Razvoj aplikacije za upravljanje projektima koristeći SQL, .NET Web API i Angular

Josipović, Fran

SVEUČILIŠTE JOSIPA JURJA STROSSMAYERA U OSIJEKU
FAKULTET ELEKTROTEHNIKE, RAČUNARSTVA I INFORMACIJSKIH TEHNOLOGIJA OSIJEK

Sveučilišni prijediplomski studij Računarstvo

Razvoj aplikacije za upravljanje projektima koristeći SQL, .NET Web API i Angular

Završni rad

Fran Josipović

Osijek, 2024.

FERIT
FAKULTET ELEKTROTEHNIKE, RAČUNARSTVA I INFORMACIJSKIH TEHNOLOGIJA **OSIJEK**

Obrazac Z1P: Obrazac za ocjenu završnog rada na sveučilišnom prijediplomskom studiju

Ocjena završnog rada na sveučilišnom prijediplomskom studiju

Ime i prezime pristupnika:	Fran Josipović
Studij, smjer:	Sveučilišni prijediplomski studij Računarstvo
Mat. br. pristupnika, god.	R4651, 28.07.2021.
JMBAG:	0165091212
Mentor:	izv. prof. dr. sc. Ivica Lukić
Sumentor:	
Sumentor iz tvrtke:	
Naslov završnog rada:	Razvoj aplikacije za upravljanje projektima koristeći SQL, .NET Web API i Angular
Znanstvena grana završnog rada:	**Informacijski sustavi (zn. polje računarstvo)**
Zadatak završnog rada:	Razviti web aplikaciju koja se sastoji od slijedećih tehnologija: SQL za bazu, .NET web api za backend i angular za frontend. Funkcionalnost aplikacije je upravljanje projektima za timsku suradnju, slično kao aplikacija OpenProject. Tema rezervirana za: Fran Josipović

Datum prijedloga ocjene završnog rada od strane mentora:	30.08.2024.
Prijedlog ocjene završnog rada od strane mentora:	Izvrstan (5)
Datum potvrde ocjene završnog rada od strane Odbora:	11.09.2024.
Ocjena završnog rada nakon obrane:	Izvrstan (5)
Datum potvrde mentora o predaji konačne verzije završnog rada čime je pristupnik završio sveučilišni prijediplomski studij:	12.09.2024.

FERIT
FAKULTET ELEKTROTEHNIKE, RAČUNARSTVA I INFORMACIJSKIH TEHNOLOGIJA **OSIJEK**

IZJAVA O IZVORNOSTI RADA

Osijek, 12.09.2024.

Ime i prezime Pristupnika:	Fran Josipović
Studij:	Sveučilišni prijediplomski studij Računarstvo
Mat. br. Pristupnika, godina upisa:	R4651, 28.07.2021.
Turnitin podudaranje [%]:	7

Ovom izjavom izjavljujem da je rad pod nazivom**: Razvoj aplikacije za upravljanje projektima koristeći SQL, .NET Web API i Angular**

izrađen pod vodstvom mentora izv. prof. dr. sc. Ivica Lukić

i sumentora

moj vlastiti rad i prema mom najboljem znanju ne sadrži prethodno objavljene ili neobjavljene pisane materijale drugih osoba, osim onih koji su izričito priznati navođenjem literature i drugih izvora informacija.
Izjavljujem da je intelektualni sadržaj navedenog rada proizvod mog vlastitog rada, osim u onom dijelu za koji mi je bila potrebna pomoć mentora, sumentora i drugih osoba, a što je izričito navedeno u radu.

Potpis pristupnika:

SADRŽAJ

1. UVOD

U modernom poslovnom okruženju učinkovito upravljanje projektima predstavlja ključni faktor za uspjeh organizacija. Kod mnogih projekata veliki problem predstavlja podjela i zadavanje radnih zadataka. Loša organizacija i podjela zadataka uvelike mogu utjecati na uspješnost svakog projekta stoga je važno imati odgovarajući alat koji će omogućiti jednostavno organiziranje i raspodjelu radnih zadataka. Sukladno opisanom problemu zadatak završnog rada je razvoj aplikacije za dodjeljivanje zadataka i organizaciju poslova unutar tima na određenom projektu. Aplikacija se sastoji od baze podataka, RESTful API-ja i korisničkog sučelja. Programski jezici koji su korišteni su C# (koristi se u .NET razvojnom okviru), Typescript (koristi se za definiranje funkcionalnosti korisničkog sučelja), HTML (definiranje izgleda korisničkog sučelja) i SCSS (uređivanje korisničkog sučelja) te strukturni upitni jezik SQL (skladištenje i procesiranje informacija u relacijskim bazama podataka).

U prvom sljedećem poglavlju prikazana su slična i postojeća rješenja. U drugom poglavlju detaljnije se upoznaju korištene tehnologije za izradu web aplikacije. U trećem poglavlju prikazana je specifikacija, implementacija i usporedba različitih arhitekturnih pristupa pri razvoju RESTful API-ja. U četvrtom poglavlju prikazan je izgled i glavne funkcionalnosti aplikacije.

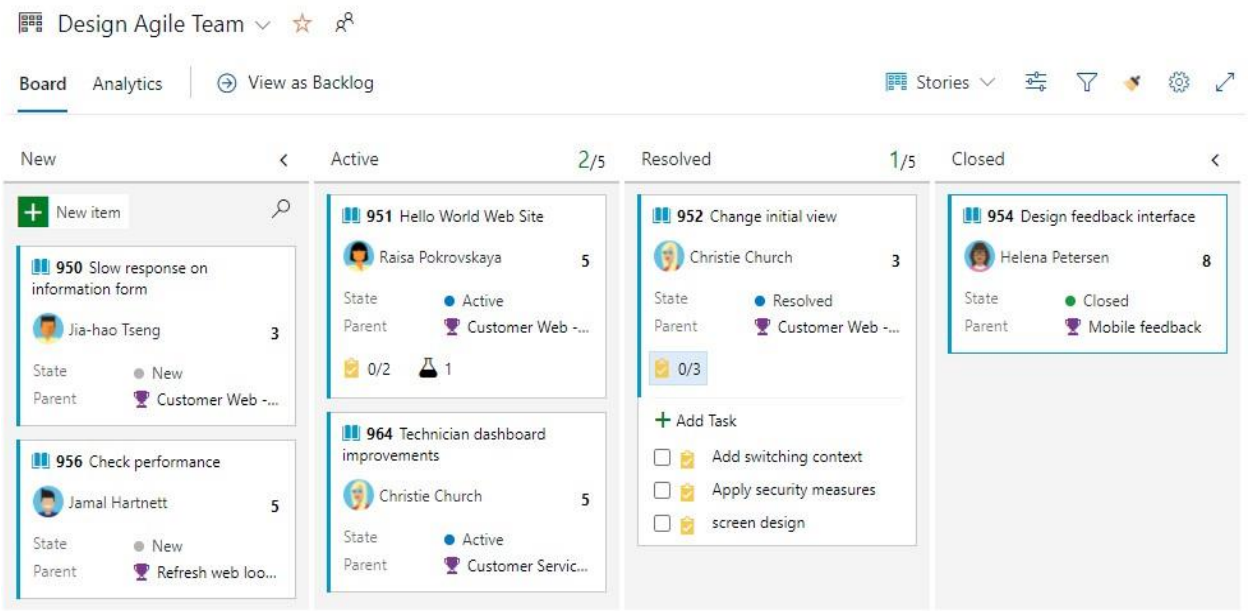
1.1. Zadatak završnog rada

Zadatak završnog rada će biti razvoj aplikacije za upravljanje projektima koristeći SQL, .Net Web APU i Angular. Za pohranu podataka i organiziranje relacija među podatcima koristit će se MsSQL (Microsoftova verzija SQL-a), za razvoj RESTful API-ja koristit će se razvojni okvir .NET Web API čija će implementacija biti pisana C# programskim jezikom. U sklopu .NET Web API okvira koristit će se razvojni okvir koji će omogućiti komunikaciju između RESTful API-ja i baze podataka, Entity Framework. Za razvoj korisničkog sučelja koristit će se Angular razvojni okvir čija će implementacija biti pisana TypeScript programskim jezikom.

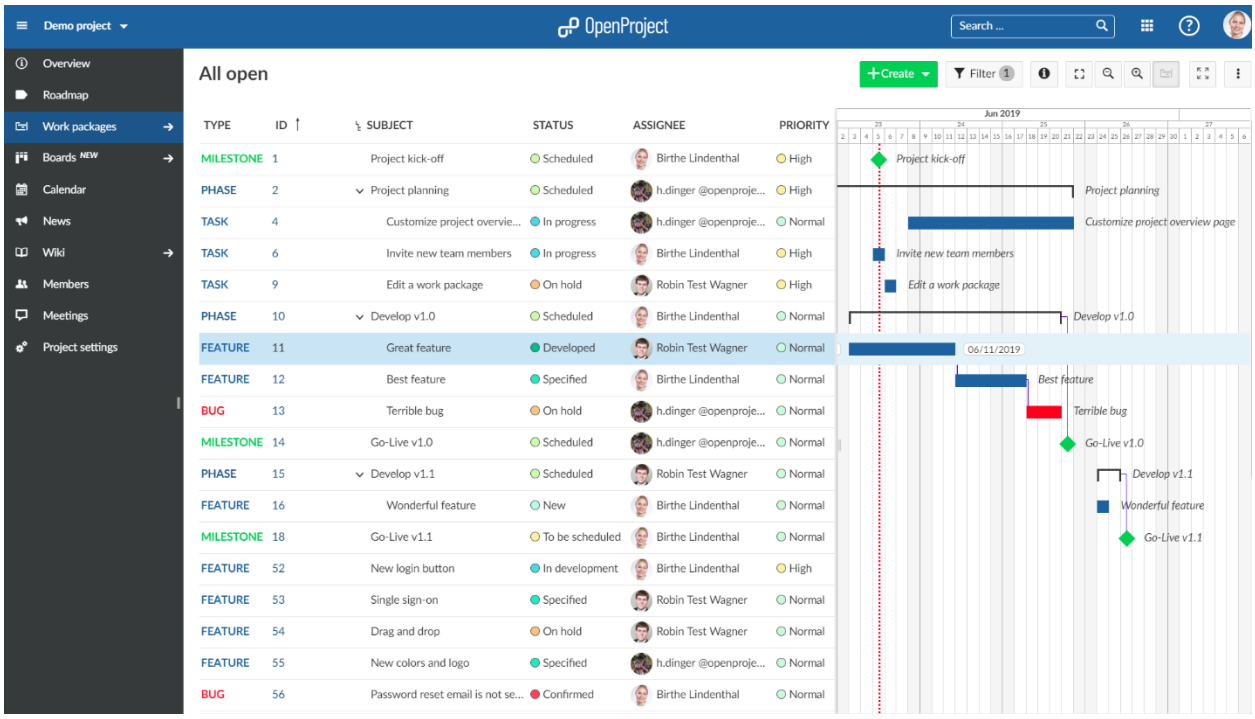
2. PREGLED PODRUČJA TEME

Danas postoje rješenja za ovakve situacije kao i za razno razne stvari pa je teško osmisliti jedinstvenu ideju za aplikaciju. Izradom ove aplikacije htio sam omogućiti i prikazati osnovnu funkcionalnost rješavanja opisanog problema.

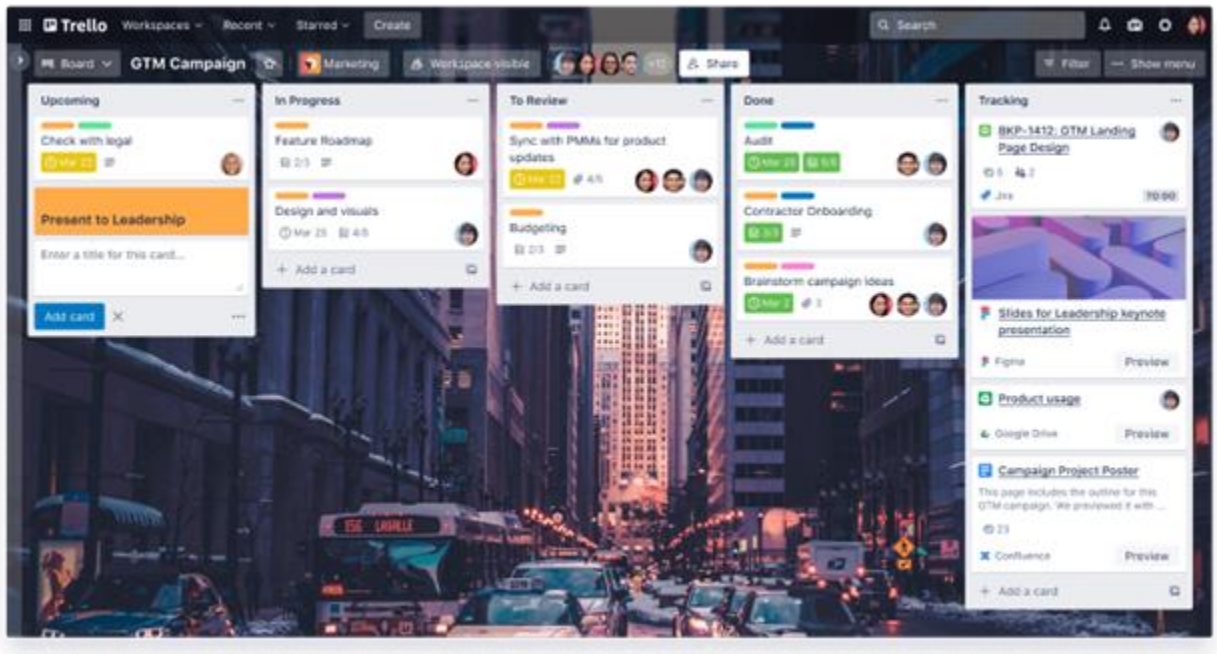
Prije izrade rada pregledana su već postojeća rješenja. Jedne od najpoznatijih aplikacija za upravljanje projektima su *OpenProject* [1], *Azure DevOps* [2]. *Jira* [3] i *Trello* [4]. Najveću inspiraciju za izradu ovog projekta potakla je aplikacija *Azure DevOps*. Slika 2.1. prikazuje isječak aplikacije *Azure DevOps*. Na njoj se može vidjeti *kanban* ploča, o čemu će se kasnije govoriti u radu. Slika 2.2. prikazuje *OpenProject* aplikaciju, Slika 2.3. prikazuje *Trello* aplikaciju i slika 2.4. prikazuje *Jira* aplikaciju.



