalR** integracija putem paketa react-signalr.

Struktura aplikacije

Struktura projekta organizovana je po funkcionalnim celinama:

- /contexts/ — definisani su globalni contexti za greške, uspešne poruke i notifikacije.
- /redux/ — sadrži store konfiguraciju i slice-ove (userSlice, notificationsSlice, themeSlice).
- /pages/ — glavni view fajlovi (kao što su SignIn, SignUp, PostPage, UpdatePost, itd).
- /components/ — višekratno upotrebljivi delovi interfejsa (Sidebar, PostCard, CommentSection, itd).
- /utils/ — pomoćne funkcije za API obradu i druge poslovne logike (handleApiError, postLikeUtils, timeAgo, itd).

Client/package.json

Datoteka package.json definiše sve zavisnosti, skripte i osnovne informacije o projektu. Ključne tačke konfiguracije su sledeće:

Skripta	Opis
"dev"	Pokreće lokalni razvojni server putem **Vite-a**.
"build"	Kreira optimizovani build za produkciju.

Skripta	Opis
"lint"	Pokreće **ESLint** radi statičke analize koda.
"preview"	Pokreće lokalni pregled build-ovane aplikacije.

Ključne zavisnosti (dependencies)

Core biblioteke:

- **react**, **react-dom** – Osnova React aplikacije.
- **react-router-dom** – Omogućava SPA (Single Page Application) navigaciju putem ruta.
- **@reduxjs/toolkit**, **react-redux** – Centralizovano upravljanje stanjem aplikacije.
- **redux-persist** – Čuvanje Redux stanja u LocalStorage-u (npr. tokeni, korisnički podaci).
- **moment**, **react-moment** – Formatiranje datuma.
- **react-icons** – Ikonice korišćene širom aplikacije.
- **flowbite-react**, **tailwindcss**, **tailwind-scrollbar** – Stilizacija i UI komponente.
- **firebase** – Integracija sa Firebase servisima (autentifikacija, storage).
- **jwt-decode** – Dekodiranje JWT tokena i ekstrakcija korisničkih podataka.
- **react-signalr** – Real-time komunikacija sa backendom.
- **react-quill**, **react-datepicker** – Rich text editor i izbor datuma.

Dev zavisnosti:

- **@vitejs/plugin-react-swc** – Brza kompilacija React koda pomoću SWC engine-a.
- **eslint**, **eslint-plugin-react**, **eslint-plugin-react-hooks** – Analiza koda i detekcija potencijalnih grešaka.
- **postcss**, **autoprefixer** – Generisanje CSS-a sa podrškom za različite browsere.

Client//vite.config.js

vite.config.js fajl sadrži konfiguraciju Vite servera i proxy mapiranja koja omogućavaju lokalnu komunikaciju sa backendom bez CORS problema.

Svi navedeni API endpointi se prosleđuju backend serveru na adresi http://localhost:5207, koji je ASP.NET Core Web API deo projekta.

Proxy konfiguracija

secure: false - omogućava korišćenje nešifrovanih HTTP konekcija tokom razvoja.

changeOrigin: true - menja izvor zahteva, čime se izbegavaju CORS greške.

ws: true - aktivira se samo za SignalR websocket rutu (/api/notificationsHub).

Pokretanje aplikacije i glavni tok renderovanja

U ovom delu pokrivam **main.jsx**, **App.jsx**, i tri ključna **Context providera** (ErrorContext, SuccessContext, NotificationsContext), jer zajedno čine **centralni mehanizam upravljanja stanjem i globalnim porukama** u React aplikaciji.

Client/src/main.js

Fajl main.jsx je **ulazna tačka React aplikacije**.
Ovde se inicijalizuje ReactDOM, povezuje globalni Redux store i definišu svi globalni provideri koji omogućavaju kontekstualnu logiku (teme, poruke, notifikacije).

Objašnjenje ključnih delova:

Komponenta	**Uloga**
<Provider>	Omogućava pristup Redux store-u iz bilo kog dela aplikacije.
<PersistGate>	Odlaže render dok se Redux stanje ne učita iz localStorage-a. Koristi se za očuvanje teme, tokena, korisnika i notifikacija.
<ThemeProvider>	Upravljanje svetlim/tamnim režimom pomoću Tailwind-a i Redux-a.
<App />	Glavna React komponenta gde se definišu sve rute i konteksti.

Client/src/App.js

Aplikacija koristi **React Router v6** sa BrowserRouter komponentom, što omogućava Single Page Application (SPA) navigaciju bez reload-a stranice.

Opis komponenata i logike:

Element	Objašnjenje
<BrowserRouter>	Upravljanje klijentskim rutama kroz URL putanje.
<ErrorProvider> i **<SuccessProvider>**	Omogućavaju prikaz globalnih poruka o greškama i uspehu (notifikacioni sistem u gornjem desnom uglu).
<NotificationsProvider>	Real-time praćenje notifikacija putem SignalR konekcije.
<ScrollToTop>	Resetuje poziciju skrola pri promeni rute.
<Header> i **<Footer>**	Zajednički delovi interfejsa koji se prikazuju na svim stranicama.
<Routes>	Definiše dostupne putanje i komponente za svaku rutu.

Zaštita ruta

Aplikacija koristi dve zaštitne komponente:

- **PrivateRoute** – dozvoljava pristup samo prijavljenim korisnicima.
- **OnlyRolePrivateRoute** – dodatna zaštita za rute namenjene samo autorima (kreiranje i ažuriranje postova, kreiranje kategorija).

Ove komponente proveravaju JWT token iz Redux store-a i dekodiraju korisničku ulogu pomoću jwt-decode.

React Context API

U nastavku su dokumentovani svi konteksti koji omogućavaju centralizovano rukovanje greškama, uspešnim porukama i real-time notifikacijama.

Aplikacija koristi tri osnovna contexta koji omogućavaju centralizovano upravljanje korisničkim iskustvom i real-time funkcionalnošću.

Client/src/context/ErrorContext.js

Svrha:

Globalno prikazuje poruke o greškama (npr. neuspeli API pozivi, token problemi).

Mehanizam rada:

- Kada se pozove showError("tekst poruke"), kreira se novi objekat sa ID-em i porukom.
- Poruka se prikazuje u gornjem desnom uglu (position: fixed), a zatim se automatski uklanja posle 4 sekunde.
- Vizuelno koristi animaciju nestajanja, što daje efekat “fading out”.

Client/src/context/SuccessContext.js

Svrha:

Analogno ErrorContext-u, ali za poruke o **uspešnim akcijama** (npr. kreiranje posta, uspešna registracija, dodavanje kategorije).

Mehanizam rada:

- Pozivom showSuccess(message) poruka se prikazuje na 7 sekundi.
- Poruke se prikazuju **jedna ispod druge** pomoću dinamičnog top izračunavanja (top: 32 + index * 80).
- Ovo omogućava paralelno prikazivanje više uspešnih akcija bez preklapanja.

Client/src/context/NotificationContext.js

Svrha:

Upravlja svim korisničkim notifikacijama i povezuje aplikaciju sa backendom putem **SignalR-a**.
Ovo omogućava **real-time primanje obaveštenja** kada, na primer:

- novi korisnik zaprati autora,
- autor objavi novi post,
- korisnik dobije komentar ili lajk.

Ključne funkcionalnosti:

1. **fetchNotifications()** – dohvata sve notifikacije sa API-ja prilikom mountovanja komponente.
2. **SignalR konekcija** – povezuje korisnika sa /api/notificationsHub koristeći JWT token.
3. **handleNewNotification()** – dinamično dodaje novu notifikaciju u lokalno stanje.
4. **updateUnreadCount()** – računa broj nepročitanih notifikacija i ažurira globalni Redux state.
5. **markAllAsReadOnPageChange()** – automatski označava sve notifikacije kao pročitane kada korisnik otvori /notifications stranicu.

Integracija Context providera sa Redux-om

NotificationsContext koristi useDispatch i setUnreadCount akciju iz Redux slice-a. Ovim pristupom **Context** služi za držanje kompleksne real-time logike, dok **Redux** ostaje zadužen za globalno stanje koje utiče na UI (npr. broj nepročitanih poruka u header-u).

