SUSTAV ZA VOĐENJE PROJEKATA

NAKIĆ, GABRIEL-LUKA

SVEUČILIŠTE U SPLITU

SVEUČILIŠNI ODJEL ZA STRUČNE STUDIJE

Preddiplomski stručni studij Informacijske tehnologije

Gabriel Luka Nakić

Z A V R Š N I R A D

Sustav za vođenje projekata

Split, rujan 2019.

SVEUČILIŠTE U SPLITU
SVEUČILIŠNI ODJEL ZA STRUČNE STUDIJE

Preddiplomski stručni studij Informacijske tehnologije

Predmet: Programiranje na internetu

Z A V R Š N I R A D

Kandidat: Gabriel Luka Nakić

Naslov rada: Sustav za vođenje projekata

Mentor: Toma Rončević, viši predavač

Split, rujan 2019.

Sadržaj

Sažetak

Cilj rada je opisati i objasniti sustav opće namjene za vođenje projekata, izrađen prema potrebama kolegija Stručne prakse, koji se održava na studijima Elektrotehnike i Elektroenergetike. Svrha razvijanja i izrada samog sustava dolazi radi potrebe pronalaska zamjene već trenutačnog načina vođenja projekata, koji se odvija putem Moodle e-sustava za učenje. Sustav za vođenje projekata je izrađen u obliku "aplikacije jedne stranice". Korisničko sučelje za aplikaciju je organizirano kao skup React komponenti, pisanih u TypeScript programskom jeziku. Mrežni poslužitelj, koji obrađuje zahtjeve korisničkog sučelja, je pisan u C# programskom jeziku i izrađen uz ASP.NET Core razvojni okvir. Za pohranu projekata i ostalih podataka sustava se koristi relacijska baza podataka, kojom upravlja MySQL sustav za upravljanjem relacijskim bazama podataka.

Ključne riječi: aplikacija jedne stranice, inverzija kontrole, React biblioteka, vođenje projekata,

Summary

Project management system

The goal of the thesis is to describe and explain a built general-purpose system for project management, modeled after the needs of the subject of Professional practice, held for the studies of Electrical Engineering and Power Engineering. The purpose of research and the development of the system itself came from the need of a suitable replacement for a current way of managing projects, as held within the Moodle e-learning system. The system for project management is designed as a single-page application. The user interface of the application is organized as a set of React components, which are written in the TypeScript programming language. The web server, which handles the requests made by the user interface, is written in the C# programming language and built with the ASP.NET Core web framework. A relational database is used for storing projects and other system related data, which is handled by MySQL relational database management system.

Keywords: inversion of control, project management, React library, single-page application

1. Uvod

Na Sveučilišnom odjelu za stručne studije, na studijima Elektroenergetike i Elektronike, održava se kolegij Stručne prakse. Studenti, koji sudjeluju u kolegiju, izrađuju praktične radove, za čije informacije o uputama, dokumentima i dodijeljenim mentorima dobivaju putem Moodle sustava za e-učenje, prikazanog na **slici 1**.

Slika 1: Kolegij Stručne prakse na sustavu Moodle

Iako se s trenutačnim pristupom vođenju kolegija Stručne prakse uspješno obavlja administracija sâmog kolegija, ovdje postoji prostor za poboljšanja. Primjerice, ne postoji strukturirani način dolaska do vlastitih projekata. Trenutačno se navedeno odvija prolaskom kroz ravnu listu projekata, što znači da bi kod velike količine projekata na kolegiju trebalo prelaziti i pregledati redom svaki projekt dok se napokon ne dođe do vlastitoga.

Također, ne postoji način da projekti pripadaju nekakvoj zajedničkoj cjelini, ovisno o domeni, mjestu ili temi. Isto tako postoji prilika da se poboljša evidencija studenata i mentorâ koji su dodijeljeni projektu i da oni, kao korisnici, mogu pregledati i filtrirati projekte koji su im dodijeljeni.

U ovom radu se opisuje izrada zamjenskog sustava, gdje su navedena poboljšanja raspravljena i oblikovana prema njegovim korisničkim zahtjevima. Novi sustav omogućava organizaciju i pretragu projekata po cjelinama, kao i metode filtriranja i sortiranja. Određen

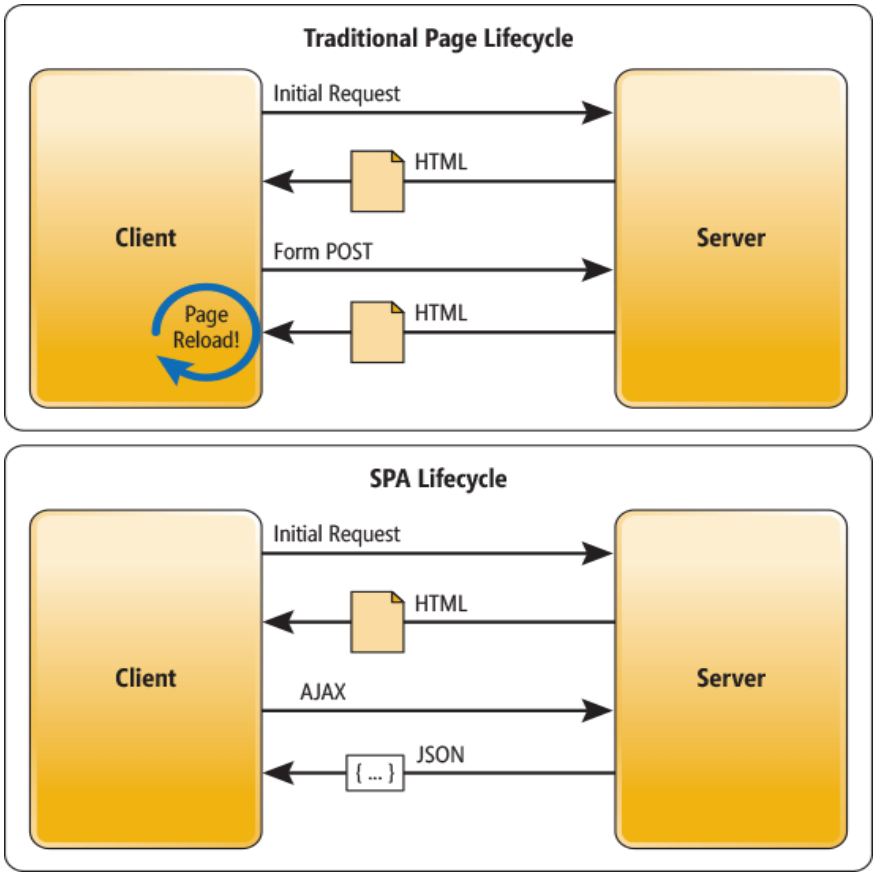
je princip kategorija, koje se određuju u sustavu i dodjeljuju projektima u ulozi deskriptivnog svojstva.

U drugom poglavlju se objašnjavaju odabrane tehnologije, kao i odabrana arhitektura sustava. Treće se poglavlje odnosi na korisničku specifikaciju i na samo korisničko iskustvo. Zaključak, kao četvrto i posljednje poglavlje, iznosi relevantnost samoga rada i preispituje njegovu učinkovitost i potencijalnu primjenu.

2. Tehnologije

2.1. Arhitektura sustava

Sustav je oblikovan kao mrežna aplikacija u *single-page application* arhitekturi. (iz engl. **aplikacija jedne stranice**; akronim SPA). Takve aplikacije dinamički prepisuju trenutačnu stranicu, umjesto da se nove stranice svaki put učitavaju s poslužitelja. Da bi pri otvaranju aplikacije u pregledniku korisničko sučelje funkcioniralo, sve što je potrebno treba dohvatiti samo jedanput s poslužitelja. To predstavlja programski kôd sâmog sučelja, napisanog u JavaScript programskom jeziku, CSS stilove sâme aplikacije ili ikone. Dinamički resursi, koji ujedno predstavljaju podatke samog sustava, se naknadno dohvaćaju upitima na poslužitelj, ovisno o odabiru korisnika. To bi, prema navedenom opisu, značilo da je kôd koji predstavlja korisničko sučelje razdvojen od kôda mrežnog poslužitelja. Pri tome funkcionalnost aplikacije ovisi o njihovoj međusobnoj komunikaciji. Usporedba tradicionalnog životnog ciklusa stranice i SPA životnog ciklusa je prikazana na **slici 2**.



Slika 2: Usporedba životnih ciklusa stranice mrežnih aplikacija. Izvor slike [1].

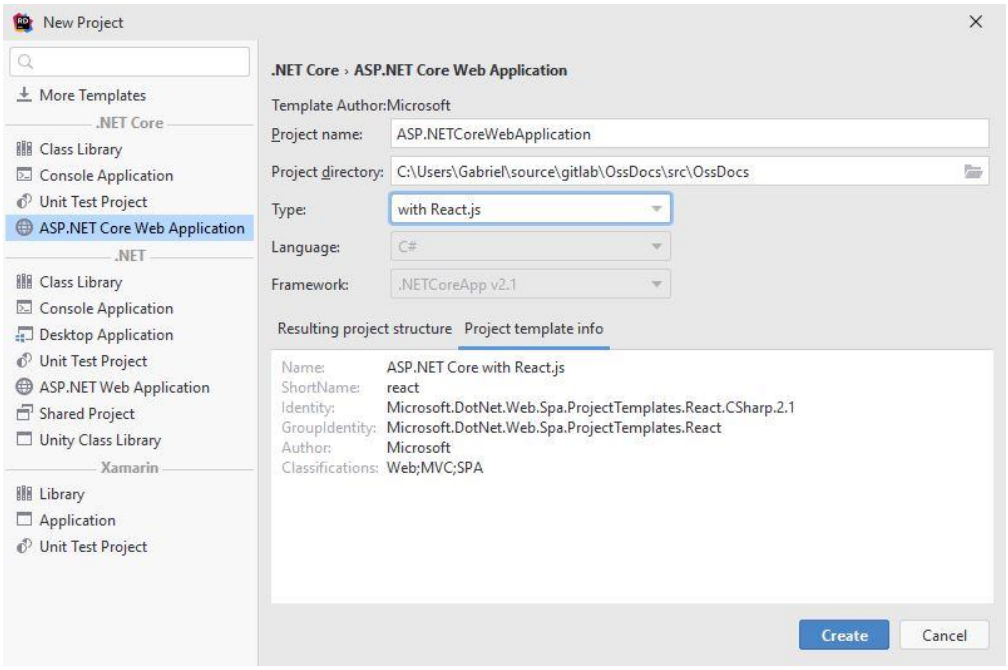
Kod mrežnih aplikacija sa SPA pristupom preglednik mora osvježavati samo dijelove stranice na kojima se izvršio proizvoljni korisnički odabir, umjesto da se cijela stranica ponovo učita.

2.1.1. Predložak projekta

Kôd za mrežni poslužitelj i korisničko sučelje se nalaze u istom projektu, koji spada u organizacijsku cjelinu koja se kod .NET okvira zove Rješenje (engl. *Solution*) [2]. Rješenje za aplikaciju je prikazano na **slici 3**.

Slika 3: Rješenje aplikacije

`OssDocs.BackOffice` predstavlja spomenutu SPA aplikaciju, dok **`OssDocs.Domain`** projekt sadrži biblioteku koja predstavlja samu logiku onoga što aplikacija radi. Potrebno je spomenuti da se projekti mogu stvarati unutar Rješenja po ponuđenim predlošcima, čiji se proces odvija u razvojnom okruženju. Važno je objasniti da je sama postava **`OssDocs.BackOffice`** aplikacije, tj. postavljeni mrežni poslužitelj koji poslužuje korisničko sučelje, stvorena po predlošku prikazanom na **slici 4**.



Slika 4: Predložak za SPA projekt

2.2. Mrežni poslužitelj

Poslužitelj djeluje kao sučelje za programiranje korisničkog sučelja. Klijentski dio aplikacije poslužuje korisnika određenim resursima i interakcijama, ovisno o stanju korisnika u sustavu i ovisno o stanju podataka same baze kojoj poslužitelj pristupa sa svrhom da se obračuna sa zahtjevima koje prima. Uz sve kazano, mrežni poslužitelj također ima svrhu da potvrdi identitet korisnika, kao i njegovu razinu autorizacije.

2.2.1. .NET Core

.NET Core predstavlja programski okvir, kojeg razvija Microsoft i služi za razvoj različitih vrsta aplikacija. Okvir pruža podlogu izvršavanja i izgradnje bilo kojeg jezika koji je podložan njegovoj specifikaciji. Glavne komponente su CLR (engl. *Common Language Runtime*) i FCL (engl. *Framework Class Libraries*). CLR služi kao jezgra za izvršavanje aplikacija, gdje prima oblik virtualne mašine koja pruža optimizaciju i upravljanje memorijom, što uključuje i sakupljač smeća. FCL je kolekcija jezično nezavisnih biblioteka koja je logički organizirana po funkcionalnosti. Verzija .NET Core okvira korištenog u projektu je 2.1.

2.2.2. C#

C# je objektno-orijentirani programski jezik koji je specificiran za .NET platformu, kojeg također razvija Microsoft. Značajke su statički sustav tipova, automatsko upravljanje memorijom, jaka provjera tipova, provjera granica nizova i otkrivanje korištenja neinicijaliziranih referenci. Iako je jezik deklariran kao objektno-orijentiran, kroz inačice su uvedene značajke uglavnom viđene u funkcionalnim jezicima, među kojima su lambda izrazi, anonimni tipovi, automatsko zaključivanje tipova (engl. *type inference*) i metode ekstenzije.

2.2.3. ASP.NET Core

ASP.NET Core je okvir otvorenog kôda visokih performansi za stvaranje mrežnih aplikacija, koje se mogu izvršavati na Linux, macOS ili Windows operativnim sustavima. U ovom je slučaju podržan razvoj navedenog okvira uz .NET Core i .NET Okvir**.** Uz ASP.NET Core pruženo je sve što je potrebno za izgradnju mrežnih aplikacija ili mrežnih usluga.

2.2.3.1. MVC Uzorak

Model-View-Controller (hrv. Model-Pogled-Kontroler) arhitekturni predložak predstavlja način organizacije aplikacije u tri kategorije komponenti: Modeli, Pogledi i Kontroleri. Cilj ovakvog uzorka je odvajanje komponenti ovisno o tome što rade.

Modeli predstavljaju nekakvo stanje aplikacije i implementaciju koja izvršava operacije i logiku aplikacije nad njim.

Pogledi su zaslužni za prezentaciju sadržaja korisniku. Složenost prezentacije može varirati od toga da se poslužuju HTML stranice koje prikazuju nekakvu interakciju korisniku, do toga da pogled može predstavljati obični odziv mrežnom zahtjevu.

Kontroleri preuzimaju zahtjeve korisnika i ovisno o zahtjevu rade s modelom i serviraju prikladni pogled. U MVC aplikaciji, kontroleri su inicijalne ulazne točke interakcije s aplikacijom.

Kontroler reagira na korisnički zahtjev, tako da obrađuje i radi na modelu, gdje na kraju vraća nazad Pogled ovisno o stanju aplikacije.

