SUSTAV ZA KOLABORACIJU MENTORA I STUDENATA

Vuletić, Ante

SVEUČILIŠTE U SPLITU

SVEUČILIŠNI ODJEL ZA STRUČNE STUDIJE

Preddiplomski stručni studij Informacijska tehnologija

ANTE VULETIĆ

Z A V R Š N I R A D

SUSTAV ZA KOLABORACIJU MENTORA I STUDENATA

Split, rujan 2021.

SVEUČILIŠTE U SPLITU
SVEUČILIŠNI ODJEL ZA STRUČNE STUDIJE

Preddiplomski stručni studij Informacijska tehnologija

Predmet: Operativni sustavi

Z A V R Š N I R A D

Kandidat: Ante Vuletić

Naslov rada: Sustav za kolaboraciju mentora i studenata

Mentor: Nikola Grgić, viši predavač

Split, rujan 2021.

Sadržaj

SAŽETAK

U završnom radu opisana je izrada web aplikacije napravljene s ciljem da olakša posao mentoru prilikom pregleda predanih završnih radova, tako da na jednom mjestu ima jednostavan pregled promjena unutar programskog koda i promjena unutar dokumenta završnog rada. Studentu se omogućava učitavanje te pregled dokumenata završnog rada po verzijama. Kao dodatna opcija omogućena je integracija s sustavom Github. Sve promjene nad dokumentima i repozitorijem unutar sustava Github, sustav Student Mentor prikazuje na pregledan način uz pomoć poruke. Navedene poruke mentori i studenti mogu komentirati te na taj način poboljšati komunikaciju u suradnji.

Ključne riječi: Github, Kolaboracija, Net Core, React, Utičnice.

SUMMARY

Student Mentor Collaborator

The purpose of the web application is to allow a mentor to manage students documentation, code and changes to their code, as well as keep track of their final test/examination papers. Any documentation uploaded is versioned to easily identify the lastest submissions. Students can also register repositories on Github that similarly allow the student to track changes as well as automatically inform the mentor of changes to that repository To allow for improved communications, all changes and updates to documentation, code or repositories are displayed as messages for both the mentor and individual student to review. This also allows for them to comment within these message threads and improves the communication between parties.

Keywords: Github, Collaboration, Net Core, React, Sockets.

1 UVOD

Izrada završnog rada na odjelu za stručne studije odvija se na sljedeći način: student se na početku javlja dodijeljenom mentoru za dogovor teme završnog rada. Nakon dogovora oko teme i uvjeta koje je potrebno ispuniti, student kreće s izradom završnog rada. Nakon dogovora teme i uvjete koje je potrebno ispuniti, student kreće s izradom završnog rada. Student se tijekom izrade s napretkom javlja mentoru za smjernice ili pregled odrađenog, ukoliko je riječ o programskom projektu, mentor pregledava kôd, zatim dokument završnog rada. Komunikacija između mentora i studenta prvenstveno se odvija putem elektroničke pošte. Elektronička pošta je službeni oblik komunikacije, sama komunikacija može biti usporena i mentoru može biti teško pratiti napredak završnog rada. Olakšavanje praćenja promjene je glavna motivacija za izradu web aplikacije za kolaboraciju mentora i studenta, cilj je olakšati komunikaciju između mentora i studenata. Web aplikacija ima cilj olakšati početnu interakciju implementacijom razmjene poruka u stvarnom vremenu *(engl. real time)* što je postignuto uz pomoć web utičnica *(engl. web socket).* Kao što je navedeno, u slučaju promjene u kodu, vrlo često mentor neće na jednostavan način primijetiti istu, dok student u slučaju da mu je potrebna pomoć od mentora treba dodatno slati elektroničku poruku mentoru. Kako bi proces bio ubrzan, aplikacija automatski šalje poruke mentoru prilikom guranja *(engl. push)* u repozitorij. Isti pristup se primjenjuje u izvedbi iteracije nad dokumentima, nakon učitavanja dokumenta u sustav, mentor dobije poruku da je on učitan. Svaki komentar nad porukom ostaje prikačen na dokument prilikom prikazivanja dokumenta, time je olakšano praćenje nužnih promjena u pisanju dokumentacije.

- U prvom poglavlju objašnjavaju se prednosti, mane i razlozi odabranih razvojnih okvira. Usporedba s drugim opcijama, povijesni razvoj istih tehnologija i raširenost primjene tehnologije.
- U drugom poglavlju objašnjava se struktura baze podataka, arhitektura poslužitelja, implementacijski detalji.
- U trećem poglavlju pojašnjava se poslovna logika izrađene aplikacije.
- U četvrtom i posljednjem poglavlju je osvrt na završni rad i moguće nadogradnje sustava.

2 TEHNOLOGIJE

U ovom poglavlju objašnjen je odabir tehnologija, usporedba s drugim izborima, mane prednosti, raširenost, uporaba i povijest.

2.1 SQL Server

SQL Server je Microsoftova implementacija relacijske baze podataka. Relacijski tip baze podataka je odabran zbog jednostavnog načina organizacije podataka. Relacijske baze podataka su najrašireniji tip baze podataka te su u primjeni od 1980. godine [1]. Za izbor relacijskih baza podataka postoji pregršt opcija poput: MySql, SQL Servera, PostgreSQL, SQLite i sl. Prilikom razvoja aplikacije dosta bitan aspekt su razvojni alati, gledajući da se razvoj iste odvija na Windows platformi za pregled tablica i baza podataka može se koristiti Microsoft SQL Server Management Studio. Uz pomoć navedenog alata pregled podataka, konfiguracija i interakcija s bazama podataka je u veliko pojednostavljen. MySql i PostgreSQL također nude slične mogućnosti, razvojni okviri prilikom korištenja ovih baza podataka se nalaze u pregledniku. Neke od prednosti MySql i PostgreSQL baza podataka je potpora na različitim platformama kao što su Unix i MacOs, dok je potpora za SQL Server ograničena. PostgreSQL je licenciran putem otvorenog koda (*engl. open source*) licence te nema dodatnih troškova prilikom korištenja istog. Zbog implementacije poslužitelja uz pomoć razvojnog okvira .Net Core, izabran je SQL Server kao baza podataka zbog jednostavnosti upotrebe i podrške prilikom spajanja na bazu podataka.

2.2 .Net Core

Razvojni okvir .Net Core omogućuje laki početak izrade poslovne logike aplikacije. Jedna od prednosti .Net Corea je to što je licenciran kao open source razvojni okvir, što omogućuje pregled izvornog koda i kontribucije. Usporedbi s drugim razvojnim okvirima kao što su PHP, Ruby on Rails, Node i Django, .Net Core nudi višestruko bolje performanse [2] iz razloga što je kompajlirani jezik što omogućava optimizacije prilikom prevođenja u CLR (*engl. Common Language Runtime*). U verziji .Net Core 5 stara verzija .Net-a je spojena sa MONO razvojnim okvirom te je sada moguće prevesti kôd u assembly, što omogućuje izvršavanje na Unix sustavima bez popratnih modula. Još jedan od pluseva, .Net Core okruženja je biblioteka Entity Framework koji pruža jednostavnu sintaksu za spremanje, čitanje, brisanje i ažuriranje podataka. Za korištenje razvojnog okvira .Net Core u upotrebi je jezik C#, koji je sada u desetom izdanju. C# je objektno orijentirani jezik koji

je čvrsto tipkan (*engl. strongly typed*), u novim izdanjima jezika povećava se podrška za koncepte iz funkcionalnog programiranja. Neke od koncepata su nepromjenjivost, podudaranje uzoraka (*engl. pattern matching*), tuple, lambda izrazi i sl. Kako se radi o razvojnom okviru u širokoj primjeni postoji velika zajednica programera koja omogućava laki pronalazak rješenja za većinu izazova prilikom razvoja aplikacije.

2.3 React

Za izradu klijentske aplikacije korištena je biblioteka React. React je Facebook-ovabiblioteka koja omogućuje laku manipulaciju objektima preglednika i organizaciju elemenata stranice u obliku komponenti. Kako bi kôd stranice bio pregledniji upotrebljava se posebna sintaksa pod nazivom JSX. JSX omogućava pisanje HTML-a i JS unutar iste datoteke tako da se logika i HTML elementi pišu unutar istih linija. U novijim inačicama biblioteke React upotrebljavaju se hook funkcije, koje omogućuju spremanje stanja sučelja unutar funkcionalnih komponenti. U starijim inačicama koristile su se klasa komponente za spremanje stanja i metode životnog vijeka (*engl. lifecycle methods*) koje su zamijenjene hook sintaksom. React je najraširenije korištena JavaScript biblioteka odmah nakon biblioteke JQuery. Jedan od glavnih značajki biblioteke React je virtualni DOM koji smanjuje broj iscrtavanja na ekran. Iscrtavanje na ekran je najskuplja operacija koja se može izvršiti unutar preglednika. Drugi izbori pored Reacta su Vue i Angular razvojni okviri, odabran je React zbog jednostavnosti upotrebe.