Pages folder

Client/src/pages/Home.jsx

Svrha

Home komponenta predstavlja **početnu stranicu** aplikacije i sadrži uvodni deo sa opisom bloga, poziv na akciju, kao i prikaz **najnovijih postova**.

Logika

- Koristi useEffect za dohvatanje podataka sa API-ja:
  const response = await fetch(`/api/posts?perPage=4`)
- Prikazuje 4 najnovija posta u okviru komponenti PostCard.
- Kod greške poziva showError iz ErrorContext-a.
- Sadrži CallToAction komponentu koja promoviše dalje čitanje ili registraciju korisnika.

Zaostala greška:
fetch se izvršava pri svakom renderu showError funkcije, jer se nalazi u useEffect dependency listi.

Preporuka: Umesto [showError], koristi prazan niz [] — showError je stabilna funkcija iz context-a i ne mora biti dependency.

Client/src/pages/Authors.jsx

Svrha

Prikazuje listu svih korisnika sa ulogom **Author**, uključujući ime, korisničko ime, email i profilnu sliku.

Logika

- API poziv:
  fetch(`/api/users?onlyAuthors=true&perPage=3&page=${currentPage}`)
- Koristi **paginaciju** iz Flowbite-a (Pagination komponenta).
- Filtrira rezultate kako bi prikazala samo korisnike sa roleName == "Author".
- Koristi pomoćnu funkciju getAvatarSrc() za siguran prikaz slike (čak i ako nedostaje URL).

Client/src/pages/CategoryPage.jsx

Svrha

Prikazuje sve postove unutar jedne kategorije.
Stranica je povezana sa rutom /category/:id.

Logika

- API poziv:
  fetch(`/api/categories/${id}?page=${currentPage}&perPage=3`)
- Renderuje sve postove unutar kategorije sa prikazom:

  - slike autora,

  - imena autora,

  - datuma objave,

  - naslova posta i pripadajućih kategorija.

- Omogućena je paginacija po 3 posta.

Client/src/pages/CreateCategory.jsx

Svrha

Stranica namenjena **autorima** za dodavanje novih kategorija postova.

Logika

- Koristi useError i useSuccess kontekste za prikaz status poruka.
- Šalje POST zahtev ka endpointu: /api/categories
- Token se preuzima iz localStorage, a zahtev se šalje uz Bearer zaglavlje.

UI elementi

- TextInput za naziv kategorije.
- Button (Flowbite) sa gradientDuoTone="purpleToPink" efektom.
- Nakon uspešnog dodavanja, polje se resetuje i prikazuje success toast.

Client/src/pages/CreatePost.jsx

Svrha

Omogućava kreiranje novih postova — jedan od **ključnih delova blog sistema**.

Logika

Učitava sve kategorije za checkbox selekciju: /api/categories?getAll=true

Učitava sliku pomoću FormData: /api/images

Nakon toga šalje glavni zahtev za kreiranje posta: /api/posts

Koristi ReactQuill kao rich text editor za unos sadržaja posta.

Nakon uspešnog dodavanja:

- briše formu,
- čisti slike i tekst,
- prikazuje showSuccess.

Dodatni elementi

- **Upload slika:** FileInput + Button
- **Pregled slike:**
- **Kategorije:** višestruki izbor checkbox-ova.
- **Sadržaj:** ReactQuill editor.

Client/src/pages/Dashboard.jsx

Svrha

Dashboard je **centralno mesto za administraciju** i menadžment sistema.
Omogućava:

- upravljanje postovima, kategorijama, korisnicima, komentarima,
- pregled logova (useCaseLogs),
- zahtev za “Author” status,
- pregled zahteva drugih korisnika (ako je admin).

Logika

- tab parametar u URL-u (npr. ?tab=posts) određuje koja se komponenta prikazuje.
- Povezuje se sa komponentama poput DashSidebar, DashProfile, DashLogs, DashComments, itd.

Client/src/pages/NotificationsPage.jsx

Svrha

Stranica za prikaz korisničkih notifikacija dobijenih putem SignalR-a.

Logika

- Koristi NotificationsContext za pristup listi notifikacija.
- Filtrira notifikacije po tipu (All, Posts, Comments, Likes).
- Automatski menja boju ivice novih notifikacija (plava = nova, siva = pročitana).
- Poziva navigate(notification.link) za otvaranje povezanog sadržaja.

Client/src/pages/PostsPage.jsx

Svrha

Stranica koja prikazuje sve objavljene postove, sa mogućnostima:

- pretrage po naslovu,
- sortiranja (Latest / Oldest),
- filtriranja po više kategorija,
- paginacije.

Logika

- API poziv: /api/posts?page=x&perPage=5&title=&sortOrder=&categoryIds[n]=
- Komponenta koristi MultiSelectDropdown za više kategorija.
- Paginacija pomoću Flowbite-ove Pagination.

Client/src/pages/UserCommentPage.jsx

Svrha

Prikazuje određeni komentar i eventualno njegov nadkomentar (parentComment).

Logika

- API poziv: /api/comments/:id
- Ako komentar ima idParent, poziva se dodatni fetchParentComment.
- Prikazuje ime autora, datum, tekst i broj lajkova.

Client/src/pages/UserPage.jsx

Svrha

Profil korisnika: prikazuje osnovne informacije, broj pratilaca, postove i komentare.

Logika

- Dohvata korisnika i proverava da li trenutni korisnik prati njega: /api/followers/${id}/check

- Omogućava Follow i Unfollow logiku sa ažuriranjem followersCount.
- Prikazuje poslednji post, komentare i ostale objave.

Client/src/pages/PostPage.jsx

Cilj: prikaz posta, lajk/dislajk, komentari, autor, kategorije.

Tok rada:

1. **Učitavanje posta po ID-u**
   ➤ Koristi se showError iz konteksta i handleApiError.
2. **Render Loading Spinner-a dok se učitava**
3. **Lajk/Dislajk logika**

- Proveravaš token.
- Ako ga nema → redirect na /sign-in.
- Proveravaš da li je korisnik već lajkovao:
  const isAlreadyVoted = checkIfAlreadyVotedOnPost(post, idPost, currentUser.id, status)
- Ovo je odlično — eksternalizovana logika u utils/postLikeUtils.js sprečava dupliranje koda.

4. **Update stanja posta posle reakcije**
   setPost(updatedPost) je čisto i jasno.
   Dobar pattern bi bio dodati optimistički update — tj. odmah ažuriraš UI, pa onda API (korisnik vidi bržu reakciju).

Client/src/pages/SignIn.jsx

Tok rada:

Komponenta omogućava prijavu putem korisničkog imena i lozinke ili Google OAuth naloga.
Nakon uspešnog logovanja:

- JWT token se dekodira pomoću jwtDecode().
- Korisnički podaci iz tokena (ActorData) se čuvaju u Redux store-u.
- Token se sprema u localStorage za buduće zahteve.
- Navigacija se automatski preusmerava na početnu stranu.

Redux + Context integracija

- useDispatch() koristiš za globalni state (signInStart, signInSuccess, signInFailure).
- useError() koristiš za trenutne UI notifikacije — što znači da razdvajaš "logic error" od "UX feedbacka".
  To je dobra praksa.

➤ Izvlačiš korisnika iz payload-a i odmah ga setuješ u Redux store.

Greške se centralno hvataju kroz handleApiError i prikazuju putem showError.

Client/src/pages/SignUp.jsx

Tok rada:

1. **Error Handling**
   ➤ Vrlo dobar sistem sortiranja i prikaza grešaka
2. **Navigacija**
   navigate('/sign-in') nakon uspešne registracije je odlično UX rešenje.
3. **Ostalo**
   ➤ OAuth integrisan — znači da tvoj sistem već podržava Google sign-in, što je sjajno.
   Samo pazi da i kod OAuth logina dodaš istu Redux logiku (signInSuccess itd.), da ne bude odvojen flow.

