Izrada web aplikacije za vođenje evidencije stručne prakse na fakultetima

Matijašić, Teo

Sveučilište u Rijeci – Fakultet informatike i digitalnih tehnologija

Sveučilišni prijediplomski studij Informatika

Teo Matijašić

Izrada web aplikacije za vođenje evidencije stručne prakse na fakultetima

Završni rad

Mentor: Doc. dr. sc. Lucia Načinović Prskalo

Rijeka, 17. 7. 2023.

www.inf.uniri.hr

FIDIT
Sveučilište u Rijeci
Fakultet informatike
i digitalnih tehnologija

Rijeka, 6.9.2023.

Zadatak za završni rad

Pristupnik: Teo Matijašić

Naziv završnog rada: Izrada web aplikacije za vođenje evidencije stručne prakse na fakultetima

Naziv završnog rada na engleskom jeziku: Creation of a web application to record professional practice at faculties

Sadržaj zadatka: Zadatak završnog rada je izraditi web aplikaciju za vođenje evidencije stručne prakse. Pritom će se koristiti odabrani alati i tehnologije za razvoj frontend i backend dijela aplikacije koji će se u radu i detaljno opisati. Također će se opisati i demonstrirati svi važni elementi i funkcionalnosti izrađene web aplikacije.

Mentor

Doc. dr. sc. Lucia Načinović Prskalo

Voditelj za završne radove

Doc. dr. sc. Miran Pobar

Zadatak preuzet: 12.6.2023.

(potpis pristupnika)

Adresa: Radmile Matejčić 2 Tel: +385(0)51 584 700
51000 Rijeka, Hrvatska E-mail: ured@inf.uniri.hr
OIB: 64218323816
IBAN: HR1524020061400006966

Sadržaj

Izrada web aplikacije za vođenje evidencije stručne prakse na fakultetima

Sažetak

Ovaj rad se bavi i opisuje postupak izrade web aplikacije za vođenje evidencije stručne prakse na fakultetima. U izradu navedene web aplikacije krenulo se je zbog današnje velike popularnosti korištenja web aplikacija, ali i zbog uočenoga nedostatka određenoga sustava ili aplikacije kojom bi fakulteti i studenti mogli lakše voditi evidenciju stručne prakse. Za izradu web aplikacije, korištene su vrlo aktualne tehnologije poput PostgreSQL-a, ASP.NET Core-a te .NET Core-a i React-a, a za pisanje koda korišten je Visual Studio 2022.

Web aplikacije se sastoje od nekoliko ključnih elemenata, svaki sa svojom ulogom i bitnim udjelom u njenom funkcioniranju i radu. Tako je i u ovome radu opisana izrada stražnjega dijela („backend“) web aplikacije, odnosno baze podataka, Web API-a te središnjega sloja i sloja pristupa podacima, ali i njenoga prednjega dijela („frontend“) koji služi za prikazivanje podataka i rezultata krajnjim korisnicima.

Kao jedan od rezultata završnog rada, kreirana je web aplikacija koja je namijenjena fakultetima kako bi im olakšala vođenje evidencije stručne prakse svojih studenata te su implementirani i opisani svi njeni ključni dijelovi i elementi.

Ključne riječi: web, aplikacija, backend, frontend, .NET, React, PostgreSQL

Creation of web application to record professional practice at faculties

Summary

This thesis deals with and describes the process of creating a web application for keeping records of professional practice at faculties. The creation of the aforementioned web application was undertaken due to the current popularity of web applications, but also due to the identified lack of a specific system or application that would allow faculties and students to more easily keep records of professional practice. The web application was built using very current technologies such as PostgreSQL, ASP.NET Core, .NET Core, and React, and Visual Studio 2022 was used to write the code.

Web applications consist of several key elements, each of which has its own role and essential part in the functioning and operation of the application. Therefore, the thesis also describes the creation of the backend of the web application, i.e., the database, the web API, the middle layer, and the data access layer, as well as the frontend, which is used to display data and results to end users.

As a result, a web application was created to facilitate faculties to keep records of their students' professional practice, and all important parts and elements were implemented and described.

Keywords: web, application, backend, frontend, .NET, React, PostgreSQL

1. Uvod

Ovim radom opisan je postupak i faze izrade web aplikacije za vođenje evidencije stručne prakse na fakultetima. Kako u današnjemu društvu web aplikacije postaju sve popularnije zbog svoje fleksibilnosti i mogućnosti korištenja bez instalacije, ali i zbog uočenoga nedostatka određenoga programa, sustava ili aplikacije za praćenje i vođenje evidencije stručne prakse studenata na fakultetima, krenulo se je u izradu i implementaciju web aplikacije koja bi fakultetima i studentima služila kao organizirano i intuitivno mjesto na kojemu mogu efikasnije i lakše pratiti i voditi evidenciju stručne prakse. Današnje, sve popularnije web aplikacije, sastoje se od stražnjega dijela kojega najčešće čine server koji prima zahtjeve, baza podataka gdje se pohranjuju podaci te sama aplikacija ili Web API[1] koji obrađuju zahtjeve korisnika prema logici definiranoj u središnjem sloju, pristupaju bazi podataka pomoću sloja pristupa podacima i vraćaju odgovore na primljene zahtjeve. Drugi najvažniji dio svake web aplikacije je prednji dio, odnosno korisničko sučelje preko kojega korisnik izvodi određene akcije i pregledava njihove rezultate.

Time bi cilj ovoga rada bio opisati, ali i implementirati web aplikaciju koja bi fakultetima omogućila lakše i preglednije vođenje evidencije stručnih praksi svojih studenata. Također, ovim radom se nastoje opisati i izraditi svi najvažniji dijelovi web aplikacije, uključujući bazu podataka, Web API, središnji sloj za poslovnu logiku, sloj pristupa podacima te samo korisničko sučelje.

Za izradu same web aplikacije nastojalo se koristiti vrlo aktualne i moderne tehnologije. Tako je za izradu baze podataka korišten PostgreSQL koji predstavlja vrlo popularan i moćan sustav za upravljanje relacijskim bazama podataka s brojnim modernim značajkama. S druge strane za izradu Web API-a je korišten ASP.NET Core, razvojni okvir otvorenoga koda koji je orijentiran na razvoj modernih i mrežnih aplikacija za različite platforme. Za implementaciju središnjega dijela i sloja pristupa podacima korišten je .NET Core, nešto općenitija platforma i razvojni okvir za razvoj različitih vrsta aplikacija za različite platforme, a za izradu prednjeg dijela je korišten, danas vrlo popularan React koji predstavlja JavaScript biblioteku za razvoj i implementaciju korisničkih sučelja. Za pisanje samoga koda, korišten je Visual Studio 2022 koji uvelike olakšava izradu web aplikacija te rad s navedenim tehnologijama. Prilikom pisanja tekstualnoga dijela ovoga rada, većinom su korištene internetske dokumentacije navedenih tehnologija, ali i poneki drugi internetski izvori.

Sam rad je podijeljen na pet glavnih poglavlja te tako nakon uvoda slijedi kratki, općeniti opis web aplikacija i njihove upotrebe. Zatim slijedi opis kreirane web aplikacije za vođenje evidencije stručne prakse na fakultetima. U idućem poglavlju su detaljnije opisane tehnologije koje su korištene za izradu navedene aplikacije, a zatim slijede prikazi i pripadajući opisi pojedinih glavnih dijelova koda web aplikacije. Na kraju je dan općeniti zaključak na izradu ovoga rada te su navedeni popis slika, popis korištene literature te prilozi.

[1] Web API (Application programming interface) – programsko sučelje web aplikacije za razmjenu resursa i interakciju različitih aplikacija pomoću HTTP metoda (Wikipedia, Web API, 2023.).

2. Što je web aplikacija?

U ovome poglavlju je dan općeniti opis i definicija web aplikacija, njihovih sastavnih dijelova i glavnih prednosti i nedostataka kako bi se prikazao temeljni uvod i kontekst u sam nastavak cijeloga rada. Također, prikazan je i glavni razlog kreiranja web aplikacije za vođenje evidencije stručne prakse studenata na fakultetima te je dan njen kratki opis i temeljne mogućnosti.

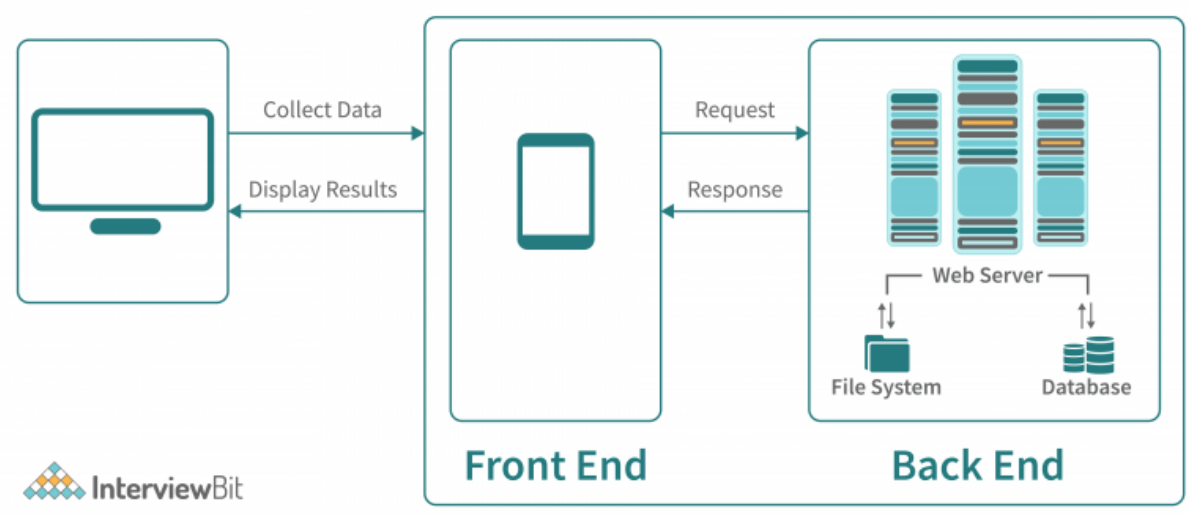
Pojavom Web-a došlo je do brojnih promjena u tehnologijama i samome razvoju aplikacija, a time čak i u ljudskim životima. Od svoga prvoga pojavljivanja, Web je utjecao na brojna područja ljudskoga djelovanja od kojih je jedno od njih razvoj i izrada brojnih poslovnih, komercijalnih te društvenih web aplikacija koje su postale široko dostupne i s vremenom sve više korištene. Kako je došlo do napretka tehnologija, ali i načina razvoja web aplikacija i samoga Web-a, web aplikacije su postajale sve korisnije, lakše za upotrebu, sigurnije, ali i pouzdanije te je to sve uzrokovalo njihovu veliku popularnost, raširenost i dostupnost. Svojim daljnjim napretkom i razvojem, web aplikacije uspjele su održati svojstvo jednostavnosti njihove izrade, no postajale su i složenije za održavanje zbog uključivanja pohrane podataka putem baze podataka, ali i obrade brojnih poslovnih podataka. Svime time, web aplikacije dosegle su razinu lokalnih i ostalih vrsta aplikacija te danas postaju sve popularnije, no sve te vrste aplikacija imaju vlastite prednosti i nedostatke koje treba uzeti u obzir prilikom početka njihovoga kreiranja (Murugesan, 2008.).

Prema (Volle, 2022.) „*web aplikacija je računalni program koji je pohranjen na udaljenom serveru i koji se pokreće od strane korisnika putem web preglednika*.“ Iz te definicije mogu se uočiti temeljni i glavni koncepti svake web aplikacije koji su potrebni za njeno djelovanje. Također, iz te definicije mogu se izvesti i određene najvažnije prednosti web aplikacija poput široke dostupnosti i fleksibilnosti jer nisu ovisne o određenom operacijskom sustavu, pregledniku niti uređaju na kojem se izvode. Kako se pokreće na serveru, web aplikaciju nije potrebno instalirati na računalo te time ne zauzima dodatnu memoriju. Razvojem brojnih tehnologija, njihova izrada je postala dosta jednostavna i troškovno te vremenski učinkovita što je uvelike utjecalo na njihovu popularnost. Iako web aplikacije imaju brojne prednosti, neki od njihovih glavnih nedostataka su moguća nešto sporija brzina rada zbog ovisnosti o pregledniku na kojemu se pokreće te prisutnost mrežne konekcije koja je potrebna za njihov ispravan rad.