2.4 SignalR

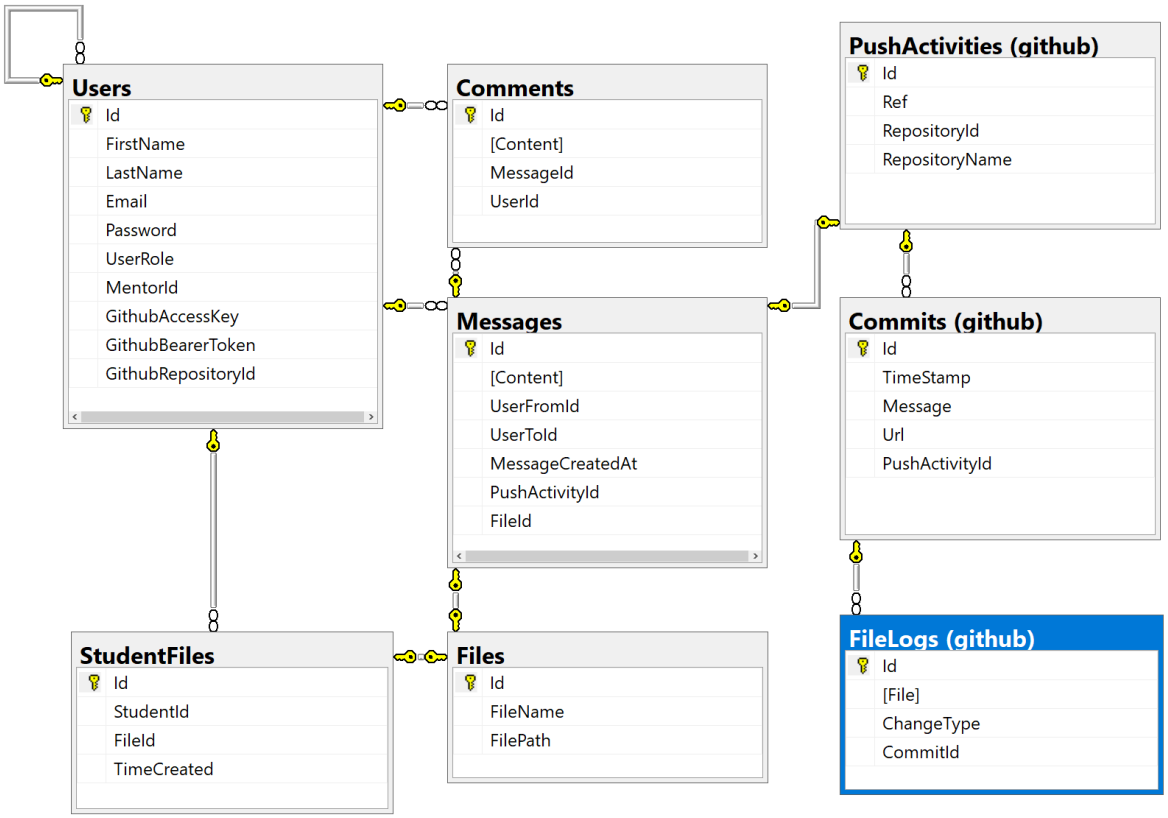
Biblioteka SignalR je dio razvojnog okvira .Net Core koja omogućava razmjenu poruka između klijenta i poslužitelja u stvarnom vremenu. Implementirana uz pomoć web utičnica te za starije preglednike koristi različite tehnike kako bi se postigao isti efekt. Web utičnice su napredna web tehnologija koja omogućuje otvaranje dvostrane sesije za komunikaciju između klijenta i poslužitelja. Moguće je slati događaje putem sesije i slušati iste događaje bez da se koristi metoda *long polling* za odgovor od poslužitelja. WebSocket protokol je definiran od strane Internet Engineering Task Force-a [3] te je standardiziran za implementaciju u više različitih tipova poslužitelja kao što je Node, Java i .Net Core.

3 IZVEDBA APLIKACIJSKE LOGIKE

U ovom poglavlju biti će pojašnjenja struktura relacijske baze podataka, arhitektura poslužitelja, arhitektura klijenta, implementacijski detalji i poslovna logika.

3.1 Struktura relacijske baze podataka

U svrhu spremanja podataka koji su nužni za izvedbu web aplikacije, korišten je relacijski dijagram prikazan na `Slika 1`.



Slika 1 Relacijski dijagram

Na početku rada napravljena je tablica korisnici (*engl. users*), jer od nje kreće glavni dio implementacije. Jedan od osnovnih zahtjeva aplikacije je podrška za različite korisničke uloge kao što su: mentor, student i administrator aplikacije. Te je postignuto uz pomoć atributa korisnička uloga (*engl. user role*) koja se nalazi u tablici korisnici. Za prijavu u sustav nužna je elektronička pošta. U svrhu razlikovanja različitih korisnika, elektroničke pošte trebaju biti jedinstvene da bi bilo poznato o kojem korisniku je riječ. Kako bi elektronička pošta bila jedinstvena unutar sustava koristi se sljedeći odsječak koda prilikom konfiguracije entiteta korisnik.

```
builder.HasIndex(u => u.Email)
       .IsUnique();
```

Ispis 1 Konfiguracija svojstva

Navedeni odsječak dodaje indeks nad atributom elektronička pošta, što onemogućuje spremanje više istih elektroničkih adresa. Tip indeksa u upotrebi je neskupljeni (*engl. non-clustered*) jedinstveni indeks. Zbog potrebe povezivanja više različitih korisničkih uloga odlučena je uporaba tehnike tablica po hijerarhiji (*engl. TPH, table per hierarchy*). Što omogućuje stvaranje različitih vrsta objekata iz jedne tablice. Kako bi TPH radio, potrebno je definirati stupac diskriminator (*engl. discriminator column*) po kojem će se različiti objekti stvarati [4].

```
builder
       .HasDiscriminator(u => u.UserRole)
       .HasValue<Student>(UserRole.Student)
       .HasValue<Mentor>(UserRole.Mentor)
       .HasValue<Admin>(UserRole.Admin);
```

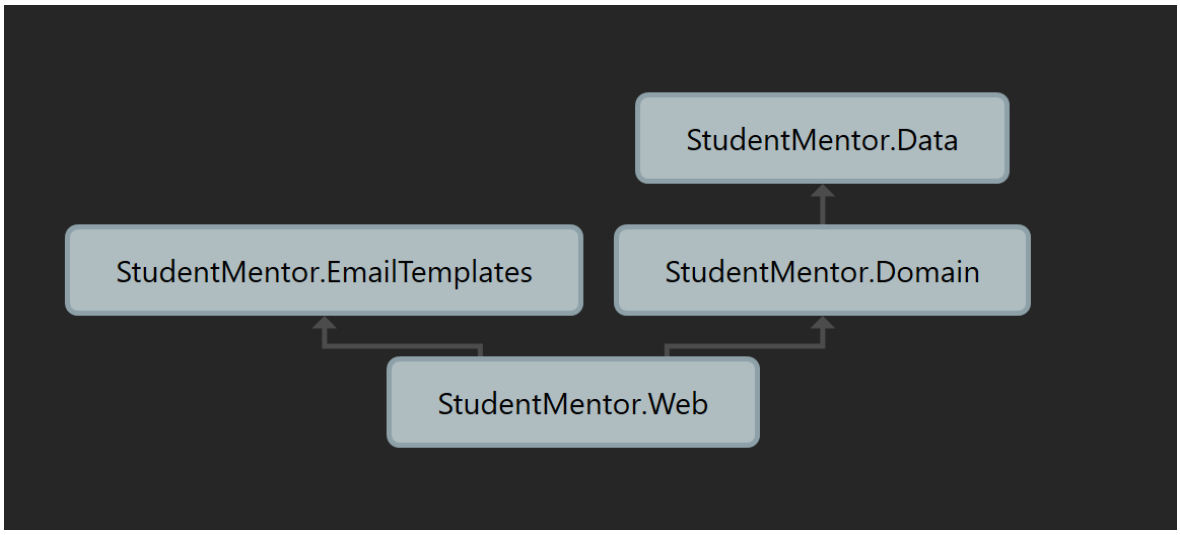
Ispis 2 Konfiguracija diskriminatora

Navedeni odsječak koda definira vrijednosti koje atribut uloga korisnika treba poprimiti da bi se kreirale točne vrste objekata, više o tome u potpoglavlju Arhitektura poslužitelja. Kako bi se postigla asocijacija mentora i studenta dodan je strani ključ `MentorId` nad tablicom korisnik, vrsta relacije u uporabi je korisnik jedan na prema više. Očekivana upotreba integracije s Github-om je da će korisnik, uloge student povezati samo jedan korisnički račun iz sustava s Github sustavom. Kako bi navedeno bilo omogućeno korišteni su atributi `GithubAccessKey` i `GithubBearerToken`, više o njihovoj upotrebi u potpoglavlju Github integracija. Nakon povezivanja računa, od studenta je očekivano biranje repozitorija koji sadrži kôd završnog rada, što je postignuto dodavanjem atributa `GithubRepositoryId`. Tablica `Messages` se koristi u svrhu razmjene poruka među različitim korisnicima. Glavna svrha je komunikacija između korisničkih uloga student i mentor. Kako bi mogli znati tko je izmijenio poruku koriste se atributi `UserFromId` i `UserToId`. Oblici poruka koji su podržani su: tekstualna poruka, automatizirana poruke za aktivnost nad repozitorijem i poruka prilikom predaje datoteke završnog rada. Tekstualni sadržaj poruke se sprema unutar atributa `Content`. Aktivnost nad repozitorijem se asocira pute stranog ključa `PushActivityId`. Poruka za obavijest o predaji verzije dokumenta završnog rada asocira se putem stranog ključa `FileId`. Tablica `Comments` se koristi za komentiranje dolazećih poruka te je u relaciji jedan na prema više s tablicom `Messages`, ostvarena stranim ključem `MessageId`. Sadržaj komentara je u tekstualnom obliku i sprema se u atribut `Content`. Kako bi bilo poznato koji je korisnik

poslao poruku također se sprema njegov jedinstveni identifikator u atribut `UserId`. Tablica `Files` koristi se za predaju završnih radova, na način da se spremaju podaci o datoteci koja je pohranjena na poslužitelju što je postignuto atributima `FilePath` i `FileName`. Gdje atribut `FilePath` predstavlja putanju do datoteke i atribut `FileName` predstavlja naziv datoteke. Tablica `StudentFiles` služi za verziranje završnih radova te je ujedno i među tablica relacije više na prema više između entiteta `Student` i `File`. Kako bi bila poznata zadnja verzije završnog rada koriste se atribut `TimeCreated` koji predstavlja vrijeme pohranjivanja datoteke. Tablica `PushActivity` koristi se za spremanje aktivnosti nad Github repozitorijem. Atribut `RepositoryId` predstavlja jedinstveni identifikator, atribut `RepositoryName` naziv repozitorija nad kojem se dogodila aktivnost. Tablica `Commits` služi za pohranjivanje grupa izmjena nad repozitorijem. Sadrži atribute `TimeStamp`, `Uri` i strani ključ `PushActivityId`. `TimeStamp` predstavlja vrijeme stvaranja grupe izmjena, dok `Uri` predstavlja poveznicu na promjenu i `PushActivityId` je strani ključ za asocijaciju na aktivnost. Tablica `FileLogs` pohranjuju jednu izmjenu u grupi izmjena. Atribut `ChangeType` sadrži podatak o tipu promjene koji može biti dodavanje, brisanje i izmjena. Atribut `File` sadrži naziv datoteke nad kojom se dogodila izmjena. Strani ključ `CommitId` služi za asocijaciju u grupu izmjena.