Client/src/pages/UpdatePost.jsx

Tok rada:

1. **Dva useEffect-a za učitavanje podataka**
   - Prvi za post (fetchPost)
   - Drugi za kategorije (fetchCategoriesForUpdatePost)
     ➤ Dobro odvojeno, ali mogli bi se kombinovati sa Promise.all() ako želiš paralelno učitavanje.
2. **Prikaz već postojećih podataka**
   ✅ defaultValue={editData.title || ''}
   ✅ value={content ? content : editData.content}
   ➤ Lepo razdvojeno između starog i novog sadržaja.

3. **Update logika**
   const url = currentUser.id == editData.idUser ?
   `/api/posts/${editData.id}/personal` : `/api/posts/${editData.id}`

➤ Vrlo dobar uslov — sprečavaš da korisnik menja tuđe postove ako nije autor.

4. **Image upload**
    - Koristiš FormData i ručno dodaješ header sa tokenom.
      Ali obrati pažnju: fetch ne treba da ima "Content-Type" ako koristiš FormData.
      Browser ga automatski setuje — inače može doći do greške.

5. **ReactQuill**
   ✅ value={content ? content : editData.content}
   ➤ Odlično rešeno za dinamično ažuriranje teksta.

6. **UX poboljšanja (predlozi)**
    - Dodaj spinner dok se post učitava.
    - Dodaj “Cancel” dugme koje vraća korisnika na prethodnu stranu.
    - Možeš prikazati preview slike u modalu pre slanja.

Zaključak za Pages sloj

Pages folder predstavlja **glavni sloj prikaza korisničkog interfejsa**, kroz koji se obavlja komunikacija između Context i Redux slojeva sa API-jem.
Svi pozivi su REST-based, a greške se propagiraju kroz ErrorContext.

Komponente

Komponente predstavljaju osnovne građevinske blokove korisničkog interfejsa.
Svaka komponenta je napisana kao **funkcionalna komponenta (function component)** i koristi **React Hooks** za upravljanje stanjem i životnim ciklusom.
UI se gradi pomoću **TailwindCSS** i **Flowbite-React** biblioteke, dok se ikone uvoze iz **react-icons** paketa.
U ovom poglavlju biće objašnjene sve komponente koje čine kompletan korisnički interfejs blog sistema.

Client/src/components/AdminDashboard.jsx

Opis

Komponenta **AdminDashboard** prikazuje ključne administrativne statistike i pregled najnovijih korisnika, komentara i postova.
Dostupna je samo korisnicima sa ulogom **Admin**.

Funkcionalnosti

- Dohvata agregirane podatke o:
    - ukupnom broju korisnika, komentara i postova,
    - broju novih korisnika, komentara i postova u poslednjih mesec dana.
- Prikazuje tabele sa pet najnovijih zapisa iz svake kategorije.
- Omogućava navigaciju do potpunih lista putem Link komponente (/dashboard?tab=users, /dashboard?tab=comments, itd.).
- Svaki entitet (user, comment, post) ima vizuelni prikaz preko Flowbite tabela (<Table>).

Tehnička struktura

- Hook **useEffect** pokreće asinhronu funkciju fetchDashboardData koja paralelno dohvata sve podatke pomoću Promise.all().
- U slučaju greške (npr. nedostajućeg tokena), poziva se showError() iz **ErrorContext-a**.
- Podaci se filtriraju i formatiraju za prikaz (samo prvih 5 zapisa po tipu).

Upotrebljene biblioteke

- react-icons za vizuelne oznake (HiOutlineUserGroup, FaRegComments, HiDocumentText).
- flowbite-react za UI elemente (Button, Table).
- redux i useSelector za dobijanje podataka o trenutnom korisniku.
- getAvatarSrc() helper funkcija — vraća validnu URL adresu slike profila.

Client/src/components/CallToAction.jsx

Opis

Jednostavna i statična marketinška komponenta koja prikazuje promotivni poziv ka spoljnjem resursu.
Koristi se uglavnom na početnoj stranici.

Funkcionalnosti

- Prikazuje tekstualni poziv (“Want to learn more about JavaScript?”).
- Sadrži dugme koje vodi na eksterni sajt *100 JavaScript Projects* (https://www.100jsprojects.com).
- Prikazuje prateću ilustraciju u desnom delu komponente.

Tehnička struktura

- Koristi **responsive flex layout** (sm:flex-row) i **TailwindCSS** klase za stilizaciju.
- Eksterni link otvara se u novom tabu pomoću atributa target="_blank" i rel="noopener noreferrer" (sprečava potencijalni XSS vektor).

Client/src/components/ChildComment.jsx

Opis

Pomoćna komponenta koja rekurzivno prikazuje hijerarhiju podkomentara (odgovora) na glavni komentar.
Koristi se unutar komponente **Comment**.

Funkcionalnosti

- Filtrira niz childComments prema idParentComment.
- Za svaki pronađeni “child” komentar renderuje **Comment** komponentu.
- Uređuje vizuelni padding u zavisnosti od dubine rekurzije i veličine ekrana (isSmallScreen).

Tehnička struktura

- Koristi **prop-types** za validaciju ulaznih podataka.
- Prosleđuje sve callback funkcije dalje ka **Comment** komponenti (onDeleteComment, onLikeComment, itd.).

- U style objektu koristi dinamičke vrednosti za paddingLeft i paddingBottom radi konzistentnog prikaza.

Client/src/components/Comment.jsx

Opis

Centralna komponenta sistema komentarisanja.
Zadužena je za prikaz pojedinačnog komentara, omogućavanje lajkovanja, editovanja, brisanja i dodavanja odgovora.

Funkcionalnosti

- Prikazuje podatke o korisniku koji je postavio komentar.
- Prikazuje vreme kreiranja komentara pomoću **moment.js** (fromNow() format).
- Omogućava:
    - lajkovanje/dislajkovanje komentara,
    - uređivanje i brisanje komentara (samo vlasnik ili admin),
    - odgovaranje na komentare putem rekurzije (ChildComment),
    - prikaz oznake “Comment is removed” za obrisane komentare.

Tehnička struktura

- useEffect asinhrono dohvaća podatke o autoru komentara sa /api/users/{idUser}.
- handleSave() PUT metodom ažurira sadržaj komentara.
- Funkcija toggleReplyForm() otvara formu za odgovor i automatski ubacuje @username.
- Komponenta podržava višeslojnu strukturu rekurzivnih komentara.

Integracije

- **useError** iz ErrorContexta za hvatanje grešaka.
- **useSelector** iz Redux-a za proveru identiteta trenutnog korisnika.
- **ChildComment** komponenta za prikaz svih rekurzivnih odgovora.

Client/src/components/CommentSection.jsx

Opis

Glavna komponenta za rad sa komentarima na stranici posta.
Upravlja svim CRUD operacijama (Create, Read, Update, Delete) za komentare i njihove odgovore.

Funkcionalnosti

- Dohvata post i sve povezane komentare (/api/posts/{idPost}).
- Prikazuje komentare i odgovore koristeći Comment i ChildComment.
- Omogućava dodavanje novih komentara i odgovora.
- Omogućava lajkovanje/dislajkovanje komentara.
- Prikazuje i obrađuje modale za potvrdu brisanja komentara.
- Automatski ažurira broj komentara pomoću callback funkcije onCommentsNumberChange.

Tehnička struktura

- Koristi useState za više logičkih stanja (komentar, childComments, broj komentara itd.).
- useEffect inicijalno učitava post i komentare, sortira ih po datumu i filtrira obrisane.
- handleVoteComment() razlikuje POST i DELETE pozive prema trenutnom statusu korisnikovog lajka.
- Modal iz Flowbite biblioteke koristi se za potvrdu brisanja komentara.

Integracije

- useError za prikaz grešaka.
- handleApiError za standardizovanu obradu API odgovora.
- getAvatarSrc za dinamičko određivanje slike korisnika.

Client/src/components/DashAuthorRequests.jsx

Opis

Komponenta **DashAuthorRequests** prikazuje zahteve korisnika koji žele postati autori. Pristup ima samo administrator.

Funkcionalnosti

- Dohvata listu zahteva paginirano (/api/authorrequests?page={currentPage}).
- Omogućava administratoru da prihvati ili odbije zahtev.
- Prikazuje trenutni status zahteva (Pending, Accepted, Rejected).
- Prikazuje paginaciju pomoću Pagination komponente.