Arhitektura web aplikacija se temelji na klijent-server konfiguraciji gdje klijent šalje HTTP zahtjeve serveru, a server zatim obrađuje pristigle zahtjeve i izvršava aplikaciju te vraća odgovore. Bitno je naglasiti da se na strani klijenta, web aplikacije koriste putem web preglednika koji prikazuje glavne stranice i glavno korisničko sučelje putem kojega se šalju određeni HTTP zahtjevi, prikazuju odgovori na njih te putem kojega korisnici izvršavaju temeljne akcije i različite obrade podataka. Te glave stranice korisničkoga sučelja web aplikacije se najčešće kreiraju putem jezika za označavanje poput HTML-a, ali i programskog jezika JavaScript. U početku samih web aplikacija te stranice su bile statičnoga karaktera te su prikazivale nepromjenjivi sadržaj, no njihovim razvojem te pojavom JavaScript programskog

jezika, omogućeno je prikazivanje dinamičkoga i prilagodljivoga sadržaja i dijelova web aplikacija. Svime time se klijentska strana web aplikacije još naziva i prednji dio ili „frontend[2]“

Na serverskoj strani web aplikacije, koja se još naziva stražnji dio ili „backend[3]“ najčešće se definira sva logika rada te funkcioniranja web aplikacije i njene obrade zahtjeva od klijenata. Tako se na stražnjem dijelu web aplikacija nalazi server ili poslužitelj koji je zadužen za primanje HTTP zahtjeva poslanih na određene rute, odnosno URL-ove[4] od strane klijenata. Server (poslužitelj), također često služi i kao domaćin za Web API te pritom prosljeđuje zaprimljene zahtjeve na odgovarajuće rute Web API-a gdje je definirana glavna logika usmjeravanja i vraćanja odgovora. Osim Web API-a, na stražnjem dijelu aplikacije često se nalazi i središnji sloj („middle layer“) u kojemu je definirana glavna poslovna logika obrade zaprimljenih zahtjeva, ali i sloj pristupa podacima („data access layer“) koji obavlja komunikaciju sa samom bazom podataka. Sa zaprimljenim podacima iz baze podataka, Web API generira HTTP odgovor koji se onda vraća klijentu te mu se rezultati prikazuju na korisničkom sučelju, odnosno prednjem dijelu web aplikacije. Opisana struktura je korištena i prilikom implementacije kreirane web aplikacije za vođenje evidencije stručne prakse na fakultetima. Vrlo jednostavna skica koja generalizirano prikazuje osnovnu opisanu strukturu web aplikacija prikazana je na slici 1.



Slika 1 - Prikaz arhitekture web aplikacije
Preuzeto iz: (InterviewBit, 2022.)

Iz svega navedenoga, mogu se prepoznati i određeni temeljni slojevi svake web aplikacije, a to su aplikacijski sloj koji predstavlja samo korisničko sučelje putem kojega korisnik stupa u interakciju s web aplikacijom, sloj poslovne logike koji definira glavnu

[2] „frontend“ – razvoj grafičkih korisničkih sučelja putem kojega korisnici pregledavaju i stupaju u interakciju s sadržajem (Wikipedia, Front-end web development, 2023.).

[3] „backend“ – razvoj koda s serverske strane koji definira ispravno funkcioniranje i svu logiku rada aplikacije koja nije vidljiva krajnjim korisnicima (Wikipedia, Frontend and backend, 2023.).

[4] URL (Uniform Resource Locator) – jedinstveni lokator sadržaja koji se koristi za identifikaciju i pronalaženje resursa (Wikipedia, URL, 2022.).

poslovnu logiku obrade podataka i korisničkih zahtjeva, sloj usluga podataka koji razdvaja poslovnu logiku od prezentacijskoga sloja te služi za manipulaciju i razmjenu podataka između ta dva sloja te sloj pristupa podacima koji služi za pristupanje samim podacima pohranjenim u bazama podataka (Altynpara & Bestaieva, 2023.).

Početni dio ovoga poglavlja poslužio je kao kratki, općeniti opis web aplikacija i njihovih glavnih značajki i načina funkcioniranja kako bi se dao općeniti kontekst i omogućilo lakše praćenje ostalih poglavlja u ovome radu.

2.1 Web aplikacija za vođenje evidencije stručne prakse

U ovome poglavlju dan je općeniti opis razloga kreiranja web aplikacije za evidenciju stručne prakse studenata na fakultetima te su opisane njene osnovne mogućnosti i funkcionalnosti, a njena implementacija je djelomično objašnjena u četvrtom poglavlju.

Web aplikacija koja je kreirana, služi fakultetima i njihovim studentima kao mjesto na kojemu mogu lakše i efikasnije voditi evidenciju stručne prakse. Naime, u izradu same web aplikacije krenulo se je zbog potrebe fakulteta, ali i uočenoga nedostatka određenoga sustava ili aplikacije koja bi omogućila fakultetima i studentima jednostavniju evidenciju stručne prakse. Kreirana web aplikacija, osim za praćenje stručnih praksi za potrebe fakulteta, također studentima služi kako bi oni sebi pratili i organizirali napredak kroz stručnu praksu na način da upisuju odrađeni broj sati ili recimo pišu kratki dnevnik prakse.

Sama web aplikacija se sastoji od četiri glavnih uloga od kojih svaka ima određene ovlasti i mogućnosti izvođenja pojedinih akcija, a to su administrator, student, profesor i asistent. Administrator ima sve ovlasti i mogućnosti u aplikaciji te je zamišljeno kako bi on prvo kreirao potrebne korisničke račune s inicijalnim korisničkim imenom i lozinkom koje bi onda podijelio korisnicima koji su i taj račun zatražili. Korisnici bi se tada prijavili u aplikaciju s dobivenim, inicijalnim korisničkim imenom i lozinkom koju su dobili od administratora te bi u odjelu popisa korisnika uredili svoje podatke tako da bi ažurirali korisničko ime i lozinku. Tada bi za ponovnu prijavu koristili upravo te novo ažurirane podatke te bi tako ostvarili pristup korištenju same web aplikacije. Prilikom prijave je moguće označiti opciju „Zapamti me“ kako bi korisnik ostao prijavljen i nakon ponovnoga pokretanja web aplikacije.

Nakon uspješne prijave i aktiviranja svoga korisničkoga računa, korisniku su omogućene tri glavne mogućnosti unutar aplikacije, a to su upravljanje podacima korisnika, unos i upravljanje podacima kompanija u kojima se obavlja stručna praksa te unos i upravljanje podacima stručnih praksi studenata. Također, korisnik neovisno o ulozi može pregledati vlastite podatke na zasebnoj kategoriji u navigaciji koja dinamički mijenja naziv ovisno o korisničkom imenu korisnika. Osim glavne navigacije koja se sastoji od naslova web aplikacije te kategorija koje se prikazuju jedino prijavljenim korisnicima i koje vode na podatke o prijavljenom korisniku, popise korisnika, popise kompanija i popise stručnih praksi,

kreirana je i sekundarna navigacija koja se nalazi na lijevoj strani te sadrži opcije za povratak na naslovnicu te, u obliku padajuće liste, opcije za odlazak na već navedene popise.

Svaka od četiri dostupnih uloga ima vlastita ograničenja i mogućnosti te se te uloge mogu hijerarhijski prikazati tako da se na vrhu nalazi administrator, ispod njega se nalaze profesor pa asistent i na kraju se nalazi sam student koji ima najmanje ovlasti. Tako administrator može obavljati sve aktivnosti unutar web aplikacije, odnosno može kreirati, ažurirati i brisati sve korisnike, može kreirati, brisati i ažurirati kompanije te može kreirati, brisati i ažurirati stručne prakse. On jedini može kreirati nove korisnike u web aplikaciji te jedini može mijenjati role, odnosno uloge određenog korisnika. Profesor ima nešto manje ovlasti od administratora pa tako on ne može kreirati nove korisnike, no može uređivati i brisati studente i asistente, a ne može uređivati i brisati administratore te druge profesore. Na taj način profesor ima određene ovlasti pa se za neke aktivnosti ne treba uvijek kontaktirati administratora. Što se tiče kompanija i praksi, profesor može kreirati, ažurirati i brisati sve njihove podatke. Nakon uloge profesora, slijedi asistent koji ima nešto manje ovlasti, ali opet dovoljne da napravi određene izmijene na zamolbu studenata bez potrebe za kontaktiranjem administratora ili profesora. Tako on također ne može kreirati niti brisati korisnike te može uređivati samo svoje podatke unutar web aplikacije. No za kompanije i prakse su mu omogućene sve ovlasti kreiranja, uređivanja i brisanja redaka. Na samome dnu hijerarhije, nalazi se i uloga studenta koja ima najmanje ovlasti te kojemu bi aplikacija služila za vođenje vlastitih podataka, ali i podataka o svojim stručnim praksama. Tako student također ne može kreirati ni brisati korisnike te može ažurirati samo svoje podatke. Kod popisa kompanija, student može samo kreirati i uređivati podatke pojedinih kompanija, a kod popisa praksi može kreirati, urediti i brisati samo svoje prakse. Na taj način je onemogućeno studentima da pregledavaju, uređuju i brišu podatke ostalih studenata.

Na vrhu svake stranice s popisima i tablicama, ovisno o dozvolama, prikazan je gumb za dodavanje novih unosa te se svima prikazuje gumb za osvježavanje podataka. Prilikom dodavanja novih unosa korisniku su, pored polja za unos, s crvenim zvjezdicama, naznačena obavezna polja gdje se mora unijeti određena vrijednost. Tako se kod dodavanja korisnika upisuju korisničko ime, e-mail, ime, prezime, JMBAG (ako postoji), lozinka, potvrda lozinke te dozvole. Kod dodavanja kompanija upisuju se naziv kompanije, adresa, grad i e-mail, a kod dodavanja praksi se odabiru dostupni studenti i kompanije te se upisuju akademska godina, studij, datum početka i završetka, pozicija, odrađeni sati, mentor, komentar mentora te opis poslova ili dnevnik prakse. Prilikom dodavanja novih unosa, korisnik u bilo kojem trenutku može odustati odabirom crvenoga gumba „Odustani“ na dnu forme ili može potvrditi svoj unos odabirom zelenoga gumba „Dodaj“ nakon kojega će se automatski dodati novi unos ili će se korisniku prikazati crvene napomene ispod određenih polja koja ne zadovoljavaju određene uvjete i ograničenja (razne validacije koje provjeravaju može li polje biti prazno, ima li unos dovoljan broj znakova, je li unesen ispravan e-mail, je li unesena lozinka prema traženim pravilima i sl.). Crvene napomene se također prikazuju prilikom unošenja vrijednosti u polje e-maila, potvrdne lozinke ili JMBAG-a kako bi uputile korisnika da unese ispravne vrijednosti. Odabirom gumba za dodavanje novih unosa, otvara se forma sa poljima za unos gdje su implementirane različite vrste polja poput najčešće korištenog polja za unos teksta,

polja za unos brojeva kod odrađenog broja sati pri kreiranju praksi, padajućih lista kod odabira rola, odnosno uloga korisnika koja je omogućena jedino administratoru te je ostalim ulogama dostupna samo za čitanje kako si korisnici ne bi mogli dodjeljivati veće dozvole. Također, padajuća lista je korištena i kod odabira dostupnih studenta i kompanija pri kreiranju praksi gdje je implementirano da se studentu ponude samo njegovi podaci (ime i prezime) kako ne bi mogao dodavati prakse za ostale studente dok se ostalim ulogama prikazuju samo svi dostupni studenti, a ne svi korisnici iz baze jer profesori i asistenti ne mogu ići na praksu. Također su korišteni i radio gumbi za odabir studija pošto se mogu odabrati samo dvije vrijednosti (Preddiplomski i Diplomski) te kalendari za odabir datuma početka i završetka prakse. Na stranicama glavnih kategorija (Korisnici, Kompanije, Prakse) se, osim gumba za dodavanje novih unosa i osvježavanje, nalaze i tablice s popisom svih dostupnih podataka i njihovih stupaca, a u zadnjem stupcu svake tablice se nalaze i gumbi za uređivanje i brisanje pojedinih redaka koji se prikazuju ovisno o dozvolama. Odabirom gumba za uređivanje se otvara ista forma kao i kod dodavanja novih redaka, samo su ovdje već forme ispunjene s dostupnim, postojećim podacima. Prilikom brisanja redaka, odabirom gumba „Izbriši“, prikazuje se lebdeći okvir iznad gumba s potvrdom brisanja.