3.2 Arhitektura poslužitelja

Za izvedbu poslužitelja korištena je višeslojna arhitektura (*engl. n-layer architecture*) [5], koja sadrži sljedeće slojeve: `Data`, `Domain`, `EmailTemplates` i `Web`. Na slici prikazan je dijagram relacija između slojeva. U ovom potpoglavlju biti će pojašnjen svaki sloj arhitekture.



Slika 2 Dijagram relacija među slojevima

3.2.1 Data

U Data sloju koristi se prvo kôd (*engl. code first*) pristup rada. Prvo se definiraju klase koje predstavljaju entitete unutar baze podataka. Zatim se uz pomoć biblioteka `EntityFrameworkCore.Design` i `EntityFrameworkCore.Tools` izrađuju migracije korištene za izmjene baze podataka SQL Server. U datoteci `Configurations` pohranjuje se kôd vezan za konfiguriranje entiteta. Datoteka `Entities` pohranjuje kôd vezan za definicije entiteta. Datoteka `Enum` pohranjuje sve fiksne tipove podataka poput `FileChangeType.cs`, koji sadrži tip izmjene datoteke unutar tablice `FileLogs`. Datoteka `Migrations` sadrži sve migracije korištene za izmjenu strukture baze podataka.

3.2.2 Domain

Sloj Domain služi za pohranjivanje koda za poslovnu logiku aplikacije. Datoteka `Abstractions` se koristi za apstraktne klase i sučelja (*engl. interface*). Datoteka `Constants` za pohranu svih potrebnih ne promjenjivih podataka. Datoteka `Helpers` se koristi za izradu statičnih klasa što je dobra praksa zbog upotrebe čistih (*engl. pure*) funkcija koje ne ovise o stanju objekta. Datoteka `Mappings` služi za pohranu definicija za mapiranje objekata iz `Data` sloja u `Web` sloj i obratno. Datoteka `Models` sprema definicije svih objekata potrebnih za izvedbu poslovne logike. `Repositories` definira klase za pristup podacima baze podataka. Datoteka `Services` služi za implementaciju poslovne logike. Datoteka `Resources` pohranjuje sve poruke koje se vraćaju korisniku, najčešće u slučaju kada se dogodila greška.

3.2.3 EmailTemplates

Sloj koji služi za pohranjivanje predložaka elektroničke pošte u obliku HTML, funkcionira uz pomoć razvojnog okruženja Razor. Sadrži implementaciju koja omogućuje pretvaranje elektroničke pošte u tekstualni oblik pogodan za slanje preko mreže. Datoteka `Views` sadrži sve predloške koji postoje u sustavu i organizirani su na način da postoji bazni predložak u kojem je definiran tlocrt (*engl. layout*) i stil elektroničke pošte.

3.2.4 Web

Web sloj je pristupna točka poslužitelju, sadrži upravitelje (e*ngl. controller*) i akcije koji se pozivaju uz pomoć protokola HTTP. Osim upravitelja ovaj sloj služi kao početna točka aplikacije i sadrži implementaciju spremnika za inverziju kontrole (*engl. inversion of control container*). Spremnik za inverziju kontrole omogućava laku ponovnu (*engl.*

reusability) upotrebu koda uz pomoć korištenja injekcije ovisnosti (*engl. dependency injection*).

3.3 Arhitektura klijenta

Za implementaciju klijenta korištena je biblioteka React. Za razliku od poslužitelja, arhitektura klijenta ne sadrži arhitekturu u slojevima. Arhitektura klijenta organizirana je uz pomoć datoteka grupiranim po srodnim funkcionalnostima. Datoteke u upotrebi su `Components`, `Screens`, `Services`, `Themes` i `Utils`. Datoteka `Components` sadrži sve komponente koje se koriste na više različitih zaslona (*engl. Screens*). Primjer jedne takve komponente je komponenta `Message`, koja se koristi na zaslonima za uloge student i mentor. Datoteka `Screens` sadrži zaslone za sve rute unutar sustava, kako se mijenja ruta unutar preglednika tako se i jedna od komponenti koja se nalazi u datoteci `Screens` prikazuje na zaslonu. Datoteka `Services` se koristi za imlementaciju konteksta (*engl. context*), uz pomoć kojeg se implementira uzorak dizajna (*engl. design pattern*) koji se zove Redux. Redux design pattern se koristi kako bi se stanje sučelja premjestilo na zajedničko mjesto, u svrhu korištenja na više različitih komponenti. Datoteka `Themes` definira sve konfiguracije sučelja poput tipografije i boja. Datoteka `Utils` služi za pohranjivanje funkcionalnosti koji se koriste na više različitih mjesta. `Utils` ima sličnu svrhu poput datoteke `Helpers` na poslužitelja po tome kako su sve funkcionalnosti ostvarene uz pomoć pure funkcija.

3.4 Implementacijski detalji

U ovom potpoglavlju biti će objašnjene funkcionalnosti koje se koriste u aplikaciji i omogućavaju njen nesmetan rad.

3.4.1 Kreacija tokena

U svrhu autorizacije korisnika upotrjebljen je JSON Web Token (JWT), korištenjem biblioteke `Jose.JWT` koja sadrži implementaciju najčešće korištenih enkripcijskih algoritama, u svrhu enkripcije podataka. U trenutku kada se korisnik uspješno ovjeri (*engl. authenticated*) poziva se metoda `GetJwtTokenForUser(User user)`, koja stvara token uz pomoć konfiguriranog enkripcijskog algoritma. U token se spremaju vrijednosti vezane za trenutnog korisnika, poput korisnikovog identifikatora i uloge. Kako je određeno RFC7519 standardom dodaju se sljedeća svojstva: `iss`, `aud` i `exp` [6]. Gdje `iss` predstavlja naziv poslužitelja koji je izradio token, `aud` predstavlja klijenta za kojeg je kreiran token i `exp` predstavlja vrijeme istjecanja valjanosti relativno od 01/01/1970.

```
public string GetJwtTokenForUser(User user)
{
	var expiresAt = GetCurrentSeconds + _jwtConfiguration.ExpiryMinutes *
60;
	var payload = new Dictionary<string, string>
	{
		{"iss", _jwtConfiguration.Issuer},
		{"aud", _jwtConfiguration.AudienceId},
		{"exp", expiresAt.ToString(CultureInfo.InvariantCulture)},
		{Claims.UserId, user.Id.ToString()},
		{Claims.Role, user.UserRole.ToString()}
	};

	return JWT.Encode(payload, _jwtConfiguration.GetAudienceSecretBytes(),
Algorithm);
}
```

Ispis 3 Enkripcija tokena JWT

U svrhu lakšeg upravljanja argumentima prilikom kreiranja tokena korištena je ugrađena funkcionalnost razvojnog okvira .Net Core koja omogućava dinamično konfiguriranje aplikacije putem JSON konfiguracijske datoteke.

```
"JwtConfiguration": {
    "Issuer": "student-mentor",
    "AudienceId": "student-mentor-audience",
    "AudienceSecret":
"735035B417D1BFE4F4B426A590CD31EF8966A9E714B5BB1B683D939B53A66C77",
    "ExpiryMinutes": 10
},
```

Ispis 4 Konfiguracija tokena JWT

Navedene vrijednosti se registriraju u IoC Container kako bi bile korištene unutar aplikacije. Za provjeru valjanosti izdanog tokena koristi se ugrađena funkcionalnost razvojnog okvira .Net Core.

```
services.AddAuthentication(JwtBearerDefaults.AuthenticationScheme)
    .AddJwtBearer(options =>
    {
        options.TokenValidationParameters = new TokenValidationParameters
        {
            ValidateIssuer = true,
            ValidIssuer = jwtConfiguration.Issuer,
            ValidateAudience = true,
            ValidAudience = jwtConfiguration.AudienceId,
            ValidateIssuerSigningKey = true,
            IssuerSigningKey = new
SymmetricSecurityKey(jwtConfiguration.GetAudienceSecretBytes())
        };
        options.Events = new JwtBearerEvents
        {
            OnMessageReceived = context =>
            {
                var accessToken = context.Request.Query["access_token"];
                if (!string.IsNullOrEmpty(accessToken))
                {
                    context.Token = accessToken;
                }
                return Task.CompletedTask;
```

Ispis 5 Konfiguracija provjere tokena

Kako bi se zaštitile poslužiteljske rute pomoću atributa `Authorize` potrebno je uz pomoć metoda `AddAuthentication` i `AddJwtBearer` specificirati vrijednosti za provjeru. Navedena svojstva `iss`, `aud` i `exp` se provjeravaju ukoliko zahtjev poslan protokolom HTTP sadrži token. Dodatno je potrebno dodati `JwtBearerEvents` kao dio konfiguracije u svrhu autoriziranja web utičnica korištenih u sklopu biblioteke SignalR. U svrhu stvaranja mentorskih korisničkih računa korišten je jednak pristup za kreiranje tokena. Stvoreni token se šalje kroz poveznicu u elektroničkoj pošti prilikom poziva mentora u sustav. Nakon što korisnik klikne na poveznicu unutar svog elektroničkog pretinca otvara se registracijska forma. Klijent u tom trenutku pošalje upit na poslužitelja gdje se dekriptira token i dohvaćaju se podaci o trenutačnom korisniku. Od korisnika se zahtjeva unos lozinke kako bi se dovršio registracijski proces.

```
public string GetTokenForEmailInvite(int userId)
{
    var payload = new Dictionary<string, string>
    {
        {"iss", _jwtConfiguration.Issuer},
        {"aud", _jwtConfiguration.AudienceId},
        {Claims.UserId, userId.ToString()},
    };

    return JWT.Encode(payload, _jwtConfiguration.GetAudienceSecretBytes(),
Algorithm);
}
```