2.2.3.2. Povezivanje modela

Kada se u trenutačnom kontekstu kaže vezanje modela, misli se na funkcionalnost okvira da automatski veže svojstva nekog zahtjeva na model ili svojstvo modela. Po tome se može zahtjev vezati direktno na model u aplikaciji, uz minimalnu konfiguraciju. Nije potrebno ručno pisati sloj kôda zaslužan za prevođenje zahtjeva u model, gdje to prevođenje vrši sami okvir. **Primjer 1** sadrži kontroler koji obrađuje zahtjeve vezane za brojač.

```
using Microsoft.AspNetCore.Mvc;
using Primjer.Models;

namespace Primjer.Controllers
{
    [Route("api/counters")]
    public class CounterController: Controller
    {
        [HttpPost("increase")]
        public Counter Increase(Counter counter)
        {
            counter.Increase();
            return counter;
        }
    }
```

Primjer 1: Kontroler za obrađivanje zahtjeva brojača

Nadalje, potrebno je naslijediti klasu **`Controller`**. Nasljeđivanje se vrši tako da se u deklaraciji klase dvotočkom odvoji ime bazne klase koju se želi naslijediti (**`Controller`**) od imena klase koja se piše (**`CounterController`**). Važno je istaknuti specifičnost kod ovog primjera, a to je da se kod svake deklaracije, bila to metoda ili klasa, u uglatim zagradama navodi atribut.

Atributi su značajka jezika koja daje mogućnost da se definiraju dodatni metapodaci o kôdu. Mogu se definirati iznad skoro svake deklaracije, gdje se unutar uglatih zagrada piše njihovo ime. Također se mogu navesti i parametri atributa kao što se navode kod metoda. Kod **primjera 1** je definirana putanja na kojoj će kontroler obrađivati zahtjeve. Atribut **`Route`** prima tekstualni parametar **`"api/counters"`**, koji je ujedno i prefiks putanje svake metode u kontroleru. Primjerice, za metodu **`Increase`** je definirano da obrađuje HTTP POST zahtjeve na putanji **`"increase"`**, gdje će prava putanja kod pokrenute aplikacije biti **`"/api/counters/increase"`**. Ono što radi metoda **`Increase`** kada obrađuje zahtjeve na toj putanji je da primi zahtjev oblika po tipu **`Counter`**, poveća brojač i na kraju vrati ažurirani brojač kao odziv na zahtjev.

Kod napisane metode se ništa dodatno ne treba specificirati za prevođenje podataka zahtjeva u model napisan u kôdu, jer je ta značajka vezivanja modela uključena u ASP.NET okvir. Parametar **`counter`** će se vezati na tijelo zahtjeva ako se parametri tijela HTTP zahtjeva podudaraju sa svojstvima ili poljima napisane klase.

2.2.3.3. Validacija modela

Uz funkcionalnost vezivanja modela, također je podržana validacija stanja modela**.** Prvi izvor validacije su greške koje nastaju pri vezivanju modela. Primjer može biti da se na nekom svojstvu ne podudaraju tipovi (očekuje se broj, ali se u zahtjevu primi tekst ili ugniježđeni objekt). Drugi je validacija samog modela koju definira autor modela. Da bi se vršila validacija nad modelom, potrebno je stanje modela opisati validacijskim atributima.

S obzirom da vezanje modela i njegova validacija se događaju prije izvršavanja akcije kontrolera, može se provjerit stanje modela pomoću svojstva **`ModelState.IsValid`** unutar kontrolera. **Primjer 2** provjerava stanje modela i uvjetno prema tome vraća nazad brojač ili ga povećava.

```
using Microsoft.AspNetCore.Mvc;
using Primjer.Models;

namespace Primjer.Controllers
{
    [Route("api/counters")]
    public class CounterController: Controller
    {
        [HttpPost("increase")]
        public Counter Increase(Counter counter)
        {
            if (!ModelState.IsValid)
                return counter;

            counter.Increase();

            return counter;
        }
    }
}
```

Primjer 2: Vraćanje nepromijenjenog brojača pod uvjetom da stanje modela nije validno

2.2.4. Baza podataka

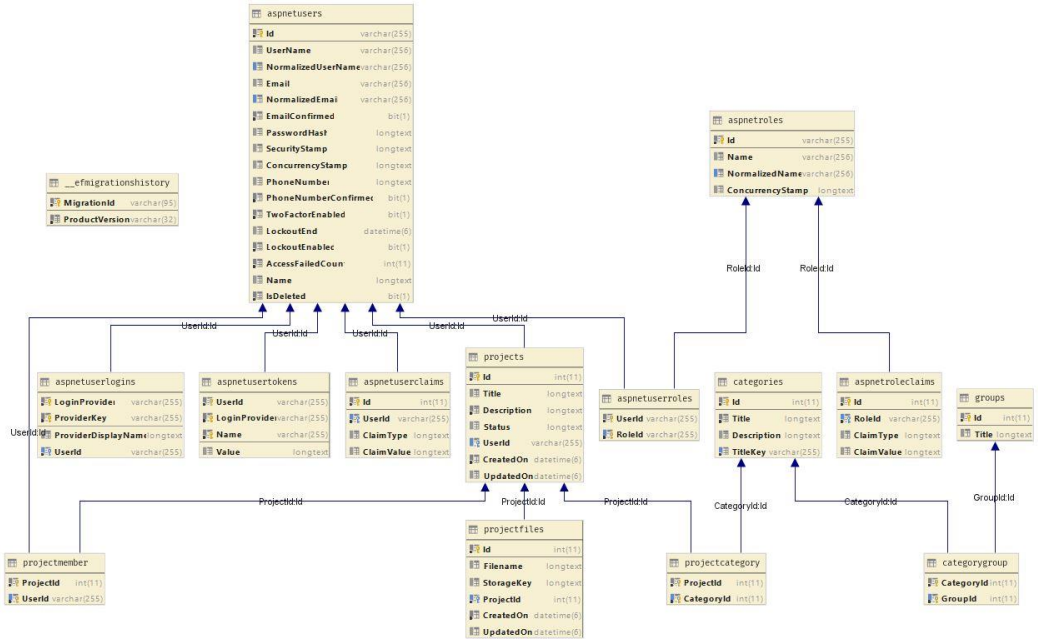
Relacijska baza podataka služiti kao pohrana svega što se u aplikaciji smatra perzistentnim. Odnose između podataka, kao primjerice između kategorije i grupe ili između projekta i kategorije, se definira tablicama i relacijama između njih.

2.2.4.1. MySQL

MySQL je sustav upravljanja relacijskim bazama podataka. Služi za upravljanje, stvaranje i održavanje kontrole nad relacijskim bazama podataka. Podržan je na širokom spektru platformi, od Windows, macOS, Linux do UNIX derivata. MySQL je softver otvorenog kôda, izdan pod GNU licencom. Prva verzija je izašla 1995. godine, gdje se, od pisanja ovog rada, verzija 5.x može pronaći u stabilnijim i popularnijim distribucijama, dok je najviša verzija 8.0. S obzirom da je rad razvijan na Windows i Linux operativnim sustavima, korištene verzije variraju između 5.7 i 8.0.

2.2.4.2. Shema

Shema **relacijske baze** aplikacije je prikazana na **slici 5**, gdje je važno istaknuti da imena tablica mogu varirati o okolini.



Slika 5: Shema relacijske baze aplikacije. U korištenoj MySQL inačici je konfigurirano da se tablice stvaraju s imenima u malim slovima.

Promatrajući shemu na **slici 5**, mogu se predvidjeti korisnički zahtjevi. Vidljivo je da projekti mogu imati više kategorija i kategorije mogu pripadati u više projekata, kao što je vidljivo u relaciji više na više između tablice **`projects`** i **`categories`** pomoću tablice **`projectcategory`**. Sličan odnos se može promatrati između kategorija i grupa, kroz tablice **`categories`**, **`groups`** i **`categorygroup`**.

Prije nego što se opiše odnos između projekta i korisnika, gdje se korisnički podaci spremaju u tablicu **`aspnetusers`**, naglasiti će se detalj gdje nazivi određenih tablica sadrže prefiks "aspnet". Nazivi bez spomenutog prefiksa su omeđeni od svih tablica definiranih od strane autora sustava i tablice koje ga sadrže su zapravo definirane kroz ASP.NET Core Identitet (engl. *Identity*). Radi se o sustavu članstva, kojeg se dodaje u aplikaciju u obliku paketa, koji omogućava da se dodaje funkcionalnost prijave u aplikaciju. Nadalje, kroz ostatak rada će se na ASP.NET Core Identitet oslovljavati samo kao Identitet, zbog jednostavnosti.

Korisnici su u relaciji jedan na više s projektom, gdje svi projekti pripadaju samo onom korisniku koji ih je stvorio u sustavu, kroz strani ključ **`UserId`** u **`projects`** tablici. Korisnici su također u relaciji više na više kroz **`projectmembers`** tablicu, gdje više korisnika može biti dodijeljeno na više projekata u sustavu.

Kako je sustav članstva koji se koristi u aplikaciji ekstenzivan, takva je i njegova definicija tablica. Iako su kroz specifikaciju definirane samo tri uloge korisnika, postoji tablica **`aspnetroles`,** gdje se spremaju definirane uloge i gdje se u slučaju aplikacije iste definirane uloge sustava spreme nakon stvaranja tablice u bazi podataka.

Iako je definirana shema baze, nije definirano kako je konzumira sama aplikacija. Nije još razjašnjeno kako se tablica i njezini stupci preslikavaju u nešto što je razumljivo u kôdu u kojem je napisan dio sustava koji koristi samu bazu. Nedostajala poveznica ovom problemu je Entitetski Okvir.

2.2.4.3. Entitetski Okvir

S obzirom da je odabrani jezik u kojemu se piše mrežni poslužitelj objektno-orijentiran, to dozvoljava pristup da se definiciju tablice zamisli kao klasu, njezine stupce kao svojstva klase i retke tablice kao instance iste klase, što znači objekte. Tablice koje su u relaciji se isto mogu zamisliti kao svojstva klase, kompozitna ili enumerirana ovisno o kakvoj se relaciji radi. Ono što će to ostvariti je Entitetski Okvir (engl. *Entity Framework*). Alat služi kao apstrakcija nad odnosom aplikacije i baze podataka tako da mapira podatkovni model (tablice, stupce, retke u tablici) u konceptualni model (klase, svojstva, instance klase). Klasa, koja predstavlja pripadnika modela podataka, se naziva entitet. U **primjeru 3** su definirani entiteti koji predstavljaju kategorije i grupe. Definicije su nepotpune od prave implementacije radi lakšeg objašnjavanja.

```
public class Category
{
    public int Id { get; set; }
    public string Title { get; set; }
    public string TitleKey { get; set; }
    public string Description { get; set; }
}

public class Group
{
    public int Id { get; set; }
    public string Title { get; set; }
}
```

Primjer 3: Entiteti za kategorije i grupe

Sva svojstva iz **primjera 3** se mogu prepoznati iz pripadajućih tablica na **slici 5**. Važno je istaknuti da iako imena klasa koje predstavljaju entitete su napisana u jednini, Entitetski Okvir njihove tablične ekvivalente po konvenciji imenuje u množini. Zbog navedenog **`categories`** tablica ima svoje ime, kao što je viđeno na **slici 5**, iako entitet koji ju predstavlja se naziva **`Category`**. Nedostaje još kako definirati relaciju između entiteta kategorije i entiteta grupe, što je vidljivo u **primjeru 4**.

```
public class Category
{
    public int Id { get; set; }
    public string Title { get; set; }
    public string TitleKey { get; set; }
    public string Description { get; set; }

    public ICollection<CategoryGroup> CategoryGroups { get; set; }
}

public class Group
{
    public int Id { get; set; }
    public string Title { get; set; }

    public ICollection<CategoryGroup> CategoryGroups { get; set; }
}
```

Primjer 4: Definiranje relacija između kategorije i grupe

S obzirom da se radi o slučaju relacije više na više, svaki entitet sadrži svojstvo kolekcije entiteta koje predstavlja retke u međutablici. Definicija međutabličnog entiteta, pod nazivom **`CategoryGroup`**, je prikazana u **primjeru 5**.

```
public class CategoryGroup
{
    public int CategoryId { get; set; }
    public Category Category { get; set; }

    public int GroupId { get; set; }
    public Group Group { get; set; }
}
```

Primjer 5: Međutablični entitet

U Entitetskom Okviru, kada se definira međutablični entitet, potrebno je ručno napisati registraciju samih relacija. Razlog tome je što se entitet ne piše po podržanoj konvenciji, zbog čega ga okvir ne može automatski prepoznati. Način na koji okvir pruža da

se napravi prilagodba je da se napiše konfiguracijska klasa, koja implementira ponuđeno sučelje okvira. Od sučelja se implementira konfiguracijska metoda, u kojoj se definira sve što je potrebno da se ostvari registracija. Napisana konfiguracijska klasa se onda registrira u kontekst baze podataka.

2.2.4.4. Kontekst baze podataka

Da bi se dohvaćalo, spremalo i mijenjalo entitete, sve operacije se vrše kroz objekt koji predstavlja bazu podataka aplikacije i koji sadrži kolekcije svih raspoloživih entiteta. Takav objekt se naziva kontekst baze podataka**.** Entitete se registrira u kontekst kao svojstva klase koja predstavljaju njihove kolekcije. Konfiguracije entiteta se registrira u kontekst tako da se preoptereti metoda pod nazivom **`OnModelCreating` (**hrv. na stvaranje modela), koja se poziva kada se stvori kontekst da se izgrade entiteti i njihova mapiranja u memoriji. Nepotpuni kontekst je prikazan u **primjeru 6**.

```
public class DatabaseContext
{
    public DbSet<Group> Groups { set; get; }
    public DbSet<Category> Categories { get; set; }

    protected override void OnModelCreating(ModelBuilder builder)
    {
        builder.ApplyConfiguration(new CategoryGroupConfiguration());
    }
}
```

Primjer 6: Kontekst baze podataka

Definirana je klasa imenom **`DatabaseContext`**. Grupe i kategorije su definirane kao svojstva konteksta tipa **`DbSet`,** koji predstavlja entitetski skup**.** Nad **`DbSet`** objektom se mogu graditi upiti, spremati, osvježavati i brisati instance koje pripadaju definiranom entitetu.

2.2.4.5. Migracije

Sinkronizacija baze podataka s konceptualnim modelom podataka se odvija inkrementalno, osvježavanjem baze s migracijama. Migracija je automatski generirana klasa koja predstavlja promjene konceptualnog modela s kojima se osvježava baza podataka. Generira se izvršavanjem naredbe uz naredbeni redak. Kroz **primjer 7** se prikazuje dodavanje novog svojstva **`NewColumn`** entitetu **`Category`** .

```
public class Category

{
     public int Id { get; set; }
     public string Title { get; set; }
     public string TitleKey { get; set; }
     public string Description { get; set; }
     public string NewColumn { get; set; }

     public ICollection<CategoryGroup> CategoryGroups { get; set; }
}
```

Primjer 7: Dodavanje svojstva **`NewColumn`** na entitet **`Category`**

Nakon što se napravi promjena, izvrši se naredba koja generira migraciju. Sve naredbe vezane uz Entitetski okvir se pozivaju s **`dotnet ef`,** čije se podržane naredbe se nalaze u referenci za alate Entitetskog Okvira [3]. Puna naredba za dobiti migraciju koja predstavlja promjenu je u **primjeru 8**.

```
> dotnet ef migrations add Add_NewColumn_To_Category
```

Primjer 8: Naredba za dodavanje migracije koja predstavlja promjenu u primjeru 6

Izvršavanjem naredbe za dodavanje migracije se generira klasa koja predstavlja migraciju. Ona je sadržana u datoteci pod putanjom

`OssDocs.Domain/Migrations/20190616145324_Add_NewColumn_To_Category.cs`.