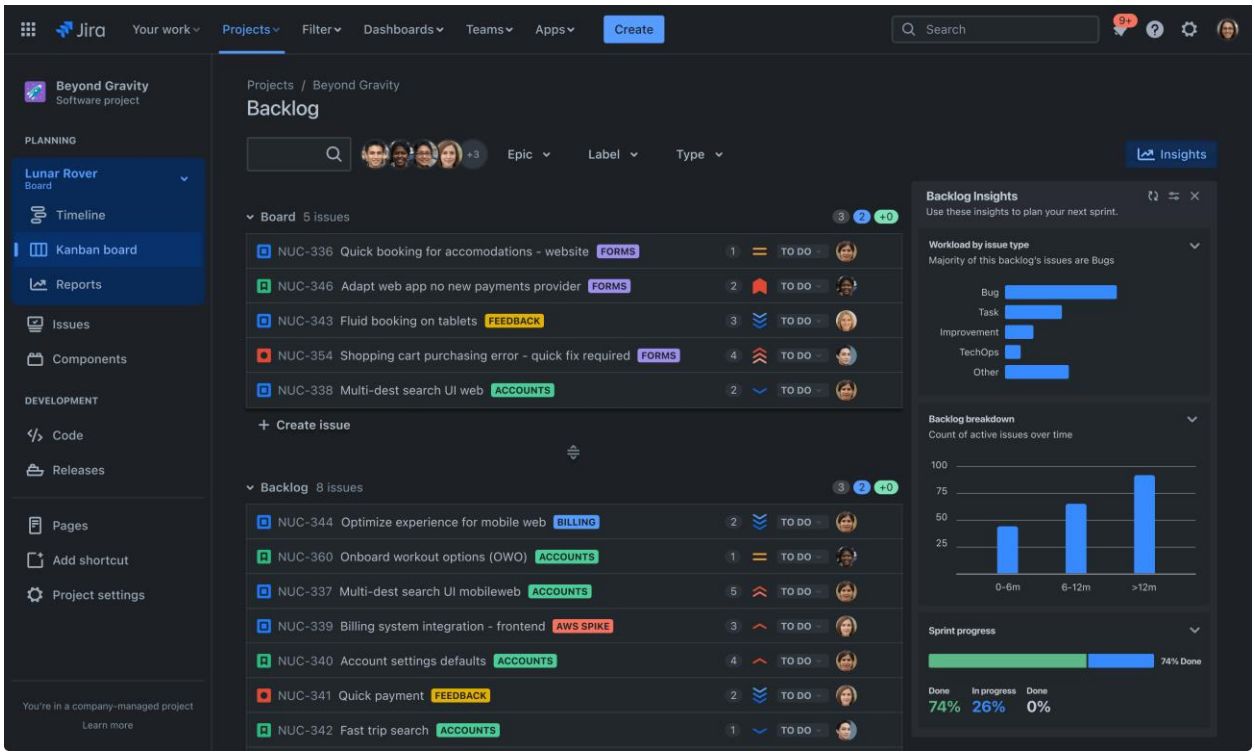
Sl. 2.1. AzureDevops kanban ploča, preuzeto iz [5]



Sl. 2.2. OpenProject aplikacija, preuzeto iz [1]



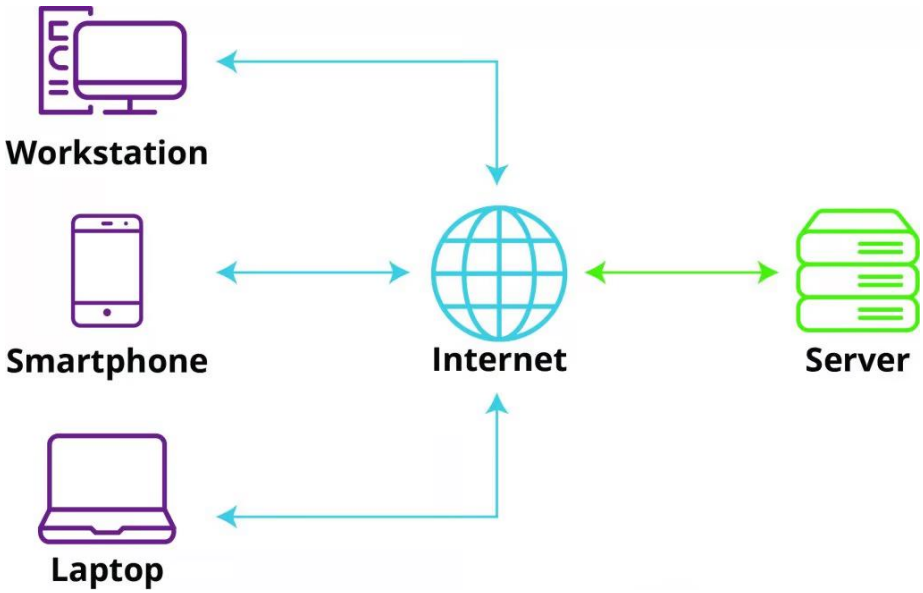
Sl. 2.3. Trello aplikacija, preuzeto iz [6]



Sl. 2.4. Jira aplikacija, preuzeto iz [3]

3. KORIŠTENE TEHNOLOGIJE

Za izradu aplikacije predstavljene u završnom radu korištena je klijent-poslužitelj arhitektura. Klijent-poslužitelj arhitektura omogućava učinkovito razdvajanje obaveza. Poslužiteljska strana obično je zadužena za sigurnost, autentifikaciju korisnika, enkripciju podataka i slično. S druge pak strane, to omogućava programerima koji razvijaju klijentske aplikacije da se mogu više usredotočiti na korisničko iskustvo [7]. Slika 3.1. prikazuje klijent-poslužitelj arhitekturu.



Sl. 3.1. klijent-poslužitelj arhitektura, preuzeto iz [8]

Ovakav oblik se također može precizirati kao “SPA (eng. Single Page Aplication) arhitektura sa RESTful API-jem”. U tom modelu klijent (eng. Frontend) je SPA, što znači da se aplikacija učitava jednom i dinamički ažurira sadržaj putem API poziva, bez potrebe za učitavanjem cijele stranice. Backend (poslužitelj) izlaže vlastiti RESTful API, koji klijent koristi za komunikaciju i pristup podacima. Baza podataka je centralizirana na poslužitelju i koristi se za skladištenje i upravljanje podacima. Za razumijevanje ove arhitekture važno je objasniti šta je RESTful API koji se spominje. RESTful API je arhitektonski stil za programsko sučelje aplikacija koje koristi HTTP zahtjeve za pristup i korištenje podataka. RESTful arhitektura pruža glavne i osnovne operacije poput GET, PUT, POST, DELETE koje se odnose na operacije čitanja, ažuriranja, kreiranja I brisanja podataka skladištenih u bazi podataka [9]. Tehnologije pomoću kojih je ostvarena aplikacija završnog rada u navedenoj arhitekturi su Angular (klijentska strana), .NET (poslužitelj) i MsSQL (baza podataka).

3.1. Angular

Angular je razvojni okvir otvorenog koda (eng. open-source) razvijen od strane Google-a, namijenjen je za izradu dinamičkih web aplikacija. Prvobitno je izdan 2010. Godine pod imenom *AngularJs*. Nakon toga je značajno evoluirao te je verzija poznata kao Angular predstavljena 2016. godine. Programski jezik koji se koristi u Angular razvojnom okviru je *TypeScript*, jezik vrlo sličan poznatom *JavaScriptu* koji podržava definiranje tipova varijabli, funkcija, struktura i složenih objekata [10]. Neke od ključnih karakteristika Angulara su:

- Komponentna arhitektura – komponenti pristup omogućava dijeljenje aplikacije na manje komponente koje se mogu koristiti u više navrata u različitim dijelovima aplikacije, to pridonosi modularnosti aplikacije i jednostavnom slaganju korisničkog sučelja
- Dvostrano povezivanje podataka – ova funkcionalnost omogućava sinkronizaciju podataka između modela (podataka) i korisničkog sučelja. Npr. kada se vrijednost promijeni u korisničkom sučelju automatski će se ažurirati u modelu i obrnuto, bez potrebe za dodatnim ručnim kodiranjem.
- Ubrizgavanje ovisnosti (eng. Dependency injection) – ovo je jedna od najvažnijih značajki ovog razvojnog okvira jer omogućava jednostavno ubrizgavanje ovisnosti među različitim komponentama aplikacije. Pogoduje ponovnoj upotrebi koda, jednostavnom razvoju i testiranju aplikacije.
- Usmjeravanje (eng. Routing) – Angular omogućava sistem za upravljanje navigacijom unutar aplikacije, time korisnicima pruža mogućnost usmjeravanja između različitih stranica i dijelova aplikacije bez ponovnog učitavanja čitave stranice što je jedna od specifičnosti Single-page aplikacija
- Templating – koristi vlastiti sistem koji omogućava dinamičko generiranje sadržaja na osnovu modela podataka

3.2. .NET Web aPI

.NET Web API je robustan razvojni okvir stvoren od strane Microsofta. .NET Web API proizlazi kao jedna od verzija Microsoftovog .NET Framework-a koji je u proizvodnju pušten prvi puta 2002. godine. .NET Web API dolazi sa puno značajki koje omogućuju stvaranje potpunih servisa baziranih na HTTP-u [11]. Zbog svoje robusnosti, pouzdanosti, jednostavnosti i sigurnosti koristi se u mnogim firmama koje se bave razvojem programske podrške te sam ga zbog tog razloga i ja odlučio iskoristiti za izradu ovog rada. .NET Web API razvojni okvir pruža mnogo značajki poput

autorizacije, autentifikacije, kreiranje API krajnjih točaka (eng. Endpointova), jednostavno rukovanje iznimkama, serijalizacija objekata, također dolazi sa značajkama koje olakšavaju kontinuiranu integraciju i kontinuiranu isporuku (eng. CI/CD).

3.3. Microsoft SQL Server

Microsoft SQL Server, skraćeno MsSQL, je sustav za upravljanje relacijskim bazama podataka kojeg je razvio Microsoft [12]. Prva verzija je izdana 1989. godine, a od tada postaje jedan od najpopularnijih sustava za upravljanje relacijskim bazama podataka na tržištu. MsSQL podržava Transact-SQL koje je proširenje standardnog SQL jezika koje omogućava napredne funkcionalnosti kao što su proceduralna logika, deklaracija varijabli, upravljanje pogreškama i transakcije. Neke od ključnih karakteristika MsSQL-a su:

- Visoke performanse i skalabilnost – dizajniran je za rukovanje velikim količinama podataka i velikim brojem istovremenih korisnika.
- Sigurnost – MsSQL nudi napredne sigurnosne značajke: enkripcija podataka, kontrola pristupa na razini korisnika i uloga te integracija sa *Active Directory* za centraliziranu autentifikaciju.
- Jednostavna integracija sa drugim Microsoftovim razvojnim alatima – neki od njih su: *Azure, Power BI, Visual Studio* i za ovaj rad najvažnije jednostavno povezivanje sa .NET Web API razvojnim alatom

Zbog stabilnosti, performansi i velikog broja značajki, MsSQL se često koristi u raznim industrijama za poslovne aplikacije, skladišta podataka i web aplikacije pa sam ga zbog toga i odabrao za izradu ovog završnog rada.

4. TEORIJSKA POZADINA I IMPLEMENTACIJA

Radi boljeg razumijevanja ove aplikacije važno je proučiti teorijsku podlogu vezanu za sami cilj aplikacije.

4.1. Agile model

Ova aplikacija temelji se na *agile* modelu. *Agile* model je inkrementalni i iterativni proces razvoja programske podrške. Stavlja manji naglasak na unaprijed i strogo definirane planove i više se oslanja na neformalnu suradnju, koordinaciju i učenje [13]. Unaprijed definira broj, trajanje i opseg svake iteracije. Svaka iteracija smatra se kratkim okvirom *agile* procesa, svaka iteracije uglavnom traje od dva do četiri radna tjedna. Agilni model dijeli zadatke u određenom vremenu kako bi se u jednoj iteraciji pružila odgovarajuća funkcionalnost programske podrške koja se razvija. Podjela projekta na male dijelove tj. iteracije pomaže minimizirati rizik projekta i ukupno vrijeme isporuke samog projekta. Evo nekih od prednosti agilnog modela:

- Fleksibilnost i prilagodljivost – agilni model omogućava jednostavne prilagodbe promjenama zahtjeva tijekom razvoja što omogućava brzi odgovor na promjene na tržištu ili korisničkim potrebama.
- Brza isporuka funkcionalnosti – Svaka iteracija uglavnom rezultira isporukom određene funkcionalnosti. Ako se funkcionalnost u iteraciji ne dovrši u potpunosti, lagano se dio funkcionalnosti može prenijeti na iduću iteraciju.
- Bolja uključenost korisnika – korisnici su često uključeni u proces razvoja kroz redovite preglede i povratne informacije što pridonosi zadovoljstvu korisnika i konačnoj kvaliteti proizvoda.
- Kontrola rizika – zbog podijeljenosti projekta na iteracije lakše je identificirati probleme i zaostatke te je zbog toga lakše upravljati rizicima u ranijim fazama razvoja.

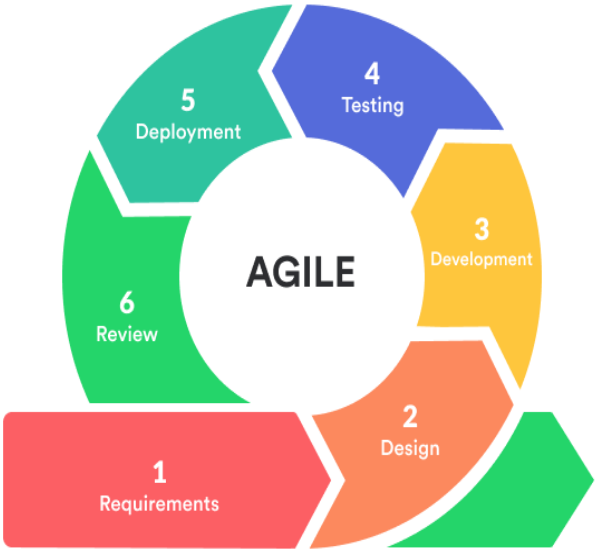
Faze agilnog modela:

- Prikupljanje zahtjeva – U ovoj fazi se definiraju zahtjevi, razgovara se o poslovnim mogućnostima te vremenu i trudu potrebnim za projekt. Analizom ovih zahtjeva može se odrediti ekonomska i tehnička izvedivost sustava.
- Dizajniranje zahtjeva – Nakon prikupljenih i određenih zahtjeva isti se dizajniraju. Procjenjuje se prioritet različitih dijelova zahtjeva. Zahtjevi se najčešće dijele na *User*

Story-e (podjedinice svakog zahtjeva), najčešće jedan *user story* pripada jednom programeru.

- Razvoj – nakon definiranja i dizajniranja zahtjeva počinje njihovo izvođenje/razvoj. Svaki *user story* se dijeli na manje zadatke. Tu postoje dva pristupa, ili developer sam sebi određuje zadatke ili mu ih određuje vođa tima. Tu dolazimo do dva važna pojma: *kanban* i *scrum*. O ta dva pojma ću govoriti kasnije.
- Test – Samo ime govori da se radi o testiranju. Služi za osiguravanje kvalitete, provjeravanje performansi sustava i prijavljuju se pogreške tijekom faze.
- Implementacija – Proizvod se isporučuje kupcu/korisniku

Slika 4.1. prikazuje faze agilnog razvoja.



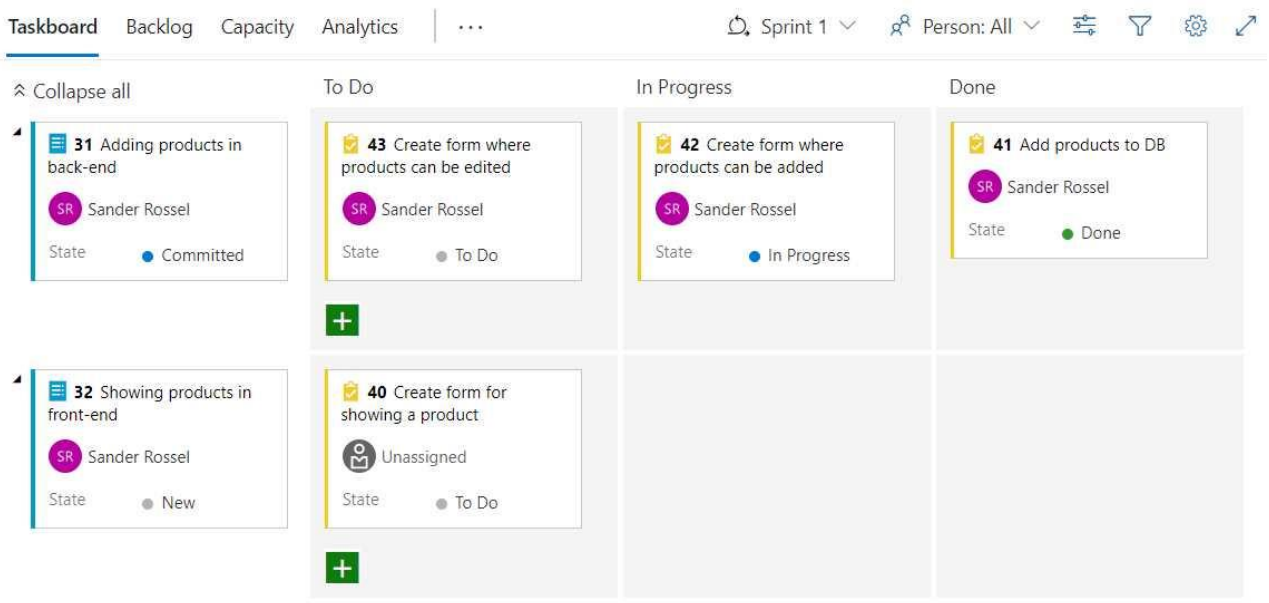
Sl. 4.1. Agile faze, preuzeto iz [14]

4.2. Scrum i kanban

Iako su *scrum* i *kanban* glavni dijelovi *agile* modela i jako su slični postoje neke razlike. *Kanban* je dobar za vizualizaciju rada: Od početka projekta definirani su zadatci koji kada se izvrše trebali bi činiti gotovi proizvod. *Kanban* ploča je podijeljena na tri glavna stupca, a to su: *Za obaviti (eng. To-do), U tijeku (eng. In progress)* i *Završeno (eng. Done)*. Programeri sami odabiru zadatke i stavljaju ih u stupac *U tijeku* te rade na njima, kada završe sa zadatkom premještaju ga u stupac *Završeno* i odabiru novi zadatak. Slika 4.2. prikazuje *kanban* ploču.