Na stranicama s popisima, odnosno tablicama s podacima o korisnicima, kompanijama te praksi, omogućeno je pretraživanje podataka preko polja za pretragu koje se nalazi iznad tablica, a kod popisa praksi je dodatno omogućen i filter po akademskim godinama u obliku padajuće liste kako bi se korisnicima omogućio lakši i brži pronalazak traženih podataka.

Na temelju ovih opisa, korisnik, odnosno student, nakon prijave u aplikaciju može urediti vlastite podatke, može unijeti te uređivati podatke o kompaniji na kojoj je odrađivao prasku i na kraju može unijeti i voditi evidenciju svojih stručnih praksi. Na taj način, student može kontinuirano pratiti svoje napredovanje kroz stručnu praksu, a fakultet pri tome automatski dobiva intuitivno mjesto na kojemu mu se nalazi popis svih studenata, kompanija u kojima su oni odrađivali stručnu praksu i popis svih stručnih praksi određenih studenata te je time olakšan posao i studentima i fakultetu. Profesori i asistenti također mogu uređivati i brisati određene podatke kako bi i oni mogli brinuti o aktualnosti podataka ili uređivati i dodavati unose u slučaju da studenti imaju određenih poteškoća.

U ovom poglavlju je prikazan temeljni opis svih glavnih funkcionalnosti u kreiranoj web aplikaciji za evidenciju stručne prakse na fakultetima kako bi se prikazao dodatni kontekst o njenoj svrsi i mogućnostima te kako bi se definirao određeni uvod za sljedeća poglavlja o korištenim tehnologijama izrade te objašnjenjima pojedinih dijelova kodova.

3. Korištene tehnologije

U ovome poglavlju opisane su sve tehnologije koje su korištene prilikom izrade svih ključnih dijelova kreirane web aplikacije, uključujući tehnologije korištene za stražnji dio, odnosno bazu podataka, Web API, središnji sloj te sloj pristupa podacima ali i prednji dio, odnosno korisničko sučelje. Time je prvo prikazan kratki opis PostgreSQL baze podataka te su dani kratki opisi i prikazi same implementacije tablica, a zatim su dani opisi i glavne mogućnosti ASP.NET Core i .NET Core razvojnih okvira koji su korišten za izradu Web API-a, središnjega sloja i sloja pristupa podacima te React-a koji je korišten za izradu korisničkoga sučelja.

3.1 PostgreSQL (pgAdmin 4)

Za implementaciju i izradu baze podataka za kreiranu web aplikaciju je korišten PostgreSQL, jedan od najpopularnijih sustava za upravljanje relacijskim bazama podataka. Naime, PostgreSQL je vrlo moćan i pouzdan SUBP[5] otvorenoga koda koji pruža brojne mogućosti za sigurnu pohranu i upravljanje vrlo složenim podacima. Svoju popularnost i korištenost je postigao zbog više od 35 godina aktivnoga razvijanja i pružanja novih značajki te su neke od njegovih glavnih prednosti pouzdanost pohrane podataka, jednostavno očuvanje integriteta podataka i veliki raspon mogućnosti korištenja SQL[6] jezika i upravljanja podacima (PGDG, 2023.).

Izabran je za implementaciju baze podataka kreirane web aplikacije za evidenciju stručne prakse na fakultetima prvenstveno zbog toga što je otvorenoga koda te zbog podrške za različite platforme i operacijske sustave. Njegova jednostavnost korištenja i lakoća definiranja tablica, njihovih stupaca, ali i različitih, specifičnih ograničenja nad njima je također uvelike utjecalo na njegov odabir. Još jedna velika prednost PostgreSQL-a je i njegova prilagodljivost zbog koje se može koristiti za pohranu skupova podataka različitih veličina, ali i za integraciju s velikim brojem programskih jezika. Veliki broj različitih i već dostupnih tipova podataka, ograničenja za održavanje integriteta podataka, postavke za poboljšanje performansi, pouzdanost te sigurna pohrana podataka samo su neke od glavnih mogućnosti PostgreSQL-a te su poneka od njih korištena i pri samoj izradi kreirane web aplikacije (PGDG, 2023.).

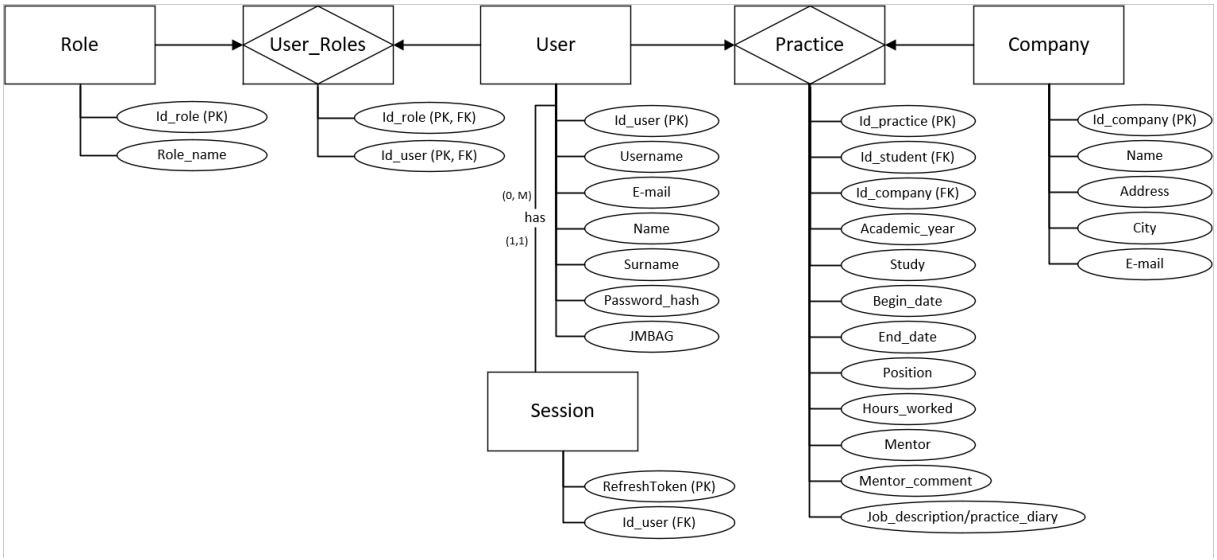
Kako bi se lakše implementirala sama baza podataka u PostgreSQL-u, korišten je pomoćni alat pgAdmin 4. On je jedan od najpopularnijih i najefikasnijih PostgreSQL alata koji predstavlja platformu za efikasno upravljanje i razvijanje baza podataka. Kao i sam PostgreSQL, tako se i ovaj alat može koristiti i podržava različite platforme i operacijske sustave što ga čini vrlo popularnim i široko dostupnim. pgAdmin 4 također pruža vrlo intuitivno grafičko korisničko sučelje zbog kojega mi je bilo vrlo jednostavno i lako kreirati

[5] SUBP – Sustav za upravljanje bazom podataka (Wikipedia, Sustav za upravljanje bazom podataka, 2022.).

[6] SQL – Structured Query Language – programski, upitni jezik za upravljanje podacima u bazi podataka (w3schools, SQL Tutorial, 2023.).

tablice u bazi podataka te njihove stupce. Osim kreiranja i upravljanja bazom podataka, odnosno tablicama, pgAdmin 4 karakterizira i mogućnost pisanja vlastitih SQL upita, kreiranje velikog broja različitih objekata baze podataka, upravljanje sigurnošću i performansama baze podataka, ali i kreiranje sigurnosnih kopija baze podataka i to sve preko vrlo intuitivnoga i jednostavnoga sučelja (pgAdmin, 2023.).

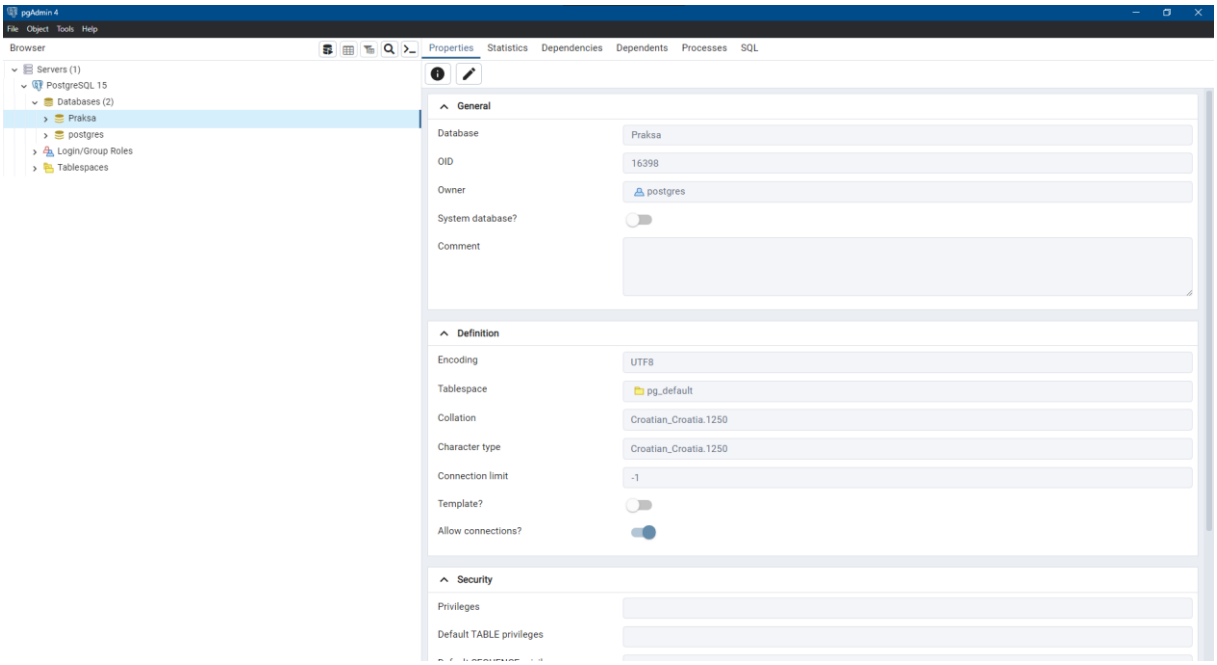
U nastavku je prikazan kratki opis i prikaz samog kreiranja baze podataka u pgAdmin 4 alatu. Općeniti model podataka koji je kreiran za kreiranu web aplikaciju prikazan je na slici 2. Svaki pravokutnik je implementiran kao zasebna tablica osim entiteta Role odnosno uloga korisnika, koji je u kodu implementiran kao nabrojani tip podataka („enum“) zbog uočene nepotrebnosti za upravljanjem podataka pojedinih uloga, ali i zbog unaprijed definiranih, statičnih uloga od kojih se sama web aplikacija i sastoji (Administrator, Profesor, Asistent, Student). U modelu je entitet Role ostao prikazan radi lakšega razumijevanja. Svaki entitet ima označene atribute od kojih se sastoji te su sa „PK[7]“ označeni primarni ključevi, a s „FK[8]“ vanjski ključevi. User_Roles predstavlja agregaciju između korisnika i rola odnosno uloga zbog više-više veze, dok Practice predstavlja agregaciju između korisnika i kompanije te ona ima vlastiti primarni ključ radi lakšega pronalaska pojedine prakse, ali i zbog mogućnosti da na dvije različite prakse bude isti student u istoj kompaniji. Pravokutnik Session služi za upravljanje prijavom i sesijama pojedinih korisnika te sadrži „RefreshToken“ kao primarni ključ, koji služi za identifikaciju sesije određenoga korisnika te za njeno produljenje, što je objašnjeno u četvrtom poglavlju te sadrži identifikator korisnika kao vanjski ključ te on služi kako bi se znalo kojem korisniku pripada određena sesija.



Slika 2 - Prikaz modela podataka web aplikacije za evidenciju stručne prakse na fakultetima

Nakon definiranja kompletnog modela, krenulo se je u implementaciju same baze podataka u pgAdmin 4 alatu gdje je korišteno intuitivno grafičko korisničko sučelje koje zadatke kreiranja tablica i njenih stupaca čini vrlo jednostavnim (slika 3).