Uz dodatak ekstenzije koja predstavlja C# datoteku, dodana je također vremenska oznaka na početku imena datoteke. Novo stvorena migracija je prikazana u **primjeru 9.**

```
using Microsoft.EntityFrameworkCore.Migrations;

namespace OssDocs.Domain.Migrations
{
    public partial class Add_NewColumn_To_Category : Migration
    {
        protected override void Up(MigrationBuilder migrationBuilder)
        {
            migrationBuilder.AddColumn<string>(
                name: "NewColumn",
                table: "Categories",
                nullable: true);
        }
        protected override void Down(MigrationBuilder migrationBuilder)
        {
            migrationBuilder.DropColumn(
                name: "NewColumn",
                table: "Categories");
        }
    }
}
```

Primjer 9: Migracija **`Add_NewColumn_To_Category`**

Migracija iz primjera 9 je klasa koja nasljeđuje klasu **`Migration`,** ponuđenu od strane Entitetskog Okvira. Preopterećene su dvije metode naslijeđene klase, metoda **`Up`** (hrv. Gore), koja ažurira bazu podataka i metoda **`Down`** (hrv. Dolje), koja vraća promjenu. Parametar svake metode je od tipa **`MigrationBuilder`**, koji specificira promjene koje će se osvježiti na bazi. U ovom slučaju, vidi se da pozivom metode **`migrationBuilder.AddColumn`** unutar metode **`Up`** se dodaje stupac u tablicu **`Categories`,** koji je tekstualan i nije obavezan.

2.2.5. Inverzija kontrole i struktura logike

Nastavlja se s pretpostavkom da su definirani svi potrebni entiteti i kontekst podataka koji se koristiti. Primjer ide od pretpostavke da je potrebno imati klasu koja sadrži logiku svih operacija vezanih za entitet kategorije. Svrha toga bi bila objediniti zajedničku funkcionalnost, tako da se može koristiti u ostalim mjestima u kôdu. Takve klase se u aplikaciji po konvenciji nazivaju uslugama (engl. *Service*) i tako će ih se zvati do daljnjeg. Klasa koja predstavlja svu logiku vezanu za entitet **`Category`** se naziva **`CategoryService`.** U **primjeru 10** se početnom implementacijom usluge zaslužne za kategorije započinje objašnjavanje procesa i načina na koji se pišu i koriste usluge u aplikaciji.

```
public class CategoryService

{
    public List<Category> All()
    {
        return new List<Category>();
    }
}
```

Primjer 10: Usluga kategorija

Usluga sadrži metodu **`All`** koja, sudeći po nazivu (hrv. **Sve**), vraća sve kategorije. Kada se kaže sve, misli se na sve kategorije spremljene u bazi podataka pa tako se može uvesti kontekst podataka u uslugu i dohvatiti sve iz baze, kao što je prikazano u **primjeru 11**.

```
public class CategoryService

{
    public List<Category> All()
    {
        var databaseContext = InitializeDatabaseContext();
        return databaseContext.Categories.ToList();
    }
    public DatabaseContext InitializeDatabaseContext()
    {
        var databaseContextOptions = new DbContextOptions<DatabaseContext>();
        return new DatabaseContext(databaseContextOptions);
    }
}
```

Primjer 11: Dodavanje konteksta baze podataka

Dodana je nova metoda, **`InitializeDatabaseContext`**, koja stvara novi objekt tipa **`DatabaseContext`,** tip koji je ranije spomenut. Metoda se poziva u metodi **`All`** i svojstvo konteksta podataka **`Categories`** je povratna vrijednost metode **`ToList`.** Možda je važno spomenuti da pretvaranje svojstva **`Categories`** u običnu listu se odvija jer ta pretvorba tek izvršava zahtjev na bazu za dohvaćanje podataka.

Prvi problem pristupa u **primjeru 11** je što je inicijaliziran kontekst podataka, ali nije oslobođen. Za većinu objekata za koje se mogu napraviti instance, .NET-ov sakupljač smeća (engl. *garbage collector*) će se pobrinuti osloboditi ih iz memorije, ali se neupravljani resursi, kao što je veza s bazom podataka, moraju ručno oslobađati. Klase, čiji objekti rukovode s neupravljanim resursima, implementiraju sučelje **`IDisposable`**, što znači da objekti takve klase se koriste jednokratno i oslobađaju se kada više nisu potrebni. Klase koje

implementiraju takvo sučelje moraju implementirati metodu **`Dispose`**, čijim se pozivom odmah oslobađaju neupravljani resursi. Umjesto da se direktno poziva **`databaseContext.Dispose()`** kad je potrebno, u jeziku se može koristiti **`using`** izraz, koji kao parametar prima član tipa **`IDisposable`** i automatski ga oslobađa kada izvršavanje **`using`** izraza završi. **Primjer 12** oslobađa kontekst baze podataka kada više nije potreban.

```
public class CategoryService

{
    public List<Category> All()
    {
        using (var databaseContext = InitializeDatabaseContext())
        {
            return databaseContext.Categories.ToList();
        }
    }
    public DatabaseContext InitializeDatabaseContext()
    {
        var databaseContextOptions = new DbContextOptions<DatabaseContext>();
        return new DatabaseContext(databaseContextOptions);
    }
}
```

Primjer 12: Oslobađanje konteksta podataka **`using`** tvrdnjom

Metoda **`All`** počinje s inicijalizacijom konteksta podataka unutar zagrada, gdje se u tijelu izraza vraća lista kategorija.

Rješenje prvog problema je naglasilo drugi problem. Usluga za kategorije se, uz svrhu upravljanja kategorija, još mora brinuti za postavljanje i inicijalizaciju konteksta podataka. Bilo bi poželjno da je kontekst podataka samo član usluge i da se može neopterećeno konzumirati, kao što je prikazano za uslugu kategorija u **primjeru 13**.

```
public class CategoryService

{
    private readonly DatabaseContext _databaseContext;
    public CategoryService(DatabaseContext databaseContext)
    {
        _databaseContext = databaseContext;
    }
    public List<Category> All()
    {
        return _databaseContext.Categories.ToList();
    }
}
```

Primjer 13: Korištenje konteksta podataka kao polja usluge

Svojstvo konteksta podataka unutar usluge se može još zvati ovisnost (engl. *dependency*). Za neko svojstvo usluge se kaže da je ovisnost kada se usluga oslanja na njega da obavlja svoju svrhu.

Pristup dovodi u pitanje kako postavljati ovisnosti pojedinih klasa i kako zapravo definirati njihov životni vijek. S obzirom da objekt dobiva svoje ovisnosti kroz konstruktor, moguće je ručno napisati inicijalizaciju ovisnosti i onda ih na kraju proslijediti kao parametre konstruktoru željenog objekta kojemu su potrebni.

Iako na prvu misao predstavlja nekakav početak upravljanja ovisnostima kroz aplikaciju, ostavlja otvoreno pitanje kako upravljati životnim vijekom neke ovisnosti. Upitno je još kako usluge, zajedno s njihovim ovisnostima, konzumira aplikacija u odvojenom projektu. Usluge i kontekst podataka su sadržani u **`OssDocs.Domain`** projektu, oblikovanom kao biblioteka klasa, koji upravo služi da drugi projekti, napisani kao aplikacije, konzumiraju i pružaju njene funkcionalnosti. Problem koji se nalazi u konzumiraju **`OssDocs.Domain`** projekta je da je moguća nesigurnost i složenost kod postavljanja ovisnosti aplikacije, kao i nesigurnost kod postavljanja životnog vijeka tih ovisnosti.

Ono što bi riješilo spomenuti problem je oduzeti dužnost tradicionalnom toku aplikacije da upravlja svojim ovisnostima i tu dužnost dodijeliti okviru aplikacije. Takav okvir, kao i slični okviri za pisanje različitih oblika aplikacija, uglavnom sadrže funkcionalnost injekcije ovisnosti (engl. *dependency injection*). Kod takve funkcionalnosti, ovisnosti aplikacije (usluge, konteksti podataka i slično), se registriraju unutar metode klase koja predstavlja postavke. Uz registraciju se također definira i životni vijek neke ovisnosti.

Okvir u trajanju aplikacije postavlja ovisnosti kakve su definirane u njegovim postavkama i također ih oslobađa. Može se još reći da se događa **inverzija kontrole** [8], gdje okvir uzima vlasništvo nad upravljanim resursima, umjesto da se upravljanje resursima direktno piše u implementaciji aplikacije. Kako su dužnosti postavljanja ovisnosti dodijeljene okviru, mogu se pisati usluge u obliku kakav je u **primjeru 13**.

Sada kada postoji znanje kako koristiti ovisnosti iz **`OssDocs.Domain`** projekta, može ih se registrirati uz postavke okvira aplikacije. Uz definicije ovisnosti također je poželjno zapakirati njihove registracije i životni vijek, tako da ih svaka aplikacija konzumira konzistentno na isti način. Registracija i definicija ovisnosti se, uz svojstvo da se sve ovisnosti izlažu kao cjelina, ostvaruje pomoću kontejnera (engl. *container*). Kontejner je objekt koji sadrži sve definicije i postavke ovisnosti. Objekt koji izgrađuje kontejner jer modul (engl. *module*). **`OssDocs.Domain`** projekt izlaže modul kao nešto što će ostale aplikacije registrirati u svoj sustav inverzije kontrole. Pojednostavljeni modul, koji registrira uslugu kategorije i kontekst podataka, je prikazan u **primjeru 14**.

```
public class IocModule: Module
{
    protected override void Load(ContainerBuilder builder)
    {
        builder.RegisterType<CategoryService>().AsSelf();
        builder.RegisterType<DatabaseContext>().AsSelf()
            .InstancePerLifetimeScope();

        base.Load(builder);
    }
}
```

Primjer 14: Modul ovisnosti

U .NET Rješenju aplikacije, koristi se biblioteka za injekciju ovisnosti pod imenom Autofac [4]**,** u kojoj je sadržana klasa **`Module`,** koju klasa **`IocModule`** nasljeđuje. Preopterećuje se metoda **`Load`**, s čijim parametrom **`builder`** **se** registriraju ovisnosti. Koristeći **`builder`**, registrira se tipove ovisnosti, metodom **`RegisterType`**, gdje se prosljeđuje tip kroz generički parametar (**`CategoryService`** i **`DatabaseContext`**). Nakon poziva **`RegisterType`**, poziva se metoda **`AsSelf`,** koja definira da se želi poslužiti osnovni tip.

Osim definiranja kako se poslužuje tip, postoji mogućnost postavljanja djelovanja instance nekog tipa. Može se primijetiti da se za registraciju tipa **`DatabaseContext`**

poziva još metoda **`InstancePerLifetimeScope`**, koja definira da se želi jedna instanca kroz cijeli vijek trajanja.

2.2.5.1. Vijek trajanja

Aplikacije, dok rade, mogu izvršavati razne zadaće, gdje je za svaku zadaću definiran početak i kraj. Upit na bazu, odziv na HTTP zahtjev ili obrada nekakvog skupa podataka se može definirati kao takva zadaća ili također pod termin jedinica rada. Svaka jedinica rada također ima i svoj vijek trajanja. Korištenjem kontejnera ovisnosti, kao što je Autofac biblioteka, se definira vijek trajanja komponenti vezanih uz jedinicu rada, tako da ih kontejner sam oslobodi kada više nisu potrebne. Kontejner u pozadini otvori vijek trajanja (engl. *lifetime scope*), u kojem opredijeli sve zajedničke komponente po pravilima koja su definirana za njih, kao u metodi **`Load`** u **primjeru 14**. Sada je moguće reći, kada se na registraciju konteksta podataka poziva metoda **`InstancePerLifetimeScope`**, želi se jedan kontekst podataka po vijeku trajanja neke jedinice rada. Želi se imati veza na bazu po jedinici rada, tako da se spriječe mogući problemi korištenja jedne veze za sve jedinice rada. Ostale metode za definiciju vijeka trajanja se mogu pronaći u dokumentaciji [5].

2.2.6. Provjera autentičnosti

Da bi klijent pristupio resursima aplikacije koje mrežni poslužitelj nudi, potrebno je svakom zahtjevu uključiti pristupni žeton (engl. *access token*).

Pristupni žeton se generira od strane poslužitelja, koji se vraća klijentu u odzivu zahtjeva provjeravanja autentičnosti, gdje onda klijent koristi isti žeton za daljnje zahtjeve prema zaštićenim resursima. Žeton je kriptografski potpisan, ali nije šifriran, zbog čega je potrebno da se spremanje žetona i zahtjevi koji ga uključuju odvija slanjem kroz sigurnosni protokol (HTTPS).

Pristupni žeton se može zapravo zamisliti kao osobna iskaznica osobe, gdje se njezina autentičnost može provjeriti, ali njezine podatke može svatko vidjeti tko ima njoj pristup. Zbog toga, podatke osjetljive za aplikaciju nije preporučljivo spremati u sadržaj žetona. U slučaju pisanog sustava, korisničko ime i uloga korisnika je dovoljan sadržaj žetona da se odvijaju pravilni zahtjevi između klijenta i servera.

Način na koji se događa razmjena pristupnog žetona je da se poslužuje kontroler zaslužan za provjeru autentičnosti, koji je prikazan u **primjeru 15**.

```
[AllowAnonymous]
[Route("api/login")]
public class LoginController: Controller
{
    private readonly LoginService _service;
    public LoginController(LoginService loginService) =>
        _service = loginService;

    [HttpPost("")]
    public async Task<IActionResult> Login([FromBody] LoginModel model)
    {
        if (!ModelState.IsValid)
            return BadRequest(ModelState);

        var authToken = await _service.SignIn(model);

        if (!string.IsNullOrEmpty(authToken))
            return Ok(authToken);

        ModelState.AddModelError("Attempt", "FAILED");

        return BadRequest(ModelState);
    }
}
```

Primjer 15: Kontroler provjere autentičnosti

Kontroler kao ovisnost prima uslugu, koja sadrži logiku za provjeru i izradu žetona. Metoda **`Login`** se obračunava sa zahtjevom koji u svom tijelu mora sadržavati potreban model za provjeru. Potrebna svojstva tog modela su korisničko ime i lozinka. Dužnost metode je da poslužuje prikladne odzive zahtjeva ovisno o stanju modela. Ako model ne sadrži potrebna svojstva ili ako je provjera neuspjela, vratit će se u odzivu da je zahtjev loš, gdje će tijelo odziva sadržavati stanje modela. Ovisno o stanju modela u tijelu odziva, korisničko sučelje može prikazati korisniku što je otišlo po zlu. Ukoliko je provjera uspješna i ukoliko poziv metode usluge vrati vrijednost žetona, vraća se da je zahtjev bio uspješan. Kod uspješnog zahtjeva, tijelo odziva sadrži žeton provjere koji se može koristit za pristup poslužitelju.