Sl. 4.2. Kanban ploča, preuzeto iz [15]

Za razliku od *kanban-a scrum* ima i podjelu po redcima, gdje svaki redak predstavlja jedan *user story*. *User story* je dodijeljen programeru te on sam definira zadatke vezane za taj *user story*. Važan dio *scrum* metodologije su dnevni *scrum* sastanci (eng. Daily Standup) na kojima programeri izvještavaju o odrađenom radu, izazovima s kojima su se susreli, planovima za taj dan te na temelju toga definiraju zadatke. Moja aplikacija je napravljena tako da prati *scrum* metodologiju rada. Slika 4.3. prikazuje *scrum* ploču.



Sl. 4.3. Scrum ploča, preuzeto iz [16]

4.3. Implementacija

U ovom poglavlju je opisana implementacija za tri glavna sastavna dijela ove aplikacije: korisničko sučelje (Angular), implementacija poslužitelja (.NET web API), konfiguracija i dizajn baze podataka.

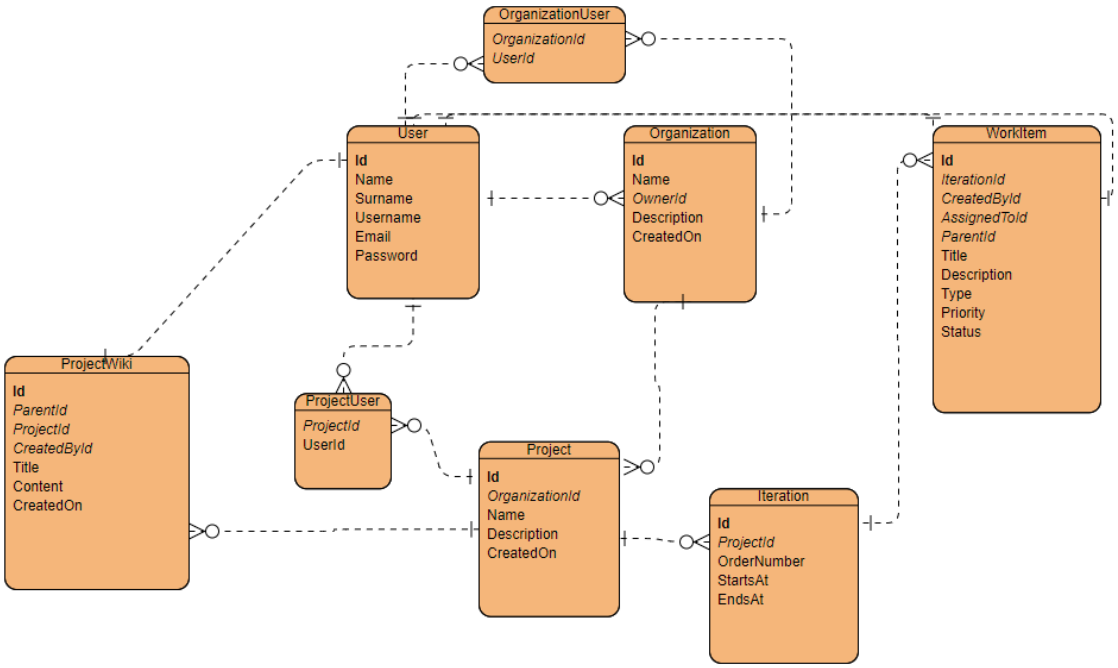
4.3.1. Baza podataka

Kako je prije spomenuto, za bazu podataka je korišten MsSQL kao upitni jezik i SSMS (eng. Sql Server Management Studio) kao razvojni alat. Da bi što bolje razumjeli bazu podataka važno je znati koji su entiteti koji tvore ovu aplikaciju. Ti entiteti su: *User*, *Organization*, *Project*, *Iteration*, *Project Wiki*, *Work Item*.

- *User* – entitet koji predstavlja korisnika, sadrži glavne informacije o korisniku te nema nikakvih vanjskih ključeva.
- *Organization* – entitet koji predstavlja organizaciju. Kao atribute sadrži polja koja opisuju organizaciju (naslov i opis organizacije) te sadrži jedan vanjski ključ koji daje vezu na vlasnika organizacije, taj ključ referencira entitet *User*
- *OrganizationUser* – pošto više *User-a* može pripadati organizaciji, ali i jedan *User* može posjedovati više organizacija imamo vezu više na prema više (N:M). Stoga nam je potrebna tzv. pivot tablica kako bi se ostvario odnos N:M

- *Project* – ovaj entitet sadrži glavne informacije o svakom projektu kao što su naziv projekta, opis projekta i vremenska štampa kada je započet. Pošto jedna organizacija može sadržavati više projekata (odnos 1:N), entitet Project ima vanjski ključ koji referencira organizaciju
- *ProjectUser* – također kao i pivot tablica *OrganizationUser* iz istog razloga postoji *ProjectUser* tablica koja omogućuje odnos N:M između *User* i *Project* entiteta.
- *ProjectWiki* – ovaj entitet služi za prikazivanje dokumentacije svakog projekta. Kao atribute sadrži: naslov, sadržaj, vremensku štampu kada je stvoren, kada je zadnji put modificiran, sadrži vanjski ključ na entitet *User* koji predstavlja tko je taj entitet stvorio, sadrži vanjski ključ kao poveznicu na projekt kojemu pripada i sadrži ključ koji referencira sami entitet *ProjectWiki* zbog funkcionalnosti koje pruža ova aplikacija. Referenca na samog sebe omogućava stvaranje stabla i odnosa roditelj dijete gdje roditelj može imati N djece i sva ta djeca mogu imati svoju djecu. To omogućuje da dokumentaciju podijelimo na manje dijelove, ali da ipak tvore smislenu cjelinu.
- *Iteration* – ovaj entitet predstavlja iteraciju u projektu. Sadržava atribute koji predstavljaju redni broj iteracije, vremensku štampu kada iteracija počinje i kada završava i naravno vanjski ključ koji referencira projekt kojemu ta iteracija pripada.
- *WorkItem* – ovo je entitet koji predstavlja jednu radnu jedinicu projekta. *WorkItem* pripada jednoj iteraciju stoga ima vanjski ključ na određenu iteraciju. Ima vanjski ključ koji predstavlja stvaratelja *WorkItem*-a, ima vanjski ključ koji referencira *User*-a kojemu je dodijeljen taj *WorkItem*. Sadrži atribute koji daju naslov, opis, vrstu (*user story, task, bug*), prioritet, status (stvoren, aktivan, završen). I ono što je najvažnije opet postoji vanjski ključ kojim entitet referencira samog sebe zbog toga ako je tip *WorkItem*-a *user story* onda on može sadržavati N *work item*-a koji su tipa *task* ili *bug*. Znači ako je *WorkItem* tipa *task* ili *bug* on će imati spremljenu vrijednost pod tim vanjskim ključem, no ukoliko je on tipa *user* story tada atribut *ParentId* neće spremati nikakvu vrijednost tj. spremati će vrijednost *null*. Naravno ovaj problem se može riješiti na drugačiji način da *WorkItem* ne referencira samog sebe, a to bi bilo tako sto bi napravili entitet *UserStory* sa pripadajućim atributima i zasebni entitet *WorkItem* koji ne bi imao referencu samog na sebe, već bi imao vanjski ključ koji referencira entitet *UserStory*

Na idućoj slici 4.4. je prikazan izgled E-R dijagrama koji prikazuje odnos između entiteta zajedno sa njihovim atributima.



Sl. 4.4. E-R dijagram

Nakon što je baza podataka teorijski i dijagramom definirana potrebno ju je izgraditi. Za izgradnju modela baze podataka koristi se DDL (eng. Data Definition Language). Prije nego što nastavim sa objašnjavanjem stvaranja baze podataka važno je objasniti što je to objektno-relacijsko mapiranje.

Objektno-relacijsko mapiranje (eng. ORM) je tehnika programiranja za pretvorbu podataka između relacijskih baza podataka i objektno orijentiranih programskih jezika. Tako se zapravo stvara baza podataka virtualnih objekata koji se mogu koristiti unutar programskog jezika. Razvojni alat koji služi za objektno-relacijsko mapiranje jest *Entity Framework*. Koristeći *Entity Framework* omogućena su dva pristupa za definiranje baze podataka. Ta dva pristupa su *„db first“* i *„code first“* pristupi. Sa *„db first“* pristupom pišemo sami sql kod za stvaranje entiteta i isti taj kod automatski izvršavamo u razvojnom okviru SSMS. Nakon što je DDL kod izvršen potrebno je pokrenuti komandu unutar komandne linije u kojoj će biti definiran *connection string* koji opisuje povezanost alata sa bazom podataka. Nakon izvršavanja komande sam *Entity Framework* će se pobrinuti da stvori odgovarajuće objekte koji odgovaraju entitetima definiranim u samoj bazi podataka. Nedostatak ovog pristupa je što sami moramo brinuti o verzioniranju baze podataka i vođenju migracija što baš i nije jednostavno i često puta može dovesti do komplikacija. Drugi pristup je puno jednostavniji, potrebno je samo u kodu definirati klase koje će predstavljati entitete i izvršiti par komandnih naredbi. Svakom promjenom klasa stvarat će se nove migracije što olakšava praćenje i verzioniranje baze podataka. Naravno ključan dio komandnih naredbi za „code

first“ pristup je *connection string* koji definira povezanost sa bazom podataka. Slika 4.5. prikazuje *connection string*. Slika 4.6. prikazuje jednu od migracija koje se automatski generiraju.

```
"ConnectionStrings": {
  "TeamOpsConnectionString": "Data Source=(localdb)\\TeamOpsDb;Database=TeamOpsDb;Connection Timeout=30; Integrated Security=true;TrustServerCertificate=True;"
},
```

Sl. 4.5. Connection string

```
public partial class additerrationsmigration : Migration
{
    /// <inheritdoc />
    0 references
    protected override void Up(MigrationBuilder migrationBuilder)
    {
        migrationBuilder.CreateTable(
            name: "Iterrations",
            columns: table => new
            {
                Id = table.Column<Guid>(type: "uniqueidentifier", nullable: false),
                ProjectId = table.Column<Guid>(type: "uniqueidentifier", nullable: false),
                OrderNumber = table.Column<int>(type: "int", nullable: false),
                StartsAt = table.Column<DateTime>(type: "datetime2", nullable: false),
                EndsAt = table.Column<DateTime>(type: "datetime2", nullable: false)
            },
            constraints: table =>
            {
                table.PrimaryKey("PK_Iterrations", x => x.Id);
                table.ForeignKey(
                    name: "FK_Iterrations_Projects_ProjectId",
                    column: x => x.ProjectId,
                    principalTable: "Projects",
                    principalColumn: "Id",
                    onDelete: ReferentialAction.Cascade);
            });

        migrationBuilder.CreateIndex(
            name: "IX_Iterrations_ProjectId",
            table: "Iterrations",
            column: "ProjectId");
    }

    /// <inheritdoc />
    0 references
    protected override void Down(MigrationBuilder migrationBuilder)
    {
        migrationBuilder.DropTable(
            name: "Iterrations");
    }
}
```

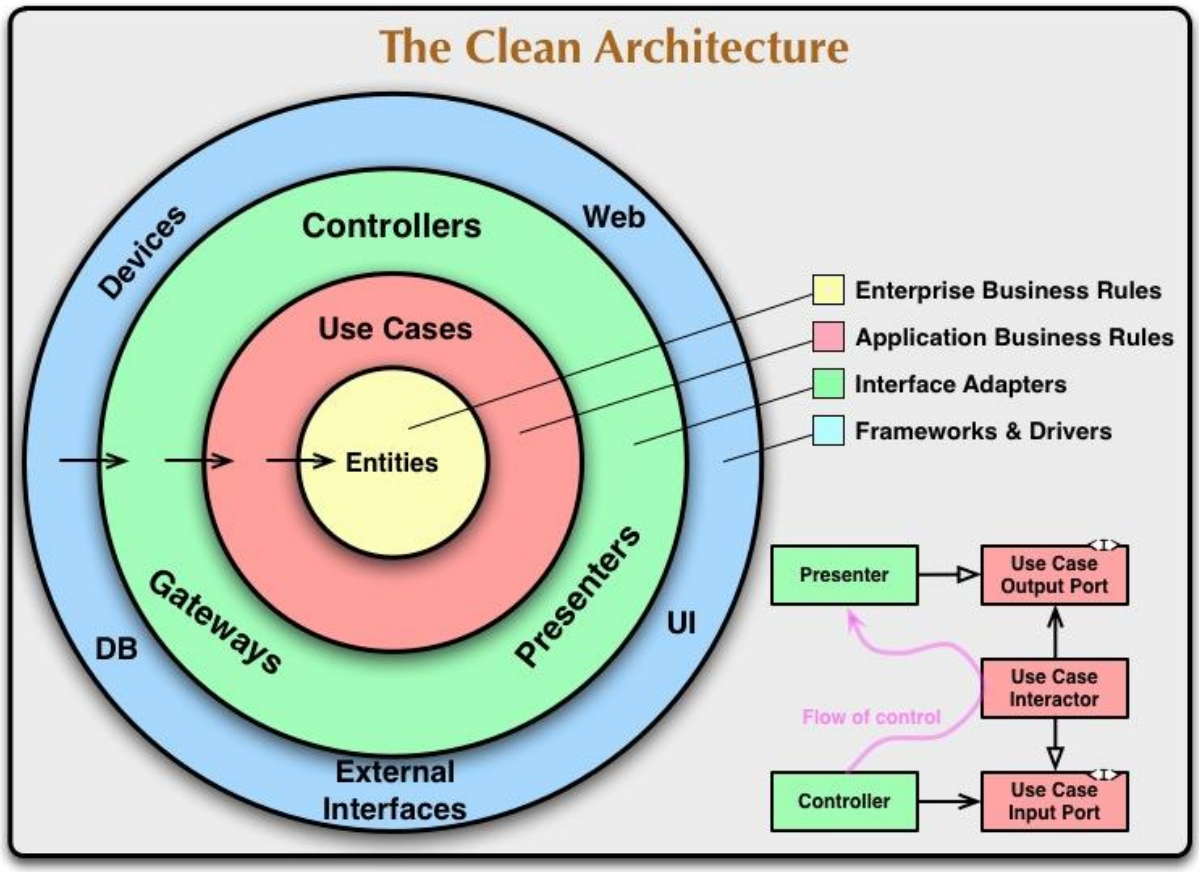
Sl. 4.6. Migracija

4.3.2. Poslužitelj

Strana poslužitelja je implementirana pomoću već spomenutog razvojnog alata .NET Web API. U ovom poglavlju će biti opisana korištena datotečna arhitektura projekta, uzorci rada i glavne i neizostavne komponente svakog .NET Web API projekta.