Tehnička struktura

- Funkcije handleAcceptAuthorRequest i handleRejectAuthorRequest šalju PUT zahteve backendu.
- Funkcija updateAuthorRequestStatus odmah ažurira stanje u interfejsu (optimistički update).
- Koristi statusIcon mapu za prikaz statusa sa različitim bojama i ikonama.

Integracije

- **useError** za greške.
- **getAvatarSrc** helper za prikaz slike korisnika.
- **Flowbite** komponente: Table, Pagination, Button.

Client/src/components/DashComments.jsx

Opis

Komponenta namenjena administratorima za upravljanje komentarima korisnika. Omogućava pregled svih komentara, brisanje i paginaciju.

Funkcionalnosti

- Dohvata komentare sa backend-a paginirano (/api/comments?page={currentPage}).
- Prikazuje:
    - datum kreiranja komentara,

    - sadržaj (truncate ako je duži od 35 karaktera),
    - broj lajkova,
    - post na koji se odnosi,
    - autora komentara.
- Omogućava brisanje komentara putem modala (Flowbite Modal).
- Prikazuje uspešnu poruku nakon brisanja (showSuccess).

Tehnička struktura

- useEffect za inicijalno učitavanje podataka.
- handleDeleteComment() izvršava DELETE zahtev i uklanja komentar iz lokalnog stanja.
- Koristi showError i showSuccess za centralizovano UX obaveštavanje.

Client/src/components/DashCategories.jsx

Opis

Komponenta **DashCategories** služi za prikaz, uređivanje i brisanje kategorija unutar administrativnog panela.
Pristup ovoj funkcionalnosti ima isključivo korisnik sa ulogom **Admin**.

Funkcionalnosti

- Dohvata kategorije sa servera putem paginacije (/api/categories?page={x}&perPage=12).
- Prikazuje listu kategorija u tabelarnom prikazu koristeći Flowbite Table.
- Omogućava **inline uređivanje naziva kategorije** i **brisanje kategorija** putem podkomponente EditableCategoryRow.
- Omogućava promenu stranice pomoću Flowbite Pagination.

Tehnička struktura

- Stanja:

    - categories — niz svih kategorija za trenutnu stranicu.
    - currentPage, pageCount — kontrola paginacije.
- useEffect se pokreće pri svakoj promeni currentPage i asinhrono dohvata podatke.
- Token se učitava iz localStorage i koristi za autorizaciju svakog API zahteva.
- U slučaju greške, koristi se **ErrorContext** i handleApiError() helper.

Integracije

- **EditableCategoryRow**: podkomponenta za editovanje i brisanje kategorija.
- **useError** (ErrorContext) za prikaz grešaka.
- **Redux (useSelector)** za pristup trenutnom korisniku.
- **Flowbite** komponente: Table, Pagination.

Client/src/components/EditableCategoryRow.jsx

Opis

Podkomponenta koja omogućava **inline izmenu** i **brisanje kategorija** iz tabele prikazane u **DashCategories** komponenti.
Predstavlja reusable red tabele koji podržava režime “prikaz” i “uređivanje”.

Funkcionalnosti

- Prikazuje naziv kategorije i omogućava dvostruki klik (ili klik) za prelazak u mod uređivanja.
- U režimu uređivanja:
    - Prikazuje <input> polje sa trenutnim nazivom.
    - Dugmad sa ikonama (✓ i ✗) za potvrdu i otkazivanje promene.
- Omogućava brisanje kategorije uz modalno potvrđivanje (Flowbite Modal).
- Automatski obaveštava roditeljsku komponentu (onSave, onDelete).

Tehnička struktura

- isEditing — boolean za mod uređivanja.
- newName — lokalno stanje za izmenjeni naziv.
- showDeleteModal — prikazuje modal za potvrdu brisanja.
- API pozivi:
    - PUT /api/categories/{id} — za izmenu naziva.
    - DELETE /api/categories/{id} — za brisanje.
- Koristi PropTypes za validaciju ulaznih podataka.

Integracije

- **useError**, **useSuccess** za UX obaveštenja.
- **handleApiError** za standardizovanu obradu grešaka.
- **Flowbite** komponente (Table, Button, Modal).
- **react-icons** za HiOutlineCheck, HiOutlineX, HiOutlineExclamationCircle.

Client/src/components/DashLogs.jsx

Opis

Komponenta **DashLogs** služi za prikaz svih logova use case operacija u sistemu (CQRS logovanje).
Dostupna isključivo korisniku sa ulogom **Admin**.
Predstavlja administratorski alat za nadzor nad svim radnjama koje se izvršavaju na backendu.

Funkcionalnosti

- Prikazuje **istoriju izvršenih use case operacija** (ime, korisnik, vreme, podatke).
- Omogućava **filtriranje** po:
    - korisniku (*Actor*),
    - use case imenu (*UseCaseName*),
    - vremenskom intervalu (*DateFrom*, *DateTo*).
- Podržava **sortiranje po datumu** (asc/desc).

- Klikom na “View Details” otvara modal sa detaljnim prikazom loga (uključujući JSON podatke).
- Paginacija se automatski računa na osnovu totalCount.

Tehnička struktura

- Stanja:
    - logs, loading, total, error, selectedLog, isModalOpen.
    - filters objekt sa parametrima za pretragu i paginaciju.
- API zahtev:
  GET /api/usecaselogs sa query parametrima generisanim iz filters.
- toggleSort() menja filters.sortOrder između "asc" i "desc".
- DatePicker biblioteka koristi se za unos datuma filtriranja.

Integracije

- **Flowbite** komponente: Table, Button, Pagination, Modal, TextInput, Spinner.
- **react-datepicker** za odabir vremenskog intervala.
- **localStorage token** za autentifikaciju.

Client/src/components/DashPosts.jsx

Opis

Komponenta za prikaz svih postova u administrativnom panelu.
Dostupna isključivo korisniku sa ulogom **Admin**.

Funkcionalnosti

- Dohvata postove sa servera (/api/posts?page={currentPage}).
- Prikazuje:
    - datum kreiranja,
    - naslov posta,

    - pripadajuće kategorije,
    - sliku posta (thumbnail),
    - opcije za brisanje i uređivanje.
- Omogućava **brisanje posta** putem modalnog dijaloga.

Tehnička struktura

- Hook useEffect pokreće asinhrono dohvatanje postova pri svakoj promeni currentPage ili postDeleted.
- API pozivi:
    - GET /api/posts — za listu postova.
    - DELETE /api/posts/{id} — za brisanje posta.
- handleDeletePost() obavlja backend poziv i optimistično uklanja post iz lokalnog state-a.

Integracije

- **useError**, **useSuccess** — za UX poruke.
- **handleApiError** — za backend greške.
- **Flowbite** komponente: Table, Modal, Pagination, Button.
- **react-icons**: HiOutlineExclamationCircle.
- **react-router-dom**: Link za navigaciju.

Client/src/components/DashProfile.jsx

Opis

Komponenta **DashProfile** omogućava korisnicima da pregledaju i menjaju svoj profil, kao i da obrišu nalog ili se odjave.
Dostupna svim korisnicima sistema.

Funkcionalnosti

- Ažuriranje osnovnih informacija (ime, prezime, korisničko ime, email, lozinka).
- Ažuriranje profilne slike (sa preview prikazom).

- Brisanje naloga (DELETE /api/users/{id}).
- Odjava korisnika (brisanje tokena i Redux stanja).

Tehnička struktura

- useRef koristi se za upravljanje <input type="file">.
- FormData koristi se za slanje multipart zahteva.
- Nakon uspešnog ažuriranja:
    - Redux se ažurira pomoću updateUserSuccess() i updateProfilePictureSuccess().
    - Prikazuje se showSuccess() poruka.
- Brisanje naloga prikazuje Flowbite Modal za potvrdu.

Integracije

- **Redux** akcije: updateUserSuccess, updateProfilePictureSuccess, deleteUserSuccess, signoutSuccess.
- **ErrorContext** i **SuccessContext**.
- **Flowbite**: Modal, Button, TextInput.
- **getAvatarSrc** helper za slike.