Ispis 6 Metoda za stvaranje tokena za elektroničku poštu

3.4.2 Heširanje osjetljivih podataka

Kako bi se osigurala sigurnost osjetljivih podataka, oni se heširaju uz pomoć jednosmjernog algoritma definiranog od strane radne skupine za internetsko inženjerstvo. Algoritam pripada obitelji enkripcijskih algoritama definiranih standardima za javne enkripcijske algoritme iz laboratorija RSA. Naziv obitelji algoritama je PKCS. Korištena je klasa `RNGCryptoServiceProvider` za dohvaćanje nasumičnih vrijednosti korištenih u enkripcijskom algoritmu. Nasumične vrijednosti se spremaju u varijablu `salt`, svrha nasumičnih vrijednosti je da dvije jednake šifre nemaju istu heširanu vrijednost. Korištenjem klase `Rfc2898DeriveBytes` inicijalizira se objekt `pbkdf2` i metodom `GetBytes` generira se heširana vrijednost u obliku niza bajtova. Niz bajtova zatim konvertiramo u base64 string vrijednost koja je spremna za spremanje u bazu podataka.

```
private const int SaltSize = 16;
private const int HashSize = 20;
private const int NumberOfIterations = 100;

public static string Hash(string password)
{
       byte[] salt;
       new RNGCryptoServiceProvider().GetBytes(salt = new byte[SaltSize]);

       var pbkdf2 = new Rfc2898DeriveBytes(password, salt, NumberOfIterations);
       var hash = pbkdf2.GetBytes(HashSize);

       var hashBytes = new byte[SaltSize + HashSize];
       Array.Copy(salt, 0, hashBytes, 0, SaltSize);
       Array.Copy(hash, 0, hashBytes, SaltSize, HashSize);

       return Convert.ToBase64String(hashBytes);
}
```

Ispis 7 Metoda za heširanje lozinke

3.4.3 SignalR i razmjena poruka

Kako bi se poruke mogle izmjenjivati u stvarnom vremenu potrebno je primijeniti jednu od sljedećih tehnika: dugo povlačenje (*engl. long polling*), kratko povlačenje (*engl. short polling*) ili web utičnica (*engl. web socket*) [7]. Web utičnica pruža najbolje korisničko iskustvo, razlog tome je što se omogućava poslužitelju šalje poruke prema klijentu. Web utičnice nisu podržani u svim preglednicima i nisu pogodni u svim mrežnim uvjetima. Zbog toga je potrebno pratiti mrežne uvjete i podršku preglednika kako bi se primijenila tehnika long ili short polling. Long polling je tehnika kojem poslužitelj ne odgovara na poruku koju zahtjeva klijent dok informacija koju je zatražio ne postane dostupna. Dok implementacijom short polling klijent periodično šalje zahtjeve poslužitelju da provjeri da li je informacija postala dostupna. Upravo zbog navedenih kompleksnosti pogodno je koristiti biblioteku koja automatski odabire najpogodniju tehniku umjesto nas, ta biblioteka u razvojnom okviru .Net Core je SignalR. Prvotno je potrebno registrirati biblioteku SignalR u container. Sljedeće se postiže korištenjem metode `AddSignalR` u fazi konfiguracije usluga (*engl. services*) sustava.

```
services.AddSignalR();
```

Ispis 8 Konfiguracija servisa SignalR

Jedinica organizacije prilikom korištenja biblioteke SignalR je čvor (*engl. hub*). Navedeni čvorovi se trebaju registrirati u sustav usmjerivanja ruta okruženja .Net Core. Sljedeće se postiže korištenjem metode `MapHub<T>`, unutar metode za konfiguriranje ruta `UseEndpoints`

```
app.UseEndpoints(endpoints =>
{
    endpoints.MapControllers();
    endpoints.MapHub<MessagesHub>($"/{nameof(MessagesHub)}");
});
```

Ispis 9 Konfiguracija čvora

Na taj način registriran je čvor `MessageHub` u listu poznatih ruta razvojnog okvira. S klijentske strane nužno je također registrirati utičnica na čvor. Kako bi implementacija bila što jednostavnija koristi se paket za JavaScript naziva `@microsoft/signalr`. Upotrebom klase `HubConnectionBuilder` stvara se objekt koji prati stanje veze s poslužiteljem. To je postignuto uz pomoć sljedećeg koda:

```
import { HubConnectionBuilder } from "@microsoft/signalr";
const connection = new HubConnectionBuilder()
  .withUrl("/MessagesHub", {accessTokenFactory: () => localStorage.getItem("to
ken"),})
  .withAutomaticReconnect()
  .build();
async function start() {
  try {
    await connection.start();
  } catch (err) {
    setTimeout(start, 5000);
  }
}
connection.onclose(start);
export const startMessageHubConnection = async () => {
  await start();
};
```

Ispis 10 Klijentska konfiguracija

Gdje se funkcija `startMessageHubConnection` poziva u trenutku kada se pokrene osluškivanje događaja nad čvorom. Kada se ostvari veza između klijenta i poslužitelja moguće je početi razmjenjivati poruke u oba smjera. Poruke se razmjenjuju na način da se koriste identifikatori za vrste poruka. `MessageHub` klasa na poslužitelju određuje kojim korisnicima je potrebno poslati poruku. Kako bi bilo poznato koji korisnik pripada kojoj vezi nužno je slati token kao parametar poruka. Poslužitelj pristupa tokenu uz pomoć izvršavajući konteksta. Više informacija o registraciji tokena u čvor moguće je pročitati u potpoglavlju Kreacija tokena. Za razmjenu poruka koristi se metoda `SendMessage`

```
public const string SendMessageMethod = "MessageRecieved";

public async Task SendMessage(SendMessageModel message)
{
    var userId = Context.UserIdentifier;
    var messageResponse = await
_messageRepository.SendMessage(int.Parse(userId), message);
    var userToTask =
Clients.User(message.UserToId.ToString()).SendAsync(SendMessageMethod,
messageResponse);
    var userFromTask = Clients.User(userId).SendAsync(SendMessageMethod,
messageResponse);

    Task.WaitAll(userToTask, userFromTask);
}
```

Ispis 11 Metoda za razmjenu poruka

Prilikom klika na dugme Send unutar forme za slanje poruke potrebno je spremiti poruku u bazu podataka i obavijestiti sve odgovarajuće korisnike da je poruka poslana. Da bi se ubrzao proces slanja poruke upotrjebljen je paralelan način izvođenja prilikom slanja poruke svim korisnicima definiranih poslovnom logikom. Na klijentskoj strani izvedbe registriran je slušatelj događaja (*engl. event listener*) pomoću kojeg prisluškuju poruke pristigne putem utičnice. Poruka odgovarajućeg tipa izmijeni stanje `Message` konteksta objašnjenog u potpoglavlju Stanje korisničkog sučelja. Za registraciju event listener koristi se sljedeći kôd:

```
export const subscribeToMessageHub = (callback) => {
  connection.on("MessageRecieved", callback);
};
```

Ispis 12 Registracija slušatelja događaja

Gdje je `connection` objekt stanje veze s poslužiteljem. Implementacija je provedena na isti način za izmjenu komentara nad porukama.

3.4.4 Stanje korisničkog sučelja

U svrhu lakšeg praćenja stanja sučelja upotrebom biblioteke React, koristi se ugrađena funkcionalnost konteksta. Kako bi kontekst bio izrađen, biblioteka React pruža više različitih funkcija vrste kuka (*engl. hooks*). Za izvedbu praćenja poruka s klijentske strane korištena je `MessageProvider.js` datoteka. Pravila korištenja konteksta nalažu da je kontekst potrebno inicijalizirati sa svim nužnim svojstvima koje će kontekst pratiti. Stoga se kreira objekt inicijalnog stanja sljedećim kodom:

```
const initialState = {
  messages: [],
  isEndOfMessages: false,
  isMessageLoading: false,
  studentFilterId: 0,
};
```

Ispis 13 Postavljanje inicijalnog stanja konteksta

Inicijalno stanje konteksta zatim se prosljeđuje u hook `CreateContext()`, sljedećim kodom:

```
export const MessageContext = createContext({
  state: { ...initialState },
  handleStudentFilterId: () => {},
  setPage: () => {},
});
```

Ispis 14 Kreacija konteksta

Gdje su definirane funkcije koje manipuliraju kontekstom izvan datoteke `MessageProvider.js`. Kako bi ostatak aplikacije imao pristup podacima koji se nalaze unutar konteksta potrebno je definirati davatelja usluga, to je izvedeno sljedećim kodom:

```
<MessageContext.Provider value={value}>{children}</MessageContext.Provider>
```

Ispis 15 Postavljanje inicijalnog stanja konteksta

Davatelj usluga se zatim upotrebljava putem komponente `MessageProvider` na nužnom mjestu unutar hijerarhije komponenti. Komponenta `MessageProvider` koristi još sljedeće hook-ove izvedbe a to su: `useState` i `useEffect`. Hook `UseState` se koristi za praćenje jedinstvenog stanja komponente, primjer korištenja su poruke koje se nalaze unutar klijenta:

```
const [messages, setMessages] = useState([]);
```

Ispis 16 Primjer korištenja hooka useState

Gdje je `messages` niz poruka dostupnih klijentu i `setMessages` funkcija kojom se namješta stanje niza `messages`. Hook `UseEffect` se koristi ukoliko promjena stanja ima nuspojavu. Za primjer promjena studentskog filtera u listi poruka zahtjeva dohvaćanje novog niza poruka i osluškivanje na filtrirane događaje unutar utičnice.

```
useEffect(() => {
   startMessageHubConnection().then(() => {
     subscribeToMessageHub((response) => {
       if (!studentFilterId) {
         setMessages((prev) => [response.data, ...prev]);
         return;
       }

       if (
         studentFilterId &&
         (response.data.userFrom.id === studentFilterId ||
           response.data.userTo.id === studentFilterId)
       ) {
         setMessages((prev) => [response.data, ...prev]);
       }
     });
     subscribeToCommentHub((response) => {
       setMessages((prev) => {
         const prevCopied = [...prev];
         const messageIndex = prevCopied.findIndex(
           (m) => m.id === +response.messageId
         );
         prevCopied[messageIndex].comments = [
           ...prevCopied[messageIndex].comments,
           response,
         ];

         return prevCopied;
       });
     });
   });
 }, [setMessages, studentFilterId]);
```

Ispis 17 Registracija slušatelja poruka

Navedeni kôd registrira sve slušatelje događaja nad čvorom te ukoliko je nužno i poruka koja dolazi iz čvora pripada definiranom filteru dodaje poruku u niz. Ukoliko nije definiran filter za studenta sve nadolazeće poruke se spremaju u niz poruka korištenjem metode `setMessages`. Kako je funkcija `setMessages` također promjenjiva jer je definirana od strane hooka `useState` potrebno je izmijeniti hook `useEffect` na način da promjeni svoju definiciju u slučaju izmjene `setMessages` funkcije.