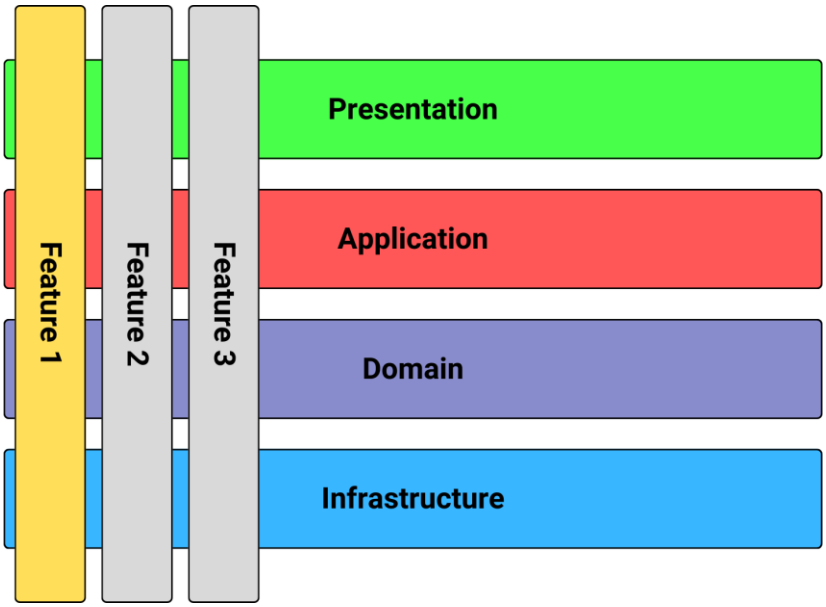
Dvije najpoznatije vrste datotečnih arhitektura jesu *Clean Architecture* i *Vertical Slice Architecture.* Slojevita Arhitektura (eng. Clean Architecture) je prikazana kao koncentrične kružnice, koncentrične kružnice predstavljaju različita područja software-a. Općenito, što se ide

dalje, software postiže višu razinu. Vanjski krugovi su mehanizmi. Unutarnji krugovi predstavljaju politiku software-a [17]. Slika 4.7. prikazuje slojevitu arhitekturu.



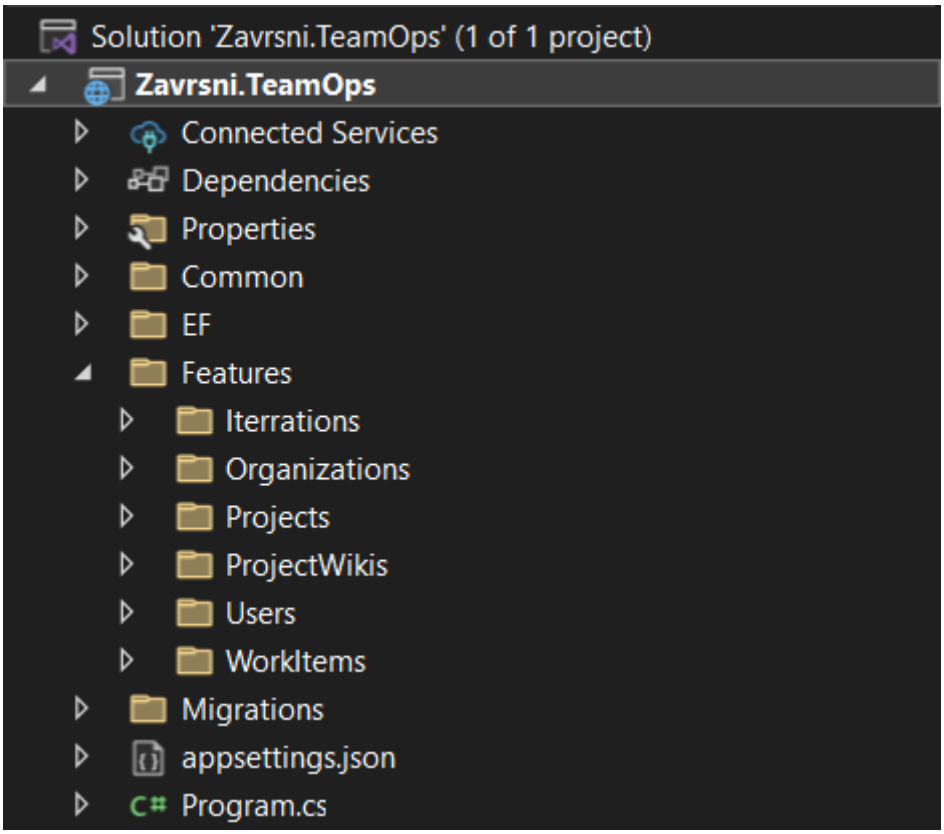
Sl. 4.7. Slojevita arhitektura, preuzeto iz [17]

Veliku važnost imaju crne isprekidane strelice koje označavaju smjer ovisnosti. Dakle prezentacijski sloj ovisi o aplikacijskom i domenskom sloju, aplikacijski sloj ovisi samo o domenskom sloju, ne i o prezentacijskom sloju. Slojevita arhitektura fokusira se na odvajanje ovisnosti među različitim komponentama što olakšava razumijevanje i održavanje programske podrške. Postoje mnoge prednosti ove arhitekture kao što su mogućnosti održavanja, fleksibilnost i labava veza među objektima tj. komponentama. Zbog labave ovisnosti među komponentama sustav zahtjeva puno apstrakcije što kod većih projekata znatno može povećati samu kompleksnost i otežati razumijevanje npr. dodavanjem nove značajke morali bi mijenjati i prilagođavati svaki sloj ove arhitekture. Kako bi se riješio taj problem uvedena je *vertical slice* arhitektura. Slika 4.8. prikazuje *vertical slice* arhitekturu.



Sl. 4.8. Vertical slice arhitektura, preuzeto iz [18]

Koristeći *vertical slice* arhitekturu sve datoteke vezane za jednu značajku nalaze se unutar jednog direktorija pa je zbog toga kohezija značajke jako visoka što znači da će se promjenom jedne značajke mijenjati samo jedan direktorij vezan za tu značajku [18]. *Vertical slice* arhitektura je pogodnija za manje projekte jer ipak zahtjeva puno više koda od slojevite arhitekture stoga sam ja za ovu aplikaciju koristio *vertical slice* arhitekturu. Slika 4.9. prikazuje primjenu *vertical slice* arhitekture unutar projekta za ovaj rad.



Sl. 4.9. Vertical slice arhitektura u projektu

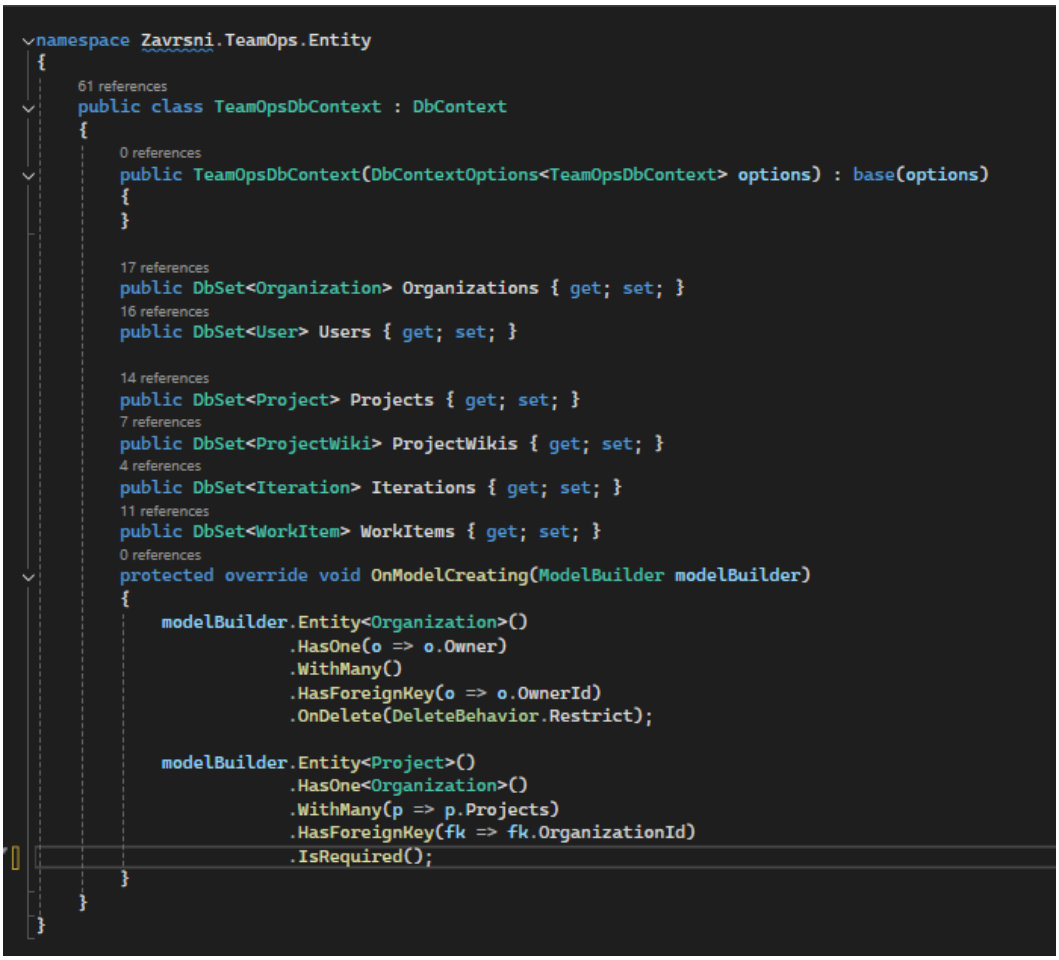
Za implementaciju *vertical slice* arhitekture koristio sam dva obrasca, a to su *CQRS* (eng. Command Query Responsibility Segregation) obrazac i *Repository* obrazac. Praveći API kao stražnju stranu (eng. Backend) neizostavne su HTTP naredbe: POST, PUT, DELETE i GET. Neki HTTP zahtjevi mogu biti puno kompleksniji od drugih, upravo zbog toga koristio sam *CQRS* obrazac koji zahtjeve razdvaja na naredbe i upite. Ovaj obrazac omogućuje korištenje raznih modela: komunikator s bazom, validator ulaza i izlaza, povezivači sa vanjskim servisima itd. unutar jedne klase te na taj način olakšava kompleksne upite i naredbe. Naravno, kompleksnih zahtjeva je puno manje od običnih pa sam za jednostavne zahtjeve koristio *Repository* obrazac koji unutar jedne klase implementira manipulaciju entiteta na bazi podataka poput dohvaćanja svih zapisa, dohvaćanja zapisa po nekakvim atributima, brisanje određenih zapisa, osvježavanje određenih zapisa i još puno osnovnih i jednostavnih radnji sa bazom podataka. Najčešće jedna metoda te klase se koristi za jedan zahtjev izložen putem javnih krajnjih točaka (eng. Endpoint).

Glavne i neizostavne komponente svakog .NET Web API projekta su: *Properties* direktorij koji sadržava *launchSettings.json* datoteku, *DatabaseContext.cs* datoteka i migracijski direktorij ukoliko koristimo *Entity Framework*, *appsettings.json* datoteka i *Program.cs* datoteka.

- *launchSettings.json* – u ovoj datoteci su definirana pravila izgradnje projekta, aplikacijski url, aplikacijski portovi, vrsta okoline u kojoj se pokreće aplikacija npr. razvojna okolina, testna okolina, produkcijska okolina itd. Slika 4.10. prikazuje *launchSettings.json* datoteku.

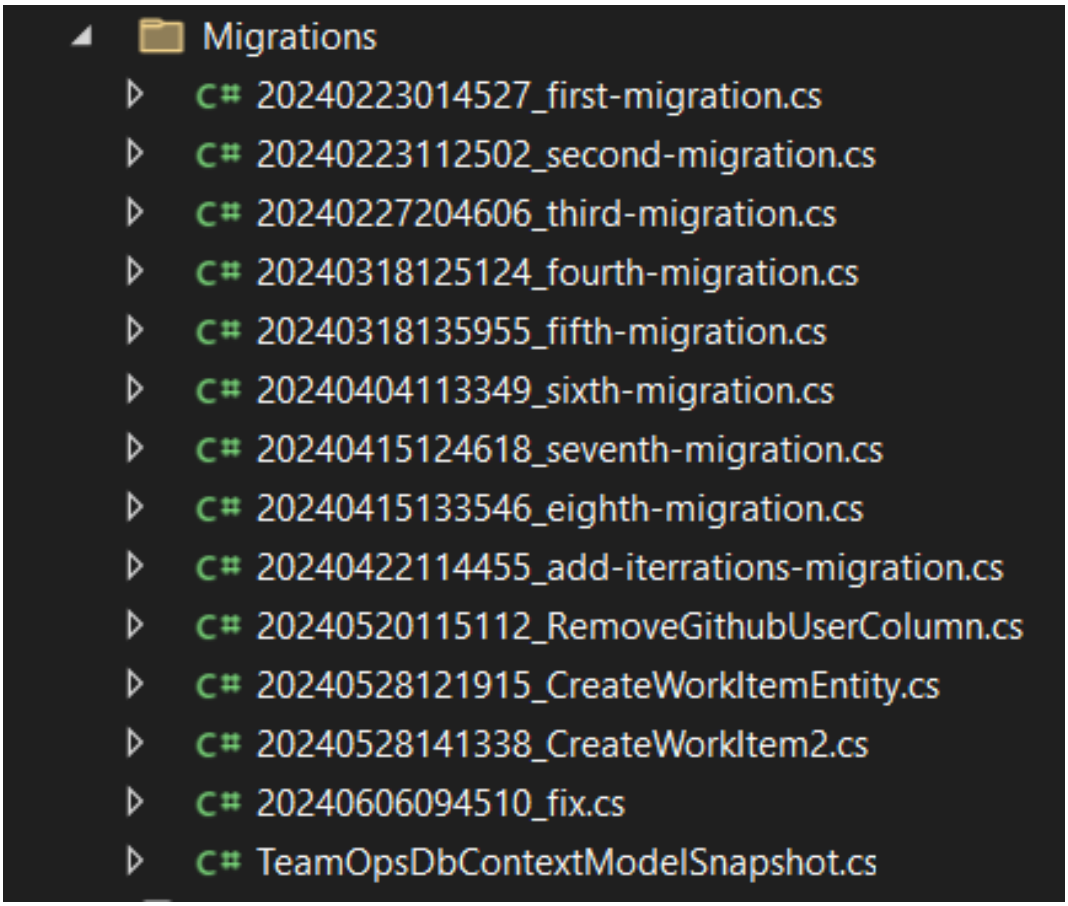
Sl. 4.10. launchSettings.json datoteka

- *DatabaseContext.cs* – ova datoteka sadrži klasu unutar koje se definiraju virtualni objekti koji predstavljaju entitete baze podataka. Također definiraju se pravila tih entiteta poput primarnih ključeva, vanjskih ključeva, odnosa među entitetima, ograničenja itd. Slika 4.11. prikazuje *DatabaseContext.cs* datoteku.



Sl. 4.11. DatabaseContext.cs datoteka

- Direktorij sa migracijama – u ovom direktoriju se nalaze sve migracije vezane za bazu podataka. Promjenom *DatabaseContext.cs* datoteke generiraju se nove migracijske datoteke i spremaju u taj direktorij. Migracijske datoteke se mogu ručno prilagođavati. Važno je napomenuti da ovaj direktorij sadržava *snapshot* datoteku baze podataka koja pamti zadnje stanje baze podataka prije izvršavanja migracija na samoj bazi. Slika 4.12. prikazuje migracijski direktorij sa migracijama i *snapshot* datotekom.