Client/src/components/DashSidebar.jsx

Opis

Komponenta **DashSidebar** prikazuje navigacioni meni unutar dashboarda.
Prilagođava prikaz stavki u zavisnosti od korisničke uloge (Admin, Author, User).

Funkcionalnosti

- Dinamički renderuje sidebar linkove na osnovu currentUser.roleName.
- Dohvata dodatne informacije o korisniku (/api/users/{id}) – broj postova, pratilaca, praćenih.
- Obezbeđuje **odjavu korisnika** (signoutSuccess + brisanje tokena).

- Aktivni tab detektuje iz URLSearchParams.

Tehnička struktura

- useLocation prati URL query parametar ?tab=.
- useEffect za sinhronizaciju tab stanja sa URL-om.
- fetchUser() poziv za osvežavanje korisničkih metrika.
- Prikaz broja postova, followers i following u badge-ovima.

Integracije

- **Redux (useSelector, useDispatch)**.
- **Flowbite Sidebar i Button** komponente.
- **react-icons** za vizuelne oznake (HiUser, FaRegComments, BsListColumns, itd.).
- **ErrorContext** i **handleApiError**.

Client/src/components/DashUsers.jsx

Opis

Administrativna komponenta za pregled i brisanje korisnika.
Dostupna isključivo korisniku sa ulogom **Admin**.

Funkcionalnosti

- Dohvata korisnike sa servera (/api/users?page={x}).
- Prikazuje:
    - datum kreiranja,
    - profilnu sliku,
    - korisničko ime, email, ulogu.
- Omogućava brisanje korisnika uz modalnu potvrdu.

Tehnička struktura

- Stanja: users, currentPage, pageCount, showModal, idUserToDelete.

- useEffect za inicijalno učitavanje i paginaciju.
- handleDeleteUser() vrši DELETE zahtev i ažurira lokalno stanje.

Integracije

- **useError**, **useSuccess** za UX poruke.
- **handleApiError** za backend greške.
- **Flowbite**: Table, Modal, Pagination, Button.
- **getAvatarSrc** helper za slike korisnika.

Client/src/components/FollowList.jsx

Opis

Komponenta **FollowList** prikazuje listu korisnika koji prate trenutnog korisnika (*Followers*) ili koje trenutni korisnik prati (*Following*).
Koristi se unutar dashboard sekcije za korisnike sa ulogom **Author**.

Funkcionalnosti

- Dinamičko učitavanje podataka sa API endpoint-a:
  - /api/followers/{userId}/followers
  - /api/followers/{userId}/following
- Prikaz korisničkih profila u grid layout-u sa avatarima, imenima, korisničkim imenom i emailom.
- Paginacija rezultata (perPage=5) putem Flowbite Pagination komponente.

Tehnička struktura

- Stanja:
  - listUsers — trenutni set korisnika za prikaz.
  - currentPage i pageCount — za navigaciju kroz strane.
- useEffect:
  - Dinamički poziva odgovarajući endpoint u zavisnosti od prop-a isFollowersTab.

- Ažurira prikaz pri promeni stranice ili taba.

Integracije

- **Redux (useSelector)** — pristup trenutnom korisniku.
- **ErrorContext** — za prikaz grešaka.
- **Flowbite Pagination** — za UX kontrolu stranica.
- **getAvatarSrc** — siguran izvor avatara čak i ako korisnik nema sliku.

Client/src/components/Footer.jsx

Opis

Komponenta **Footer** dinamički prikazuje sve dostupne kategorije bloga u dnu stranice, preuzete sa backend API-ja.
Ovo omogućava korisnicima da brzo pristupe kategorijama iz bilo kog dela sajta.

Funkcionalnosti

- Fetch svih kategorija pomoću query parametra getAll=true.
- Generiše listu linkova ka stranicama kategorija (/category/{id}).
- Koristi **TailwindCSS gradient pozadinu** za estetski efekat.

Tehnička struktura

- Jednostavan useEffect koji se pokreće jednom (componentDidMount).
- handleApiError() obezbeđuje standardizovanu obradu neuspješnih odgovora.
- showError() prikazuje toast grešku kroz globalni ErrorContext.

Integracije

- **ErrorContext** i **handleApiError**.
- **react-router-dom/Link** za navigaciju.

Client/src/components/Header.jsx

Opis

Glavna navigaciona traka aplikacije, uključuje logotip, pretragu, dugmad za akcije (kreiranje posta, dodavanje kategorije), promenu teme, notifikacije i korisnički meni. Ovo je jedna od centralnih komponenti koja povezuje **UX, autentifikaciju, notifikacije i navigaciju**.

Funkcionalnosti

- Prikaz različitih elemenata zavisno od uloge korisnika:
    - **Admin** — može dodati kategorije i postove.
    - **Author** — može dodati postove i vidi notifikacije.
    - **User** — nema administratorske kontrole.
- Prikaz broja nepročitanih notifikacija u realnom vremenu.
- Pretraga postova po ključnim rečima (/posts?search=...).
- Odjava korisnika (uklanjanje tokena i Redux reset).
- Promena teme između *light* i *dark* moda.

Tehnička struktura

- Stanja:
    - headerSearchTerm — lokalno polje pretrage.
- Hookovi:
    - useSelector — tema i korisnik.
    - useContext(NotificationsContext) — notifikacije.
    - useError — za prikaz grešaka.
    - useNavigate, useLocation — za rutu i navigaciju.
- Komponente:
    - Flowbite Navbar, Button, Avatar, Dropdown, TextInput.
    - Ikone: FaMoon, FaSun, FaRegBell, AiOutlineSearch.

Integracije

- **NotificationsContext** — brojač i real-time status.

- **Redux theme slice** — toggleTheme().
- **Redux user slice** — signoutSuccess().

Client/src/components/MultiSelectDropdown.jsx

Opis

Komponenta **MultiSelectDropdown** koristi se za izbor jedne ili više kategorija pri kreiranju ili uređivanju posta.
Podržava opcije „Select All“ i „Clear Selection“.

Funkcionalnosti

- Omogućava višestruki izbor pomoću checkbox polja.
- Prikazuje broj izabranih opcija ((x selected)).
- „Select All“ i „Clear Selection“ dugmad se automatski aktiviraju/deaktiviraju.
- Dropdown je potpuno kontrolisan pomoću Tailwind pseudo-klasa i peer mehanizma.

Tehnička struktura

- selectedOptions — lista ID-jeva odabranih kategorija.
- optionsListRef — referenca na DOM element liste radi manipulacije checkboxovima.
- handleChange() — ažurira set selektovanih ID-jeva i poziva onChange().
- isJsEnabled koristi se da spreči render pre nego što se komponente mountuju (sprečava SSR bagove).

Integracije

- **PropTypes** za validaciju tipova.
- **TailwindCSS** pseudo-klase: peer-checked, :has(), dark:hover:bg-gray-500.
- Potpuno nezavisna komponenta, lako reusable u bilo kom formu.

Client/src/components/OAuth.jsx

Opis

Komponenta **OAuth** omogućava prijavu korisnika putem **Google naloga** korišćenjem Firebase Authentication-a.

Funkcionalnosti

- Otvara Google popup kroz signInWithPopup().
- Nakon uspešne prijave:
    - Backend endpoint /api/auth prima podatke o korisniku.
    - Backend vraća JWT.
    - Token se dekodira pomoću jwtDecode().
    - Token se čuva u localStorage i Redux store se ažurira (signInSuccess()).
- Navigacija na početnu stranu nakon prijave.

Tehnička struktura

- useState(loading) sprečava višestruko klikanje.
- Firebase GoogleAuthProvider koristi parametar prompt: 'select_account' za izbor naloga.
- useError() prikazuje greške u toku autentifikacije.
- UX signalizacija: “Loading…” dok traje prijava.

Integracije

- **Firebase Auth** (signInWithPopup, getAuth, GoogleAuthProvider).
- **Redux user slice** — signInSuccess().
- **ErrorContext** i **handleApiError**.
- **Flowbite Button** i AiFillGoogleCircle ikona.