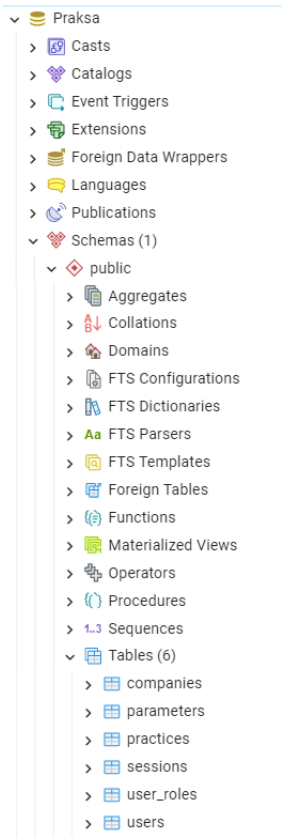
[7] PK – Primarni ključ (Primary key) (Microsoft, Primary and Foreign Key Constraints, 2023.).
[8] FK – Vanjski ključ (Foreign key) (Microsoft, Primary and Foreign Key Constraints, 2023.).



Slika 3 - Prikaz korisničkoga sučelja pgAdmin 4 alata

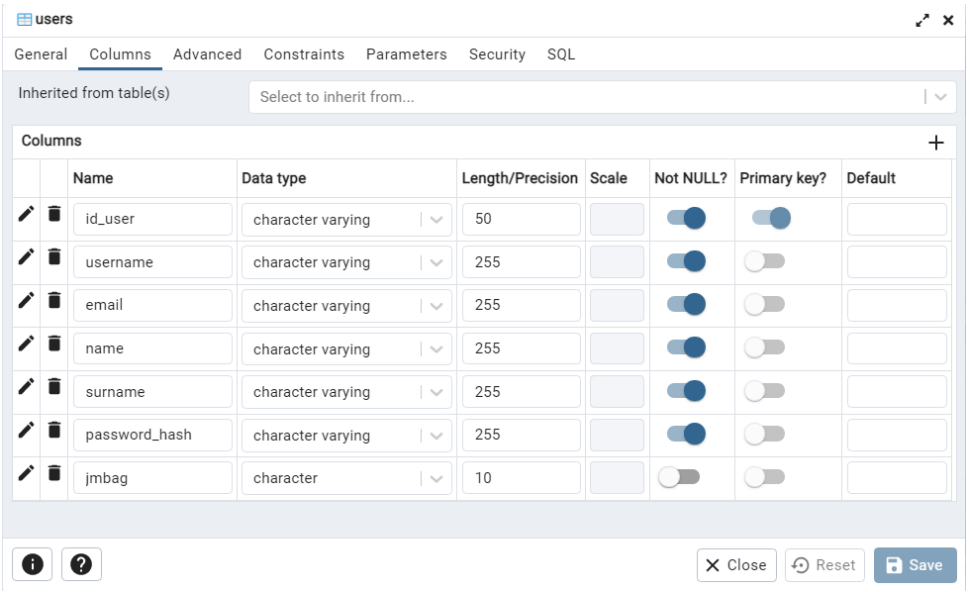
Za većinu aktivnosti i zadatke kreiranja baze podataka u ovome alatu korištena je opcija desnoga klika na mišu te odabira odgovarajuće opcije i unosa podataka. Tako se je prvo, desnim klikom na mapu „Databases“ odabrala opcija „Create“ pa „Database“ nakon čega se pojavio novi prozor u kojemu se definiraju glavne postavke tablice poput naziva, kodiranja, tabličnog prostora, sigurnosti i sl. Tako je kreirana baza podataka s nazivom Praksa kojoj su postavljeni zadani tablični prostor te UTF-8[9] standard kodiranja. Proširenjem novokreirane baze podataka Praksa, prikazale su se brojne dostupne mape i mogućnosti koje pruža pgAdmin 4 alat. Među svima njima, najvažnija je bila mapa „Schemas“ u kojoj se definiraju svi glavni objekti baze podataka pa tako i tablice. Tako se je, unutar mape „public“, desnim klikom na mapu „Tables“, koristila opcija „Create“ pa zatim „Table“ kako bi se kreirale sve tablice. Time se je otvorio novi prozor koji je služio za definiranje stupaca, primarnih i vanjskih ključeva te određenih ograničenja tablica na vrlo jednostavan način kroz intuitivno sučelje te su na taj način kreirane sve potrebne tablice (na slici 4 prikazana je i tablica „parameters“ koja se kasnije koristila za migracije te ona nije bitna koliko i ostale tablice).

[9] UTF-8 – Unicode Transformation Format – standard kodiranja znakova varijabilne duljine koji podržava znakove i slova iz različitih svjetskih jezika (Wikipedia, UTF-8, 2023.).

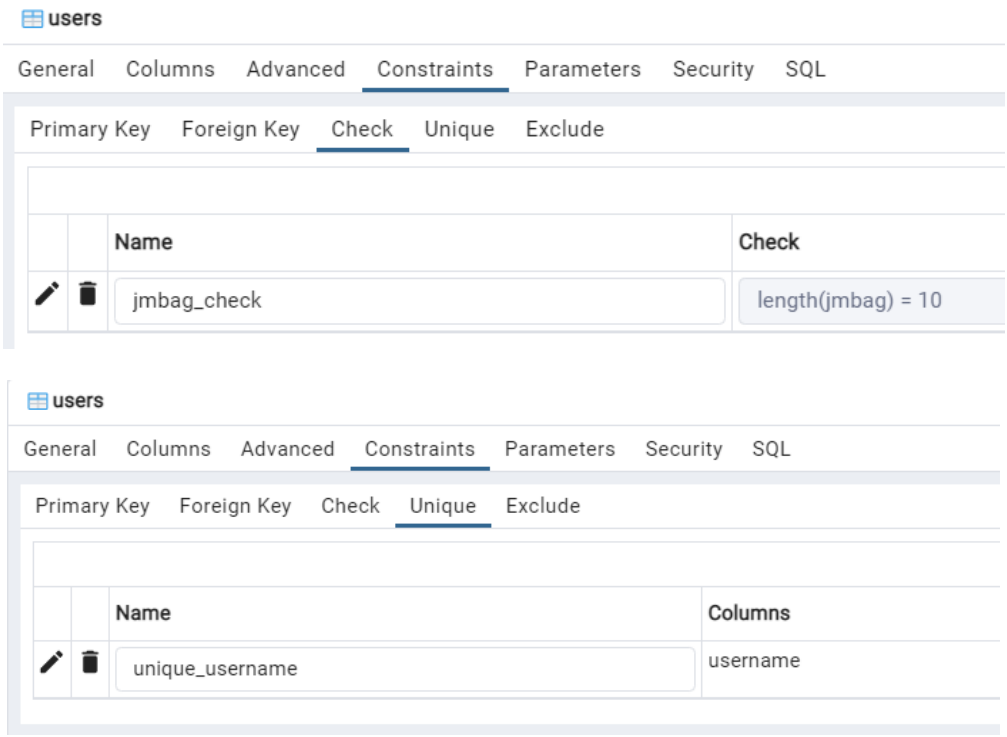


Slika 4 - Prikaz strukture baze podataka u pgAdmin 4 alatu

Kod kreiranja određenih tablica su se, osim definiranja stupaca, njihovih tipova podataka, duljine i mogućnosti da nema vrijednosti (slika 5), definirala i potrebna ograničenja poput „unique“ ograničenja za korisničko ime korisnika (kako korisnici ne bi mogli imati jednako korisničko ime), duljine JMBAG-a te vanjskih ključeva (slika 6). To je sve implementirano odabirom odgovarajućih kartica s prikladnim nazivom na vrhu prozora pri kreiranju tablica.

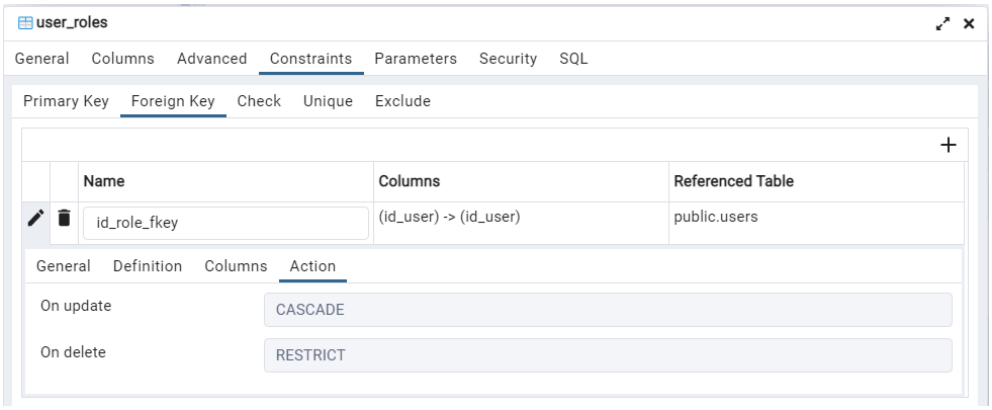


Slika 5 - Prikaz definiranja stupaca u pgAdmin 4 alatu



Slika 6 - Prikaz definiranja ograničenja u tablicama unutar pgAdmin 4 alata

Prilikom definiranja vanjskih ključeva se je, osim definiranja stupca kojega treba referencirati i koji predstavlja vanjski ključ, definiralo i ponašanje u slučaju ažuriranja i brisanja podataka u tablicama u kojima je to polje primarni ključ. Tako se je postavilo da se ažuriranjem toga polja, ono kaskadno ažurira i u tablici gdje je vanjski ključ, dok je brisanje toga polja onemogućeno (postavljanjem opcije „RESTRICT“ što je prikazano na slici 7).



Slika 7 - Prikaz definiranja vanjskoga ključa putem pgAdmin 4 alata

Na taj način su se kreirale i ostale potrebne tablice i njihovi stupci te se je uspješno implementirala i dovršila baza podataka u PostgreSQL-u koja obavlja funkciju pohrane svih podataka koji se koriste kroz web aplikaciju za evidenciju stručne prakse na fakultetima.

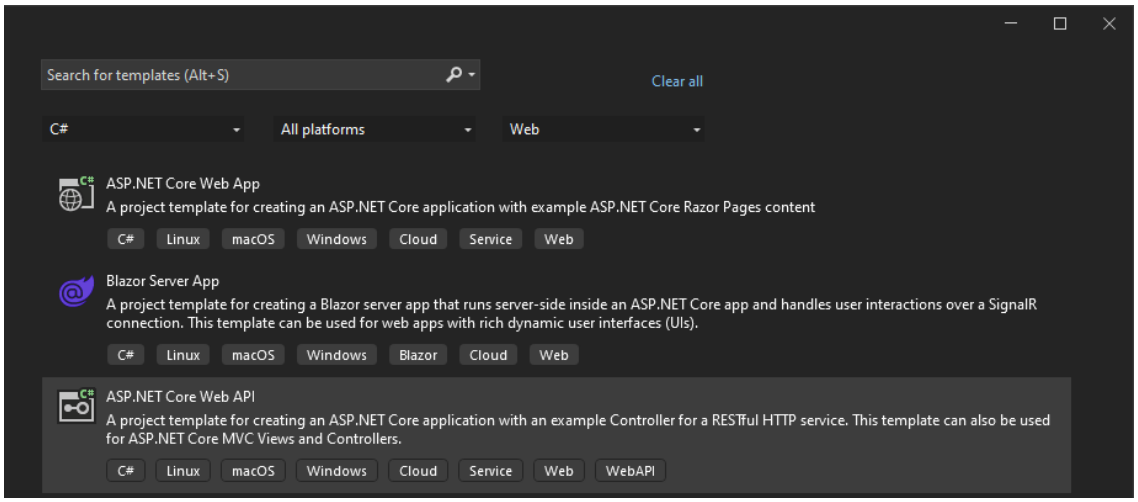
3.2 ASP.NET Core i .NET Core razvojni okviri (.NET 7)

Za implementaciju samog Web API-a korišten je ASP.NET Core razvojni okvir koji primarno koristi C# programski jezik. ASP.NET Core je samo jedan manji dio velike .NET Core platforme (nadogradnja klasičnoga .NET razvojnog okvira) koja ima puno općenitiju ulogu te služi za razvoj različitih vrsta aplikacija, dok sam ASP.NET Core ima nešto specifičniju ulogu u razvoju modernih i mrežnih aplikacija. Za razliku od .NET Core platforme, koja omogućuje i olakšava razvoj raznovrsnih aplikacija za različite platforme, ASP.NET Core je samo jedan njen dio, odnosno razvojni okvir koji se većinom specificira na razne aspekte web razvoja, poput web aplikacija, Web API-a i sličnih komponenti te je to bio jedan od glavnih razloga njegovog odabira za izradu Web API-a kreirane web aplikacije.