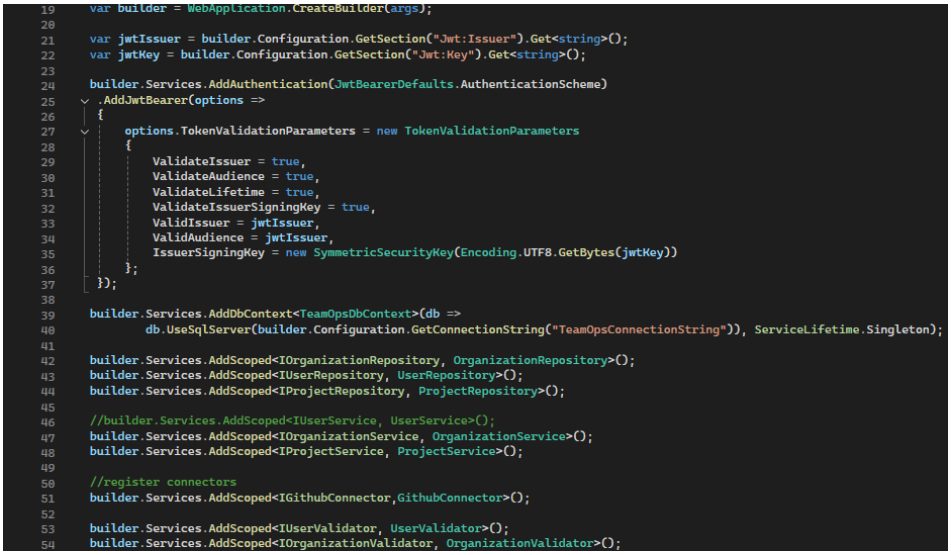


Sl. 4.12. Migracijski direktorij

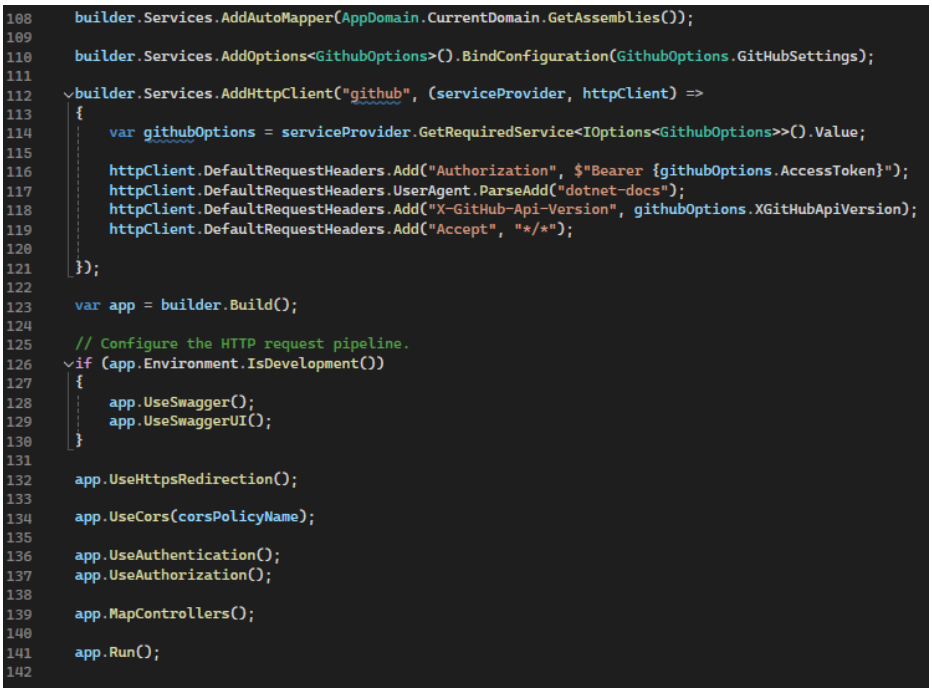
- *appsettings.json* datoteka – ovo je konfiguracijska datoteka koja sadrži podešavanja i vrijednosti koje aplikacija koristi tijekom izvršavanja, umjesto hard kodiranja vrijednosti direktno u kod, konfiguracija se čuva u ovoj datoteci, što omogućava lakše održavanje, promjenu i testiranje aplikacije. Najčešći dijelovi ove datoteke su *Logging* koji služi za podešavanje razina logiranja za aplikaciju, *AllowedHosts* koji definira domene koje aplikacija prihvaća, *ConnectionStrings* koja čuva podatke za povezivanje sa bazom podataka, *AppSettings* koja sadrži specifična podešavanja tajnih riječi, ključeva itd. koji se koriste u aplikaciji. Sam razvojni okvir podržava serijalizaciju ove datoteke u objekt programskog jezika kako bi se vrijednosti te datoteke mogle koristiti. Slika 4.13. prikazuje *appsettings.json* datoteku korištenu u projektu.

Sl. 4.13. appsettings.json datoteka

- *Program.cs* datoteka – ova datoteka definira ulaznu točku aplikacije i sadrži osnovnu konfiguraciju potrebnu za njezino pokretanje. Ključni dijelovi su: *builder* objekt koji postavlja servise i konfiguracije, *app* objekt koji omogućava korištenje raznih podrški za aplikaciju npr. definiranja HTTPS-a umjesto HTTP-a, dodavanje podrške za autorizaciju, mapiranje putanja na određene kontrolere i samo pokretanje aplikacije. Slike 4.14. i 4.15. prikazuju Program.cs datoteku.



Sl. 4.14. Program.cs datoteka 1. dio



```
builder.Services.AddAutoMapper(AppDomain.CurrentDomain.GetAssemblies());

builder.Services.AddOptions<GithubOptions>().BindConfiguration(GithubOptions.GitHubSettings);

builder.Services.AddHttpClient("github", (serviceProvider, httpClient) =>
{
    var githubOptions = serviceProvider.GetRequiredService<IOptions<GithubOptions>>().Value;

    httpClient.DefaultRequestHeaders.Add("Authorization", $"Bearer {githubOptions.AccessToken}");
    httpClient.DefaultRequestHeaders.UserAgent.ParseAdd("dotnet-docs");
    httpClient.DefaultRequestHeaders.Add("X-GitHub-Api-Version", githubOptions.XGitHubApiVersion);
    httpClient.DefaultRequestHeaders.Add("Accept", "*/*");

});

var app = builder.Build();

// Configure the HTTP request pipeline.
if (app.Environment.IsDevelopment())
{
    app.UseSwagger();
    app.UseSwaggerUI();
}

app.UseHttpsRedirection();

app.UseCors(corsPolicyName);

app.UseAuthentication();
app.UseAuthorization();

app.MapControllers();

app.Run();
```

Sl. 4.15. Program.cs datoteka 2. dio

4.3.3. Klijent

Klijentska strana je implementirana pomoću Angular razvojnog okvira. U ovom poglavlju govorit će se o Angular datotečnoj arhitekturi, glavnim komponentama i najboljim praksama za izradu Angular web aplikacije.

Glavne komponente koje čine svaku Angular aplikaciju su: *.angular* direktorij, *node_modules* direktorij, *src* direktorij, *angular.json* datoteka, *package-lock.json* i *package.json* datoteke, *server.ts* datoteka i *tsconfig.json* datoteka.

- *.angular* direktorij – služi za pohrani privremenih datoteka koje su potrebne za razvoj aplikacije. Koristi Angular CLI (eng. Command Line Interface) za spremanje podataka o keširanju, kompajliranju i drugim procesima kako bi ubrzao razvojne zadatke i optimizirao procese izgradnje. Može ga se ignorirati prilikom verzioniranja koda.
- *node_modules* direktorij – služi za pohranu svih vanjskih paketa i ovisnosti koje aplikacija koristi. Kada se pokrene naredba *npm install*, svi potrebni paketi iz *package.json* datoteke preuzimaju se i pohranjuju u ovaj direktorij.
- *src* direktorij – ovaj direktorij sadrži sve poddirektorije i datoteke koje implementiraju samu Angular aplikaciju.

- *angular.json* datoteka – ova datoteka služi za konfiguraciju Angular projekta. U njoj su definirane postavke projekta kao što su putanje do izvornog koda, opcije za izgradnju i testiranje aplikacije, konfiguracije za različita okruženja te specifične postavke za alate kao što su *build, serve, test* i *lint*. Također omogućava prilagodbu načina na koji Angular CLI upravlja projektom.
- *package-lock.json* i *package.json* datoteke – ove dvije datoteke služe kao glavne konfiguracijske datoteke za svaku vrstu Node.js projekta pa samim time i Angular projekta. U *package.josn* datoteci su definirane osnovne informacije o projektu kao što su naziv i verzija, popis ovisnosti i skripte koje projekt koristi. Također omogućava ručnu specifikaciju verzija vanjskih paketa koje projekt koristi. *package-lock.json* datoteka služi za zaključavanje verzija ovisnosti koje su instalirane u projektu kako bi se svi paketi instalirali s točno određenim verzijama, čime se osigurava dosljednost među različitim okruženjima i prilikom verzioniranja projekta.
- *server.ts* datoteka – ova datoteka u Angular projektu služi za konfiguraciju i pokretanje poslužiteljske strane aplikacije. U njoj se postavlja *Express* poslužitelj koji služi za renderiranje Angular aplikacije na poslužitelju, čime omogućava brže učitavanje stranica i bolju optimizaciju za pretraživače (eng. Search Engine Optimization).
- *tsconfig.json* datoteka – ova datoteka služi za konfiguraciju *TypeScript* kompajlera u projektu. U njoj se definiraju verzija JavaScripta, pravila za provjeru tipova podataka, putanje do datoteka te uključivanje ili isključivanje određenih datoteka ili direktorija. Ova konfiguracija omogućuje prevođenje *TypeScript* koda u *JavaScript* kod.

Najbolja praksa kod Angular aplikacija jest korištenje već prije spomenute *vertical slice* arhitekture. Direktorij u kojemu se piše glavna implementacija aplikacije je *app* direktorij. *app* direktorij sadržava tri poddirektorija, *core* i *features* te sadrži korijenske datoteke koje definiraju ulaznu točku u Angular aplikaciju. Te korijenske datoteke imaju prefiks *app*.

- *core* direktorij sadrži jezgrene datoteke koje opisuju opće ponašanje aplikacije. Sadrže klase u kojima se definira zaštita različitih putanja, modeli koji se koriste u čitavoj aplikaciji, *pipes* klasa koja definira metode koje se direktno mogu pozivati unutar HTML-a, validacijske datoteke i mnoge druge.
- *features* direktorij – opisuje implementaciju svake značajke pojedinačno. Najčešće je jedna stranica (putanja) jedna značajka. Svaka značajka se sastoji od četiri komponente. Prva komponenta je html komponenta koja definira izgled te značajke. Druga komponenta je

css ili *scss* datoteka koja služi za uređivanje te stranice. Treća komponenta je klasa koja definira poslovnu logiku te značajke. Posljednja komponenta definira testove za tu značajku.

- Korijenske datoteke – jedne od ovih datoteka su također četiri osnovne komponente koje opisuju svaku značajku, no postoji još nekoliko komponenti. To su: *module* komponenta koja služi za deklaraciju vanjskih modula potrebnih za izgradnju web aplikacije i *routing* komponenta koja služi za definiranje putanja i pridruživanje značajki svakoj putanji.

5. IZGLED I GLAVNE FUNKCIONALNOSTI

U ovom poglavlju je prikazan izgled aplikacije i glavne funkcionalnosti zajedno za važnim dijelovima svake aplikacije, a to su autentifikacija i autorizacija.

5.1. Autorizacija i autentifikacija

Razlikovanje autorizacije i autentifikacije važno je u svijetu razvoja programske podrške. Autentifikacija je proces potvrde identiteta korisnika, tj. provjerava se jesu li korisnički podatci točni. Autorizacija je proces dodjeljivanja prava pristupa resursima na temelju korisničkog identiteta i uloga. Za implementaciju autentifikacije i autorizacije korištene se sljedeće tehnologije: .Net Web API, Angular i JWT (eng. Json web token). Da bih nastavio dalje s objašnjavanjem ključno je objasniti šta je JWT. JWT je način za sigurno prenošenje informacija između dviju strana kao JSON objekata. Najčešće se koristi za autorizaciju i autentifikaciju. Token se sastoji od tri dijela: zaglavlje, podatkovni dio i potpis. Zaglavlje sadrži informacije o algoritmu za potpisivanje i tipu tokena. Podatkovni dio sadrži podatke koji se prenose između klijenta i poslužitelja. Potpis služi za verifikaciju autentičnosti tokena, generira se kombiniranjem tajnog ključa sa podatcima iz zaglavlja i podatkovnog dijela. JWT omogućava da se jednom autentificirani korisnici prepoznaju bez potrebe za ponovnim slanjem novog tokena prilikom svakog zahtjeva.

5.1.1. Implementacija na strani poslužitelja

Prvi korak implementacije je registracija i prijava korisnika u sustav. Prilikom registracije stvara se novi entitet koji se sprema u bazu podataka u tablicu *User*. Da bi se korisnik registrirao potrebno je u tijelu HTTP zahtjeva poslati određene podatke definirane .NET objektom u JSON obliku. Slika 5.1. prikazuje Definiciju klase koja predstavlja .NET objekt za registraciju korisnika, a Slika 5.2. prikazuje glavnu funkciju koja implementira HTTP zahtjev za registraciju korisnika u aplikaciju.

```
public class Command : IRequest<ServiceActionResult>
{
    0 references
    public string Name { get; set; } = string.Empty;
    0 references
    public string Surname { get; set; } = string.Empty;
    1 reference
    public string Username { get; set; } = string.Empty;
    1 reference
    public string Email { get; set; } = string.Empty;
    3 references
    public string Password { get; set; } = string.Empty;
    0 references
    public int InviteeId { get; set; }
}
```

Sl. 5.1. Definicija .NET objekta za registraciju korisnika

```
public async Task<ServiceActionResult> Handle(Command request, CancellationToken cancellationToken)
{
    var serviceActionResult = new ServiceActionResult();
    var validationResult = await _userValidator.ValidateUserSignUp(request.Username, request.Email, request.Password);
    if (!validationResult.IsValid)
    {
        serviceActionResult.SetBadRequest(validationResult.Messages[0]);
        return serviceActionResult;
    }

    var hashedPassword = PasswordHelper.HashPassword(request.Password);

    request.Password = hashedPassword;

    User user = _mapper.Map<User>(request);
    try
    {
        await _dbContext.Users.AddAsync(user, cancellationToken);
        await _dbContext.SaveChangesAsync(cancellationToken);
    }
    catch (Exception)
    {
        throw;
    }

    UserNoSensitiveInfoDTO createdUser = _mapper.Map<UserNoSensitiveInfoDTO>(user);

    serviceActionResult.SetResultCreated(new
    {
        User = createdUser
    });
    return serviceActionResult;
}
```

Sl. 5.2. Glavna funkcija registracije korisnika

Naravno kako bi se očuvala sigurnost aplikacije i korisničkih podataka važno je da se lozinka u bazu podataka ne sprema kao čisti tekst, već tu lozinku treba zaštititi enkripcijom tj. *hashirati*. Razlika između *hashiranja* i enkripcije je ta da se enkriptirana poruka može dekripcijom pronaći njezin izvorni oblik dok je *hashiranu* poruku nemoguće razotkriti. Na slici 5.2. može se pronaći *HashPassword* metoda koja je zadužena za *hashiranje* lozinke. Slika 5.3. prikazuje funkciju za *hashiranje* lozinke. Funkcija prima dva parametra, a to su sama lozinka i tzv. *salt*. Naravno funkcija lozinku *hashira* pomoću HMAC i SHA256 algoritma. Zanimljiviji je dio sa *salt*. Kada se

lozinka *hashira* bez *salt*, identične lozinke će proizvesti isti *hash*. Ako napadač ima pristup velikoj bazi podataka sa lozinkama koje su *hashirane* bez *salt-a*, može lako prepoznati iste lozinke, čak i ako ne zna njihov izvorni tekst. *Salt* dodaje dodatnu nasumičnu vrijednost lozinki prije nego se njen *hash* izračuna, što znači da čak i ako dvije osobe imaju istu lozinku, njihove *hashirane* vrijednosti će biti različite zbog različitih *salt* vrijednosti.

```
public static string HashPassword(string password, byte[]? salt = null)
{
    bool isNewSaltGenerated = false;
    if (salt == null)
    {
        salt = RandomNumberGenerator.GetBytes(128 / 8);
        isNewSaltGenerated = true;
    }

    string hashed = Convert.ToBase64String(KeyDerivation.Pbkdf2(
        password: password,
        salt: salt,
        prf: KeyDerivationPrf.HMACSHA256,
        iterationCount: 100000,
        numBytesRequested: 256 / 8));

    return isNewSaltGenerated ? $"{Convert.ToBase64String(salt)}:{hashed}" : hashed;
}
```