Client/src/components/OnlyRolePrivateRoute.jsx

Opis

Ruta koja štiti pristup stranicama namenjenim **Admin** i **Author** korisnicima.
Koristi react-router-dom Outlet mehanizam da renderuje child rute samo ako korisnik ima odgovarajuću ulogu.

Funkcionalnosti

- Ako je korisnik Admin ili Author, dozvoljava pristup (<Outlet />).
- Ako nije ulogovan ili nema dovoljnu ulogu, preusmerava na /sign-in.

Integracije

- **Redux user slice** za proveru trenutnog korisnika.
- **react-router-dom** za Navigate i Outlet.

Client/src/components/PrivateRoute.jsx

Opis

Jednostavna zaštitna ruta koja dozvoljava pristup samo ulogovanim korisnicima.

Funkcionalnosti

- Ako postoji currentUser u Redux store-u → renderuje child rute.
- Ako korisnik nije ulogovan → preusmerava na /sign-in.

Integracije

- **Redux user slice**.
- **react-router-dom** Navigate, Outlet.

Client/src/components/PostCard.jsx

Opis

Komponenta **PostCard** prikazuje pregled jednog blog posta u obliku kartice, sa naslovom, slikom, kategorijama i autorom.
Koristi se u listama postova (npr. na početnoj stranici, u pretragama, na stranicama kategorija).

Funkcionalnosti

- Klik na sliku ili na dugme **Read article** vodi na stranicu pojedinačnog posta (/post/{id}).
- Trunkuje naslove duže od 25 karaktera (da bi izgled bio ujednačen).

- Vizuelni efekat pri hover-u: povećanje slike i podizanje dugmeta „Read article“.
- Prikaz kategorija uz automatsko dodavanje zareza između njih.

Tehnička struktura

- Prop post je striktno tipiziran pomoću PropTypes.
- Tailwind klase upravljaju rasporedom i animacijama (group-hover, transition-all, rounded-lg).
- API slike se poziva preko /api/images/{imageName}.

Client/src/components/PostLikeButtons.jsx

Opis

Komponenta **PostLikeButtons** prikazuje broj komentara, lajkova i dislajkova na postu. Omogućava korisnicima da glasaju (like/dislike) ukoliko nisu autor posta.

Funkcionalnosti

- Brojač komentara i prikaz u formatu „1 Comment“ / „N Comments“.
- Dugmad za glasanje (FaThumbsUp, FaThumbsDown) uz boje koje označavaju reakciju.
- Onemogućava korisniku da lajkuje ili dislajkuje sopstvene postove.
- Ažurira broj glasova u realnom vremenu kroz callback onPostVote.

Tehnička struktura

- Redux useSelector dohvaća currentUser da proveri vlasništvo posta.
- Brojevi lajkova i dislajkova računaju se pomoću post.likes.filter((like) => like.status === 1).

Integracije

- **Parent komponenta** prosleđuje onPostVote handler koji vrši backend POST/PATCH zahtev.
- Ikone: FaThumbsUp, FaThumbsDown, FaRegCommentDots.

Client/src/components/RequestAuthorForm.jsx

Opis

Komponenta **RequestAuthorForm** omogućava korisniku da podnese zahtev da postane autor.
Slanje ide prema endpoint-u /api/authorrequests.

Funkcionalnosti

- Forma sa jednim obaveznim poljem (reason).
- Validacija i prikaz grešaka kroz **ErrorContext**.
- Prikaz poruke o uspehu putem **SuccessContext**.
- Reset polja nakon uspešnog slanja.

Tehnička struktura

- handleSubmit poziva fetch sa POST metodom i JSON telom.
- Status se čuva lokalno u authorRequests nizu (radi kasnijeg prikaza ako zatreba).
- Koristi **Flowbite** komponente: Label, Textarea, Button.

Integracije

- **Redux user slice** — koristi currentUser za ID korisnika.
- **ErrorContext** i **SuccessContext** — za poruke korisniku.
- **handleApiError** — standardizovana obrada grešaka.

Client/src/components/ScrollToTop.jsx

Opis

Helper komponenta koja resetuje poziciju skrola na vrh stranice svaki put kad se promeni ruta.
Koristi se u App.jsx odmah ispod BrowserRouter.

Funkcionalnosti

- Detektuje promenu rute pomoću useLocation().
- Poziva window.scrollTo(0, 0) u useEffect().

Client/src/components/ThemeProvider.jsx

Opis

Komponenta **ThemeProvider** upravlja globalnim izgledom aplikacije — light ili dark modom.
Obavija ostatak aplikacije i primenjuje odgovarajuće Tailwind klase.

Funkcionalnosti

- Čita trenutno aktivnu temu iz Redux store-a (theme slice).
- Postavlja osnovne Tailwind klase za pozadinu i tekst.
- Omogućava min-h-screen tako da sadržaj uvek pokriva ekran.

Integracije

- **Redux theme slice**.
- **TailwindCSS** klase: dark:bg-[rgb(16,32,42)], dark:text-gray-200.

Client/src/components/UserDashPosts.jsx

Opis

Komponenta **UserDashPosts** prikazuje listu postova koje je korisnik objavio, omogućava pregled, uređivanje i brisanje.
Pristup imaju korisnici sa rolom **Author**.

Funkcionalnosti

- Učitavanje postova korisnika putem /api/users/{id}.
- Prikaz u tabeli (Flowbite Table) sa kolonama:
    - Datum, naslov, kategorije, dugmad za *Edit* i *Delete*.
- Brisanje posta uz modalno potvrđivanje.
- Vizuelni feedback za uspeh ili grešku.

Tehnička struktura

- Stanja: userPosts, currentPage, pageCount, showModal, idPostToDelete.
- handleDeletePost():

    - Pronalazi post koji se briše.
    - Poziva DELETE endpoint:
        - /api/posts/{id}/personal ako je korisnik autor.
        - /api/posts/{id} ako je admin.
    - Ažurira lokalni state bez ponovnog učitavanja.
- Koristi **Flowbite** komponente: Table, Pagination, Modal, Button.

Integracije

- **Redux user slice** — currentUser.
- **ErrorContext** i **SuccessContext**.
- **handleApiError** — obrada neuspeha API poziva.

Redux

1. Šta je uopšte Redux?

Redux je **centralizovani menadžer stanja**.
Zamisli ga kao *jedan veliki objekat (store)* koji sadrži sve što aplikacija zna o:

- trenutnom korisniku,
- temi (light/dark),
- notifikacijama,
- i svemu što želiš da deliš između komponenti.

Dakle, umesto da:

- App.jsx šalje propsove u Header, pa Header u UserMenu, pa UserMenu u Avatar...
  ti jednostavno **uzimaš podatke iz globalnog store-a** bilo gde u aplikaciji pomoću: const { currentUser } = useSelector((state) => state.user)

- I menjaš ih pomoću:

const dispatch = useDispatch()

dispatch(signInSuccess(user))

2. Šta radi createSlice u userSlice.js

Ovo je moderna Redux Toolkit (RTK) verzija Redux-a — **pojednostavljuje sve**.

initialState

Početno stanje tvog user segmenta.
U startu nema korisnika (null), nema greške (null), i ništa se ne učitava (false).

createSlice

Ovaj deo: kreira **mini Redux modul** koji sadrži i *state* i *funkcije za promenu tog state-a (reducere)*.

Reduceri (metode za menjanje stanja)

Npr:

```
signInStart: (state) => {
  state.loading = true
  state.error = null
}
```

Znači: kad krene logovanje — aktiviraj loader i poništi greške.

signInSuccess

Ovo je srce — ono što se dešava kad backend vrati uspešan login:

1. prima payload iz dispatch-a, koji sadrži podatke iz JWT tokena,
2. mapira ga u *frontend-friendly* format (camelCase),
3. snima u globalni state.currentUser.

Sada svaka komponenta u aplikaciji zna ko je prijavljen, bez prosleđivanja propsova. Tvoj Header, Dashboard, PrivateRoute — svi koriste **isti podatak iz Redux-a**.