ASP.NET Core je razvojni okvir otvorenoga koda koji glavni fokus stavlja na podršku za različite platforme i performanse modernih web aplikacija. Fokusiranost na web razvoj, mogućnost pokretanja kreiranih web aplikacija na različitim platformama, performanse, otvorenost koda, ali i dosta velika korištenost i popularnost su neki od glavnih razloga koji su utjecali na odabir upravo ovoga razvojnoga okvira za implementaciju Web API-a (Roth, Anderson, & Luttin, 2022.).

Kod same implementacije Web API-a, korišten je C# programski jezik koji je pisan unutar Visual Studio 2022 integriranog razvojnoga okruženja, što je detaljnije opisano u četvrtome poglavlju. Kako bi se ASP.NET Core uopće koristio unutar Visual Studio 2022 razvojnoga okruženja, bilo je potrebno instalirati .NET SDK[10] koji predstavlja osnovu za početak izgradnje bilo koje vrste .NET aplikacija. Nakon njegove uspješne instalacije, trebalo je kreirati sam projekt unutar Visual Studio 2022 razvojnoga okruženja. Tu se je prilikom kreiranja projekta, odabrao gotovi predložak pod nazivom „ASP.NET Core Web API“ (slika 8) koji uvelike olakšava izgradnju samoga Web API-a jer odmah stvara osnovne mape i dijelove koji su potrebni za njegovu izgradnju. Također se je u postavkama kreiranja projekta odabrala .NET 7 verzija razvojnoga okvira koja integrira i uključuje najnoviju verziju i ASP.NET Core, ali i .NET Core razvojnih okvira i njihovih funkcionalnosti. Time se može reći da je za Web API-a i ostalih dijelova stražnjega sloja („backend“) korišten samo .NET 7 razvojni okvir, bez razdvajanja na ASP.NET Core i .NET Core jer novije verzije integriraju cijelu .NET platformu u jedan razvojni okvir s različitim mogućnostima, a projekti za Web API onda koriste mogućnosti i implementiraju ASP.NET Core razvojni okvir dok ostali dijelovi koriste mogućnosti i implementiraju .NET Core razvojni okvir koji je potreban za ispravno funkcioniranje.

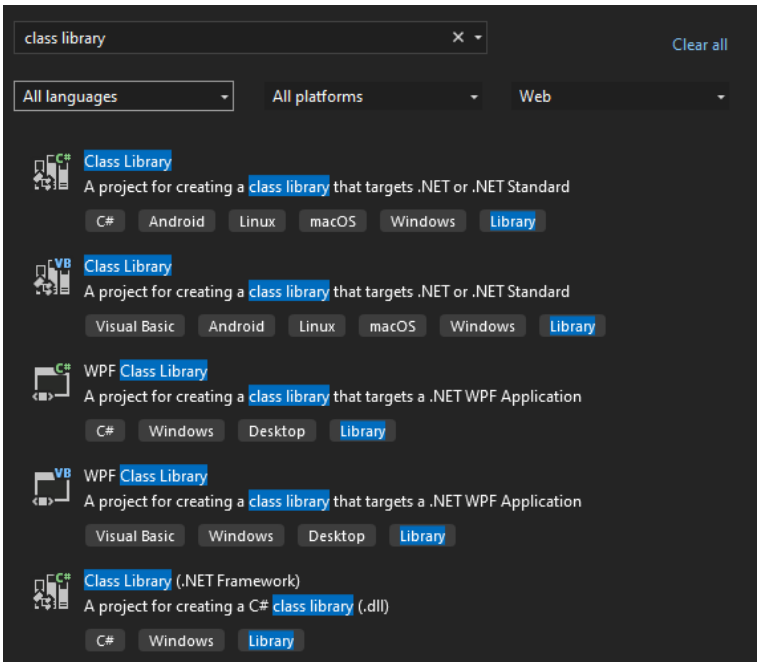
[10] SDK – Software Development Kit – skup alata i biblioteka koji predstavljaju osnovu za razvoj .NET aplikacija (Microsoft, What is the .NET SDK?, 2022.).



Slika 8 - Prikaz predloška za izradu Web API-a unutar Visual Studio 2022 razvojnoga okruženja

Kod implementacije središnjega sloja („middle layer“) i sloja pristupa podacima („data access layer“) korišten je već spomenuti .NET Core razvojni okvir koji se ne fokusira na web razvoj, već pruža mogućnosti za kreiranje različitih vrsta aplikacija i njenih ključnih dijelova. On je nastao kao nadogradnja već postojećeg .NET razvojnoga okvira, dodajući mu podršku za različite platforme što je jedna od njegovih glavnih prednosti. Također, .NET Core je na postojeći .NET razvojni okvir dodao i brojne snažnije i veće mogućnosti prilagodbi performansi te je zbog toga, ali i zbog većinom istih razloga kao i kod ASP.NET Core razvojnoga okvira, poput otvorenosti koda i poboljšane sigurnosti, odabran za implementaciju središnjega sloja i sloja pristupa podacima. Kod njihovog kreiranja je također korišten C# programski jezik jer predstavlja primarni programski jezik kojega koristi .NET razvojni okvir.

Središnji sloj, modeli te sloj pristupa podacima kreirane web aplikacije su dodani kao zasebni projekti unutar istoga rješenja („Solution“ koji služi kao kontejner za projekte) u Visual Studio 2022 razvojnom okruženju gdje je kreiran i projekt za Web API što je detaljnije objašnjeno u četvrtom poglavlju. Prilikom kreiranja tih projekata, osim odabira opcije Add → New Project i .NET 7 verzije razvojnoga okvira, za vrstu projekta odabrala se je opcija „Class library“ koja služi za kreiranje različitih klasa i sučelja koje se mogu koristiti i dijeliti kako bi se njihove funkcionalnosti mogle koristiti kroz više aplikacija i projekata. Odabrana je ta vrsta projekta kako bi kreirani projekti za Web API, središnji sloj te sloj pristupa podacima mogli zajedno razmjenjivati i koristiti definirane funkcionalnosti. Prilikom odabira vrste projekta, ponudile su se različite vrste „Class library-a“, ovisno o željenom razvojnom okviru i programskom jeziku te je odabrana prva ponuđena opcija koja automatski koristi .NET Core razvojni okvir te C# programski jezik (slika 9).



Slika 9 - Prikaz odabira vrste projekta za implementaciju središnjega sloja i sloja pristupa podacima

3.3 React

Za implementaciju prednjega dijela, odnosno samoga korisničkoga sučelja kreirane web aplikacije korišten je React. On predstavlja biblioteku temeljenu na JavaScript programskom jeziku koja se koristi za razvoj grafičkih korisničkih sučelja različitih vrsta aplikacija. JavaScript programski jezik je uz HTML i CSS, najkorišteniji programski jezik za razvoj web stranica. React je odabran za izradu samoga korisničkoga sučelja kreirane web aplikacije zbog široke korištenosti i popularnosti, jednostavne izrade ugodnih korisničkih sučelja i temeljenosti na ponovnom iskorištavanju koda (ReactTeam, 2023.).

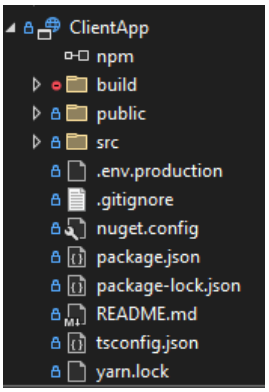
React se temelji na nekoliko ključnih elemenata koji omogućuju izradu različitih korisničkih sučelja. Jedni od najvažnijih od tih elemenata, koji su korišteni i prilikom izrade kreirane web aplikacije, su komponente koje predstavljaju pojedinačne dijelove korisničkoga sučelja sa svojstvom ponovne iskoristivosti i vlastite funkcionalnosti. One su zapravo određene JavaScript funkcije koje se kreiraju korištenjem ključne riječi „function“ te one najčešće služe za upravljanje stanjem, različitim svojstvima te postavkama samoga prikazivanja pojedinih dijelova korisničkih sučelja. Komponente u React-u najčešće vraćaju vrstu jezika za označavanje („markup“) koji sliči HTML-u[11] kroz JSX[12] sintaksu koja predstavlja drugi temeljni element React-a. JSX omogućuje da JavaScript funkcije, odnosno komponente vraćaju određenu vrstu HTML-a (najčešće se nalazi u „return“ dijelu komponenti) i na taj način definiraju glavne izglede pojedinih dijelova korisničkoga sučelja. Još poneki od glavnih elemenata na kojima se React temelji su stanja, svojstva te kuke

[11] HTML – HyperText Markup Language, temeljni jezik za kreiranje web stranica (w3schools, HTML tutorial, 2023.).

[12] JSX – JavaScript XML – sintaksa koja se koristi za pisanje jezika za označavanje u JavaScript-u (w3schools, React JSX, 2023.).

(„Hooks“). Stanja omogućuju izgradnju različitih interaktivnih elemenata korisničkoga sučelja. Ona se najčešće koriste za pohranu i upravljanje stanjima i podacima unutar pojedinih komponenti kada se podaci dinamički mijenjaju kao kod korisnikovog unosa, što je i implementirano u kreiranoj web aplikaciji. Stanja se koriste pomoću „useState“ kuke unutar komponenti. Svojstva ili „props“ su vrlo slični običnim parametrima u funkcijama u ostalim programskim jezicima te služe za prosljeđivanje određenih varijabli ili vrijednosti unutar komponenti. Konkretnim vrijednostima unutar svojstava komponenti se pristupa s operatorom točke (na primjer ako je proslijeđeno svojstvo s imenom, konkretnoj vrijednosti imena bi se pristupalo sa svojstvo.ime). Kuke ili „Hooks“ su zapravo React funkcije koje omogućuju implementacije različitih React značajki poput već spomenutoga korištenja stanja ili pokretanja dodatnih aktivnosti unutar React komponenti. Jedna od najčešćih React kuka je „useState“ koja omogućuje praćenje stanja pojedinih komponenti. Još jedna od najpoznatijih React kuka je i „useEffect“ koja omogućuje izvršavanje određenih dodatnih i vanjskih aktivnosti poput dohvaćanja određenih dodatnih podataka unutar komponenti. Svi navedeni elementi su samo neki od glavnih i temeljnih komponenti na kojima se React temelji te on pruža još veliki broj različitih značajki i funkcionalnosti koje omogućuju jednostavno kreiranje korisničkih sučelja (ReactTeam, 2023.).

Sama React aplikacija je dodana u postojeće rješenje („Solution“) unutar Visual Studio 2022 razvojnog okruženja kako bi se sve nalazilo na jednome organiziranome mjestu. React aplikacija je kreirana pomoću naredbe „create-react-app“, s parametrom „–template typescript“ kako bi se kreirale TypeScript, a ne JavaScript datoteke, koja automatski stvara početnu strukturu temeljne React aplikacije sa svim potrebnim mapama i datotekama. Na taj način je omogućena izuzetno brza, efikasna i jednostavna kreacija React aplikacija bez potrebe za ručnim definiranjem njenih pojedinih glavnih dijelova. Nakon kreiranja React aplikacije, ona je dodana kao novi projekt („TypeScript React Project“) unutar postojećeg rješenja u kojemu se nalaze projekti za Web API, središnji dio Web aplikacije te za sloj pristupa podacima. Na taj način kreirala se React aplikacija koja je sadržavala src mapu gdje se definira glavni kod React aplikacije, public mapu s raznim statičnim sadržajima koji se ne mijenjaju te s indeks.html datotekom koja predstavlja glavni predložak za prikazivanje React aplikacije, datoteku package.json koja predstavlja konfiguracijsku datoteku s glavnim podacima o React aplikaciji poput zavisnostima, verziji, konfiguraciji te raznim skriptama (slika 10). Kreirale su se i ostale osnovne datoteke i mape, no ovdje su spomenute najvažnije.