Sl. 5.3. Funkcija za enkripciju lozinke

Prijava korisnika je slična registraciji, razlika je u tome što se prilikom prijave u tijelu HTTP zahtjeva šalju dvije vrijednosti, a to su korisničko ime ili email adresa te lozinka. Slika 5.4. prikazuje .NET objekt koji predstavlja tijelo HTTP zahtjeva za prijavu korisnika. Slika 5.5. prikazuje glavnu funkciju prijave korisnika

```
public class Command : IRequest<ServiceActionResult>
{
    2 references
    public string UsernameOrEmail { get; set; } = string.Empty;
    1 reference
    public string Password { get; set; } = string.Empty;
}
```

Sl. 5.4. Definicija .NET objekta za prijavu korisnika

```
public async Task<ServiceActionResult> Handle(Command request, CancellationToken cancellationToken)
{
    try
    {
        var result = new ServiceActionResult();
        var user = await _db.Users
            .Where(u => u.Username == request.UsernameOrEmail || u.Email == request.UsernameOrEmail)
            .FirstOrDefaultAsync(cancellationToken: cancellationToken);
        if (user is null)
        {
            result.SetNotFound("Couldn't find user with username or email");
            return result;
        }
        var storedPasswordDetails = user.Password.Split(':');
        var salt = Convert.FromBase64String(storedPasswordDetails[0]);
        var storedHashedPassword = storedPasswordDetails[1];
        var hashedPassword = PasswordHelper.HashPassword(request.Password, salt);

        if (hashedPassword != storedHashedPassword)
        {
            result.SetAuthenticationFailed("Invalid password");
            return result;
        }
        else
        {
            var jwt = JwtHelper.IssueNewToken(user, _configuration);
            result.SetOk(new
            {
                Token = jwt,
                User = _mapper.Map<UserNoSensitiveInfoDTO>(user)
            }, "Successfully signed in");
            return result;
        }
    }
```

Sl. 5.5. Glavna funkcija prijave korisnika

U funkciji se vidi kako se prvo u bazi podataka pretražuje korisnik po korisničkom imenu ili email adresi. Zatim se uspoređuju pronađena i dana lozinka te ako se one podudaraju stvara se novi JWT token pomoću metode *IssueNewToken*. Slika 5.6. prikazuje stvaranje novog JWT tokena. Važni dijelovi ove metode su definiranje algoritma za *hashiranje* JWT tokena, definiranje podatkovnog dijela JWT tokena i zadavanje vremena isteka JWT tokena nakon kojeg će se JWT token morati obnoviti. Uloga JWT tokena je ta što će klijent sada biti autoriziran te će moći pristupati drugim resursima koje ovaj RESTful API nudi.

```
public static string IssueNewToken(User userData, IConfiguration configuration)
{
    var securityKey = new SymmetricSecurityKey(Encoding.UTF8.GetBytes(configuration["Jwt:Key"]));
    var credentials = new SigningCredentials(securityKey, SecurityAlgorithms.HmacSha256);

    var claims = new List<Claim>
    {
        new Claim("id", userData.Id.ToString()),
        new Claim("name", userData.Username),
        new Claim("surname",userData.Surname),
        new Claim("email", userData.Email),
        new Claim("username",userData.Username),
    };

    var Sectoken = new JwtSecurityToken(issuer: configuration["Jwt:Issuer"],
      audience: configuration["Jwt:Issuer"],
      claims: claims,
      expires: DateTime.Now.AddMinutes(120),
      signingCredentials: credentials);

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

Sl. 5.6. IssueNewToken metoda

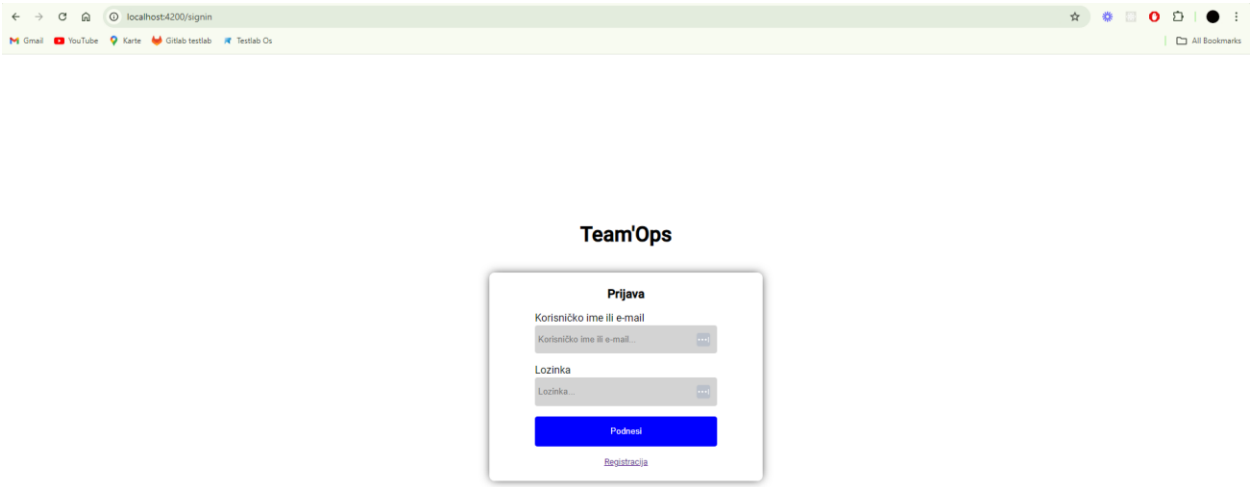
Nakon što klijent ima token, taj token je važno poslati u zaglavlju svakog HTTP zahtjeva te će on biti provjeren i validiran svaki puta. To je drugi korak implementacije na strani poslužitelja, validacija JWT tokena prilikom svakog zahtjeva. Bilo bi teško održivo pisati implementaciju validacije JWT tokena za svaku HTTP metodu koju pružamo na našem poslužitelju. Kako bi se riješio taj problem uveden je pojam *Middleware*. *Middleware* je funkcija koja se aktivira prije procesiranja svakog HTTP zahtjeva. .NET Web API dolazi sa već ugrađenim *middlewareom* za autorizaciju JWT tokena, samo je potrebno pozvati dvije metode unutar glavne *Program.cs* datoteke. Slika 5.7. prikazuje registraciju *middlewarea* za validaciju JWT tokena.

```
136     app.UseAuthentication();
137     app.UseAuthorization();
```

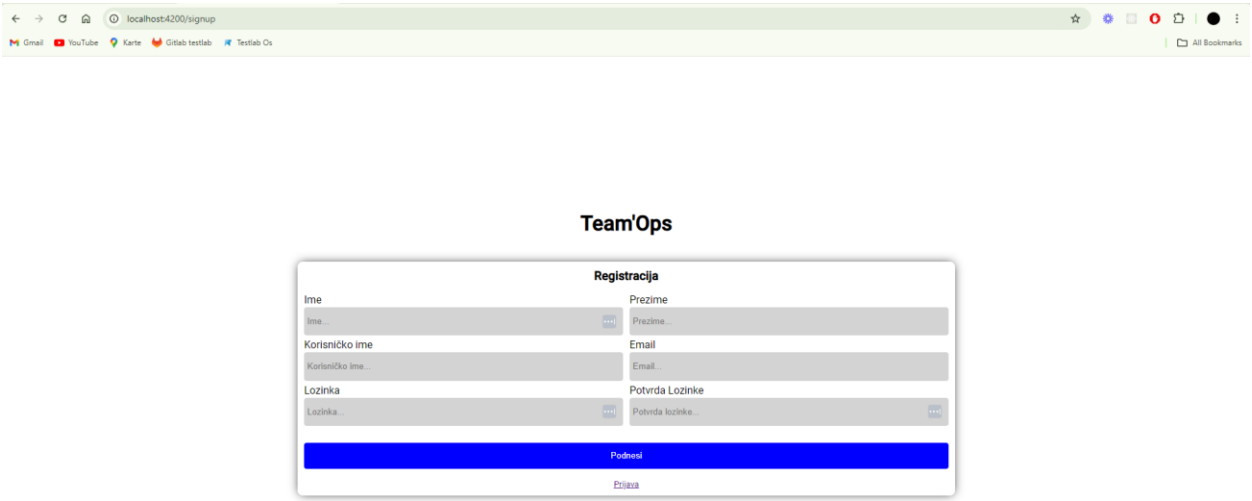
Sl. 5.7. Registracija validacije JWT tokena

5.1.2. Implementacija na klijentskoj strani

Prijava i registracija implementirana je tako što su na klijentskoj strani pružene dvije stranice koje sadržavaju običnu HTML formu u koju se unose parametri tijela potrebni za slanje HTTP zahtjeva prema poslužitelju. Slike 5.8. i 5.9. prikazuju izgled stranica na kojima su predstavljene forme za prijavu i registraciju korisnika. Nakon unošenja određenih podataka i pritiskom na *Submit* gumb poziva se metoda koja šalje HTTP zahtjev prema poslužitelju. Slike 5.10. i 5.11. prikazuju metode za slanje HTTP zahtjeva prema poslužitelju. Važno je ono što se dešava nakon dobivanja odgovora prilikom prijave. Prisjetimo se, nakon prijave poslužitelj vraća odgovor koji sadrži JWT token potreban za autorizaciju nad drugim HTTP zahtjevima. Kako bi taj token slali prilikom svakog zahtjeva potrebno ga je čuvati. Stvari poput JWT tokena skladište se unutar *localStorage* ili *sessionStorage*. Te dvije vrste skladišta omogućuju pohranu podataka na klijentskoj strani, unutar web preglednika. Razlika između ta dva skladišta je ta što su podatci u *localStorage* trajno spremljeni dok se ručno ne obrišu dok podatci unutar *sessionStorage* se brišu prilikom prekidanja sesije tj. prilikom zatvaranja kartice ili preglednika. Slika 5.12. prikazuje metodu za spremanje tokena u *localStorage*. Na slici se također može vidjeti kako se uz token sprema *bool* vrijednost jeli korisnik autentificiran te njegovi osnovni podatci.



Sl. 5.8. Stranica za prijavu



Sl. 5.9. Stranica za registraciju

```
31  public signUp(
32    requestModel: SignUpRequest
33  ): Observable<HttpResponseModel<SignUpResponse>> {
34    console.log(requestModel)
35    return this.httpClient
36      .post('https://localhost:7048/api/User/signup', requestModel)
37      .pipe(
38        map((response: any) => response),
39        catchError((error) => {
40          return of({
41            statusCode: error.error.statusCode,
42            isSuccess: error.error.isSuccess,
43            data: error.error.data,
44            message: error.error.message,
45          });
46        })
47      );
48  }
```

Sl. 5.10. Metoda za slanje HTTP zahtjeva za registraciju

```
13  public signIn(
14    requestModel: SignInRequest
15  ): Observable<HttpResponseModel<SignInResponse>> {
16    return this.httpClient
17      .post('https://localhost:7048/api/User/signin', requestModel)
18      .pipe(
19        map((response: any) => response),
20        catchError((error) => {
21          return of({
22            statusCode: error.error.statusCode,
23            isSuccess: error.error.isSuccess,
24            data: error.error.data,
25            message: error.error.message,
26          });
27        })
28      );
29  }
```

Sl. 5.11. Metoda za slanje HTTP zahtjeva za prijavu

```
11  constructor(@Inject(DOCUMENT) private document: Document) {
12    this.ls = document.defaultView?.localStorage!;
13  }
14
15  signIn(user: User, token: string) {
16    this.ls?.setItem(
17      'state',
18      JSON.stringify({ token: token, user: user, isAuthenticated: true })
19    );
20  }
```

Sl. 5.12. Spremanje tokena u localStorage

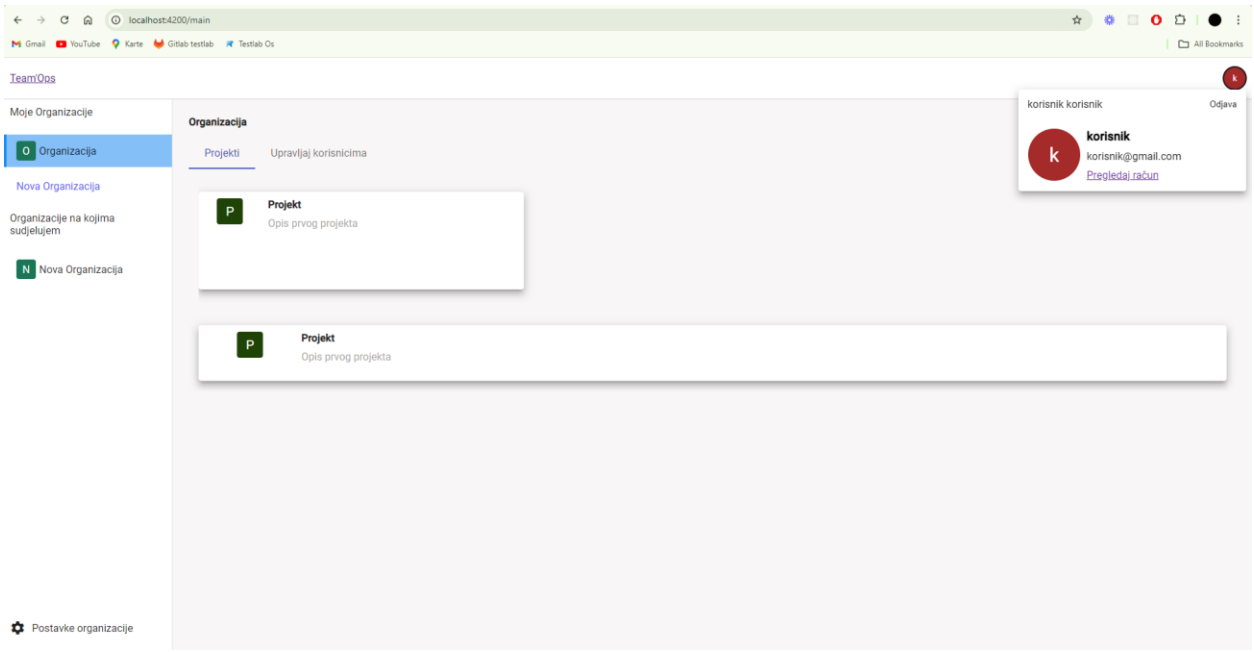
Kako JWT token ne bi morali svaki put ručno navoditi unutar zaglavlja svakog HTTP zahtjeva, slično kao .NET Web API i Angular ima podršku za to, a to je pomoću tzv. *interceptora*. *Interceptor* su metode koje okidaju prilikom slanja svakog HTTP zahtjeva. Ovdje je *interceptor* metoda korištena za nadodavanje JWT tokena u zaglavlje svakog HTTP zahtjeva. Slika 5.13. prikazuje implementaciju JWT *interceptora*

```
import { Injectable } from '@angular/core';
import {
  HttpRequest,
  HttpHandler,
  HttpEvent,
  HttpInterceptor,
} from '@angular/common/http';
import { Observable } from 'rxjs';