Ostali reducer-i:

Reducer	Namena
signInFailure	Ako login padne — upisuje poruku greške
updateUserSuccess	Ažurira currentUser kad korisnik promeni podatke
updateProfilePictureSuccess	Menja samo profilnu sliku
deleteUserSuccess / deleteUserFailure	Briše korisnika iz globalnog state-a
signoutSuccess	Resetuje sve na početno stanje (kao odjava)

Bitno: Redux sam po sebi *ne komunicira sa API-em*.
API pozivi se rade u komponentama (SignIn.jsx, UpdateProfile.jsx, itd.),
a **rezultat** se prosleđuje kroz dispatch(signInSuccess(data)).

3. redux/store.js — centralno čvorište

Ovaj deo pravi “mozak” aplikacije.

combineReducers

Spaja sve pojedinačne slice-ove:

persistReducer i persistStore

Tu nastupa magija: **podaci se čuvaju u localStorage**.

```
const persistConfig = {
  key: 'root',
```

```
  storage,
  version: 1
}
```

Ovo kaže: “sve iz Redux-a sačuvaj pod ključem root u localStorage”.

Zato se tvoj korisnik **ne odjavljuje kad refreshuješ stranicu** — currentUser ostaje u memoriji.

configureStore

Ovim se kreira globalni store.
serializableCheck: false — isključuje proveru da li su svi podaci serializabilni (npr. neki contexti, error objekti, itd.).

4. notificationsSlice.js

Vrlo jednostavan slice:

Kada SignalR ili backend pošalju nove notifikacije,
pozivaš dispatch(setUnreadCount(novaVrednost)) i Header automatski prikazuje broj.

5. themeSlice.js

Minimalističan i efikasan:

Samo menja klasu theme u ThemeProvider.jsx.
Redux se koristi jer želiš da ceo frontend zna koja je tema, a ne samo jedna komponenta.

Kratak sažetak Redux arhitekture u tvom projektu

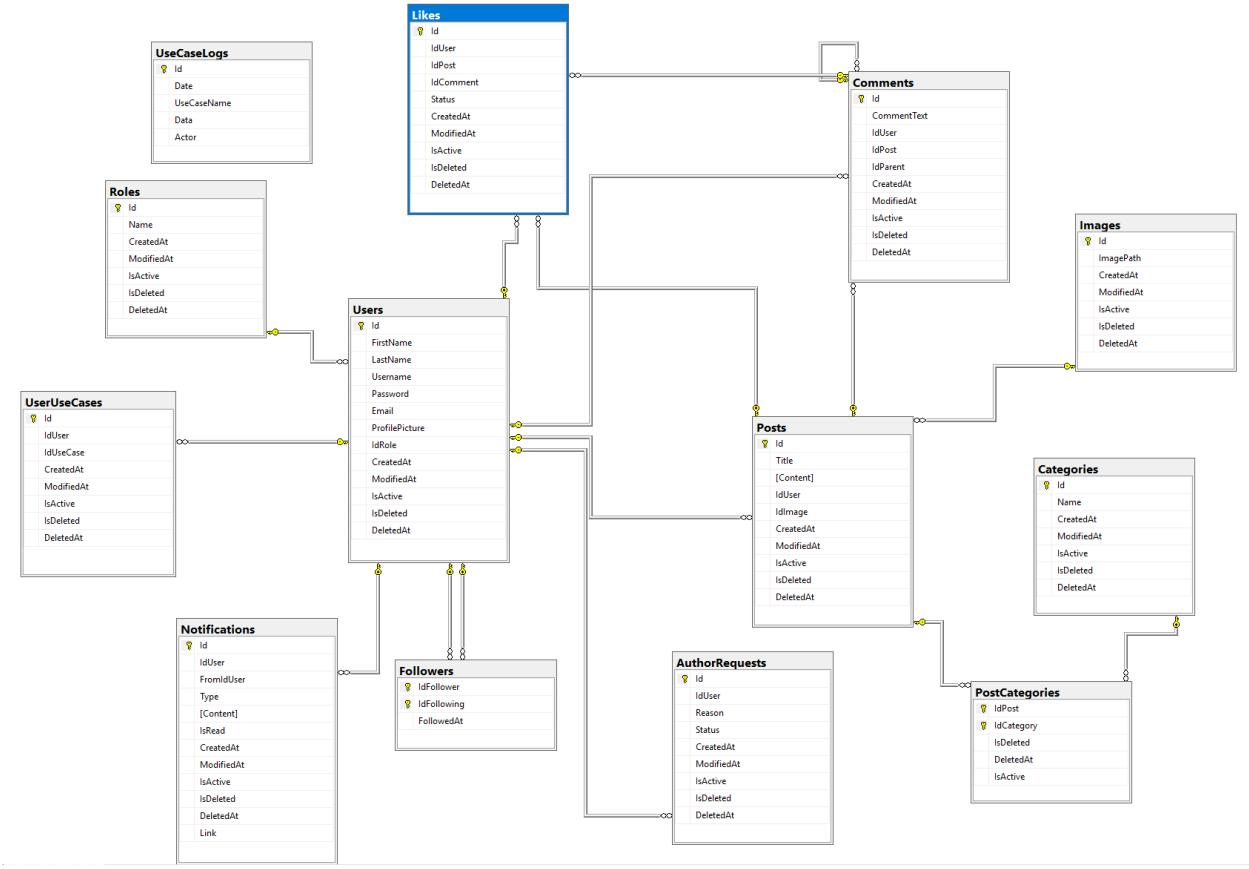
Deo	Opis

Deo	Opis
Redux Toolkit	pojednostavljena verzija Redux-a sa manje koda
createSlice()	definiše deo globalnog state-a i njegove akcije
configureStore()	spaja sve slice-ove i pravi centralni store
persistReducer()	čuva podatke u localStorage (trajni login, tema)
useSelector()	čitanje iz Redux-a u bilo kojoj komponenti
useDispatch()	slanje akcija koje menjaju globalni state

Arhitektura sistema – zaključak i analiza

1. Pregled celokupne arhitekture

Dijagram baze podataka



Projekat **My_Blog** je koncipiran kao **moderni web sistem** baziran na principima **višeslojne (layered) arhitekture**.
Sistem se sastoji od sledećih slojeva:

Sloj	**Opis**

Sloj	Opis
Domain	Sloj koji definiše osnovne entitete i interfejse aplikacije. Sadrži čiste poslovne modele, bez zavisnosti od infrastrukture.
Application	Implementira poslovnu logiku kroz use-case-ove (CQRS pristup), komandne i query handlere, validatore i servise.
EFDataAccess	Odgovoran za pristup bazi podataka korišćenjem Entity Framework-a i konfiguraciju modela.
Implementation	Konkretna implementacija servisa, handlera i komunikacija između slojeva.
API	ASP.NET Core Web API sloj koji obezbeđuje RESTful endpoint-e za komunikaciju sa frontendom.
Client (React)	Frontend deo aplikacije zadužen za interakciju sa korisnikom i prikaz podataka u realnom vremenu.

Aplikacija se zasniva na **čistoj separaciji odgovornosti**, čime se postiže visoka održivost i skalabilnost.

2. Frontend arhitektura (Client sloj)

2.1 Struktura projekta

Frontend deo je razvijen u **React-u**, uz korišćenje modernog alatnog lanca:

- **Vite** kao bundler za brze buildove i hot-reload.
- **Redux Toolkit** za centralizovano upravljanje stanjem.
- **React Router v6** za definisanje i zaštitu ruta.
- **TailwindCSS** i **Flowbite React** za UI komponentni sistem.
- **Firebase** za OAuth autentifikaciju putem Google naloga.
- **SignalR** za real-time notifikacije.
- **React Quill** kao rich text editor za uređivanje sadržaja postova.

Struktura frontenda logično je podeljena po funkcionalnim celinama:

- /pages – glavne stranice (Home, PostPage, SignIn, SignUp, Dashboard itd.)
- /components – reupotrebljive UI komponente (PostCard, Header, Footer, itd.)
- /contexts – konteksti za globalne UI poruke (ErrorContext, SuccessContext, NotificationsContext).
- /redux – globalni store i slice-ovi za temu, korisnika i notifikacije.
- /utils – pomoćne funkcije (API error handleri, formatiranje, validacije).