Za korištenje Message konteksta nužno je da su sve komponente unutar hijerarhije okružene komponentom `MessageProvider`:

```
    <Route path={mentorMenuTabs.messagesRoute}>
         <MessageProvider>
           <MessageWall />
         </MessageProvider>
       </Route>
```

Ispis 18 Dodavanje pristup kontekstu

Prilikom korištenja `MessageWall` komponente i bilo koje komponente unutar `MessageWall` komponente omogućen je pristup prethodno navedenom kontekstu. Kako bi pristupili informacijama o stanju unutar konteksta upotrebljava se hook `useContext` na sljedeći način:

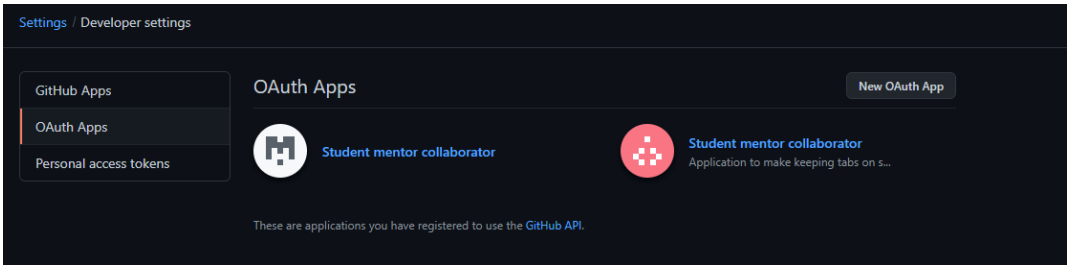
```
const {
   state: { isEndOfMessages, isMessageLoading, messages, studentFilterId },
   setPage,
   handleStudentFilterId,
 } = useContext(MessageContext);
```

Ispis 19 Pristup vrijednostima konteksta

Gdje je omogućen pristup nizu `messages`, svojstvima konteksta i definiranim funkcijama vezanih za kontekst.

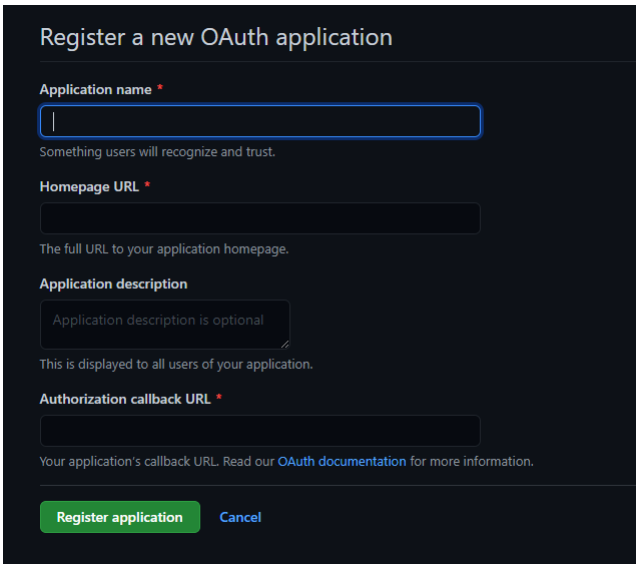
3.4.5 Github integracija

U svrhu organizacije svih akcija u interakciji s Github API-em stvoren je servis Github. Unutar servisa Github nalaze se metode za autorizaciju, dohvaćanje repozitorija, dohvaćanje JWT tokena i kreaciju web hook-a. U svrhu lakše integracije s API-em korporacija Github je izradila biblioteku `Octokit`, u kojoj su sadržane metode koje enkapsuliraju pozive na API Github. Kako bi se izradila integracija s uslugom Github nužno je pročitati dokumentaciju [8]. Prvi korak u interakciji s uslugom je autorizacija korisnika, da bi se uspješno mogli autorizirati korisnici potrebno je putem sučelja Github-a registrirati aplikaciju. Korištenjem web preglednika pristupa se sučelju Github-a te unutar navigacije se pronalazi opcija postavke za razvoj (*engl. developer settings*), prikazano na `Slika 3`.



Slika 3 Github sučelje

Gdje je ponuđena opcija registriranja nove aplikacije u sustav Github u svrhu OAuth autorizacije. Klikom na dugme “New OAuth App” otvara se forma s `Slika 4` koju je nužno ispuniti.



Slika 4 OAuth forma

Jedan od kriterija integriranja s uslugom Github je javno dostupni projekt na mreži. U procesu stvaranja aplikacije korištena je usluga ngrok, koja omogućuje pristupanje lokalnom poslužitelju putem javne mreže. Nužno se registrirati putem stranice ngrok i skinuti ngrok aplikaciju. Nakon što se ngrok aplikacija pokrene odabere se port na kojem se izvršava poslužitelj, zatim ngrok odgovara nasumično generiranom kodomenom domene ngrok. Nakon što možemo pristupiti aplikaciji putem mreže možemo nastaviti s registracijom u sustav Github. Jedno od obaveznih polja je akcija za autorizaciju koja se koristi u OAuth procesu. U svrhu ispunjavanja ovog obaveznog polja stvorena je akcija `AuthorizeGithub`.

```
[AllowAnonymous]
[HttpGet(nameof(AuthorizeGithub))]
public ActionResult AuthorizeGithub(string code, string state)
{
    return Redirect($"/home/profile/{code}");
}
```

Ispis 20 Metoda za autoriziranje Github profila

Gledajući da sustav Student Mentor već ima izvedbu korisničkih računa odlučeno je asocirati postojeće račune s Github računima. Iz toga razloga akcija za autorizaciju samo preusmjerava autoriziranog korisnika na njegov profil. Unutar komponente `ProfileScreen` definiran je parametar rute (*engl. route parameter*) koji se učitaju iz rute.

```
const { githubToken } = useParams();
```

Ispis 21 Učitavanje koda iz parametra putem hooka useParams

Korištenjem hooka `useEffect` nuspojava promjene `githubTokena` u ruti je poziv na poslužitelja za izmjenu vrijednosti `GithubAccessKey` u korisničkoj tablici.

```
useEffect(() => {
    if (githubToken) {
      axios
        .patch("/api/Student/PatchGithubAccessKey", { githubToken })
        .then(() => {
          history.push("/home/profile");
        });
    } else {
      axios
        .get("/api/Account")
        .then((response) => {
          setProfile({ ...response.data });
        })
        .finally(() => {
          setIsLoading(false);
        });
    }
  }, [history, githubToken]);
```

Ispis 22 Hook za postavljanje pristupnog ključa

Nakon uspješnog spremanja novog pristupnog ključa i ponovnog učitavanja profila biti će prikazani korisnikovi javni i privatni repozitoriji. Ukoliko je autorizirani korisnik uloge student učitava se komponenta `StudentGithubSection` koja putem akcije `GetAllRepositories` dohvaća repozitorije. Da bi to bilo moguće kao drugi korak

```
[HttpGet(nameof(GetAllRepositories))]
public async Task<ActionResult> GetAllRepositories()
{
    var response = await _githubService.GetAvailableRepositories();
    if (response.IsError)
        return BadRequest(response.Message);

    var repositoryId = await _studentRepository.GetRepositoryId();

    return Ok(new
    {
        repositoryId = repositoryId,
        repositories = response.Data
    });
}
```

Ispis 23 metoda za dohvaćanje svih repozitorija

potrebno je dohvatiti JWT token za pristup Github sustavu, kako već postoji pristupni ključ nije potrebno unijeti podatke za autorizaciju već se on koristi. Akcija `GetAllRepositories` koristi metodu `GetAvailableRepositories` iz Github servisa i dodatno vraća trenutačno odabrani studentov repozitorij.

```
public async Task<ResponseResult<IReadOnlyCollection<Repository>>>
GetAvailableRepositories()
{
    var client = await ClientWithCredentials();
    try
    {
        var repositories = await client.Repository.GetAllForCurrent(new
RepositoryRequest
        { Affiliation = RepositoryAffiliation.Owner });
        return new
ResponseResult<IReadOnlyCollection<Repository>>(repositories);
    }
    catch (Exception e)
    {
        return
ResponseResult<IReadOnlyCollection<Repository>>.Error(e.StackTrace);
    }
}
```

Ispis 24 Korištenje OctoKit paketa za upit na vidljive repozitorije

Metoda `GetAvailableRepositories` uz pomoć biblioteke `OctoKit` šalje se zahtjev HTTP prema Github sustavu za sve repozitorije gdje je korisnik vlasnik repozitorija. Privatna metoda `ClientWithCridentials` je odgovorna da klijent za pristup sustavu Github posjeduje valjani token JWT.

```
private async Task<GitHubClient> ClientWithCredentials()
{
    var response = await _studentRepository.GetGitHubAccessToken();
    if (response.Data == null)
        throw new UnauthorizedAccessException();

    return new GitHubClient(new
ProductHeaderValue(_githubConfiguration.ClientAppName), new
InMemoryCredentialStore(new Credentials(response.Data)));
}
```

Ispis 25 Metoda za inicijalizaciju objekta s skrivenim ključem

U slučaju da korisnik koji zahtjeva repozitorije ne posjeduje JWT ključ dolazi do iznimke u programu za zabranu pristupa. Ukoliko korisnik posjeduje pristupni ključ izrađuje se klijent za pristup uz pomoć konfigurirane aplikacije i ključa JWT. Unutar akcije za dodavanje pristupnog ključa šalje se još jedan zahtjev za dohvaćanje JWT tokena. Uz pomoć metode `GetBearerToken` koristi se metoda `CreateAccessToken` iz biblioteke `OctoKit` koja za argument prima id klijenta i tajni ključ, dobiven prilikom registracije putem Github sučelja i pristupni ključ kojeg je korisnik upravo dodao putem prije navedene akcije `PatchGithubAccessKey`.