Metoda **`SignIn`** usluge **`LoginService`** sadrži logiku za provjeru korisnika uz Identitet sustav članstva. Usluge Identiteta se uključuju kao ovisnosti **`LoginService`** usluge, koje su zapravo apstrakcije za provjeriti autentičnost korisnika putem baze podataka. Ukoliko provjera prođe uspješno, povrat metode **`SignIn`** će biti ispravna vrijednost pristupnog žetona. Povrat iste metode je upravo poziv privatne metode **`GenerateToken`**, koja je vidljiva u **primjeru 16**.

```
private string GenerateToken(string username, User user)
{
    var claims = new List<Claim>
    {
        new Claim(JwtRegisteredClaimNames.Sub, username),
        new Claim(JwtRegisteredClaimNames.Jti, Guid.NewGuid().ToString()),
        new Claim(ClaimTypes.NameIdentifier, user.Id)
    };

    AddUserRolesAsClaims(claims, user);

    var key = new
SymmetricSecurityKey(Encoding.UTF8.GetBytes(_configuration["JwtKey"]));
    var credentials = new SigningCredentials(key,
SecurityAlgorithms.HmacSha256);
    var expires =
DateTime.Now.AddDays(Convert.ToDouble(_configuration["JwtExpireDays"]));

    var token = new JwtSecurityToken(
        _configuration["JwtIssuer"],
        _configuration["JwtIssuer"],
        claims,
        expires: expires,
        signingCredentials: credentials
    );

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

Primjer 16: Generiranje žetona

Za pristupne žetone se koristi otvoreni JWT standard [6]. U metodi se priprema sve potrebno za izradu, gdje se rade instance različitih svojstava koja predstavljaju sami žeton. Uz potpis i rok trajanja, važno je istaknuti svojstva koja sadrže identitet korisnika, a to su tvrdnje (engl. *claims*). Tvrdnje sadržavaju vrijednosti kao korisničko ime, identifikator i ulogu samog korisnika. Takvi podaci se koristite kod obračunavanja zahtjeva s kojima se pristupa resursima specifičnima za samog korisnika. Uloga korisnika će omogućiti pristup kontrolerima i metodama za koje je potrebna autorizacija.

2.2.7. Autorizacija

Spomenuto je da pristupni žeton kao tvrdnju sadrži ulogu korisnika. Takva tvrdnja određuje razinu pristupa koju korisnik ima na određenim kontrolerima. Ista tvrdnja definira stanje dozvoljenih interakcija koje su omogućene korisniku. Identitet upravlja i rukovodi ulogama korisnika i pruža operacije definiranja i korištenja uloga da se postigne autorizacija. Za Identitet postoji tablica u bazi u kojoj se spremaju definirane uloge i postoje usluge Identiteta koje pružaju potrebne operacije. Sve dostupne uloge korisnika su sadržane unutar enumeracije, koja je vidljiva u **primjeru 17**.

```
public enum UserRole
{
    Administrator,
    Student,
    Mentor
}
```

Primjer 17: Korisničke uloge

Enumeracija služi kao centralni izvor istine o tome koje su zapravo uloge dostupne u sustavu, gdje koristi se kao tip u modelima koji predstavljaju korisnika. Također se koristi kao uvjet u raznim operacijama. Svaki član enumeracije predstavlja zapravo ime uloge koja je spremljena u bazi podataka, u tablici kojom rukovodi Identitet za skladištenje korisničkih uloga.

Na razini kontrolera se vrši zabrana pristupa dijelovima aplikacije korisnicima za čiji pristup nemaju dovoljnu ulogu. Način na koji se opisuje koju razinu pristupa pojedina metoda ili cijeli kontroler definira nad sobom je korištenjem atributa **`Authorize`**. Okviru poslužitelja se delegira logika obračunavanja zabranjenih zahtjeva tako da se definira atribut iznad određene deklaracije. Parametrom atributa se definira kojoj korisničkoj ulozi je dopušteno da radi zahtjeve. To znači da u implementaciji metode koja obrađuje zahtjev ne treba postojati logika za provjerom uloge korisnika, jer će se to provjeriti prije nego što se pozove sama metoda. To znači, primjerice, da se metoda koja obrađuje zahtjev spremanja projekta neće nikad izvršiti ako korisnik nije administrator, kao što je vidljivo u **primjeru 18**.

```
[Authorize(Roles = UserRoles.Administrator)]
[HttpPost("save")]
public IActionResult Save([FromBody] ProjectUpdateModel project)
{
    if (!ModelState.IsValid)
        return BadRequest(ModelState);

    var id = project.Id > 0
        ? _projectService.Update(project)
        : _projectService.Create(User, project);

    return Ok(new {Id = id});
}
```

Primjer 18: Zaštita resursa na temelju korisničke uloge

2.3. Korisničko sučelje

2.3.1. JavaScript

JavaScript je objektno-orijentirani programski za skriptiranje koji je podržan kroz više platformi i služi za ostvarivanje interakcije kod internetskih stranica. Dinamički je jezik, gdje se svojstva i metode mogu dinamički dodavati na objekt. Tip varijable u takvom jeziku nije potrebno definirati.

2.3.2. TypeScript

TypeScript [7] je programski jezik otvorenog kôda, kojeg razvija Microsoft, koji je ujedno i nadskup tipova (engl. *superset*) JavaScript jezika. U JavaScript se prevodi pomoću istoimenog TypeScript prevoditelja. To znači da uz TypeScript se može pisati JavaScript uz dodatnu funkcionalnost statičkog sustava tipova.

2.3.2.1. Definiranje tipova

Kao primjer za definiranje tipova se uzima funkcija koja zbraja dvije vrijednosti**.** Ako bi se reklo da je kontekst demonstracije te funkcije striktno matematički, tj. da funkcija zbraja dva broja, tipovi bi se označili brojčano kao što je vidljivo u **primjeru 19**.

```
function add(x: number, y: number): number {
    return x + y;
}
```

Primjer 19: Definiranje tipova parametara funkcije **`add`**

Sintaksa za definiranje tipa je da se nakon imena parametra i dvotočke koja slijedi napiše samo ime tipa, u ovom slučaju **`number`**, iza kojeg stoje sve brojčane vrijednosti u TypeScript jeziku. Na isti način se za funkciju definira tip za povratnu vrijednost, gdje definiranje slijedi nakon zatvarajuće oble zagrade u kojoj se definiraju parametri funkcije. Isto vrijedi i za varijable, kao što je vidljivo u **primjeru 20**.

```
let result: number = add(2, 3);
```

Primjer 20: Rezultat zbrajanja

Tip varijable se podudara s povratnim tipom funkcije s prošlog primjera. Ako bi se promijenio tip varijable **`result`** iz brojčanog tipa **`number`** u **`string`**, takav kôd se ne bih mogao prevesti i prevoditelj bi prijavio grešku kod prijevoda. U nekim situacijama, eksplicitno definiranje tipa je opcionalno, jer ako se deklaracija varijable i njezina dodjela vrijednosti događa u istoj liniji, tada može doći do automatskog zaključka tipa, ukoliko nije definiran tip za varijablu.

2.3.2.2. Klase

TypeScript podržava JavaScript klase i njihovu sintaksu, uz dodanu funkcionalnost definicije tipova i također definicije razine pristupa, kao što je vidljivo u **primjeru 21**.

```
interface O {
    a: string;
    b: string;
}

class Primjer {
    private n: number;
    public o: O;

    constructor(n: number, o: O) {
        this.n = n;
        this.o = o;
    }

    fn(): void {
        console.log(this.n)
    }
}

let primjer = new Primjer(42, { a: "a", b: "b" });
```

Primjer 21: Definiranje tipova nad članovima i njihove razine pristupa

Svojstvo **`n`** je privatni član klase **`Primjer`**, gdje je određena njegova vrijednost kao brojčani tip **`number`**. S obzirom da funkcija **`fn`** ne vraća vrijednost, povratni tip se označava s riječju **`void`**. U konstruktoru se tipovi parametara podudaraju s tipovima svojstva kojima se dodjeljuju. Iako je svojstvo javno, to nije potrebno naznačiti ključnom riječju **`public`**. Razlog tome je što su članovi pisani bez modifikatora pristupa razine implicitno definirani kao javni od strane jezika. Ono što je važno primijetiti je da svojstvo **`o`** za tip ima sučelje.

2.3.2.3. Sučelja

TypeScript podržava još jednu značajku objektno-orijentirane paradigme, a to su sučelja. Sučeljima se može definirati kakvo će ponašanje imati objekt, ali ne definira njegovu specifičnost. Uobičajena pojava je u objektno-orijentiranim jezicima da klase moraju implementirati sučelja da budu podložna njihovoj specifikaciji. U TypeScript jeziku takav proces može biti implicitan, jer su dva tipa kompatibilna ako se podudara njihova struktura ili oblik. Zato se može proslijediti objekt kao parametar u pozivu konstruktora **`Primjer`**, bez da implementira sučelje **`O`**. Kako izgleda eksplicitna implementacija sučelja je vidljivo u **primjeru 22.**

```
interface O {
    a: string;
    b: string;
}

interface IPrimjer {
    o: O;
    fn(): void;
}

class Primjer implements IPrimjer {
    private n: number;
    public o: O;

    constructor(n: number, o: O) {
        this.n = n;
        this.o = o;
    }

    fn(): void {
        console.log(this.n)
    }
}
```

Primjer 22: Implementiranje sučelja

Klasa **`Primjer`** sada implementira sučelje **`IPrimjer`**, gdje se implementiranje navodi ključnom riječju **`implements`** nakon imena klase, gdje onda slijedi ime sučelja koje se implementira.

2.3.3. React

React [9] je JavaScript biblioteka za izgradnju korisničkih sučelja koju razvija Facebook. Biblioteka također pruža podršku za korištenje u TypeScript jeziku. Rješava problem osvježavanja i ponovnog iscrtavanja korisničkog sučelja ovisno o promjeni stanja

aplikacije. Uobičajeno se kod internetskih stranica pruža interaktivnost na način da se uz izvor HTML stranice napišu skripte koje mijenjaju njezine elemente ili reagiraju na događaje. U međuvremenu su se razvili okviri i biblioteke koje dijele nekakvo sučelje logički na komponente.

Kod **React** biblioteke je slučaj da se sva logika komponente piše uz JavaScript jezik. Unutar jedne metode zaslužne za iscrtavanje može se deklarativno navesti struktura komponente. Svaki element u strukturi komponente predstavlja JavaScript objekt i piše se sintaktički kao HTML element, kroz koji se složeni oblici podataka mogu lako prosljeđivati.

2.3.3.1. Elementi i komponente

Element predstavlja ono što se želi vizualno prikazati. **Primjer 23** sadrži element koji predstavlja zaglavlje. Sintaksa za elemente koja dopušta pisanje oznaka nalik na HTML oznake se zove JSX sintaksa. Elementi pisani takvom sintaksom su JSX elementi.

```
let heading = <h1>Primjer</h1>;
```

Primjer 23: Element zaglavlja

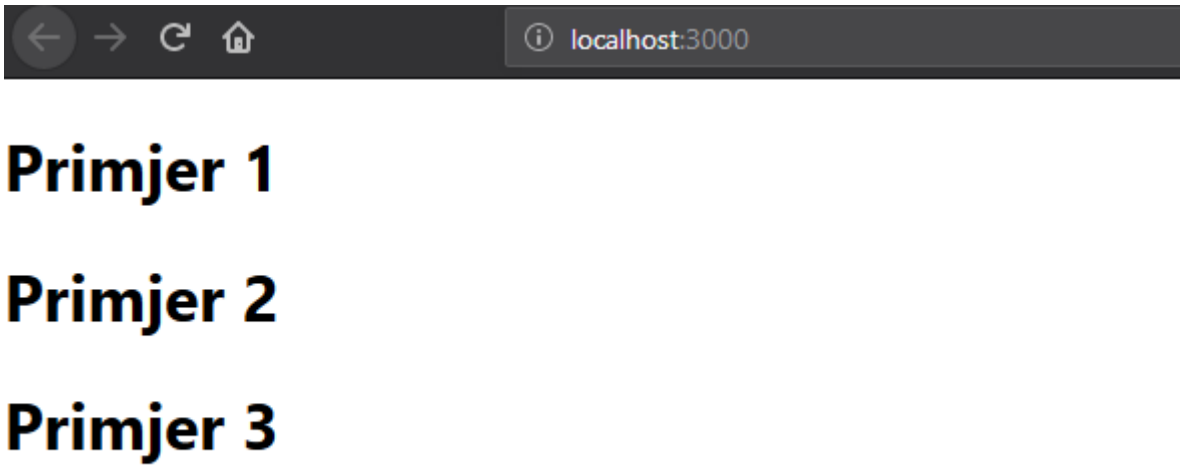
Element se može smatrati osnovnom jedinicom izgradnje komponente, gdje zapravo jedan ili više elemenata može činiti izlaz komponente, kao što je vidljivo u **primjeru 24**.

```
import React, {Component} from 'react';

class App extends Component {
    render() {
        return (
            <div>
                <h1>Primjer 1</h1>
                <h1>Primjer 2</h1>
                <h1>Primjer 3</h1>
            </div>
        );
    }
```

Primjer 24: Komponenta

Komponente se mogu pisati kao funkcije koje vraćaju elemente, ali se mogu napisati i kao klase, gdje metoda **`render`** označava izlaz komponente ili ono što se prikazuje na ekranu. Da bi klasu učinili komponentom, ona mora naslijediti klasu iz **`react`** biblioteke, pod nazivom **`Component`**. **Slika 6** predstavlja prikaz komponente **`App`**.



Slika 6: Izlaz komponente **`App`**

Strukturu komponente također mogu činiti druge komponente. **Primjer 25** predstavlja komponentu koja tekst prikazuje velikim tiskanim slovima.