Slika 10 - Prikaz React aplikacije unutar Visual Studio 2022 razvojnog okruženja

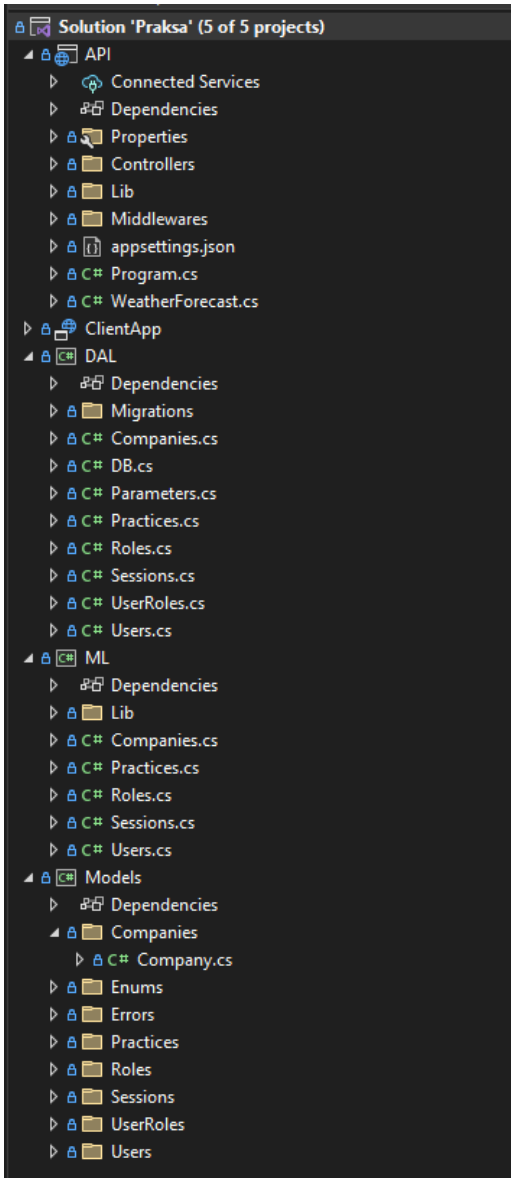
4. Objašnjenja dijelova koda

U ovome poglavlju prikazani su pojedini glavni dijelovi kodova kreirane aplikacije te njihovi opisi, ali su i odmah na početku objašnjene glavne ideje temeljne strukture i podjele cijeloga projekta te su prikazana kratka objašnjenja Visual Studio 2022 integriranoga razvojnoga okruženja te C# i TypeScript programskih jezika.

Za pisanje svih kodova kreirane web aplikacije korišten je Visual Studio 2022, integrirano razvojno okruženje te uređivač koda koji omogućuje razvoj raznovrsnih aplikacija. Glavni razlog zašto je odabran za pisanje koda kreirane web aplikacije je široka podrška za različite alate i mogućnosti .NET razvojnoga okvira čime se uvelike olakšava kreiranje modernih web aplikacija. Visual Studio 2022 također pruža podršku za različite platforme i uređaje, a intuitivne i snažne mogućnosti detekcije i pronalaska pogrešaka, koje su često korištene prilikom implementacije kreirane web aplikacije, ubrzavaju proces razvoja. Također, Visual Studio 2022 omogućuje i lako upravljanje različitim potrebnim paketima i nudi brojne gotove predloške za web aplikacije, Web API-e i brojne druge dijelove web aplikacija. Podržava i brojne programske jezike i pomaže u pisanju koda koristeći automatsko dovršavanje i predviđanje određenih naredbi (Microsoft, Visual Studio: IDE and Code Editor for Software Developers, 2023.).

Glavna ideja temeljne strukture cijeloga projekta i kreirane web aplikacije temeljila se na izradi pojedinih odvojenih projekata za zasebne dijelove web aplikacije unutar jednoga Visual Studio 2022 rješenja („Solution“). Na taj način je omogućeno da svi različiti dijelovi web aplikacije budu na jednome mjestu, ali unutar zasebnih i odvojenih projekata kako bi se zasebni dijelovi web aplikacije mogli neovisno uređivati, bez potrebe za promjenom ostalih dijelova i projekata (slika 11). Tako su kreirani zasebni projekti za Web API, središnji sloj, sloj pristupa podacima, modele te za prednji dio, odnosno za samu React aplikaciju. Pritom, kako bi se međusobno mogli koristiti podaci iz različitih dijelova web aplikacije, projektima su definirane međusobne zavisnosti („Project dependencies“, tako na primjer „API“ ovisi o „ML“ projektu i „Models“ projektu). Unutar većine projekata, radi veće i bolje organizacije, datoteke su se organizirale u zasebne mape koje su sadržavale pojedine dijelove toga projekta. Svaki projekt je imao svoju funkciju za implementaciju određenoga glavnoga dijela web aplikacije. Tako su se u projektu „Models“ kreirale datoteke za definiranje glavnih modela i podataka cijele web aplikacije te njihovih međusobnih odnosa (ti modeli se koriste kroz Web API kao zahtjevi i odgovori). Tu se zapravo za svaku tablicu iz baze kreirao zasebni model koji je sadržavao iste atribute kao i u izvornoj tablici. Unutar „API“ projekta se je implementirala glavna funkcionalnost Web API-a koji služi za definiranje logike usmjeravanja zahtjeva na pojedine rute te vraćanje odgovora. Za to su se koristili i implementirali kontroleri koju unutar Web API-a obavljaju upravo te navedene funkcionalnosti. „API“ projekt je ovisio o „ML“ projektu i „Models“ projektu jer su kontroleri vraćali određene modele kao odgovore te su pozivali određene funkcije iz „ML“ projekta. „ML“ projekt bi zapravo predstavljao središnji sloj web aplikacije odnosno „middle layer“ u kojemu je implementirana glavna poslovna logika obrade zaprimljenih

zahtjeva te koji služi kao spona između prednjega dijela, odnosno zaprimljenih zahtjeva te same baze podataka. Taj projekt je ovisio o „DAL“ i „Models“ projektima jer je također trebao koristiti podatke pojedinih modela koje je vraćao te je pozivao određene funkcije iz „DAL“ projekta. „DAL“ projekt bi zapravo predstavljao sloj pristupa podacima web aplikacije, odnosno „data access layer“ u kojemu je implementirana temeljna komunikacija s bazom podataka te različiti upiti nad samim podacima u bazi kako bi se ti podaci mogli dodavati, dohvaćati, ažurirati ili brisati. Taj projekt je ovisio samo o „Models“ projektu jer je koristio podatke pojedinih modela, odnosno samih tablica. „ClientApp“ projekt predstavlja React aplikaciju u kojoj je definiran prednji dio web aplikacije, odnosno korisničko sučelje. Na taj način korisnikov zahtjev na određenu rutu, zaprimljen iz korisničkoga sučelja, prolazi kroz kontrolere u Web API-u koji pozivaju određene funkcije iz središnjeg sloja gdje se onda pozivaju određene funkcije iz sloja za pristup podacima kako bi se vršile manipulacije nad konkretnim podacima iz baze te kako bi kontroleri iz Web API-a mogli vratiti prikladne odgovore na te zahtjeve.



Slika 11 - Prikaz podjele projekta i datoteka unutar Visual Studio 2022 razvojnoga okruženja

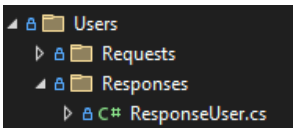
Za sve projekte stražnjega dijela aplikacije, odnosno za „API“, „Models“, „DAL“, i „ML“ projekte je korišten C# programski jezik. C# je primarni programski jezik .NET platforme te ima velikih sličnosti s generalnijim C programskim jezikom. Vrlo često se koristi pri izgradnji velikog broja različitih vrsta .NET aplikacija te se temelji na objektnoj orijentiranosti. C# pruža mogućnosti korištenja i kreiranja različitih tipova podataka te je ta činjenica uvelike pomogla prilikom izrade samih modela te njihovih atributa unutar kreirane web aplikacije. Dodatne glavne značajke C# programskoga jezika, koje ga čine vrlo popularnim za izradu web aplikacija, su automatsko upravljanje memorijom, kompatibilnost s drugim programskim jezicima te iznimno dobra podrška unutar Visual Studio 2022 razvojnoga okruženja koje je namijenjeno te pruža brojne dodatne alate za njegovu primjenu (Microsoft, A tour of the C# language, 2023.).

Za implementaciju React aplikacije, odnosno korisničkoga sučelja, korišten je TypeScript programski jezik koji svoje temelje preuzima od JavaScript programskoga jezika. JavaScript je programski jezik koji je namijenjen za web i razvoj dinamičnih web stranica. Glavni razlog zbog kojega je korišten TypeScript, a ne JavaScript programski jezik je lakše i efikasnije pronalaženje i ispravljanje pogrešaka u kodu jer se TypeScript bazira na tipovima podataka te automatski javlja poruke pogreške u samome uređivaču koda prilikom njegovoga pisanja, dok JavaScript vraća te poruke pogreške tek nakon pokretanja toga dijela programa. Također, TypeScript koristi JavaScript sintaksu te na nju dodaje dodatne mogućnosti i značajke poput strože provjere tipova podataka i bolje sigurnosti koda. Zbog temeljenosti na JavaScript programskome jeziku, TypeScript omogućuje iste mogućnosti kao i JavaScript te se često koristi za dodavanje dinamičkih dijelova web stranica, a porast njegove popularnosti dokazuju brojke o njegovoj korištenosti i raširenosti (Microsoft, TypeScript: JavaScript With Syntax For Types, 2023.).

U nastavku se nalaze opisi pojedinih dijelova koda kreirane web aplikacije. Iz svakoga projekta (dijela web aplikacije) koji je kreiran, prikazana je i opisana po jedna datoteka jer je uglavnom korištena ista struktura i principi kroz sve datoteke toga projekta.

4.1 Modeli

Unutar projekta „Models“ kreirane su zasebne mape za većinu tablica iz baze podataka. Modeli predstavljaju glavne podatke web aplikacije koje koristi Web API za zahtjeve ili odgovore. Datoteke pojedinih modela (koje su dodane kao C# klase s nastavkom .cs) su organizirane u mape radi bolje organizacije, ali i zbog dijeljenja modela na one koji se koriste prilikom slanja zahtjeva i na one koji se koriste prilikom vraćanja odgovora (slika 12). Na primjer, korisnici su, osim običnoga modela iz baze podataka, podijeljeni na model korisnika za slanje zahtjeva te model za slanje odgovora jer kod slanja zahtjeva trebamo specificirati lozinku za toga korisnika, a kod njegovoga prikazivanja tu lozinku ne želimo prikazati jer ne želimo da svi vide takve osjetljive podatke.



Slika 12 - Prikaz podjele modela na zahtjeve i odgovore

Svi modeli su kreirani kao javne klase kako bi im se moglo pristupati iz različitih projekata te definiraju glavne podatke web aplikacije. Tako recimo model korisnika za slanje zahtjeva, koji je prikazan na slici 13, sadrži sve glavne atribute iz tablice u bazi podataka, koji su definirani s različitim tipovima podataka unutar klase. Svi atributi su također javni kako bi im se moglo pristupati iz različitih dijelova. Tako je svaki atribut iz tablice korisnika iz baze podataka, kreiran kao podatkovni član klase s određenim tipom podataka i nazivom (kod ovoga primjera korisnika su svi podatkovni članovi tipa „string“, odnosno znakovni nizovi). Za atribute koji mogu poprimiti „null“ vrijednosti (što je definirano prilikom kreacije tablica u pgAdmin-u), korišten je upitnik nakon navođenja tipa podatka koji u C# programskom jeziku označava da varijabla može poprimiti „null“ vrijednost (u ovome primjeru je to samo JMBAG jer korisnici koji nisu studenti neće imati JMBAG). Nakon naziva podatkovnoga člana klase, korištena je notacija „{get; set;}“ koja je dostupna u C# programskome jeziku te koja služi za automatsko definiranje metoda za dohvaćanje i postavljanje vrijednosti pojedinih podatkovnih članova čime se eliminira potreba za njihovim ručnim definiranjem što uvelike olakšava razvoj. Podatkovnim članovima koji ne mogu biti „null“ dodijeljen je izraz „=null!“ gdje se operator uskličnika koristi nakon vrijednosti (kada se koristi prije vrijednosti označava jednostavan operator negacije) kako bi definirali vrijednosti koje su zadano prazne, ali smo za njih sigurni da neće poprimiti „null“ vrijednosti kako bi time izbjegli upozorenja kompajlera o mogućim „null“ vrijednostima prilikom korištenja tih varijabli. Iznad podatkovnih članova, korištene su anotacije podataka („data annotations“) koje u C# predstavljaju atribute koji se koriste unutar uglatih zagrada kako bi se definirale validacije prilikom unosa pojedinih podatkovnih članova. Tu je također, prilikom svake validacije, korištena i „ErrorMessage“ opcija kako bi se definirala poruka u slučaju pogreške. Tako su korišteni atributi „Required“ za definiranje vrijednosti koje ne smiju biti prazne i koje moraju poprimiti vrijednost, „StringLength“ za definiranje maksimalne duljine znakovnoga niza, „RegularExpression“ za definiranje vlastitoga regularnog izraza kako bi se dozvolio samo unos određenoga tipa, „EmailAddress“ za označavanje da je unos e-mail te automatsko provjeravanje je li upisan ispravan e-mail te atributi „MinLength“ i „MaxLength“ za definiranje minimalne i maksimalne duljine unosa. Osim atributa koji se nalaze u tablici korisnika, ovdje je dodan i podatkovni član s nazivom „Role“ tipa „IEnumerable“ koji predstavlja sučelje za pohranjivanje više vrijednosti kroz koje se može iterirati. U ovome slučaju služi za pohranjivanje kolekcije tipa „string“ (znakovnih nizova), odnosno naziva rola pojedinoga korisnika kako bi se omogućilo da prilikom unosa korisnika odmah i odaberemo njegove dozvole te ih pohranimo u tablicu koja pohranjuje korisnike i njihove dozvole. Tu se odmah i postavlja inicijalna vrijednost toga podatkovnoga člana s „new List<string>();“ kako bi se stvorila nova instanca liste znakovnih nizova i kako vrijednost podatkovnoga člana ne bi bila „null“. U ovome modelu korisnika za zahtjeve je korišteno polje „Password“ koje se razlikuje od polja „Password_hash“ u bazi jer korisnik prilikom kreiranja ili ažuriranja šalje