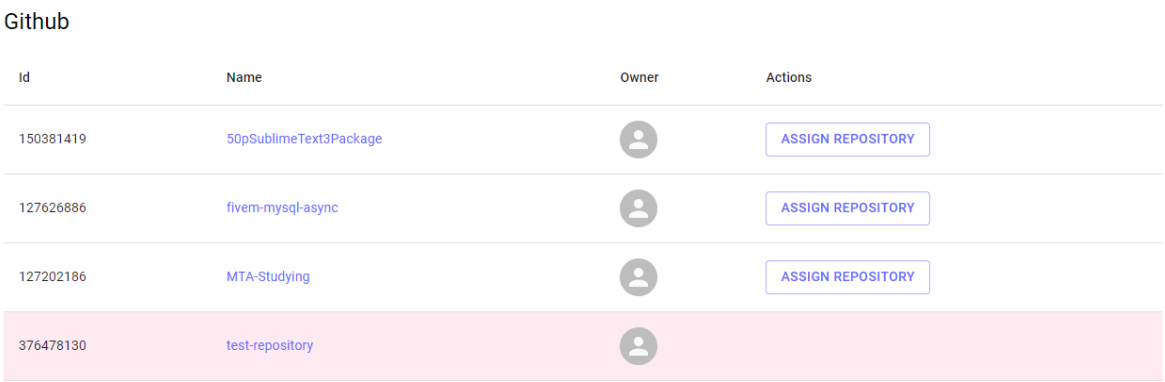
```
private GitHubClient Client => new GitHubClient(new
ProductHeaderValue(_githubConfiguration.ClientAppName));
public async Task<string> GetBearerToken(string oAuthToken = null)
{
    var oAuthTokenForRequest = "";
    if (oAuthToken == null)
    {
        var response = await _studentRepository.GetOAuthKey();
        oAuthTokenForRequest = response.Data ?? throw new
UnauthorizedAccessException();
    }

    if (oAuthToken != null)
    {
        oAuthTokenForRequest = oAuthToken;
    }

    var accessTokenResponse = await Client.Oauth.CreateAccessToken(new
OauthTokenRequest(_githubConfiguration.ClientId,
        _githubConfiguration.ClientSecret, oAuthTokenForRequest));
    return accessTokenResponse.AccessToken;
}
```

Ispis 26 Metoda za dohvaćanje ključa JWT

Nakon liste repozitorija korisniku se omogućuje biranje repozitorija prikazano na `Slika 5`, za stvaranja web hook-a. Klikom na akciju “Assign Repository” šalje se zahtjev HTTP na poslužitelja.



Slika 5 Lista repozitorija

Akcija `PatchRepositoryId` odgovorna je za stvaranje web hook-a i spremanje novo odabranog identifikatora repozitorija u korisničku tablicu.

```
[Authorize(Policy = Policies.Student)]
[HttpPatch(nameof(PatchRepositoryId))]
public async Task<ActionResult> PatchRepositoryId(RepositoryModel model)
{
    var repositoryHook = await
_githubService.CreateWebhookForRepositoryId(model.RepositoryId);
    if (repositoryHook.IsError)
        return BadRequest(repositoryHook.Data);

    var response = await
_studentRepository.PatchRepositoryId(model.RepositoryId);

    if (response.IsError)
        return BadRequest(response.Message);

    return Ok();
}
```

Ispis 27 Metoda za postavljanje odabranog repozitorija

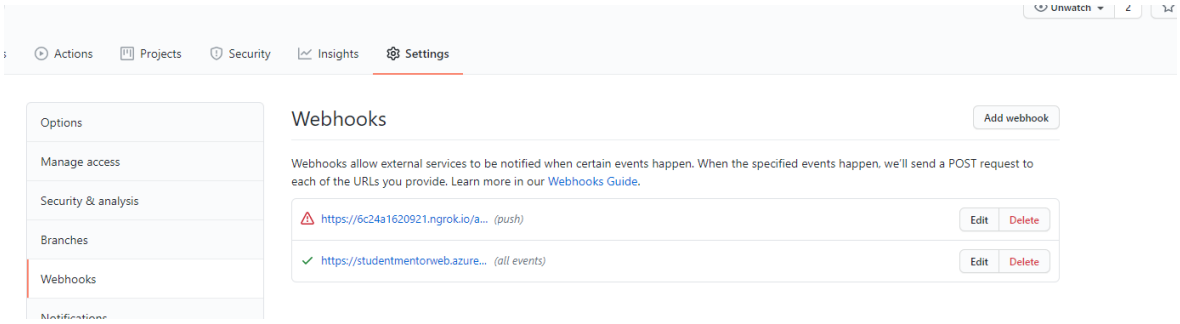
`CreateWebHook` metoda šalje zahtjev HTTP Post, gdje se definira akcija kada će Github sustav slati HTTP zahtjeve na akciju sustava Student Mentor. Kako je nužno samo pratiti izmjene nad repozitorijem odabran je event `Push` kao jedini potrebni zahtjev. Kako bi sustav Github znao točnu rutu gdje se šalje zahtjev koristi se konfiguracija u svrhu dohvaćanja odgovarajuće rute.

```
private IReadOnlyDictionary<string, string> GetDefaultConfiguration(string
method)
{
    return new Dictionary<string, string>
    {
        { "url", $"{_hookConfiguration.ExposedUrl}/api/Hook/{method}"},
        { "content-type", "application/json"}
    };
}
public async Task<ResponseResult<RepositoryHook>>
CreateWebhookForRepositoryId(long repositoryId)
{
    var client = await ClientWithCredentials();
    try
    {
        var defaultConfiguration = GetDefaultConfiguration("Push");
        var allRepositoryHooks = await
client.Repository.Hooks.GetAll(repositoryId);
        var hookForRepository =
            allRepositoryHooks.FirstOrDefault(hfr => hfr.Url ==
defaultConfiguration.GetValueOrDefault("url"));
        if (hookForRepository != null)
            return new ResponseResult<RepositoryHook>(hookForRepository);
        var response = await client.Repository.Hooks.Create(repositoryId, new
NewRepositoryHook("web", defaultConfiguration)
        {
            Events = new List<string>
            {
                "push"
            }
        });

        return new ResponseResult<RepositoryHook>(response);
    }
    catch (Exception e)
    {
        return ResponseResult<RepositoryHook>.Error(e.StackTrace);
    }
}
```

Ispis 28 Metoda za kreaciju webhooka

`ExposedUrl` sadrži vrijednost trenutačne rute poslužitelja, dok vrijednost "url" unutar definiranog rječnika (*engl. dictionary*) sadrži rutu akcije na koju se primaju HTTP zahtjevi. Prilikom pregleda odabranog repozitorija može se vidjeti lista svih stvorenih web hookova na `Slika 6`.



Slika 6 Webhooks

Metoda Push u klasi `HookController.cs` je odgovorna za spremanje i slanje poruka korisnicima koji trenutačno koriste sustav Student Mentor. Slanje poruka se izvršava po principu pojašnjenom u potpoglavlju SignalR i razmjena poruka.

```
[HttpPost(nameof(Push))]
public async Task<ActionResult> Push(PushModel model)
{
    var response = await _githubRepository.SaveGithubPushEvent(model);
    if (response.IsError)
        return NoContent();
    var messageResponse = await
_messageRepository.SendGithubMessage(response.Data);

    if (messageResponse.IsError)
        return NoContent();

    var message = messageResponse.Data;
    await _hubContext.Clients.Users(message.UserTo.Id.ToString(),
message.UserFrom.Id.ToString()).SendAsync(MessagesHub.SendMessageMethod,
messageResponse);

    return Ok();
}
```

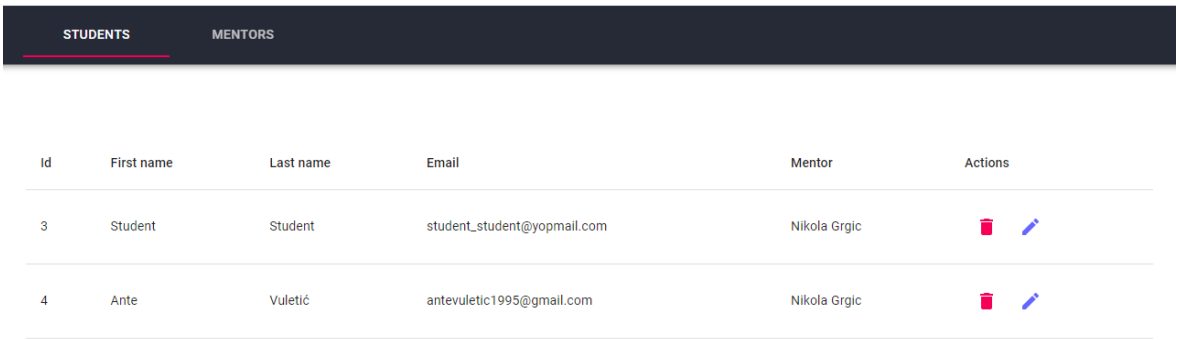
Ispis 29 Spremanje događaja push i slanje objekta SignalR klijentima

4 POSLOVNA LOGIKA

U ovom poglavlju biti će pojašnjene sve funkcionalnosti aplikacije iz perspektive aktera sustava.

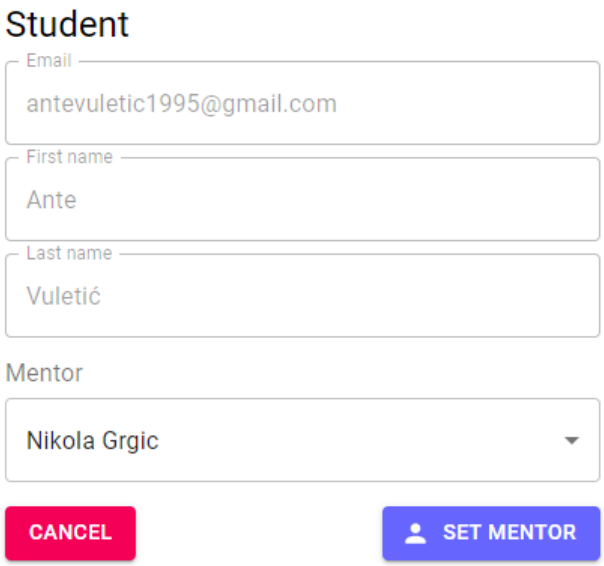
4.1 Upravljanje studentima

Akter sustava je administrator koji ima pregled svih studenata unutar sustava u obliku tablice prikazanoj na `Slika 7`.