```
class CapitalizedText extends Component {
    text = "Primjer";

    render() {
        return (
          <h1>{this.text.toUpperCase()}</h1>
        );
    }
}
```

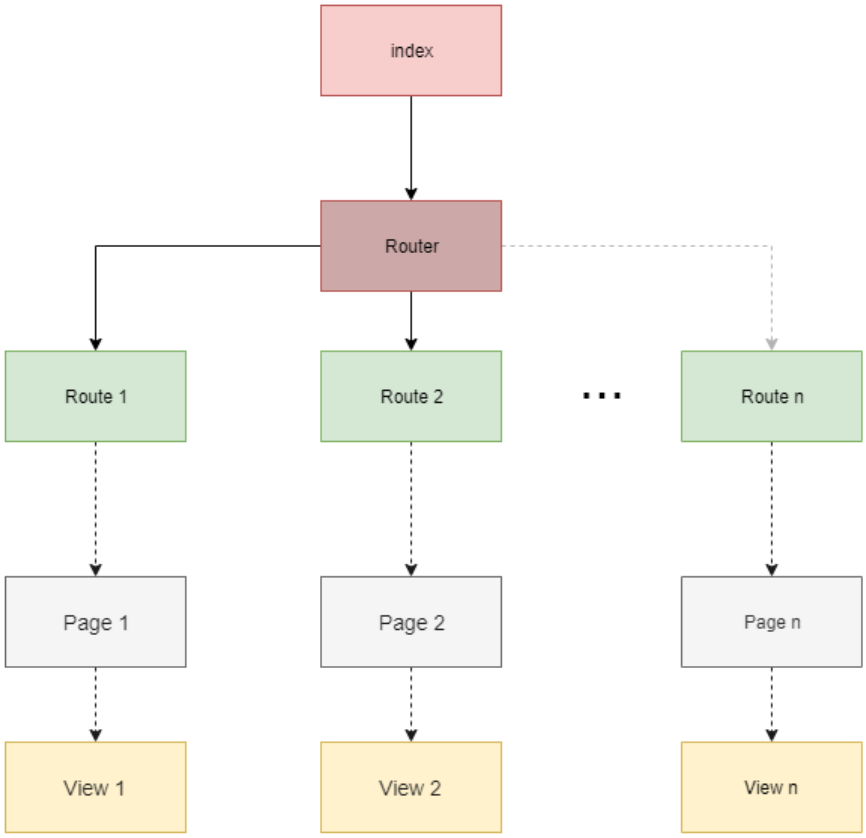
Primjer 25: Komponenta **`CapitalizedText`**

Kada se piše element, sve unutar oznaka se smatra običnim tekstom. Da bi se pristupalo kôdu izvan elementa, potrebno ga je pisati unutar zagrada. U **primjeru 25**, otvaraju se zagrade unutar elementa zaglavlja i pristupa se svojstvu **`text`** i na njemu se poziva metoda **`toUpperCase`,** koja vraća cijeli tekst u velikim tiskanim slovima.

2.3.4. Arhitektura korisničkog sučelja

Kod oblikovanja React korisničkog sučelja prema SPA pristupu, koristi se kombinacija biblioteka koje pružaju komponente s kojima se postiže potrebna implementacija. Važnu ulogu u strukturi sučelja ima komponenta usmjerivača (engl. *router*), koja poslužuje zadanu komponentu ovisno o tome je li se korisnik nalazi na zadanoj putanji stranice. Strukturu komponenti se može zamisliti kao stablo, gdje sve počinje od korijenske

roditeljske komponente, koja inicijalizira samo korisničko sučelje i registrira potrebne ovisnosti. Nakon roditeljske komponente slijede komponente koje predstavljaju djecu i čije su dužnosti raznolike, što se može vidjeti na **slici 7**.



Slika 7: Hijerarhija komponenti

Prva komponenta koja slijedi nakon korijenske komponente, gdje se korijenska komponenta u aplikaciji naziva **`index`**, je usmjerivač. Djeca usmjerivača su putanje (engl. *routes*). Ulaz svake takve komponente je putanja na kojoj će se sama komponenta nalaziti, kao i komponenta koja će se posluživati kao njezino dijete. U korisničkom sučelju su implementirane dvije moguće vrste putanja, privatne i javne. Ono po čemu se privatna razlikuje od javne je što ona ne poslužuje komponentu ako korisnik nije provjeren, nego ga usmjerava na formu za unos korisničkog imena i lozinke.

Između komponente putanje i komponente pogleda je komponenta stranice**.** Takve komponente obnašaju funkciju sličnu kao i korijenska, ali je njihovo obnašanje lokalizirano za pogled (engl. *view*) koji se treba prikazati na određenoj putanji. S obzirom da je pristup

stanja komponenti takav da je stanje odvojeno od same komponente, uz pomoć biblioteke MobX [10], potrebno je postaviti inicijalna stanja, kao i način na koji će se stanje očistiti kada se promijeni putanja.

Složenost odvajanja stanja od komponente je nadoknađena činjenicom da isto stanje može biti distribuirano od roditelja pa do njegovog zadnjeg djeteta, što otkazuje mogućnost da se kod djece dogodi zaostatak ili neželjeno stanje.

Stanje aplikacije je podijeljeno na dvije skupine, globalno stanje i stanje stranice. Globalno stanje proizlazi iz početne komponente i poslužuje se svakoj komponenti da ga koristi. Stanje stranice proizlazi iz komponente stranice i poslužuje se samo onim komponentama koje su djeca stranice.

U kontekstu same aplikacije, početna komponenta započinje tako da se postavi i prikaže na korijenski element u stranici i na kojoj se registrira usmjerivač zajedno s opskrbljivačem ovisnosti. Uloga opskrbljivača je da proslijeđene ovisnosti poslužuje svakoj komponenti putem njihovog svojstva **`props`**. (**primjer 26**)

```
ReactDOM.render(
    <Provider {...dependencies}>
        <BrowserRouter>
            <App />
        </BrowserRouter>
    </Provider>,
  rootElement
);
```

Primjer 26: Početna komponenta

Dijete početne komponente je **`App`,** komponenta koja ispunjava ulogu usmjerivača. Kao djecu sadrži komponente putanja i služi zapravo kao pregled svih mogućih putanja korisničkog sučelja (**primjer 27**).

```
@observer
class App extends Component<RouteComponentProps> {
    render() {
        return (
            <Switch>
                <PrivateAppRoute
                    exact
                    path='/settings/password'
                    component={Password}
                />
                <AppRoute
                    exact
                    path='/login'
                    component={Login}
                />
            </Switch>
        );
    }
}
```

Primjer 27: Pojednostavljeni usmjerivač aplikacije

Izlaz usmjerivača počinje s roditeljskom komponentom **`Switch`**, koja kao djecu sadrži komponente putanje. **`PrivateAppRoute`** komponente predstavljaju zaštićenu komponentu za provjerenog korisnika, dok **`AppRoute`** se poslužuje javno uz slobodan pristup. Po konvenciji i strukturi, svaka komponenta putanje prima kao ulaz komponentu stranice. Zbog toga, kada se postavi trenutačna putanja, komponenta stranice inicijalizira sve potrebne resurse za komponentu pogleda.

Komponenta stranice inicijalizira stanje i potrebne vezane ovisnosti i postavlja svoj opskrbljivač ovisnosti. Ovisno o potrebi koristi globalne ovisnosti, one postavljene od strane početne komponente, za inicijalizaciju (**primjer 28**).

```
@inject("httpClient")
@inject("auth")
@observer
class Login extends Component<LoginProps> {
    private readonly loginStore: LoginStore;

    constructor(props: LoginProps) {
        super(props);
        let {auth, httpClient} = this.props;
        let loginClient = new LoginClient(httpClient!);
        this.loginStore = new LoginStore(loginClient!, auth!);
    }
    render() {
        return (
            <Provider login={this.loginStore} location={this.props.location}>
                <LoginForm/>
            </Provider>
```

Primjer 28: Komponenta stranice za promjenu korisničke lozinke. Novost kod ovoga primjera je što se iznad deklaracije komponente koristi **`inject`** dekorator. S njime se registrira da se potrebna ovisnost pojavi unutar objekta **`props`.**

Na posljetku, unutar opskrbljivača ovisnosti se nalazi komponenta pogleda, čija je prezentacijska logika implementirana na temelju postavljenih ovisnosti komponente stranice.

2.3.4.1. Princip objave i pretplate

Ono što se barem nastojalo provesti kroz izradu aplikacije je da svaki tip predstavlja jedinicu rada i ima jedinstvenu svrhu. Tip koji predstavlja stanje forme za provjeru korisnika bi samo trebao imati svrhu držanja i ažuriranja takvog stanja. HTTP klijent bi trebao imati svrhu obavljanja poziva na mrežni poslužitelj. Sve takve jedinice rada mogu i ne moraju biti kompozicije različitih ovisnosti.

Jedan primjer pristupa bi bilo obračunavanje s događajem kada korisniku istekne provjera autentičnosti. Način na koji bi aplikacija trebala reagirati na taj događaj je da preusmjeri korisnika na formu za ponovnu provjeru. Isto tako, ako korisnik pokuša pristupiti stranici za koju nema potrebnu korisničku ulogu, potrebno ga je preusmjeriti s te stranice. Znanje o pogrešnoj interakciji korisnika sučelje ima kada se dogode zahtjevi na mrežni poslužitelj a da su označeni kao neprovjereni ili zabranjeni (odzivi takvih zahtjeva su označeni kodovima 401 ili 403).

Kada bi se promatralo implementaciju, bilo bi potrebno isprazniti globalno stanje koje predstavlja korisnika i njegov pristupni žeton, nakon čega bi se trebalo dogoditi preusmjeravanje na putanju koja poslužuje formu za ponovnu provjeru.

Moguće rješenje bi bilo da HTTP klijent kao ovisnost sadrži globalno stanje korisnika, gdje bi postojala logika koja bi pozivala akciju za isprazniti stanje u uvjetu lošeg zahtjeva. Postavlja se pitanje zašto bi uopće klijent imao ovisnost o stanju, pogotovo ako bi se koristio u drugim tipovima koji predstavljaju stanje komponente. Ono što je ideja principa objave i pretplate je što ne mora postojati ovisnost između tipa stanja i tipa klijenta da bi se pravilno obrađivao spomenuti događaj. HTTP klijent jednostavno može objaviti da je primio odziv koji je zabranjen ili neprovjeren. Stanje korisnika se može isprazniti kao pretplata na tu objavu. Na taj način, oba tipa ne sadrže znanje jedno o drugom, dok se događaji i dalje pravilno obrađuju.

Način na koji se reagira na promjenu stanja nekog objekta u aplikaciji je uz pomoć funkcije koja pripada Mobx biblioteci, s nazivom **`reaction`** (**primjer 29**).

```
reaction(() => httpClient.unauthorizedPublish, auth.logout);