@Injectable({
  providedIn: 'root',
})
export class JwtInterceptor implements HttpInterceptor {
  intercept(
    request: HttpRequest<any>,
    next: HttpHandler
  ): Observable<HttpEvent<any>> {
    const jwtToken = localStorage.getItem('jwt');

    if (jwtToken) {
      request = request.clone({
        setHeaders: {
          Authorization: `Bearer ${jwtToken}`,
        },
      });
    }

    return next.handle(request);
  }
}
```

Sl. 5.13. Implementacija JWT interceptora

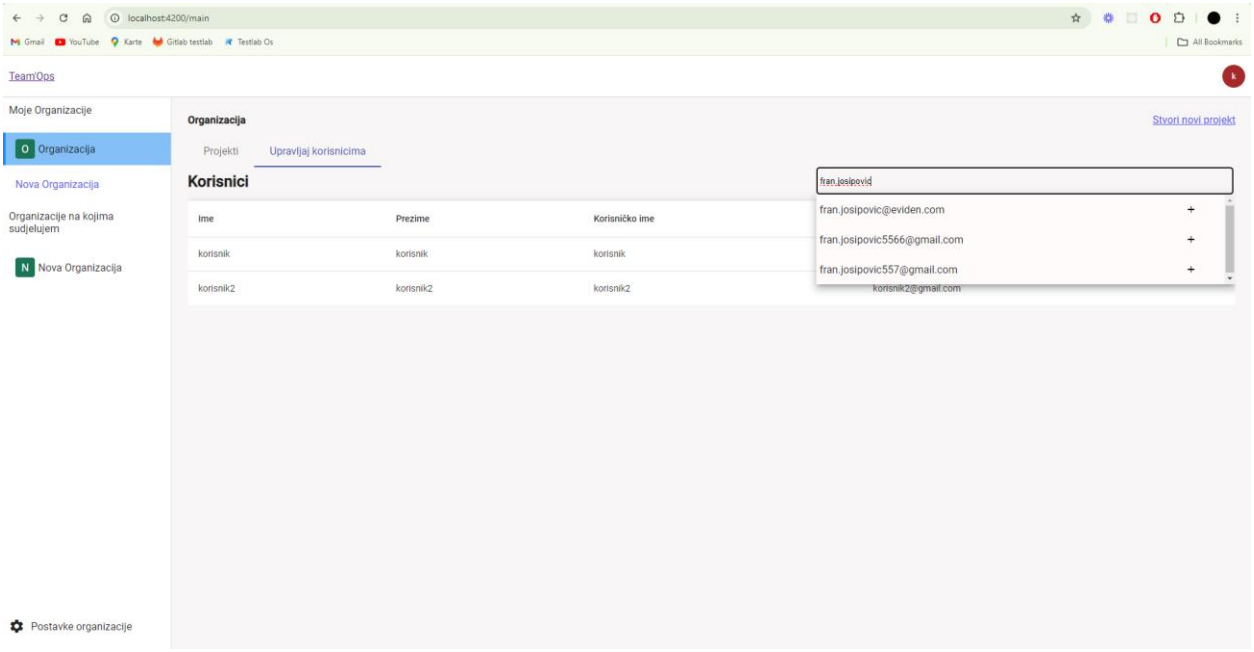
5.2. Ostale funkcionalnosti

Nakon što je korisnik uspješno registriran i prijavljen aplikacija nas odvodi na glavnu stranicu. Slika 5.14. prikazuje glavnu stranicu.



Sl. 5.14. Glavna stranica

Na glavnoj stranici je vidljivo zaglavlje koje sadrži naziv aplikacije sa lijeve strane i s desne strane se nalazi ikonica sa prvim slovom korisničkog imena, klikom na ikonicu prikazuju se dodatni detalji o korisniku te se omogućuje odjava sa aplikacije. Sa lijeve strane se može vidjeti izbornik koji prikazuje organizacije kreirane od strane korisnika te organizacije koje nije kreirao korisnik, ali ipak sudjeluje na projektima tih organizacija. U donjem lijevom kutu se može vidjeti gumb koji nas odvodi na postavke odabrane organizacije. Također u glavnom dijelu pod odjeljkom *Projekti* se vide projekti odabrane organizacije. Slika 5.15. prikazuje odjeljak *Upravljanje korisnicima* u kojemu vlasnik organizacije može dodavati druge korisnike organizaciji. Slika 5.16. prikazuje stranicu postavke organizacije na kojoj se mogu promijeniti naslov i opis organizacije također prikazan je vlasnik organizacije. U izborniku sa lijeve strane se može odabrati *projekti* koji vodi na pod stranicu stranice postavke aplikacije koja prikazuje trenutne projekte ove organizacije kao i korisnike koji sudjeluju na tim projektima.

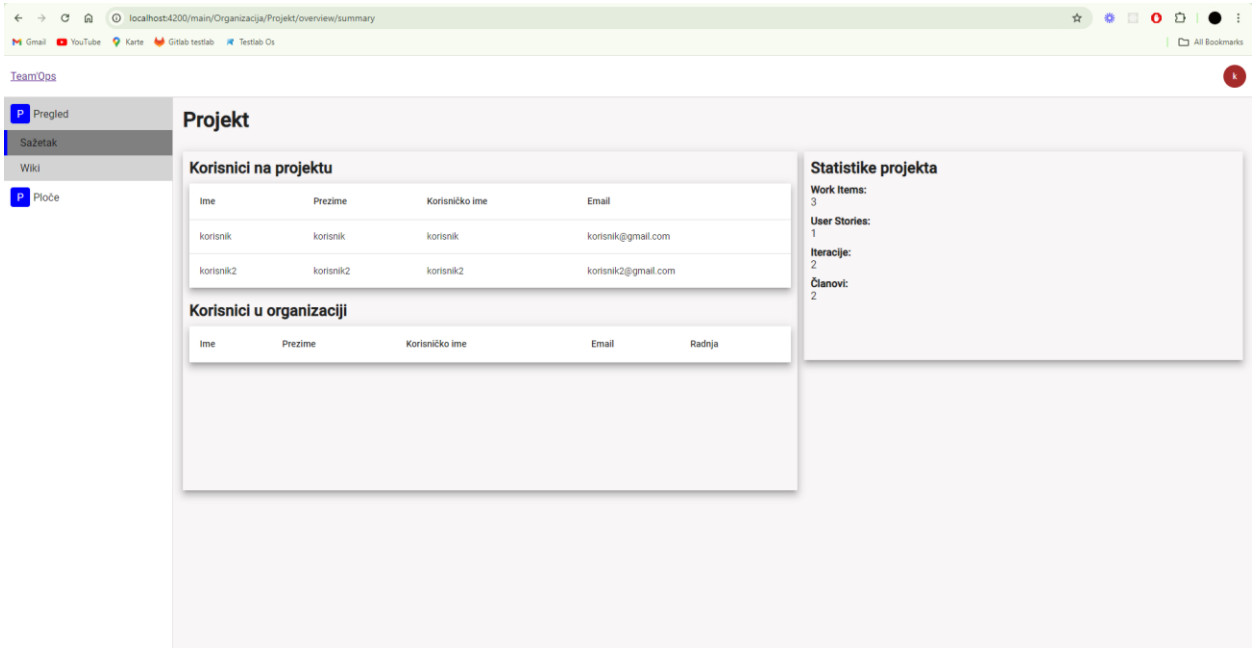


Sl. 5.15. Upravljanje Korisnicima odjeljak na glavnoj stranici

Sl. 5.16. Stranica postavke organizacije

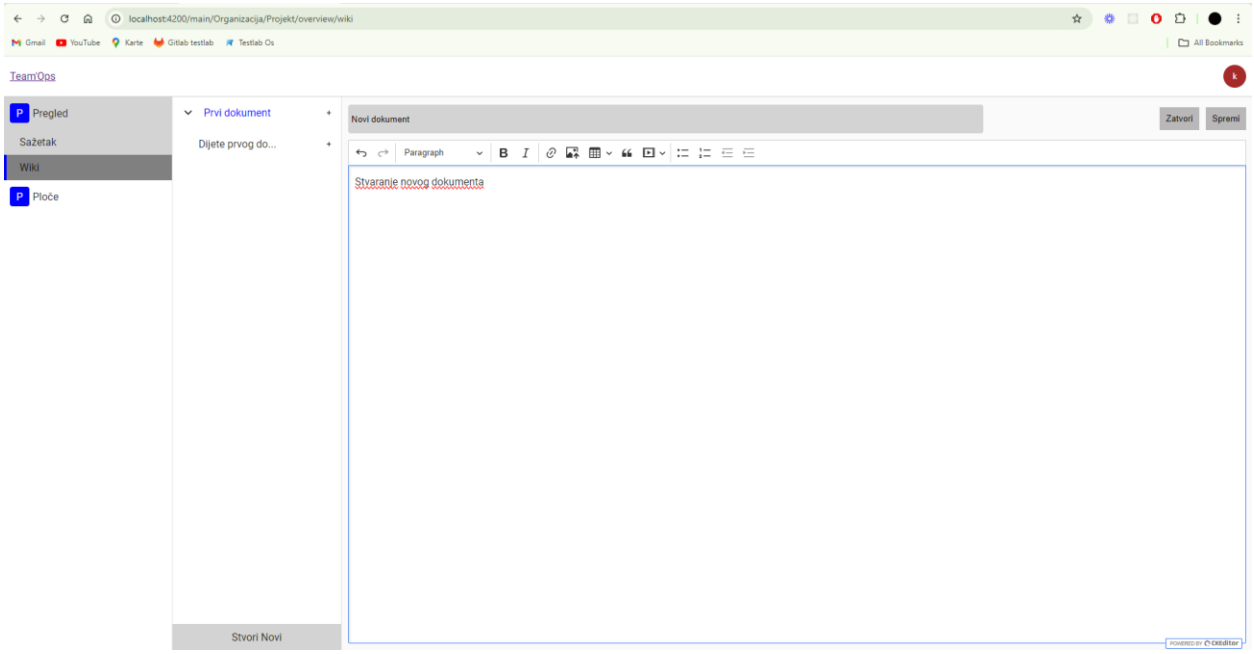
Sada kada je prikazana glavna stranica sa svim organizacijama i dodatnim detaljima slijedi prikaz stranice koja prikazuje detalje o odabranom projektu. Stranica sa detaljima o projektu sadrži četiri pod stranice: *Sažetak*, *Wiki*, *Work Items* i *Iteracije*. Na *Sažetak* pod stranici prikazuju se detalji o projektu kao nekakve statistike. Statistika o *User Story*, o iteracijama, o članovima i o radnim

jedinicama. Također vidljivi su svi korisnici koji sudjeluju na projektu te korisnici koji ne sudjeluju na projektu, a pripadaju organizaciji i koji mogu biti dodani na projekt. Slika 5.17. prikazuje S*ažetak* stranicu.

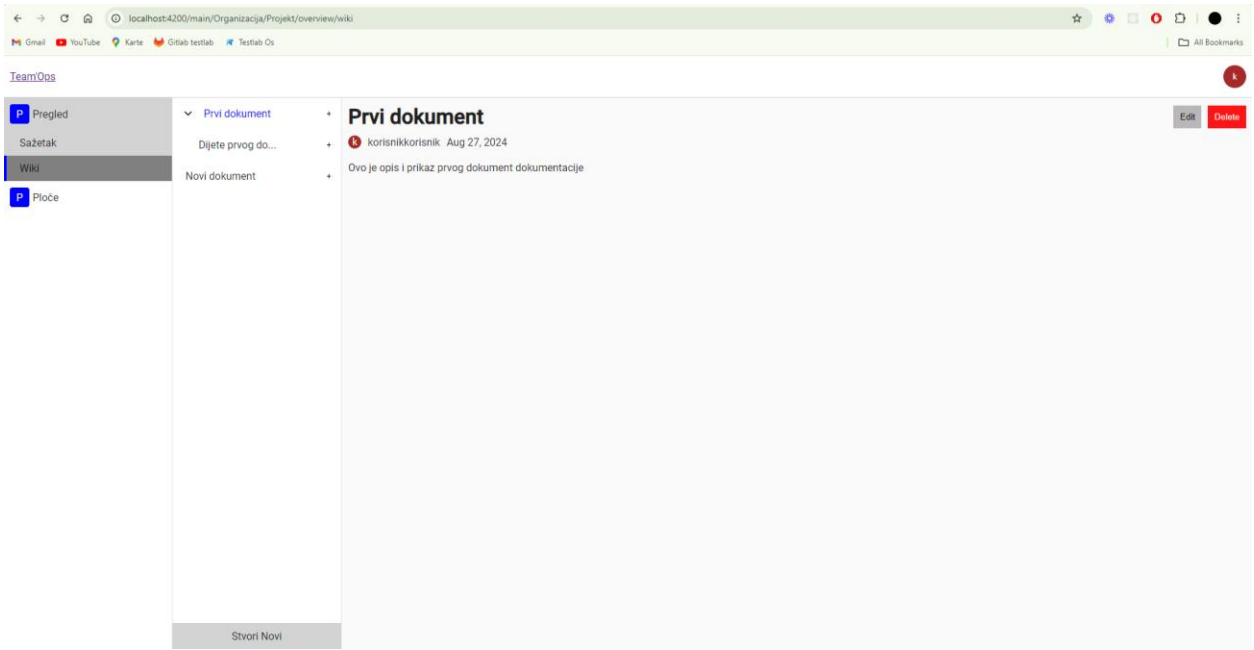


Sl. 5.17. Sažetak stranica

Iduća pod stranica je *wiki* stranica koja služi za prikaz dokumentacije projekta. Na toj stranici je moguće stvarati novi *wiki* dokument i pregledavati postojeće *wiki* dokumente. Slika 5.18. prikazuje stvaranje novog wiki dokumenta, a slika 5.19. prikazuje pregled wiki dokumenta. Također na toj slici se može primijetiti kako su dane mogućnosti uređivanja i brisanja dokumenta. S lijeve strane se može vidjeti izbornik koji nudi odabiranje svih postojećih dokumenta kao i gumb s kojim se može stvoriti novi dokument.



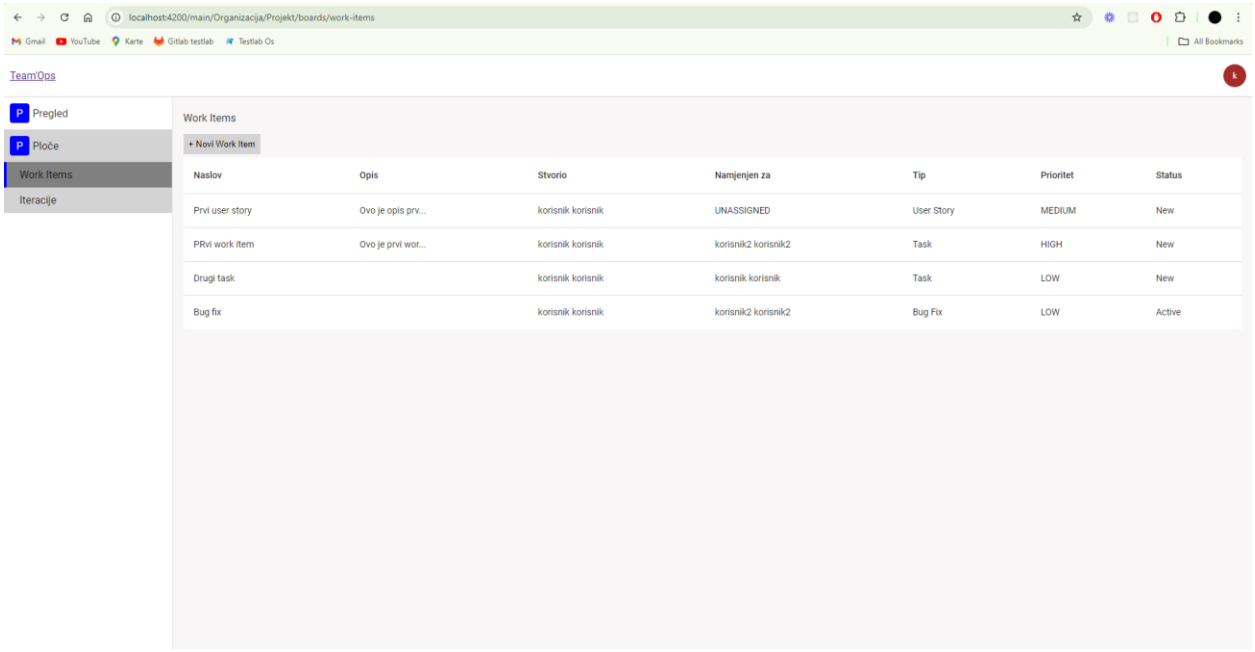
Sl. 5.18. Stvaranje novog wiki dokumenta



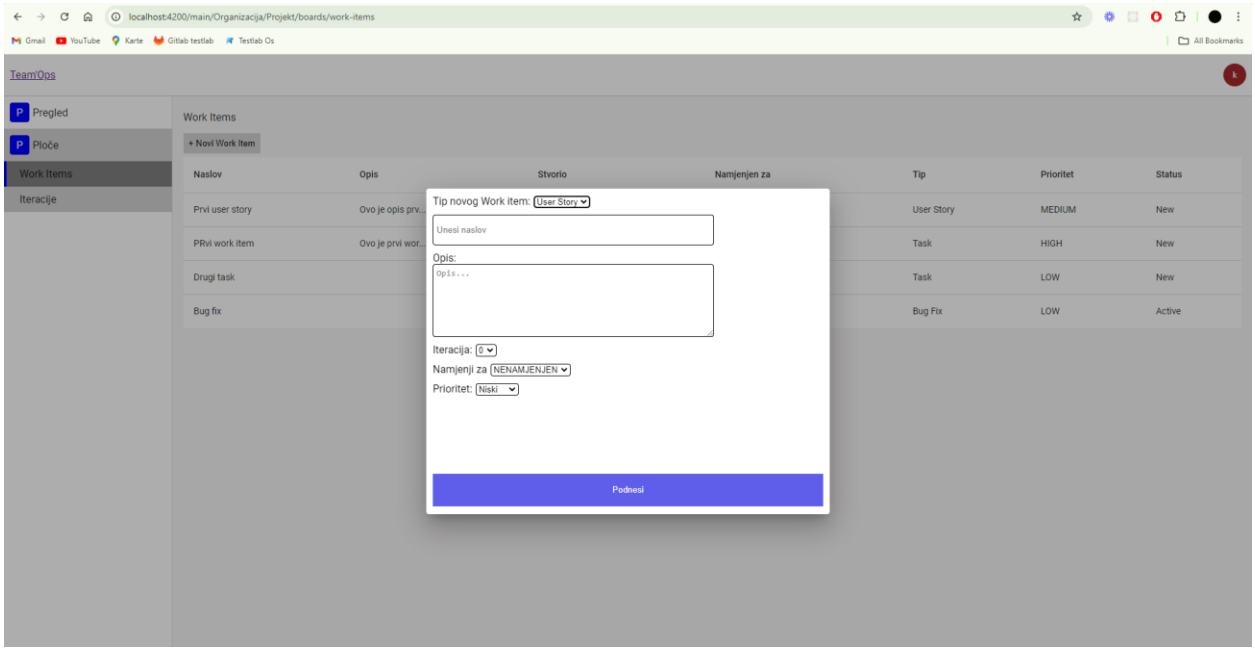
Sl. 5.19. Pregled wiki dokumenta

Iduća pod stranica je *Work Items*. Na ovoj stranici su prikazani sve vrste *Work Item*: *User Story*, *Task* i *Bug fix*. Prikazan je njihov naslov, opis, tko ih je napravio i kome pripadaju, koji je tip *work item*, koji je prioritet i status. Također na ovoj stranici se mogu stvarati novi *work item-i.* Slika 5.20. prikazuje izgled *Work Itmes* stranice. Slika 5.21. prikazuje modal pomoću kojeg se kreira

novi *work item*. U formu se unosi naslov i opis, odabire se tip, dodjeljuje se određenoj iteraciji, odabire se kome će se dodijeliti te se određuje prioritet. Klikom na *Podnesi* gumb kreira se novi *work item*.

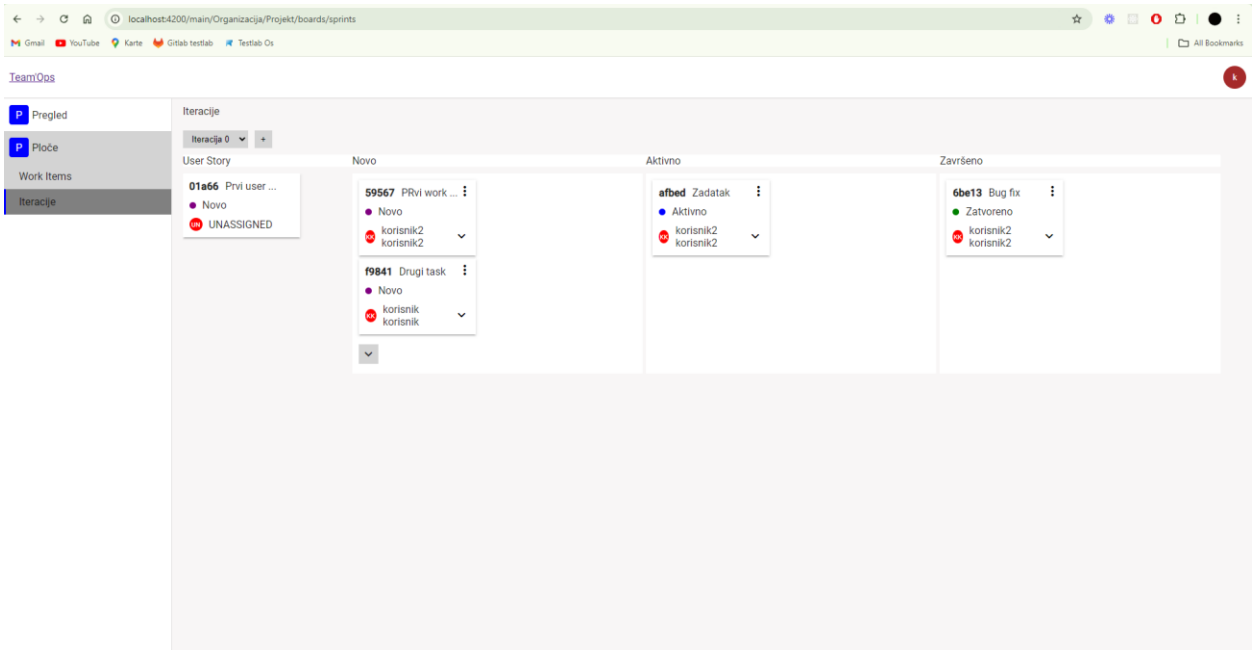


Sl. 5.20. Work Items stranica

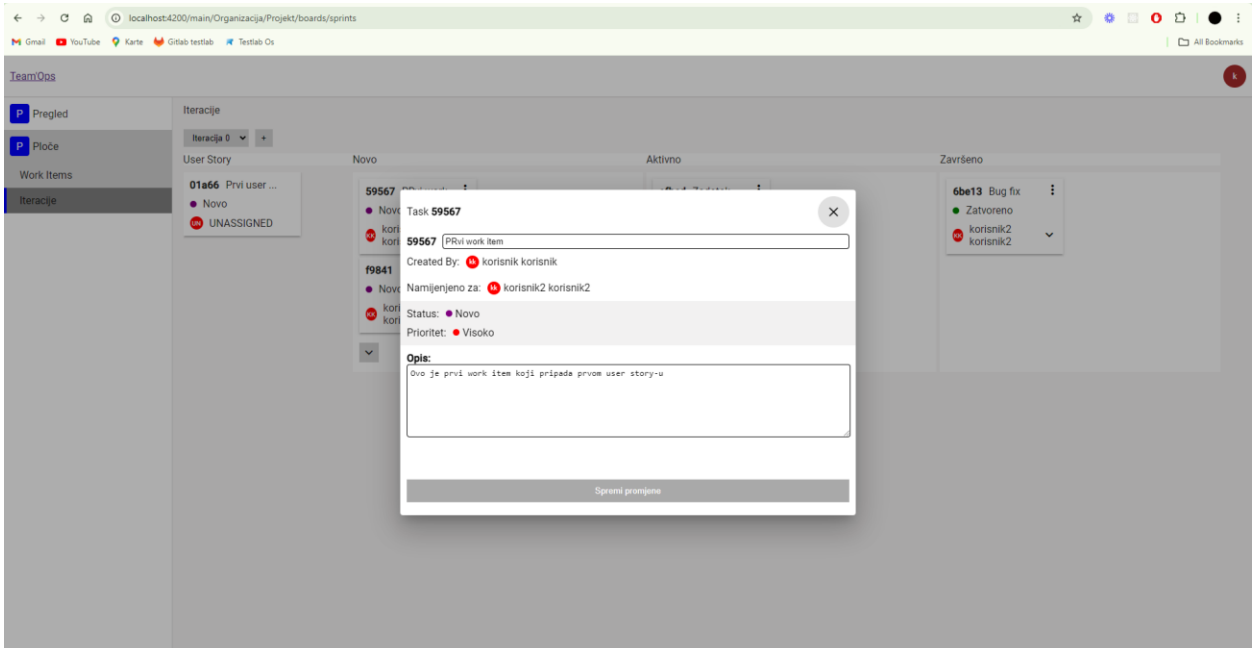


Sl. 5.21. Modal dodavanja novog work item-a

Posljednja pod stranica je *Iteracije* stranica. Ova stranica zapravo prikazuje *scrum* ploču projekta. Prikazani su *user story* vezani za odabrani sprint iliti iteraciju te zadatci vezani za određeni *user story*. Slika 5.22. prikazuje *Iteracije* stranicu. Kartice koje predstavljaju zadatke na ovoj stranici mogu se *drag and drop* metodom premještati između navedenih stupaca na slici: *Novo*, *Aktivno*, *Završeno*. Također na ovoj stranici je omogućeno stvaranje novih iteracija. Uz stvaranje novih iteracija omogućen je pregled detalja svake kartice klikom na njen naslov. Slika 5.23. prikazuje detalje odabrane kartice.



Sl. 5.22. Iteracije stranica



Sl. 5.23. Detalji kartice na iteracije stranici

6. ZAKLJUČAK