Slika 7 Pregled liste studenata

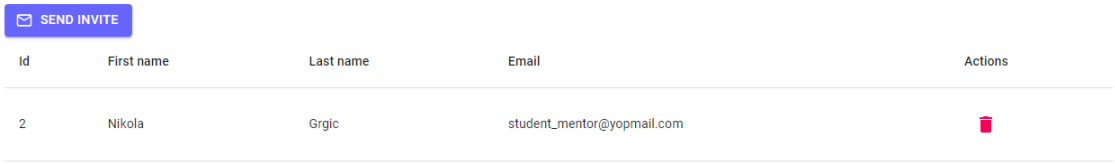
Administrator ima mogućnost brisanja i izmjene podataka vezanih za studente. Prilikom izmjene podataka dozvoljeno je postavljanje mentora studenta, dok se druga svojstva ne mogu mijenjati prikazana na `Slika 8`.



Slika 8 Izmjena mentora

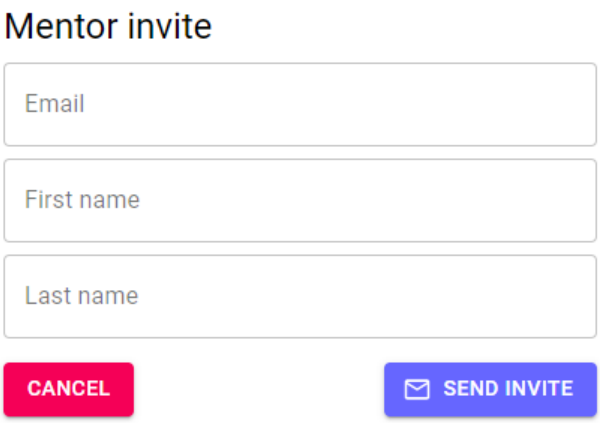
4.2 Upravljanje mentorima

Akter sustava je administrator koji ima pregled svih mentora unutar sustava u obliku tablice prikazanoj na `Slika 9`.



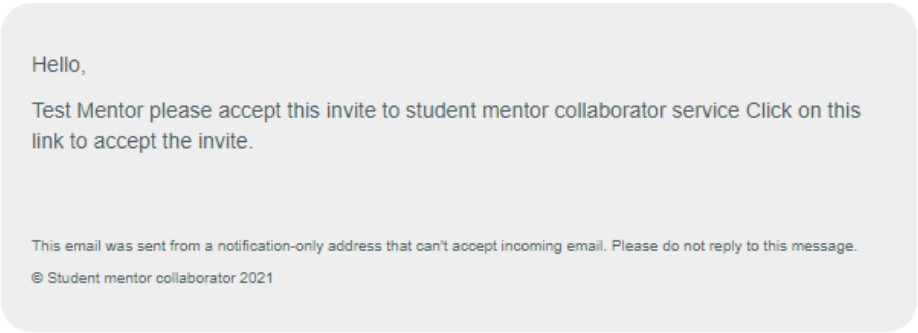
Slika 9 Lista mentora

Podaci vezani za mentora se ne mogu mijenjati, ali se mogu brisati. Kako bi se dodao novi mentor u sustav koristi se funkcionalnost pozivanja mentora u sustav. Pritiskom dugme Send Invite otvara se forma za unos osnovnih informacija o mentoru prikazana na `Slika 10`.



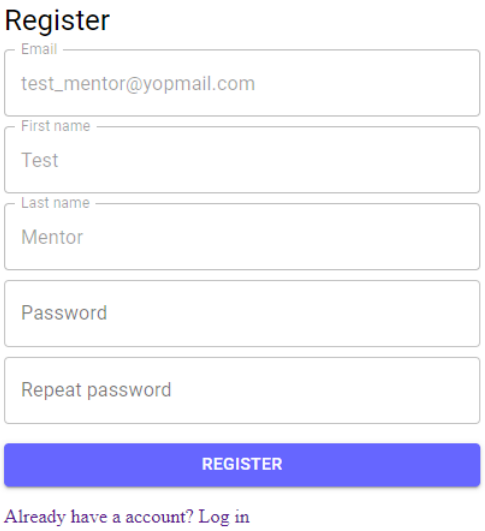
Slika 10 Unos novog mentora

Nakon što se ispune podaci vezani za mentora i potvrdi unos putem dugme Send Invite, na navedenu elektroničnu adresu se šalje elektronička pošta prikazana na `Slika 11`.



Slika 11 Primjer elektroničke pošte

Pritiskom na poveznicu “link” otvara se forma za registraciju gdje se od mentora zahtjeva unos lozinke prikazan na Slika 12.

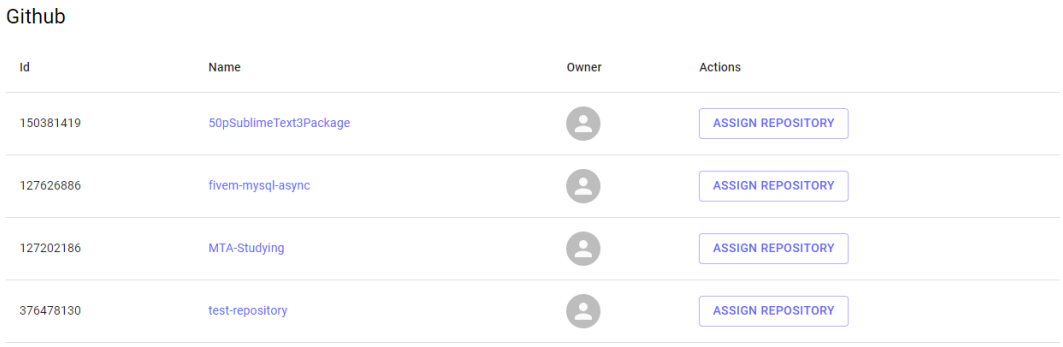


Slika 12 Forma za registraciju

Nakon unosa lozinke mentoru je omogućen pristup aplikaciji.

4.3 Profil

Akter sustava su svi korisnici aplikacije, pritiskom na dugme “Profil” otvara se korisnički profil koji sadrži osnovne informacije o trenutačnom korisniku. Osnovne informacije su ime korisnika, prezime korisnika i elektronička adresa. Ukoliko je akter sustava student na svom korisničkom profilu ima izbor spajanja računa s Github računom. Pritiskom na dugme “Authorize Github”, korisnik se preusmjerava na formu za prijavu u svoj Github korisnički račun ukoliko već postoji aktivna sesija s Github računom proces je automatiziran. Nakon spajanja korisničkih računa prikazuju se svi korisnikovi privatni i javni repozitoriji filtrirani po vlasništvu. U tabličnom obliku, prikazanom na Slika 13, korisnik odabire repozitorij na kojem se nalazi njegov završni rad.

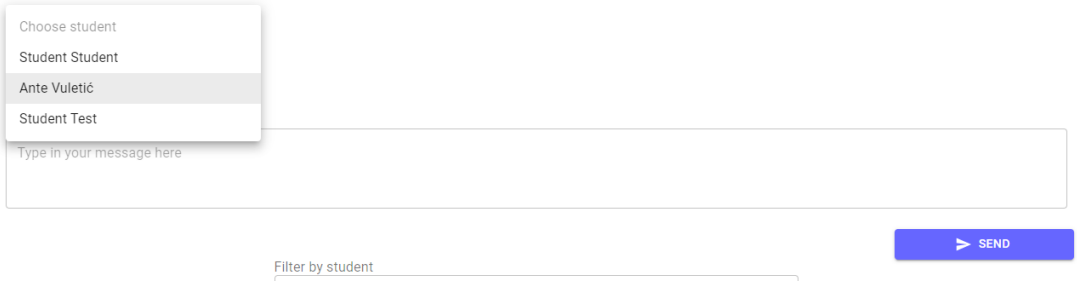


Slika 13 Lista repozitorija za odabir

U svakom trenutku, korisnik ima mogućnost izmjene repozitorija, odnosno može odabrati drugi repozitorij iz liste. Osim pregleda repozitorija studentu se omogućuje pregled dodijeljenog mentora.

4.4 Poruke

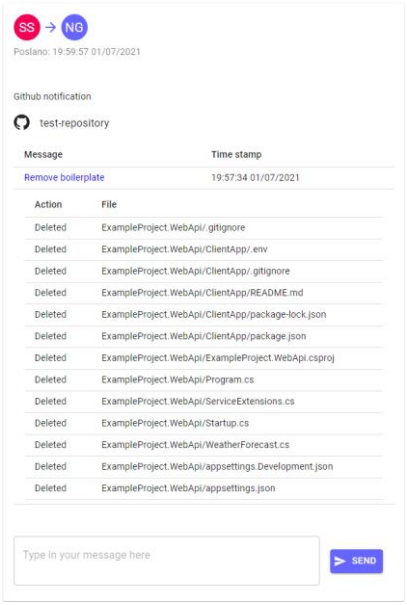
Akteri sustava su mentor i student, kako bi se omogućila lakša komunikacija između mentora i studenta implementirana je razmjena poruka u stvarnom vremenu. U formi za slanje poruke mentor odabire studenta kojem želi poslati poruku kao što je prikazano na `Slika 14.`



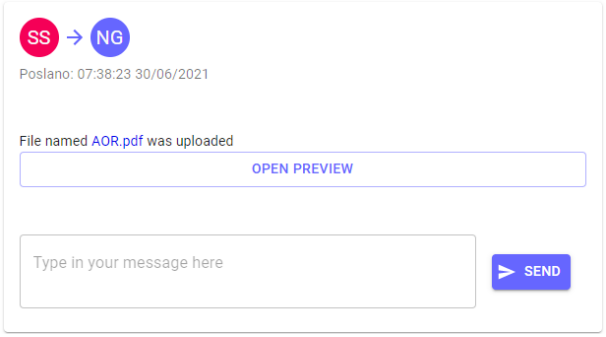
Slika 14 Izmjena poruka

Nakon odabira studenta i napisane željene poruke, pritiskom na dugme “Send” studentu se šalje poruka. Kako bi se olakšao pregled poruka, mentor na izbor ima dvije različite vrste pregleda. Omogućen je pregled svih primljenih poruka ili pregled filtriran po korisniku, odnosno studentu. Neovisno o vrsti pregleda, poruke pristižu u stvarnom vremenu. Za razliku od mentora, student može poruku samo slati odabranom mentoru te mu zbog toga nije nužan pregled uz pomoć filtriranja. Za pregled starih poruka implementiran je pregled uz pomoć paginacije i beskonačnog pomicanja (*engl. infinite scroll*). U svrhu lakšeg praćenja zbivanja, to jest napretka studenta implementirane su automatizirane poruke poput poruke

aktivnosti nad repozitorijem i poruke o pohranjivanju nove verzije završnog rada. Primjeri automatiziranih poruka vidljivi su na Slika 15 i Slika 16.