reaction(() => httpClient.forbiddenPublish, () => {
    route!.history!.push("/forbidden");
});
```

Primjer 29: Pretplata na nevaljane zahtjeve

Reakcija kao prvi parametar prima funkciju koja vraća vrijednost na koju se pretplaćuje. Drugi parametar reakcije je funkcija koja se poziva na promjenu same vrijednosti.

Prva reakcija se pretplaćuje na svojstvo klijenta **`unauthorizedPublish`**, kojem klijent promijeni vrijednost svaki put kada se zahtjev pokaže neprovjerenim. Tada se poziva **`logout`** metoda na instanci stanja **`auth`**, koja prazni stanje korisnika kao i pristupni žeton.

Druga reakcija se pretplaćuje na svojstvo **`forbiddenPublish`**, kojeg klijent ažurira kada obračuna zahtjev koji nije dozvoljen. Tada se događa preusmjeravanje na putanju koja poslužuje pogled koji prikazuje opis korisniku da je pristupio nedozvoljenom resursu.

Sličan pristup se koristi u ostalim dijelovima korisničkog sučelja. Mogući primjeri su pozivanje validacije kada se promijeni unos korisnika, učitavanje tablice podacima kada se promijeni filter i slično.

3. Implementacija

3.1. Korisnički zahtjevi

Glavna mogućnost sustava je pisanje i pohrana projekata. Naslov i opis projekta je moguće tekstualno urediti. Mentori i studenti se mogu dodati na projekt u proizvoljnom broju, i postoje u sustavu kao korisnici. Mentori i studenti mogu također biti dodijeljeni na više projekata. Na projekt se još mogu dodavati i datoteke. Također postoji svojstvo koje predstavlja status projekta. Pod status se misli na to da li je projekt nekome dodijeljen, je li u izradi i je li završen. Pregledavanje i uređivanje projekta predstavlja zasebne stranice, gdje je kod uređivanja dozvoljen pristup na razini autorizacije.

U sustavu, na razini autorizacije, postoje studenti, mentori i administratori kao korisnici aplikacije, kao što je već navedeno. Studenti i mentori mogu pregledati projekte dok administratori imaju mogućnost njihovog uređivanja, kao i uređivanja samih postavki aplikacije.

Osnovne postavke aplikacije sadrže uređivanje vlastitih podataka, kao i promjenu vlastite lozinke s kojom se pristupa sustavu. Postavke, čija prava uređivanja dobiva administrator, uključuje dodavanje novih korisnika, grupa i kategorija.

Dodavanje novih korisnika se provodi unosom osobnih podataka kao i korisničkog imena, uloge, elektroničke pošte i korisničkog imena. Administrator može mijenjati korisniku šifru i može označiti da li je korisnik aktivan ili nije, što odobrava ili ograničava pristup tog korisnika sustavu.

Kategorije služe kao deskriptivno svojstvo projekta. Projekt može imati više kategorija i jedna kategorija može pripadati u više projekata. Pružaju način da se projekti podijele u zajedničke cjeline, nešto što je navedeno kao poboljšanje trenutačnog načina vođenja projekata u prvom poglavlju. Omogućeno je pretraživanje po kategorijama, kao i pregled svih projekata pojedine kategorije. Svaka kategorija sadrži naziv, kao i njezin opis, da korisnik dobije kratak uvid kakve će projekte pregledati kada odabere kategoriju.

Grupe sadrže kategorije kao članove i njihova svrha je strogo navigacijska. Primjerice, možda je poželjno u sustavu imati grupu Kolegiji, koja sadrži sve kategorije koje predstavljaju pojedinačne kolegije i slično. Kod uređivanja grupa, dodavaju se i brišu iz nje postojeće kategorije u sustavu.

Projekti u sustavu se pregledavaju uz liste, koje podržavaju paginaciju, filtriranje po statusu projekta, pretraživanje po ključnoj riječi i sortiranje po svojstvima kao što je broj osoba na projektu, broj datoteka, vrijeme od kada je projekt zadnji put ažuriran i slično. Ista lista podržava pregled svih projekata po kategoriji, koju korisnik odabere iz navigacijske trake.

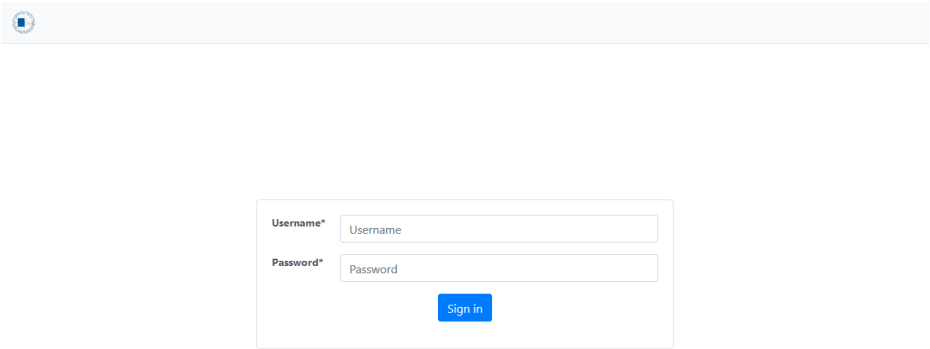
Navigacijska traka sadrži opcije za odjavu korisnika i odlazak na postavke. Za administratora je još vidljiva opcija za odlazak na uređivanje novog projekta. Na navigacijskoj traci se još nalazi i pretraga po ključnoj riječi, za čiji odabir rezultata se odvede korisnika na pretraženu kategoriju ili na pretraženi pregled projekta. Uz horizontalnu navigacijsku traku, postoji i "vertikalna navigacijska traka" (engl. *sidebar*), koja prikazuje kategorije po grupama. Također omogućuje odlazak na stranicu vlastitih projekata i stranicu svih projekata, gdje je stranica svih projekata ujedno i početna stranica.

3.2. Korisnički doživljaj

Korisnički doživljaj predstavlja niz koraka koje korisnik prolazi da bi izvršio određenu radnju u sustavu. Izrađena aplikacija sadrži više korisničkih putovanja, čije iskustvo ovisi o razini pristupa koju korisnik posjeduje.

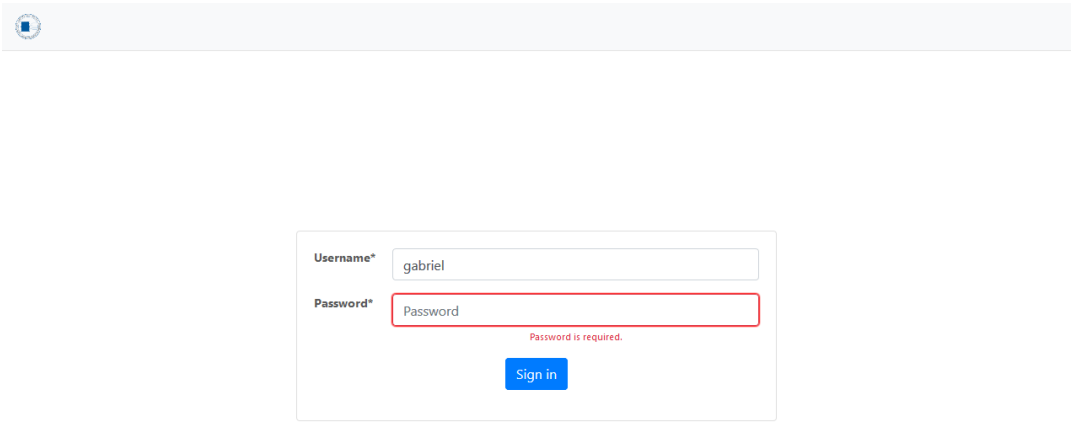
3.2.1. Provjeravanje autentičnosti

Pri prvom posjetu aplikaciji, korisnika dočekuje forma za provjeravanje autentičnosti (**slika 8**).



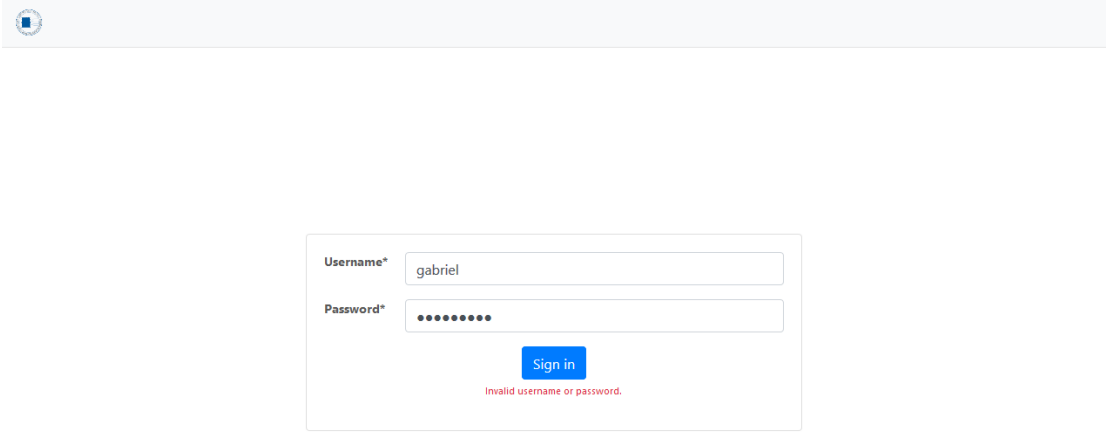
Slika 8: Forma za provjeravanje autentičnosti

Na pokušaj klika botuna koji započinje provjeru, forma podržava validaciju korisničkog unosa ukoliko nisu upisana sva polja (**slika 9**).



Slika 9: Forma za provjeravanje autentičnosti

Korisniku se također prikaže poruka ako je unos netočan. (**slika 10**).

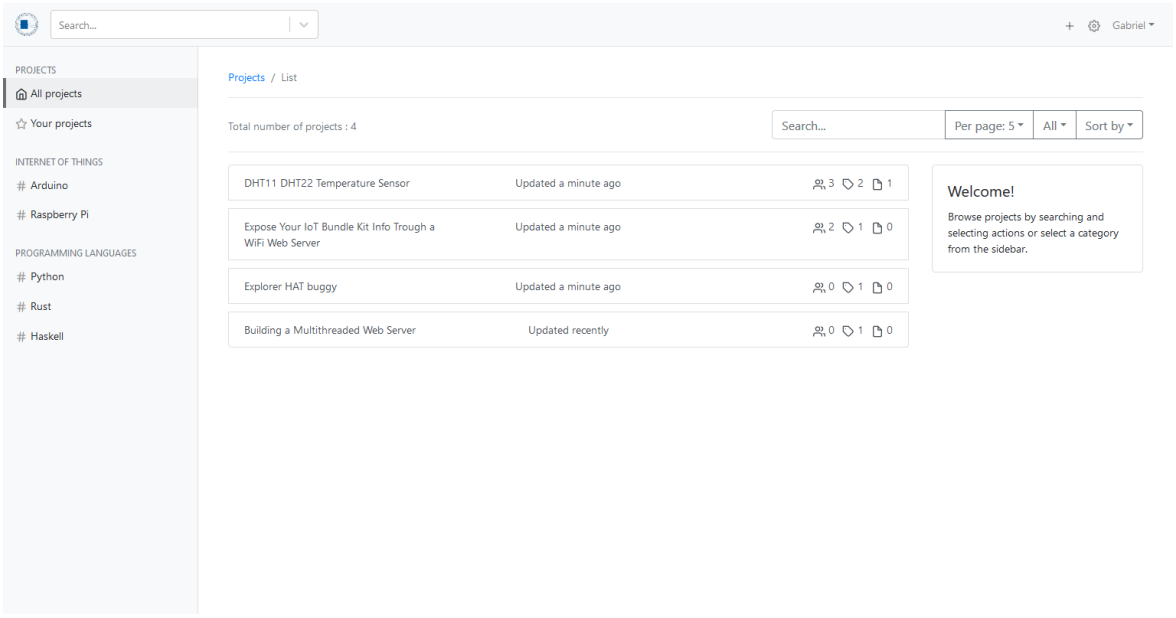


Slika 10: Signaliziranje netočnog unosa korisnika

Ukoliko je provjera autentičnosti uspješna, korisnika se preusmjerava na početnu stranicu pregledavanja projekata.

3.2.2. Pregledavanje projekata