konkretnu lozinku koja se onda kasnije „hashira[13]“ i takva se pohranjuje u bazu podataka. Kod ostalih modela su se koristili podatkovni članovi koji su odgovarali atributima u tablicama u bazi podataka. Također se kod korisnika, u „request“ mapama, kreiralo više različitih modela korisnika koji se šalju prilikom različitih aktivnosti (na primjer model korisnika za ažuriranje koji nema validacije za polje lozinke, model korisnika koji se koristi prilikom prijave te sadrži samo korisničko ime i lozinku). Modeli u „response“ mapama su većinom izgledali jednako onima u „request“ mapama samo nisu sadržavali validacije.

```
using System;
using System.Collections;
using System.Collections.Generic;
using System.ComponentModel.DataAnnotations;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace Models.Users.Requests
{
    3 references
    public class RequestUser
    {
        [Required(AllowEmptyStrings = false, ErrorMessage = "Polje {0} ne smije biti prazno!")]
        1 reference
        public string Id_user { get; set; } = null!;

        [Required(AllowEmptyStrings = false, ErrorMessage = " ne smije biti prazan!")]
        [StringLength(30, MinimumLength = 3, ErrorMessage = " mora imati minimalno 3 znakova i maksimalno 30 znakova!")]
        [RegularExpression(@"^[a-zA-Z0-9]*$", ErrorMessage = " mora biti kombinacija brojeva ili slova bez razmaka.")]
        1 reference
        public string Username { get; set; } = null!;

        [Required(AllowEmptyStrings = false, ErrorMessage = "Email ne smije biti prazan!")]
        [EmailAddress(ErrorMessage = "Molimo unesite valjani Email!")]
        1 reference
        public string Email { get; set; } = null!;

        [Required(AllowEmptyStrings = false, ErrorMessage = "Lozinka ne smije biti prazna")]
        [MinLength(6, ErrorMessage = "Lozinka mora sadržavati minimalno 6 znakova")]
        [RegularExpression(@"^(?=.*[A-Z])(?=.*[0-9]).+$", ErrorMessage = "Lozinka mora imati najmanje jedno veliko slovo i najmanje jedan broj.")]
        2 references
        public string Password { get; set; } = null!;

        [Required(AllowEmptyStrings = false, ErrorMessage = "Ime ne smije biti prazan")]
        [StringLength(30, MinimumLength = 3, ErrorMessage = "Ime mora imati minimalno 3 znakova i maksimalno 30 znakova!")]
        [RegularExpression(@"^[a-zA-ZšŠčČćĆžŽđĐ\s]*$", ErrorMessage = "Ime mora biti kombinacija slova.")]
        1 reference
        public string Name { get; set; } = null!;

        [Required(AllowEmptyStrings = false, ErrorMessage = "Prezime ne smije biti prazan")]
        [StringLength(30, MinimumLength = 3, ErrorMessage = "Prezime mora imati minimalno 3 znakova i maksimalno 30 znakova!")]
        [RegularExpression(@"^[a-zA-ZšŠčČćĆžŽđĐ\s]*$", ErrorMessage = "Prezime mora biti kombinacija slova.")]
        1 reference
        public string Surname { get; set; } = null!;

        [MinLength(10, ErrorMessage = "JMBAG mora sadržavati točno 10 znamenki!")]
        [MaxLength(10, ErrorMessage = "JMBAG mora sadržavati točno 10 znamenki!")]
        [RegularExpression(@"^[0-9]*$", ErrorMessage = "JMBAG mora sadržavati samo brojke.")]
        1 reference
        public string? JMBAG { get; set; }
        2 references
        public IEnumerable<string> Role { get; set; } = new List<string>();
    }
}
```

Slika 13 - Prikaz modela korisnika za zahtjeve

Na isti način su kreirani ostali modeli, odnosno .cs datoteke s klasama za kompanije, prakse i sesije s time da kompanije nemaju podjelu na mape „request“ i „response“. U tim datotekama su se koristile poneke druge anotacije („data annotations“) te drugačiji tipovi podataka, recimo DateTime za datume te int za brojeve. Također, pošto je odlučeno da role korisnika neće biti tablica i da će biti četiri fiksirane uloge u web aplikaciji, one su kreirane kao nabrojani tip podataka („enum“) s pripadajućim nazivima i vrijednostima za dohvaćanje (brojčanim i znakovnim) što je prikazano na slici 14.

[13] Hashiranje – pretvaranje vrijednosti u niz znakova pomoću algoritma (Wikipedia, Hash tablica, 2022.).

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace Models.Enum
{
    4 references
    public enum Rola
    {
        [Description("Student")]
        Student = 0,
        [Description("Asistent")]
        Asistent = 1,
        [Description("Profesor")]
        Profesor = 2,
        [Description("Administrator")]
        Administrator = 3
    }
}
```

Slika 14 - Prikaz modela za role korisnika

4.2 Web API (kontroleri)

Kontroleri predstavljaju jedan od temeljnih koncepata Web API-a koji je zadužen za obavljanje funkcija poput usmjeravanja zahtjeva na odgovarajuće rute te vraćanje odgovora. Time kontroleri, ali i sam Web API manipuliraju, odnosno primaju i vraćaju određene modele. Za svaki model, čijim se podacima trebalo upravljati, kreiran je jedan kontroler pa su tako kreirani kontroleri za kompanije, prakse, korisnike i sesije. Za role nije bio potreban kontroler pošto je odlučeno da se role, odnosno uloge korisnika neće moći dodavati, ažurirati i brisati. Za tablicu koja je pohranjivala korisnike i njihove role također nije bio potreban kontroler jer je njihovo dodavanje, ažuriranje i brisanje implementirano kod upravljanja podacima korisnika. Kontroleri su također kreirani kao javno dostupne klase s metodama za upravljanje zahtjevima.

U nastavku je prikazan opis kontrolera za korisnike. Na početku su, osim definiranja imenskoga prostora koji definira gdje se klasa nalazi, dodani i pomoćni atributi unutar uglatih zagrada za definiranje dodatnih postavki i konfiguracije samoga kontrolera. Tako „[Route“[controller]“]“ definira glavnu rutu kontrolera koja se u ovom slučaju postavlja pomoću dostupne opcije „[controller]“ koja uzima u obzir naziv kontrolera pa se tako kod kontrolera s nazivom „UsersController“ uzima „/users“ kao glavna ruta. Također, tu je dodan i atribut „[ApiController]“ koji služi kako bi se definiralo da kontroler treba pratiti glavne konvencije i pravila te kako bi se omogućile temeljne funkcije kontrolera poput automatske validacije. Nakon toga je definiran glavni naziv kontrolera, odnosno klase, a nakon naziva slijedi i dvotočka te „ControllerBase“ što označava da klasa nasljeđuje od klase pod nazivom „ControllerBase“ koja predstavlja već dostupnu klasu unutar ASP.NET Web API-a za definiranje različitih postavki i metoda za upravljanje zahtjevima. Unutar klase se nalaze definicije pojedinih funkcija za manipulaciju podacima korisnika poput dohvaćanja svih korisnika, dohvaćanja jednoga korisnika, dodavanja korisnika, ažuriranja korisnika te brisanja korisnika. Iznad svih funkcija su korištena dva atributa, „[Authorize]“ ili „[Authorize(Roles=“nazivi rola“)]“ za definiranje mora li korisnik biti autoriziran za njihovo korištenje te koje role, odnosno uloge imaju pristup određenoj funkciji te atribut [„HttpZahtjev“] ili [„HttpZahtjev(„{id}“)“] koji služi za definiranje vrste HTTP zahtjeva

kojega funkcija treba obraditi i njegove putanje. Također kod svih funkcija, u svim kontrolerima, ali i kroz cijeli Web API, korišteni su „async – await“ ključne riječi za definiranje asinkronosti. Time su kod definiranja svake funkcije korištene ključne riječi „async“, kako bi se definiralo da je ona asinkrona operacija, te „await“ unutar same funkcije kako bi se asinkrono čekao rezultat. Ta asinkronost je vrlo bitna prilikom pristupanja i postavljanja upita bazi podataka jer omogućuje efikasno i ispravno funkcioniranje cijele web aplikacije prilikom obrade složenih zahtjeva i operacija koji mogu trajati duže vrijeme jer omogućuje da web aplikacija nastavi obrađivati ostale zahtjeve, ne morajući čekati završetak dugih operacija koje mogu usporiti rad. Time se kod izvođenja složenih i dugih operacija ne dovodi do blokiranja izvođenja svih drugih operacija, već se omogućuje asinkrono čekanje rezultata uz mogućnost izvršavanja ostalih aktivnosti i zadataka čime se poboljšavaju performanse i efikasnost (Microsoft, Asynchronous programming with async and await, 2023.). Tako asinkrona funkcija za dohvaćanje svih korisnika (slika 15) vraća „Task“, koji predstavlja asinkronu operaciju koja se često koristi kod asinkronih funkcija te u ovom slučaju vraća kolekciju („IEnumerable“) modela korisnika za odgovore na zahtjeve (koji nemaju lozinke). Unutar funkcije se u varijablu „users“ sprema rezultat asinkrone funkcije „GetUsersAsync“ iz „ML“ projekta gdje je definirana poslovna logika. Tu je korištena ključna riječ „await“ kako bi asinkrono čekali na završetak izvršavanja te funkcije bez blokiranja izvršavanja ostatka zadataka. Nakon uspješnoga učitavanja svih korisnika u varijablu „users“, oni se transformiraju, pomoću „Adapt“ funkcije iz paketa „Mapster“, u modele korisnika za odgovore te se oni vraćaju kao odgovor na zahtjev korisnika. „Adapt“ funkcija uvelike olakšava pretvorbu i mapiranje između objekata, što uklanja potrebu za ručnim transformacijama.