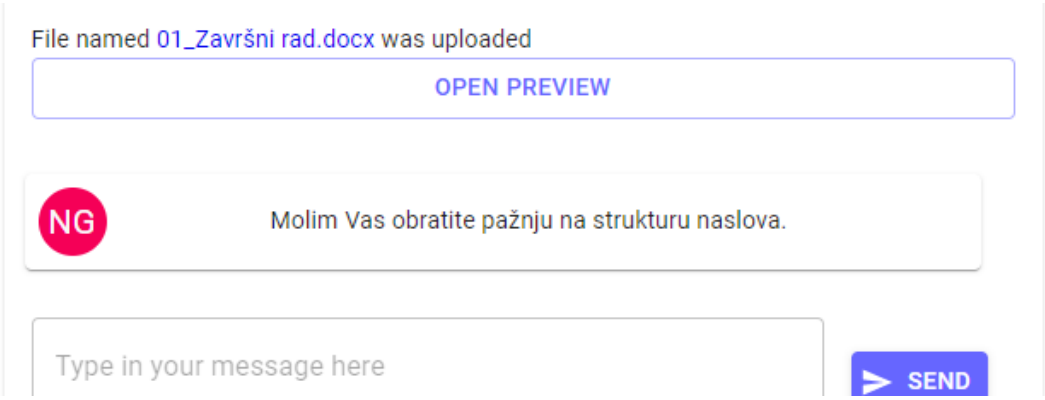


Slika 15 Izmjene nad repozitorijem



Slika 16 Predaja dokumenta završnog rada

Poruka za izmjenu na repozitoriju, prikazana na slici, omogućen je pregled svih promjena nastalih tom akcijom kao i vezu na promjenu unutar sustava Github. Za poruku o pohrani verzije završnog rada omogućen je pregled dokumenta unutar sustava, podržane ekstenzije su pdf, jpeg i docx. U svrhu lakše komunikacije omogućeno je ostavljanje komentara na poruke kako bi se mogla pružiti povratna informacija studentu i obratno. Za komentiranje poruke koristi se forma koja se nalazi na dnu poruke, primjer Slika 17.

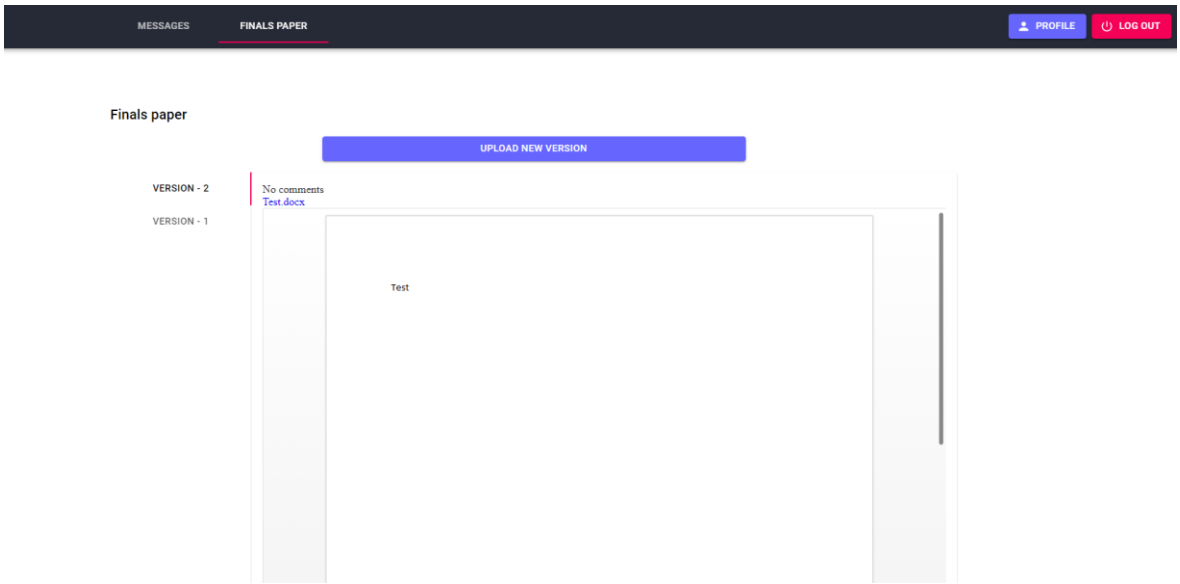


Slika 17 Primjer komentara na dokument završnog rada

Poslani komentari se prikazuju grupirani uz poruku kao što je prikazano na prethodnoj slici. Ukoliko je komentirana poruka koja sadrži datoteku, isti komentari se prikazuju zajedno s pregledom verzija dokumenata, više o tome u potpoglavlju Verziranje završnog rada.

4.5 Verziranje završnog rada

Akteri su mentor i student uloge. Uloga ovog dijela aplikacije je lako prenošenje i dokumentiranje različitih verzija završnog rada. Student ima pristup pohranjivanju verzija završnog rada, kao što je prikazano na `Slika 18`.



Slika 18 Pohrana verzije dokumenta završnog rada

Osim pregleda različitih verzija, omogućen je pregled dokumenta i ostavljenih komentara na poruci.

4.6 Github integracija

Akteri su studenti i mentori, u svrhu lakšeg praćenja napretka razvoja praktičnog dijela aplikacije omogućeno je praćenje promjena nad repozitorijem uz pomoć Github aplikativnog programskog sučelja (*engl.API, application programming interface*). Ukoliko student želi registrirati repozitorij za praćenje prvo treba omogućiti pristup svom Github računu, što se postiže uz pomoć protokola OAuth2. Nakon dopuštenih prava pristupa nužna za uspješnu interakciju s sustavom Student Mentor, korisnik odabire repozitorij za praćenje putem svog profila, kao što je prije navedeno u potpoglavlju Profil. Sustav u tom trenutku registrira web kuku putem Github API-a koja prilikom bilo kakve promjene nad repozitorijem šalje zahtjev putem protokola HTTP. Navedene promjene se prikazuju mentoru i studentu u obliku poruka, kao što je spomenuto u potpoglavlju Poruke.

5 ZAKLJUČAK

Prilikom upotrebe tehnologija najveći izazov je predstavljala izmjena poruka u trenutnom vremenu. Kako biblioteka SignalR sadrži pregršt funkcionalnosti nije bilo najlakše izabrati koncept organizacije, svrhu pojedinih dijelova korištenja poput čvorova i interakcija paketa s bibliotekom React. Prilikom upotrebe razvojnog okruženja .Net Core teško je donijeti odluku na koji način napraviti neki zahtjev, zbog velikog izbora različitih rješenja koje imaju jednaki cilj i minorne razlike u izvedbi. Zbog velikog izbora različitih rješenja trebalo je započeti izradu dijelova aplikacije iz početka u svrhu lakšeg održavanja i proširivosti. Da bi aplikacija bila lijepo dizajnirana i ugodna oku odabran je paket MaterialUI, uz pomoć kojeg je omogućena izrada sučelja s minimalnim pisanjem kaskadnih stilova.

Nakon izvedbe planiranog dijela aplikacije uočena je mogućnost proširenja funkcionalnosti. Uvođenje mogućnosti postavljanja datuma početka i kraja akademske godine. Ova značajka bi olakšala pretraživanje studenata i razmijenjenih poruka. Također bi omogućilo arhiviranje podataka iz godine u godinu na smišljen način što bi poboljšalo performanse sustava. U svrhu poboljšanja transparentnosti diplomiranih studenata, arhivirani podaci bi bili javno dostupni u sklopu pregleda završenih radova. S time bi se stvarala arhiva završnih radova koja bi poboljšala pristup obrađenim temama nadolazećim generacijama i široj zajednici.

Literatura

[1] Wikipedia, „Relational database“, https://en.wikipedia.org/wiki/Relational_database (posjećeno 3.9.2021.).

[2] TechEmpower, „Framework benchmarks“, https://www.techempower.com/benchmarks/ (posjećeno 3.9.2021.)

[3] IETF, „The WebSocket Protocol“ , https://datatracker.ietf.org/doc/html/rfc6455 (posjećeno 3.9. 2021.)

[4] Microsoft Documentation, „Inheritance – Table-per hierarchy and discriminator configuration“, https://docs.microsoft.com/en-us/ef/core/modeling/inheritance (posjećeno 3.9.2021.)

[5] Microsoft, „Common web application architectures – Traditional „N-Layer“ architecture application“, https://docs.microsoft.com/en-us/dotnet/architecture/modern-web-apps-azure/common-web-application-architectures (posjećeno 3.9.2021.)

[6] IETF, „JSON Web Token (JWT)“, https://datatracker.ietf.org/doc/html/rfc7519 (posjećeno 3.9.2021.)

[7] Microsoft, „Introduction to SignalR“, https://docs.microsoft.com/en-us/aspnet/signalr/overview/getting-started/introduction-to-signalr (posjećeno 3.9.2021.)

[8] Github, „Get Started – REST API“, https://docs.github.com/en/rest/guides/getting-started-with-the-rest-api (posjećeno 3.9.2021.)