Početni pogled aplikacije prikazuje sve spremljene projekte. Projekti se mogu pretraživati po ključnoj riječi, filtrirati i sortirati po različitim kriterijima. Također je podržana paginacija i moguće je odabrati broj projekata po stranici. Svaki redak u pregledu predstavlja jedan projekt, gdje se vidi naslov projekta, kada je zadnji put ažuriran i brojčani sažetak koji govori koliko je kategorija, osoblja i datoteka vezano za projekt. (**slika 11**)



Slika 11: Lista svih projekata

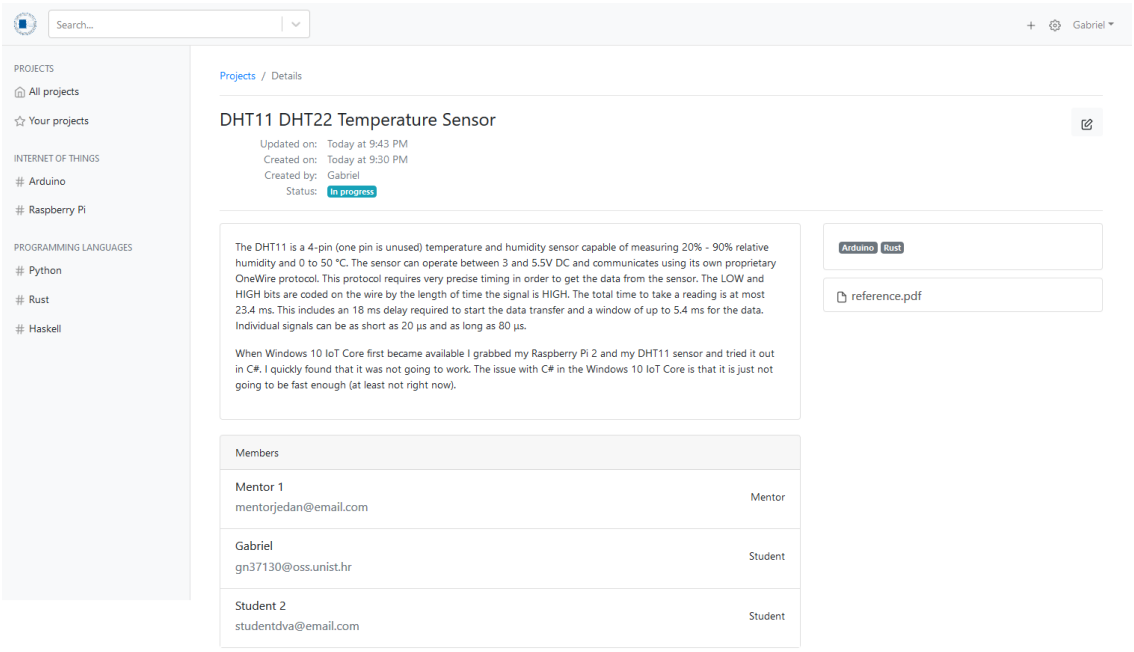
Na vrhu pogleda se nalazi horizontalna navigacijska traka koja predstavlja podlogu svakog ostalog pogleda aplikacije. Na njoj su kontrole za pretragu, odlazak na postavke i padajući izbornik u obliku korisničkog imena, preko kojeg se korisnik može odjaviti iz aplikacije.

Lijevo od liste projekata se nalazi vertikalna navigacijska traka, koja također spada u podlogu svakog pogleda i prikazuje različiti sadržaj ovisno o njegovom kontekstu. U kontekstu pogleda vezanih uz projekte, u takvoj navigacijskog traci se prikazuju kontrole za odlazak na sve projekte, odlazak na vlastite projekte i prikazane su grupe.

Grupe u aplikaciji služe navigacijskoj svrsi, gdje se ispod naslova grupe poredaju kategorije koje su vezane uz nju. Kategorije stvara i uređuje administrator u postavkama aplikacije.

3.2.2.1. Pregled pojedinog projekta

Ako se redak u listi projekata klikne, korisnika se usmjeri na sljedeći pogled koji prikazuje podatke vezane uz odabrani projekt. (**slika 12**).



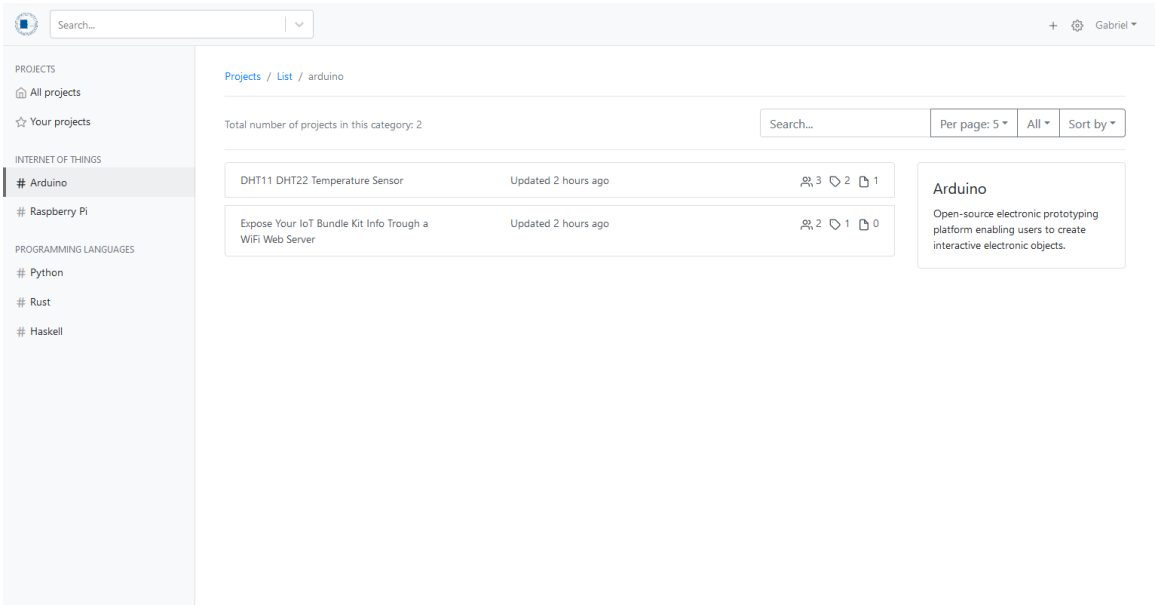
Slika 12: Lista svih projekata

Pogled projekta započinje naslovom, ispod kojeg se nalaze datumi njegove revizije, kao i tko je autor i trenutačni status projekta. Status projekta može biti u nastajanju (engl. *In progress*), neriješen (engl. *Pending*) i završen (engl. *Done*). Spremljeni projekti koji nemaju dodijeljeno osoblje su pod statusom neriješeni, gdje u suprotnome su pod statusom nastajanja. Projekti se u procesu uređivanja moraju eksplicitno označiti kao završeni ako je potrebno da budu u tom statusu.

Ispod oznake statusa projekta se nalazi sami opis projekta, ispod čega slijedi lista studenata i mentora koji su dodijeljeni projektu. Desno od opisa projekta se nalaze oznake kategorija na koje je projekt vezan i datoteke projekta koje se mogu preuzeti.

3.2.2.2. Pregled svih projekata pojedine kategorije

Grupe na vertikalnoj navigacijskoj traci predstavljaju brzi izbornik pristupanju nekoj kategoriji, gdje klik na ponuđenu usmjerava korisnika na listu svih projekata te kategorije (**slika 13**).

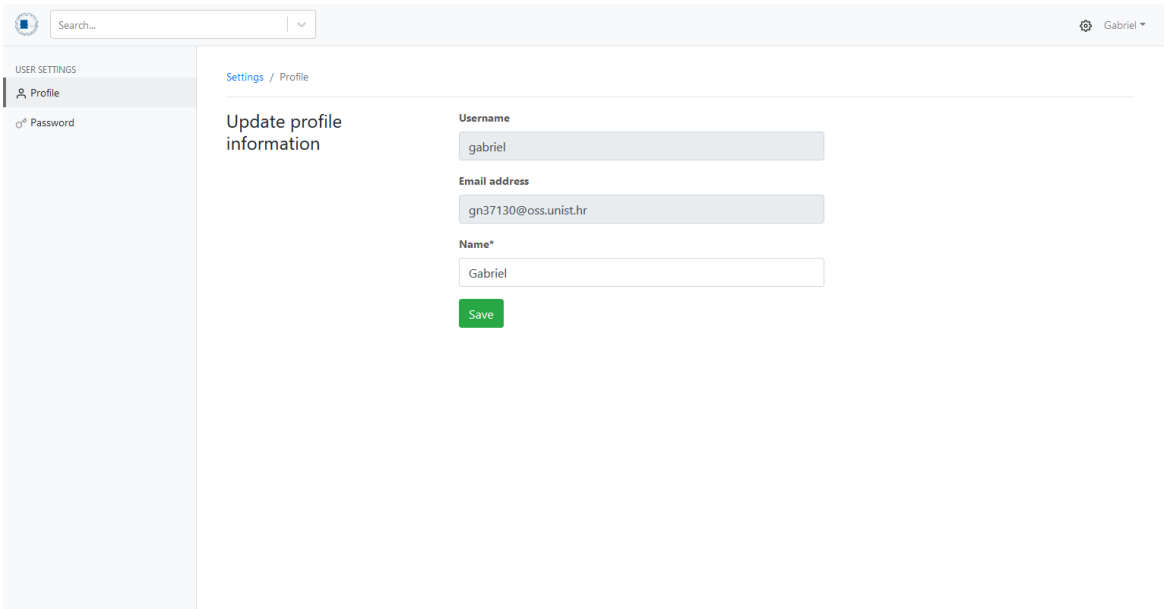


Slika 13: Projekti odabrane kategorije

Uz već spomenute funkcionalnosti pretraživanja projekata, desno od same liste je moguće vidjeti naslov kategorije o kojoj se radi, kao i njezin opis.

3.2.3. Mijenjanje korisničkih postavki

U aplikaciji je moguća promjena korisničkih postavki. Trenutačno je dostupna jednostavna funkcionalnost mijenjanja imena korisnika. Klikom botuna se ažurira entitet samog korisnika. (**slika 14**)

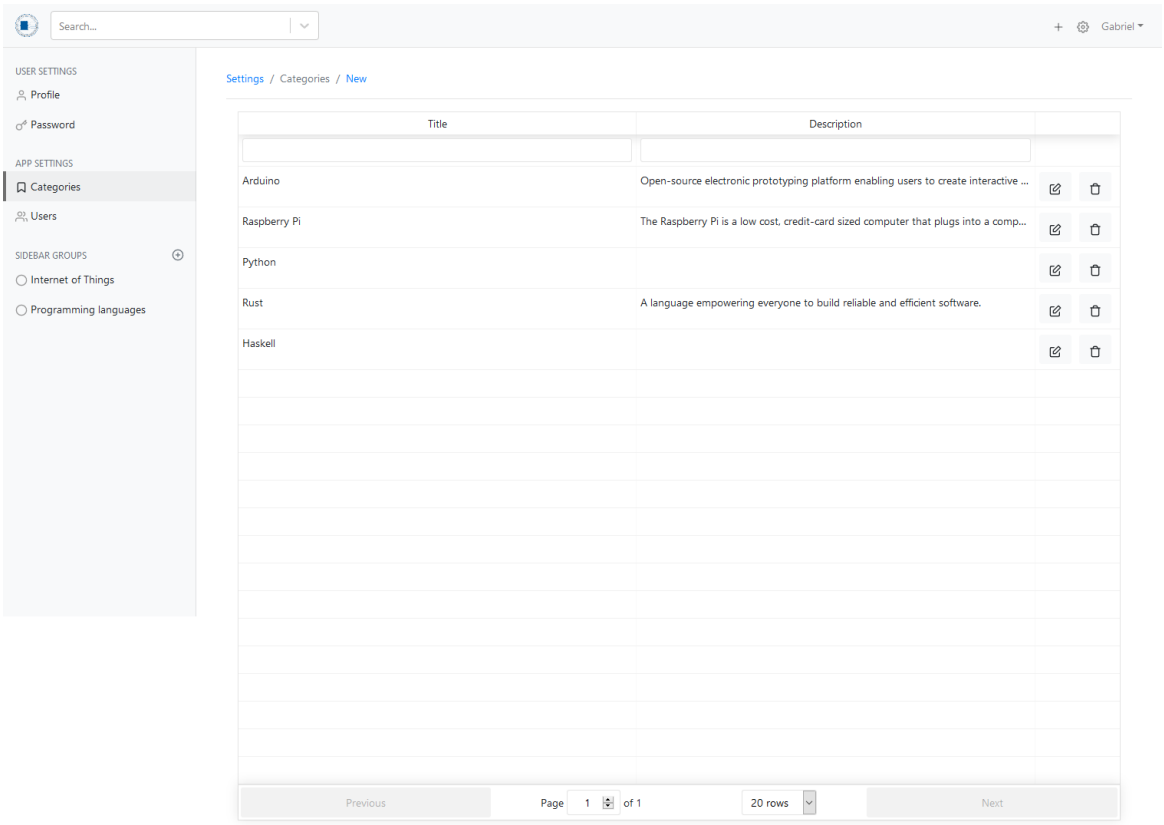


Slika 14: Promjena podataka korisnika

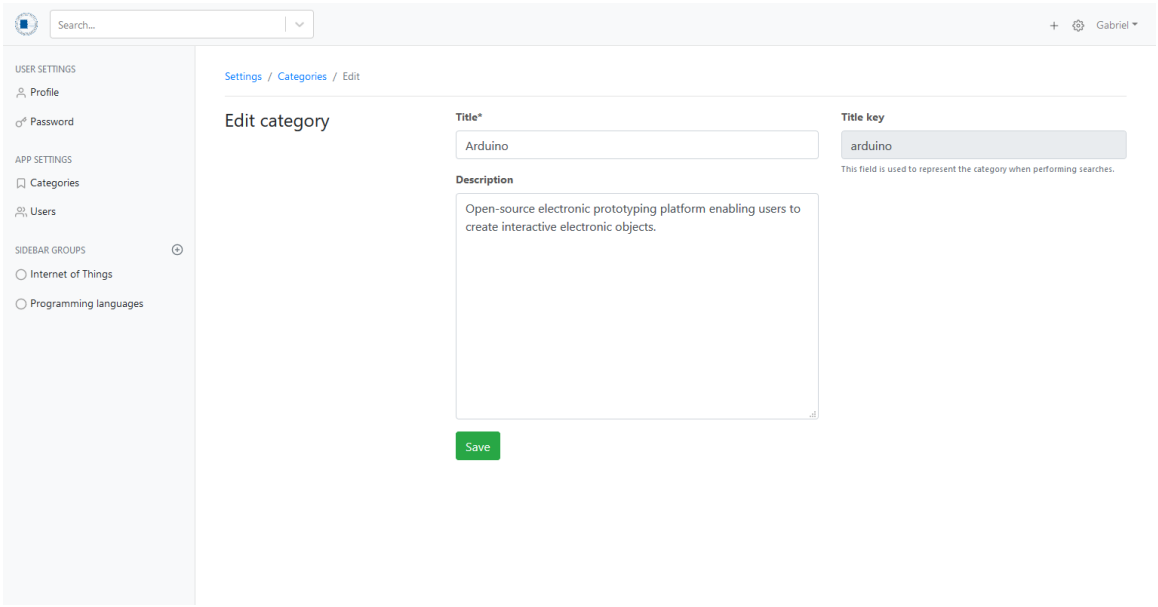
Također je moguća promjena lozinke. Potrebno je upisati trenutačnu lozinku, nakon čega slijedi ponavljano pisanje nove, gdje se obavezno nova lozinka treba podudarati s ponovljenom. Nakon što se klikne botun za ažuriranje, gdje je ažuriranje uspješno, korisnika se odjavi iz sustava, gdje ga se usmjeri na formu za provjeravanje autentičnosti.

3.2.3.1. Mijenjanje postavki aplikacije

Za korisnika koji je administrator, ponuđene su dodatne opcije u vertikalnoj traci na pogledima postavki. Moguće je otići na pregled svih kategorija (**slika 15**), gdje klikom na **`New`** u zaglavlju stranice je moguće dodavanje nove kategorije. Ista forma se koristi za uređivanje postojećih (**slika 16**).



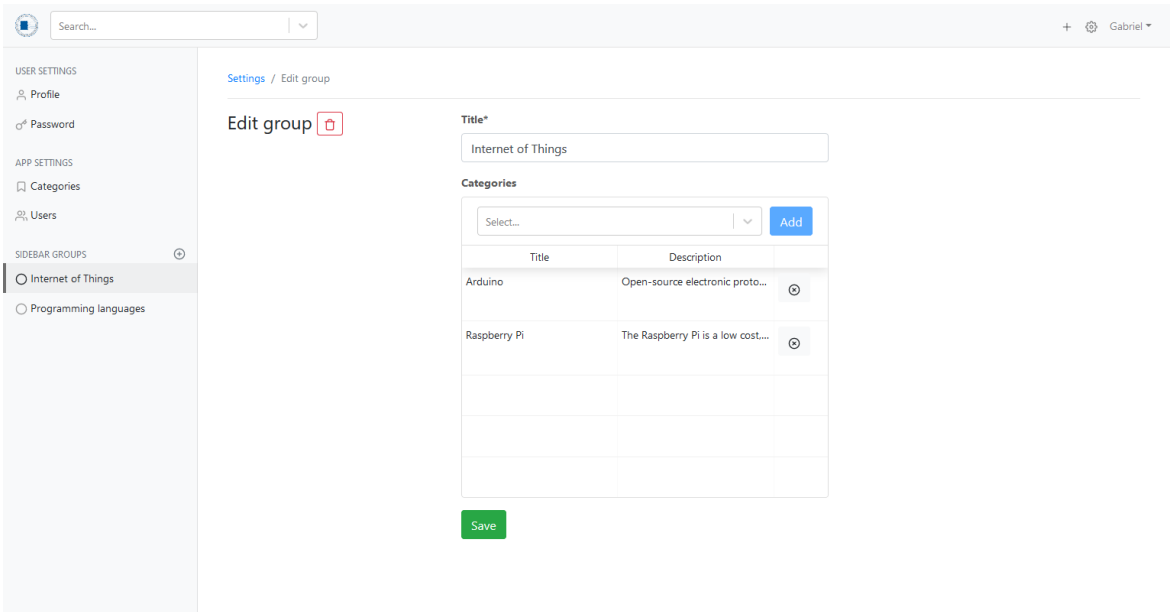
Slika 15: Pregled dostupnih kategorija



Slika 16: Uređivanje odabrane kategorije

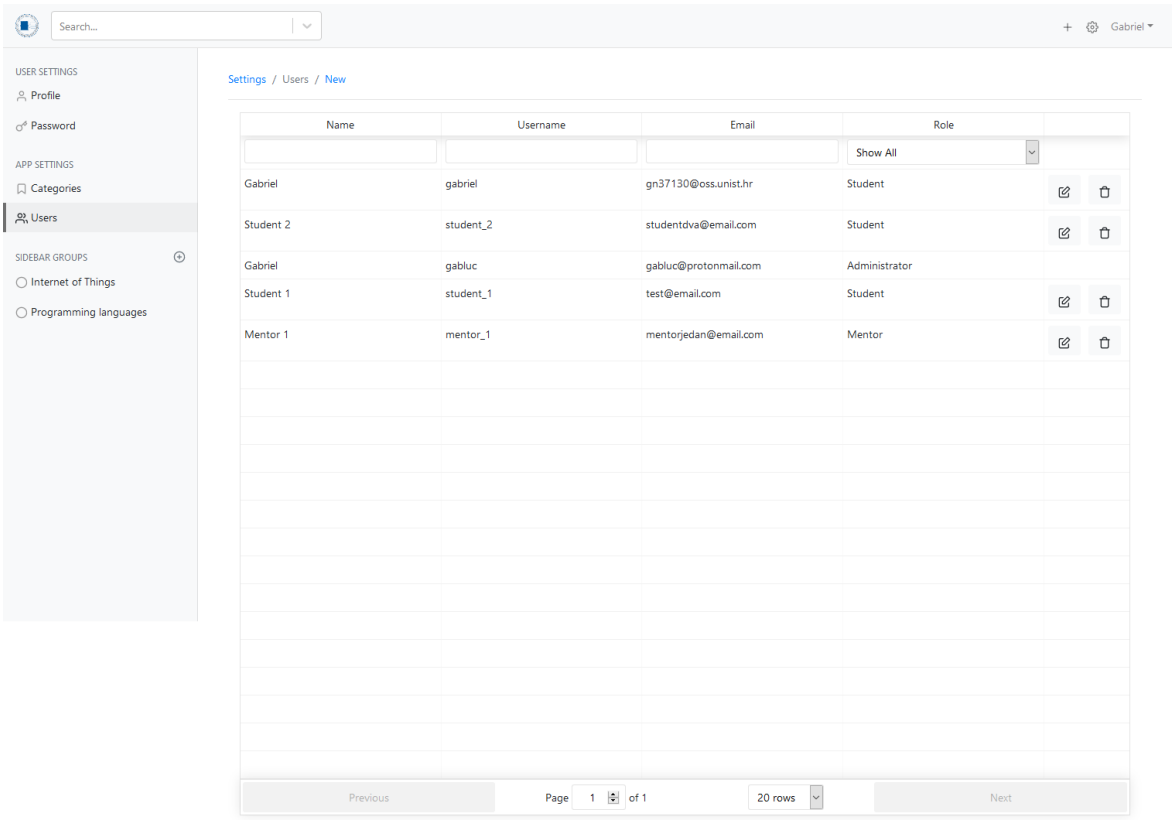
Za svaku kategoriju se generira nazivni ključ. S nazivnim ključem je moguće pristupiti projektima neke kategorije tako da se on navede u putanji aplikacije. Primjer naziva kategorije bih bio 'Nova Kategorija', čiji bi nazivni ključ bio 'nova-kategorija', gdje bi onda putanja s nazivnim ključem bila **`/projects/list/nova-kategorija`**.

Uređivanje grupe započinje navođenjem naziva grupe, gdje se onda u tablicu dodaju postojeće kategorije u aplikaciji koje će se vezati za tu grupu (**slika 17**)



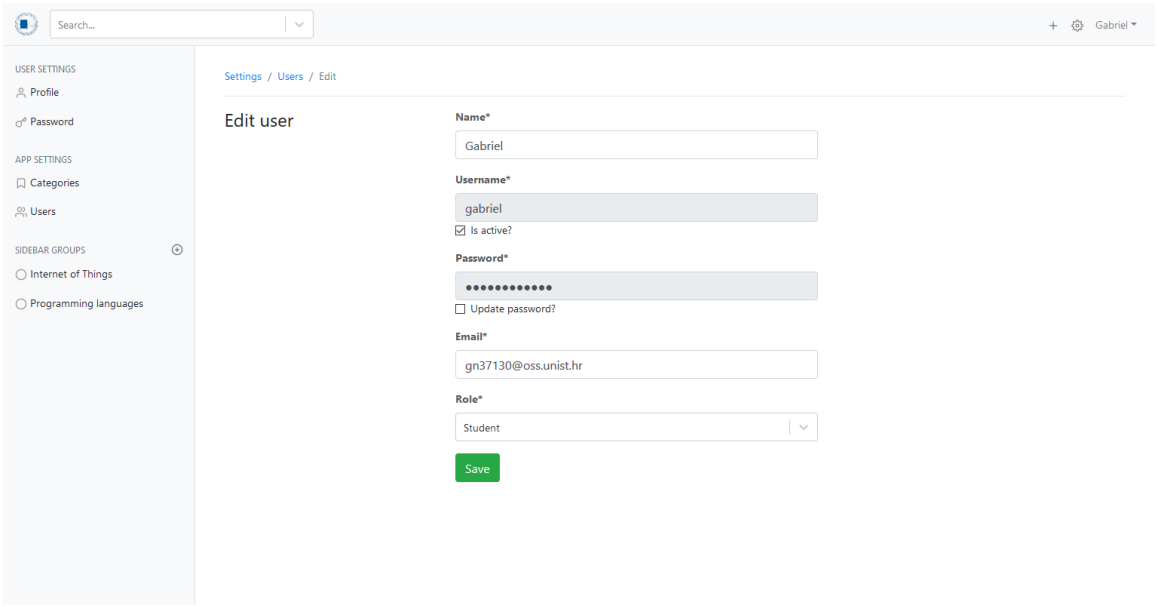
Slika 17: Uređivanje odabrane grupe

Administratori jedini imaju mogućnost pregledavanja svih korisnika u sustavu pomoću tablice (**slika 18**).



Slika 18: Tablica svih korisnika aplikacije

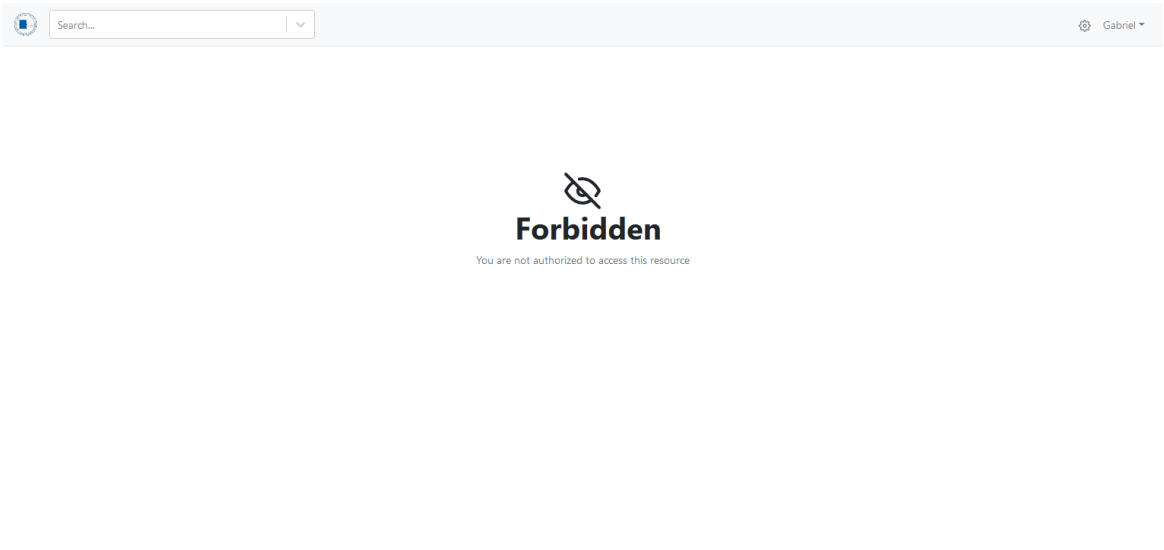
Klikom na **`New`** započinje dodavanje novog korisnika. Isto tako, moguće je odabrati postojećeg putem tablice. Forma za uređivanje dozvoljava unos imena, korisničkog imena, lozinke, elektroničke adrese i uloge korisnika. Također se može označiti je li korisnika aktivan u aplikaciji. Aktivnost korisnika predstavlja mogućnost prijave u aplikaciju s njegovim korisničkim imenom i lozinkom. Kada se uređuje korisnik, moguće je označiti je li se događa ažuriranje lozinke (**slika 19**).



Slika 19: Uređivanje korisnika

3.2.4. Dozvola pristupa

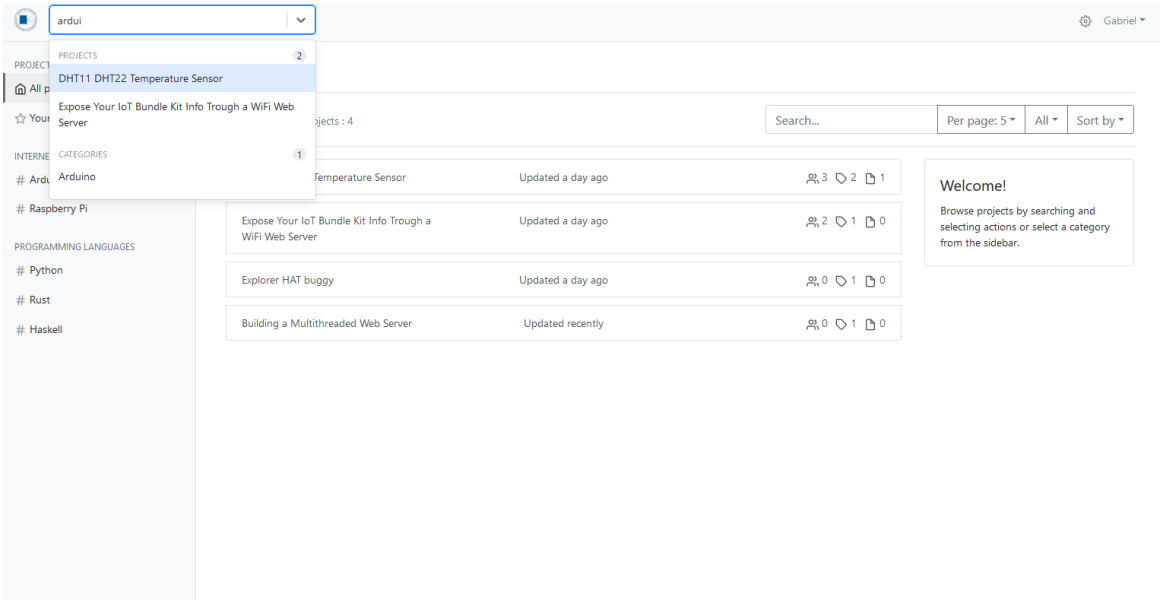
Kroz aplikaciju su određene interakcije korisnika skrivene ovisno o samoj ulozi korisnika. Za razliku od administratora, studentima i mentorima nisu vidljive kontrole za uređivanje novih projekata. Na vertikalnoj navigacijskoj traci u postavkama su samo vidljive korisničke postavke, dok su one vezane uz aplikacije skrivene. Ukoliko korisnik pokuša pristupiti spomenutim pogledima, namještajući putanju u adresnoj traci preglednika, dočekati će ga pogled koji objašnjava korisniku da pristupa zabranjenom sadržaju (**slika 20**).