3. Globalno stanje i logika – Redux i Context

Aplikacija koristi **kombinovani pristup upravljanju stanjem**:
Redux za podatke od poslovnog značaja (korisnik, tema, notifikacije),
React Context za UI notifikacije (greške, uspešne poruke, real-time feed).

3.1 Redux

- **User slice** upravlja login procesom, profilom, update-ima i odjavom.
  Zahvaljujući **Redux Persist**, stanje korisnika (token, profil) se čuva i nakon refresh-a.
- **Theme slice** kontroliše izgled aplikacije (light / dark) i koristi se u ThemeProvider komponenti.
- **Notifications slice** održava broj nepročitanih notifikacija i sinhronizuje ga sa SignalR konekcijom.

Redux omogućava **jedinstveni izvor istine (Single Source of Truth)**, čime se eliminiše potreba za prosleđivanjem propsova između komponenti.

3.2 React Context

- **ErrorContext** i **SuccessContext** obezbeđuju centralizovan sistem za prikaz grešaka i poruka uspeha.
  Na taj način svaki deo aplikacije može da pozove showError() ili showSuccess() bez dupliranja logike.
- **NotificationsContext** kombinuje **API fetch** i **SignalR real-time konekciju** za dinamičko ažuriranje notifikacija.

Ovaj spoj Redux + Context pristupa daje optimalan balans između:

- *predvidljivosti stanja (Redux)*

- *i fleksibilnosti prikaza (Context)*

4. Komunikacija Frontend–Backend

Frontend komunicira sa ASP.NET API slojem isključivo preko REST endpointa (/api/...). Komunikacija se odvija pomoću **fetch()** zahteva i **Bearer token** autentifikacije.

U slučaju grešaka, koristi se pomoćna funkcija handleApiError(response, showError) koja:

- proverava HTTP status,
- vraća prilagođenu poruku korisniku,
- i automatski prikazuje UI upozorenje kroz ErrorContext.

Za real-time notifikacije koristi se **SignalR Hub**:

- Frontend se povezuje na /api/notificationsHub.
- Svaki korisnik se pridružuje grupi po svom ID-ju (JoinGroup(currentUser.id)).
- Kada backend pošalje poruku, frontend reaguje i dodaje novu notifikaciju u lokalni state.

Ovim pristupom sistem podržava **real-time iskustvo bez osvežavanja stranice**.

5. Autentifikacija i autorizacija

Postoje dva pristupa logovanju:

1. **Klasičan login** putem korisničkog imena i lozinke (/api/login).
2. **Google OAuth** putem Firebase servisa (/api/auth).

JWT se čuva u **localStorage**, odakle ga Redux preuzima i koristi u Authorization headeru.
Komponente PrivateRoute i OnlyRolePrivateRoute štite rute u zavisnosti od korisničke uloge (Admin, Author).

6. UI logika i korisničko iskustvo (UX)

Frontend koristi **Flowbite React** komponente i **TailwindCSS** klase za moderan, responzivan i intuitivan dizajn.
Poruke o greškama i uspehu su **pozicionirane i automatski nestaju** posle određenog vremena, čime se održava čisto korisničko iskustvo.

7. SOLID principi

U okviru razvoja **My_Blog** aplikacije primenio sam principe **SOLID** arhitekture, sa ciljem da kod bude **lak za održavanje, testiranje i proširivanje**.
Projekat koristi **CQRS pristup**, **Clean Architecture** i **Dependency Injection**, što prirodno podstiče primenu ovih principa.

1. S – Single Responsibility Principle (SRP)

"A class should have only one reason to change."

- EFCreatePostCommand ima *jednu* odgovornost: kreira post.
    - Validacija: delegirana u CreatePostValidator
    - Logika: kreiranje entiteta i transakcije
    - Notifikacije: delegirane u INotificationService
- PostsController samo prima request i prosleđuje ga UseCaseExecutor-u – ne zna ništa o detaljima logike.
- Slojevi projekta:
    - Domain → model
    - Application → definicije use-case-ova
    - Implementation → konkretne implementacije
    - API → endpointi i konfiguracija
    - EFDataAccess → DB sloj

Svaki sloj ima samo jednu svrhu — i to je najčistija implementacija SRP-a.

Na primer, svaki *use-case handler* radi samo jednu stvar (npr. kreira post, markira notifikaciju kao pročitanu), dok repozitorijumi služe isključivo za pristup podacima.

2. O – Open/Closed Principle (OCP)

"Classes should be open for extension, but closed for modification."

Svaki use case ima interfejs u Application i implementaciju u Implementation.
Želiš novi način da kreiraš postove (npr. import iz CSV-a)?
Samo dodaš ICreatePostFromCsvCommand i EFCreatePostFromCsvCommand — ne diraš stare klase.

Delimično je ispoštovan — sistem lako proširujem dodavanjem novih handlera bez menjanja postojećeg koda.
Jedini deo koji bih unapredio je deo sa NotificationType enumeracijom — tu bih uveo apstrakcije da bi dodavanje novih tipova bilo još fleksibilnije.

3. L – Liskov Substitution Principle (LSP)

"Objects of a superclass should be replaceable with objects of its subclasses without breaking the application."

Gde ga koristiš:

- ICreatePostCommand može biti implementiran i sa EF, i sa Dapperom, i sa mock bazom — PostsController ne bi morao da se menja.
- IEmailSender može biti SMTPEmailSender, SendGridEmailSender, itd.
- IUseCaseLogger može biti EFDatabaseLogger ili FileLogger.

Svi ovi primeri zadovoljavaju LSP jer "visok nivo" (API, Application) ne zna *koja konkretna klasa* stoji iza interfejsa.

4. I – Interface Segregation Principle (ISP)

"Clients should not be forced to depend upon interfaces they do not use."

Vrlo čisto sprovedeno:

- Svaki command ima svoj minimalan interfejs (ICreatePostCommand, IUpdatePostCommand, …).
- Interfejsi nisu *preširoki* — imaju samo ono što treba jednom use case-u.
- IEmailSender, INotificationService, IApplicationActor — svi su fokusirani i mali.

To ti omogućava da testiraš svaki deo ponašanja izolovano — bez "fat" interfejsa koji bi te gušio.

5. D – Dependency Inversion Principle (DIP)

"High-level modules should not depend on low-level modules. Both should depend on abstractions."

Primer:

- PostsController ne zna ništa o EF-u → samo zna da prima ICreatePostCommand.
- EFCreatePostCommand zavisi od interfejsa (INotificationService, IApplicationActor), ne od konkretnih implementacija.
- Sve se injectuje kroz konstruktor i registruje kroz DI u Startup.cs i APIExtension.

Korišćenjem Dependency Injection-a, kontroleri i handleri su potpuno odvojeni od implementacionih detalja, što kod čini fleksibilnim i testabilnim.

DIP i SRP zajedno čine tvoj kod izuzetno fleksibilnim — mogao bi da menjaš bazu, logger, ili način slanja emaila *bez promene* u bilo kom kontroleru ili use-case-u.

8. Zaključna analiza

Sistem **My_Blog** predstavlja **stabilan, modularan i skalabilan projekat** koji već sada ispunjava standarde moderne full-stack arhitekture.
Slojevi su jasno odvojeni, komunikacija između backend-a i frontenda je dosledna, a globalno stanje je efikasno sinhronizovano putem Redux-a i Context API-ja.

Upotreba SignalR-a dodaje **real-time dimenziju**, dok Redux Persist i JWT autentifikacija omogućavaju **trajnu i bezbednu sesiju korisnika**.
Frontend deo je organizovan po **čistim arhitektonskim principima**, a kod je čitljiv, testabilan i spreman za dalji razvoj.

Drugim rečima — projekat My_Blog je **više od bloga**: to je kompletna platforma koja pokazuje razumevanje **modernog React + ASP.NET ekosistema**.
Sledeći korak bio bi integracija naprednih Redux asinkronih akcija, unapređenje performansi i vizuelna poliranost korisničkog interfejsa.