Cilj ovog završnog rada bio je napraviti web aplikaciju za rukovođenje timskih projekata koristeći slijedeće tehnologije: .NET Web API, Angular i MsSQL. Ova aplikacija ima jako puno funkcionalnosti. Aplikacija omogućuje registriranje i prijavu korisnika, stvaranje novih organizacija i projekata. Aplikacija omogućuje manipulaciju projektima te vođenje i praćenje razvoja projekta uz implementiranu *scrum* ploču kao jednu od glavnih značajki. Sve te značajke nije bio problem implementirati, već je veći problem bio što ih ima jako puno. Aplikacija se može proširiti na mnogo načina npr. podržavanje *kanban* ploče ili značajke za praćenje promjena na izvornom kodu (eng. Source code) projekta. Može se uvesti sistem verifikacije korisnika kao stvarnih osoba uz implementiranje strojnog učenja npr. očitavanje skenirane osobne iskaznice. Jedna od boljih značajki bi bila stvaranje kanala za razgovor unutar tima te stvaranje video sastanaka jer su sastanci ključni dijelovi svakog agilnog pristupa rada. Stvaranjem ovakvih aplikacija znatno se može poboljšati rad u timovima te se također mogu smanjiti greške koje su prouzrokovane lošom komunikacijom.

LITERATURA

[1] OpenProject (2024), OpenProject (14.0.0), dostupno na: https://www.openproject.org/ [20.08.2024.]

[2] Microsoft (2022), Azure Devops (2022), dostupno na: https://azure.microsoft.com/en-us/products/devops [20.08.2024.]

[3] Atlassian (2024), Jira (9.17), dostupno na: https://www.atlassian.com/software/jira [29.08.2024.]

[4] Atlassian (2024), Trello (2.14.9), dostupno na: https://trello.com/ [29.08.2024.]

[5] M. N. Senturk, „Azure Boards: Terminology and Core Concepts“, dostupno na: https://www.linkedin.com/pulse/azure-boards-terminology-core-concepts-murat-mert-%C5%9Fent%C3%BCrk/ [21.08.2024.]

[6] IFTTT, „The 8 best apps for freelance writers in 2024“, dostupno na: https://ifttt.com/explore/best-apps-freelance-writers [21.08.2024.]

[7] LIVAJA, Ivan; ACALIN, Jerko. Extensible Messaging and Presence Protocol (XMPP). *Zbornik radova Veleučilišta u Šibeniku*, 2018, 12.1-2: 169-181.

[8] J. Molloy, „Guide to Client-server Architecture or Model“, dostupno na: https://www.liquidweb.com/blog/client-server-architecture/ [21.08.2024.]

[9] EHSAN, Adeel, et al. RESTful API testing methodologies: Rationale, challenges, and solution directions. Applied Sciences, 2022, 12.9: 4369.

[10] C. Deshpande, ”What is Angular”, dostupno na: https://www.simplilearn.com/tutorials/angular-tutorial/what-is-angular , [21.08.2024.]

[11] K. Menon, “A One-Stop Solution To C# Web API”, dostupno na: https://www.simplilearn.com/tutorials/c-sharp-tutorial/c-sharp-web-api , [21.08.2024.]

[12] R. Awati; H. Adam; S. Craig, Microsoft SQL Server: What is Microsoft SQL Server?, dostupno na: https://www.techtarget.com/searchdatamanagement/definition/SQL-Server [21.08.2024.]

[13] DYBÅ, Tore; DINGSØYR, Torgeir; MOE, Nils Brede. Agile project management. Software project management in a changing world, 2014, 277-300.

[14] A. Chandran, „Let's talk a bit about AGILE METHODOLOGY“, dostupno na: https://dev.to/akshara_chandran_0f2b21d7/lets-talk-a-bit-about-agile-methodology-259a [21.08.2024.]

[15] A. Wermelinger, „Where it says Kanban, there is often not much Kanban inside“, dostupno na: https://acw-consulting.ch/wo-kanban-draufsteht-ist-oft-nicht-viel-kanban-drin/ [21.08.2024.]

[16] S. Rossel, „Boards“, dostupno na: https://www.syncfusion.com/succinctly-free-ebooks/azure-devops-succinctly/boards [21.08.2024.]

[17] R. C. Martin, “The Clean Architecture”, dostupno na: https://blog.cleancoder.com/uncle-bob/2012/08/13/the-clean-architecture.html [21.08.2024.]

[18] M. Jovanović, “Vertical Slice Architecture”, dostupno na: https://www.milanjovanovic.tech/blog/vertical-slice-architecture [21.08.2024.]

SAŽETAK

U ovom radu navedene su tehnologije potrebne za razvoj ovakve ili sličnih aplikacija: .NET Web API, Angular i MsSQL. .NET Web API je razvojni okvir za korišten za implementaciju poslužiteljske strane (eng. Backend), Angular je korišten za implementaciju korisničkog sučelja (eng. Frontend) i MsSQL je korišten za manipulaciju bazom podataka. Uz izgled i opis funkcionalnosti same aplikacije opisani su glavni dijelovi i komponente koje su sastavni dio svake „klijent-poslužitelj“ arhitekture koja je također opisana u ovom radu. Uz sam razvoj aplikacije naglasak je stavljen i na proučavanje agilnog pristupa rada u timovima koji je danas jedan od najkorištenijih pristupa u cijelom svijetu.

Ključne riječi: agile, Angular, kanban, klijent-poslužitelj arhitektura, MsSQL, .NET Web API, scrum

ABSTRACT

Title: Development of project management applications using SQL, .NET Web API and Angular

This thesis lists all the technologies required for the development of this or similar applications: .NET Web API, Angular and MsSQL. .NET Web API is a development framework used to implement the server side, Angular was used to implement the user interface and MsSQL was used to manipulate the database. Along with the appearance and description of the functionality of the application itself, the main parts and components that are an integral part of every "client-server" architecture that is also described in this thesis are described. Along with the development of the application itself, emphasis was also placed on studying the agile approach of working in teams, which is one of the most used approaches in the whole world today.

Keywords: agile, Angular, kanban, client-server architecture, MsSQL, .NET Web API, scrum

PRILOZI

Github link projekta: https://github.com/FranJosipovic/Zavrsni