Slika 20: Pristupanje zabranjenom resursu.

3.2.5. Globalna pretraga

Na horizontalnom navigacijskom zaglavlju se nalazi unos koji predstavlja globalnu pretragu. Korisnik unosom ključne riječi pretražuje projekte, gdje klikom na odabrani odlazi direktno na njegov pregled. Istom ključnom riječju se pretražuju projekti po nazivu projekta, nazivu kategorije, imenu člana projekta i nazivu datoteke. Ako se ključnom riječju pronađe kategorija, njezinim klikom se odlazi na pregled svih projekata te kategorije (**slika 21**).



Slika 21: Pristupanje zabranjenom resursu.

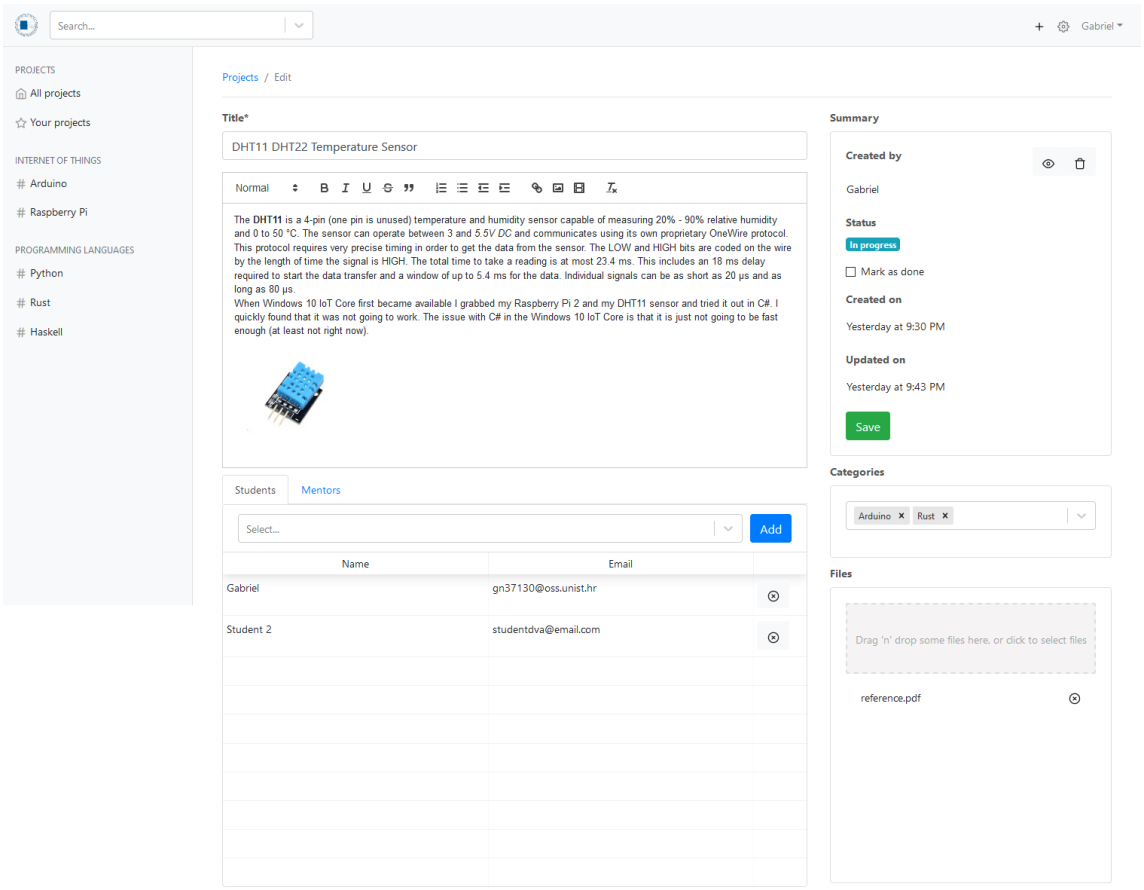
Istražena je mogućnost da se također pretražuje po opisu projekta tako da se na stupac tablice koja predstavlja opis definira indeks pretraživanja, ali je pokušaj bio neuspjeli. Problem se nalazio u definiranju spremljene procedure koja obavlja pretragu kroz Entitetski Okvir.

3.2.5. Izrada projekta

Vođenju projekta prethodi izrada projekta, kod koje se započinje unosom naziva i opisa sâmog projekta. Za opis je moguće bogato uređivanje, gdje se u tekst mogu lijepiti slike, tekst se može zacrtavati, podebljavati i slično.

Desno od unosa naslova i opisa se nalazi sažetak projekta. Kod sažetka projekta je moguće vidjeti tko je stvorio projekt, njegov status i vremena stvaranja i ažuriranja projekta, ispod čega se nalazi botun za snimanje. Ispod unosa opisa projekta se nalazi tablica za

dodavanje članova projekata**.** Karticama koje predstavljaju uloge se može odabrati dodavanje mentora ili dodavanje studenta (**slika 22**).



Slika 22: Uređivanje projekta

Ispod sažetka projekta se nalazi odabir za dodavanje kategorija na projekt, u kojem je moguće i pretraživanje po ključnoj riječi.

Ispod odabira za kategorije se nalazi kontrola za dodavanje datoteka u projekt, koja je omogućena tek kada se projekt spremi u aplikaciju. Kada se doda datoteka, spremi se u izvršni direktorij aplikacije. U tom direktoriju se nalazi jedinstveni direktorij projekta u kojemu se nalazi spremljena datoteka s jedinstvenim novim imenom. Pravo ime datoteke je spremljeno u tablicu baze podataka, gdje je također spremljeno jedinstveno ime, kao i identifikator projekta kojem pripada. S ovim pristupom je omogućeno uzastopno spremanje datoteka s istim nazivima, bez da se dogode konflikti. Kada korisnik pokuša preuzeti datoteku, mrežni poslužitelj će je dohvatiti po jedinstvenom imenu u zadanoj putanji i poslužiti će je korisniku s njezinim pravim imenom.

4. Zaključak

Kroz izradu rada se težilo ostvarenju željene specifikacije. Samo ostvarenje specifikacije zapravo nije bio krajnji cilj, nego povod da se ostavi podloga za daljnji razvoj aplikacije. U izradi se težilo koristiti dobro dokazane programske jezike, okvire i alate otvorenog kôda, čija fleksibilnost dopušta izvršavanje na operativnom sustavu vlastitog odabira. Izrađen je sustav koji sadrži osnovnu funkcionalnost za održavanje i dokumentiranje projekata. Sama njegova implementacija je organizirana u slojeve, čija je svrha ostaviti prostor za laku promjenu i nadogradnju. Takva slojevitost dopušta razvoj dodatnih aplikacija koje mogu služiti kao nadopuna postojećoj, tako da se one izgrade na samoj jezgri, koja je namijenjena za višekratnu upotrebu. **`OssDocs.Domain`** projekt je upravo ta jezgra, koja sadrži svu potrebnu poslovnu logiku na kojoj se može dalje graditi.

Usredotočenost na slojevitost i višekratnu iskoristivost dolazi pod cijenu povećane složenosti same aplikacije. Bilo bi poželjno smanjenje podloge za dodavanje nove funkcionalnosti kod arhitekture korisničkog sučelja. Organizacija izvršavanja ažuriranja baze podataka se može odvojiti u zasebni projekt, koji je zapravo neovisan o aplikaciji koja tu bazu podataka koristi. Također, stečen je osnovni mehanizam provjeravanja autentičnosti kod SPA aplikacije, gdje se više pažnje može uložiti u implementiranje najbolje prakse.

5. Literatura

[1] Microsoft, "Hands On Lab: Build a Single Page Application (SPA) with ASP.NET Web API and Angular.js" , https://docs.microsoft.com/en-us/aspnet/web-api/overview/getting-started-with-aspnet-web-api/build-a-single-page-application-spa-with-aspnet-web-api-and-angularjs (posjećeno 10.9.2019.)

[2] Microsoft, "Solutions and projects in Visual Studio", https://docs.microsoft.com/en-us/visualstudio/ide/solutions-and-projects-in-visual-studio?view=vs-2019 (posjećeno 10.9.2019.)

[3] Microsoft, "Entity Framework Core tools reference - .NET CLI", https://docs.microsoft.com/en-us/ef/core/miscellaneous/cli/dotnet (posjećeno 10.9.2019.)

[4] Autofac, "Getting Started", https://autofac.readthedocs.io/en/latest/getting-started/index.html (posjećeno 10.9.2019.)

[5] Autofac, "Instance Scope", https://autofaccn.readthedocs.io/en/latest/lifetime/instance-scope.html#instance-per-dependency (posjećeno 10.9.2019.)

[6] JWT, "Introduction to JSON Web Tokens" , https://jwt.io/introduction/ (posjećeno 10.9.2019.)

[7] TypeScript, "Documentation" , https://www.typescriptlang.org/docs/home.html (posjećeno 10.9.2019.)

[8] Martin Fowler, "Inversion of Control" , https://martinfowler.com/bliki/InversionOfControl.html (posjećeno 10.9.2019.)

[9] React, "Getting Started", https://reactjs.org/docs/getting-started.html (posjećeno 10.9.2019.)

[10] MobX, "The gist of MobX", https://mobx.js.org/intro/overview.html (posjećeno 19.9.2019.)