```
using Mapster;
using Microsoft.AspNetCore.Authorization;
using Microsoft.AspNetCore.Http;
using Microsoft.AspNetCore.Mvc;
using Models;
using Models.Users.Requests;
using System.Net;
using System.Runtime.Serialization;
using System.Security.Claims;

namespace API.Controllers
{
    [Route("[controller]")]
    [ApiController]

    0 references
    public class UsersController : ControllerBase
    {
        [Authorize]
        [HttpGet]
        0 references
        public async Task<IEnumerable<Models.Users.Responses.ResponseUser>> GetAll()
        {
            var users = await ML.Users.GetUsersAsync();
            return users.Adapt<IEnumerable<Models.Users.Responses.ResponseUser>>();
        }
```

Slika 15 - Prikaz početka kontrolera za korisnike i funkcije za dohvaćanje svih korisnika

Na slici 16 prikazana je funkcija za dohvaćanje jednoga korisnika („GetUser“) koja je implementirana na vrlo sličan način kao i funkcija za dohvaćanje svih korisnika. Tu je kod definiranja zahtjeva dodan i ID korisnika kojega želimo dohvatiti kako bi se definiralo da se funkcija aktivira prilikom „GET“ zahtjeva na putanju „/users“ s dodanim ID-om korisnika („users/id“). Također, ova funkcija vraća samo jednoga korisnika, odnosno samo jedan model korisnika za odgovore te je tu dodan i upitnik nakon povratnoga tipa kako bi se naznačilo da odgovor može biti prazan. Funkcija kao argument prima znakovni niz koji predstavlja ID korisnika te poziva asinkronu funkciju „GetUserAsync“ iz „ML“ projekta s proslijeđenim ID-om korisnika kao parametrom čiji rezultat sprema u varijablu „user“. Na kraju funkcija vraća dohvaćenoga korisnika, pretvorenoga u model korisnika za odgovore, kao odgovor na zahtjev. Funkcija za kreiranje korisnika („AddUser“) je nešto specifičnija jer je ona omogućena samo korisniku s rolom administratora te je tu u povratni tip dodan i „ActionResult“ koji predstavlja rezultat izvođenja određene metode te omogućuje vraćanje statusnih kodova (slika 16). Ova funkcija kao argument prima cijeli model korisnika za zahtjeve sa svim atributima te se aktivira prilikom izvođenja POST zahtjeva. U njoj se najprije u varijablu „vaidationErrors“ sprema rezultat izvođenja funkcije iz „ModelValidator.cs“ datoteke kako bi se provjerilo jesu li pri kreiranju korisnika prekršene kreirane i već prikazane validacije nad pojedinim poljima u modelima. Ukoliko varijabla „validationErrors“ nije prazna, znači da postoje određene nezadovoljene validacije te se vraća statusni kod 422 s prigodnim porukama definiranim u modelima. Ukoliko je varijabla „validationErrors“ prazna, znači da s je unosima korisnika sve u redu te se zatim, kao i kod dohvaćanja jednoga korisnika, poziva asinkrona funkcija „AddUserAsync“ iz „ML“ projekta i na kraju se vraća dodani korisnik kao odgovor na zahtjev.

```
[Authorize]
[HttpGet("{Id_user}")]
0 references
public async Task<Models.Users.Responses.ResponseUser?> GetUser(string Id_user)
{
    var user = await ML.Users.GetUserAsync(Id_user);
    return user?.Adapt<Models.Users.Responses.ResponseUser>();
}

[Authorize(Roles = "Administrator")]
[HttpPost]
0 references
public async Task<ActionResult<Models.Users.Responses.ResponseUser>> AddUser(Models.Users.Requests.RequestUser user)
{
    var validationErrors = Lib.ModelValidator.ValidateModel(user);
    if (validationErrors != null)
    {
        return StatusCode(StatusCodes.Status422UnprocessableEntity, validationErrors);
    }
    var User = await ML.Users.AddUserAsync(user);
    return User.Adapt<Models.Users.Responses.ResponseUser>();
}
```

Slika 16 - Prikaz funkcija za dohvaćanje i kreiranje korisnika unutar kontrolera za korisnike

Na slici 17 prikazana je funkcija za ažuriranje korisnika (UpdateUser) koja je najsloženija unutar kontrolera za korisnike. Definirano je da obrađuje „PUT“ zahtjeve s definiranim ID-om korisnika te također vraća „Task“ tipa „ActionResult“ koji vraća model korisnika za odgovore kako bi mogla vratiti i statusni kod. Kao argument prima nešto drugačiji model korisnika koji nema definirane validacije nad poljem lozinke jer korisnik možda neće ažurirati lozinku, a ako je ažurira ona se treba ponovno hashirati kako bi bila

sigurna. Unutar funkcije se prvo, u varijablu „user_id“, sprema ID prijavljenoga korisnika (koristeći dostupni „User“ objekt koji dohvaća korisnika koji izvodi zahtjev) pomoću njegovih dozvola („Claims“) i funkcije „FindFirstOrDefault“ koja vraća prvu vrijednost koja zadovoljava uvjet ili zadanu vrijednost. Tu je korištena lambda funkcija za prolazak kroz sve dozvole koje su korištene prilikom generiranja tokena za određenoga korisnika te pronalaženje one s nazivom „user_id“ kako bi se dohvatila njena vrijednost (.Value). Zatim se ponovno provjeravaju validacije pomoću „ModelValidator“ datoteke kako bi se provjerilo je li s unosima ažuriranoga korisnika sve u redu. Nakon toga se, ponovno koristeći „User“ objekt, provjerava ima li prijavljeni korisnik ulogu studenta ili asistenta, pomoću „IsInRole“ metode koja vraća „true“ ako je uvjet istinit. Ukoliko je prijavljeni korisnik student ili asistent, provjerava se je li njegov ID jednak ID-u korisnika koji je poslan u zahtjevu kao parametar u funkciju za ažuriranje kako bi se omogućilo da studenti i asistenti mogu ažurirati samo svoje podatke. U slučaju da se ID-ovi ne podudaraju, vraća se statusni kod 403 koji označava zabranjen pristup. Nakon toga se, u varijablu „trenutniKorisnik“ dohvaćaju podaci o trenutnom korisniku koji je poslan u zahtjevu (kako bi se dohvatio njegov obični model) pozivom asinkrone funkcije iz „ML“ projekta za dohvaćanje jednoga korisnika preko njegovog ID-a. Zatim se provjerava je li trenutni korisnik administrator, točnije ako nema ulogu administratora te ako je taj uvjet istinit, provjerava se jesu li role u originalnom pohranjenom modelu korisnika jednake onima u modelu korisnika koji je poslan u zahtjevu za ažuriranje koristeći „SequenceEqual“ funkciju koja uspoređuje jednakost dva niza. Ako se role korisnika razlikuju znači da je korisnik koji nema ulogu administratora pokušao promijeniti dozvole te se time vraća statusni kod 403 za zabranjen pristup. Zatim se provjerava je li korisnik ažurirao polje „Password“, točnije ima li korisnik poslan u zahtjevu ispunjeno polje lozinke kako bi se ona mogla ponovno provjeriti. Tu se prvo u varijablu „PasswordCheck“ stvara novi objekt modela „UserPasswordValidation“ koji sadrži samo podatkovni član „Password“ s validacijama (model korisnika za ažuriranje ima polje lozinke bez validacija pa se one ovdje posebno provjeravaju) te se njemu prosljeđuje ažurirana lozinka iz zaprimljenoga modela korisnika za ažuriranje koja se zatim zasebno provjerava. Pri kraju se u varijablu „UserZaBazu“ sprema rezultat izvođenja asinkrone funkcije „UpdateUserAsync“ iz „ML“ projekta kojoj je proslijeđen ažurirani korisnik iz zahtjeva. Na kraju se, kao odgovor na zahtjev, vraća „UserZaBazu“ koji je transformiran u model korisnika za odgovore.

```
[Authorize]
[HttpPut("{Id_user}")]
0 references
public async Task<ActionResult<Models.Users.Responses.ResponseUser?>> UpdateUser(Models.Users.Requests.RequestUpdateUser user)
{
    var user_id = User.Claims.FirstOrDefault(x => x.Type == "user_id")?.Value;

    var validationErrors = Lib.ModelValidator.ValidateModel(user);
    if (validationErrors != null)
    {
        return StatusCode(StatusCodes.Status422UnprocessableEntity, validationErrors);
    }

    if (User.IsInRole("Student") || User.IsInRole("Asistent"))
    {
        if (user_id != user.Id_user)
        {
            return StatusCode(StatusCodes.Status403Forbidden);
        }
    }
    var trenutniKorisnik = await ML.Users.GetUserAsync(user.Id_user);
    if (!User.IsInRole("Administrator"))
    {
        if(trenutniKorisnik != null && !trenutniKorisnik.Role.SequenceEqual(user.Role))
        {
            return StatusCode(StatusCodes.Status403Forbidden);
        }
    }

    if (user.Password != null)
    {
        var PasswordCheck = new Models.Users.UserPasswordValidation() { Password = user.Password };
        var validationPasswordErrors = Lib.ModelValidator.ValidateModel(PasswordCheck);
        if (validationPasswordErrors != null)
        {
            return StatusCode(StatusCodes.Status422UnprocessableEntity, validationPasswordErrors);
        }
    }

    var UserZaBazu = await ML.Users.UpdateUserAsync(user);
    return UserZaBazu?.Adapt<Models.Users.Responses.ResponseUser>();
}
```

Slika 17 - Prikaz funkcije za ažuriranje korisnika unutar kontrolera za korisnike

Na samome kraju kontrolera za korisnike definirana je i metoda za brisanje korisnika („RemoveUser“) koja je prikazana na slici 18. Ona je omogućena jedino korisnicima s ulogama administratora i profesora te obrađuje HTTP DELETE zahtjeve s proslijeđenim ID-om korisnika. Ona je najjednostavnija funkcija te prima ID korisnika kao parametar i nema povratnoga tipa (Task se koristi zbog asinkronosti, ali funkcija korisniku ne vraća rezultat). Unutar nje se samo asinkrono poziva funkcija „RemoveUserAsync“ iz „ML“ projekta kojoj je proslijeđen ID korisnika za brisanje.

```
[Authorize(Roles = "Administrator, Profesor")]
[HttpDelete("{Id_user}")]
0 references
public async Task RemoveUser(string Id_user)
{
    await ML.Users.RemoveUserAsync(Id_user);
}

}
```

Slika 18 - Prikaz funkcije za brisanje korisnika u kontroleru za korisnike

Na isti način su kreirani i kontroleri za kompanije i prakse, samo su oni puno jednostavniji te većinom samo koriste validacije proslijeđenih modela te pozivaju

odgovarajuće funkcije iz „ML“ projekta. Osim kontrolera, u Web API projektu kreirana je mapa „Lib“ s datotekom u kojoj je definirana korištena funkcija za validacije modela te mapa „Middlewares“ u kojoj je definirana datoteka za globalno upravljanje pogreškama i vraćanje prikladnih odgovora. Na taj način svaki zahtjev prolazi kroz „Middleware“ kako bi se uspješno hvatale pogreške i iznimke koje nastaju tijekom njihove obrade. Tu se još nalazi i „Program.cs“ datoteka koja predstavlja glavnu ulaznu točku u Web API gdje su definirane razne konfiguracije i postavke koje su potrebne za ispravan rad.

4.3 Središnji sloj (ML projekt)

Unutar „ML“ projekta, koji predstavlja središnji sloj web aplikacije u kojemu je definirana glavna logika obrade zahtjeva, kreirane su zasebne .cs datoteke za sve modele, uključujući kompanije, prakse, korisnike, sesije, ali i role (za role na kraju nije korištena ta datoteka, ali je puštena u slučaju da se želi dodati i upravljanje rolama). Svi zahtjevi koji su pristigli na Web API, pozivali su pojedine metode iz „ML“ projekta gdje je definirana njihova logika obrade i konkretan pristup podacima u bazi pa time središnji sloj („ML“) predstavlja posrednika između Web API-a i sloja pristupa podacima.

U nastavku su prikazani glavni dijelovi središnjega sloja za korisnike. U njemu je kreirana statična klasa „Users“ (statična je kako ne bi morali kreirati instance korisnika za pozivanje metoda) s statičnim asinkronim funkcijama koje su pozivane u Web API-u i koje služe za definiranje logike obrade i povezivanje sa samom bazom podataka. Unutar svih funkcija kod korisnika su korištene konekcije na bazu podataka („Dbconnection“), kako bi se mogli pozivati konkretni upiti nad bazom iz „DAL“ projekta te transakcije („DbTransaction“) koje su potrebne prilikom pristupanja i izvođenja više operacija unutar jednoga pristupa, a to je kod korisnika potrebno jer se osim njegovih podataka, upravlja i podacima njegovih rola, odnosno uloga.

Na slici 19 prikazana je asinkrona statična funkcija za dodavanje korisnika („AddUserAsync“) koja kao argument prima model korisnika iz zahtjeva koji je proslijeđen iz Web API-a te vraća „Task“, odnosno asinkronu operaciju koja vraća model običnoga korisnika. Nakon toga se asinkrono dohvaća konekcija na bazu podataka pomoću funkcije „GetConnectionAsync“ iz „DB“ datoteke koja se nalazi u „DAL“ projektu te se također započinje nova transakcija pomoću dohvaćene konekcije kako bi se moglo izvršavati i pristupati više operacija kroz jednu konekciju, odnosno transakciju. Konekcija se sprema u „DbConnection“ koja predstavlja apstraktnu klasu za pohranu određene vrste konekcije na bazu, dok se transakcija sprema u „DbTransaction“ koja predstavlja klasu za pohranu određene vrste transakcije. Tu se prvo priprema model korisnika da odgovara tablici u bazi podataka. Tako se prvo u varijablu „userZaBazu“ pohranjuje obični, originalni model korisnika tako da se proslijeđeni model za zahtjeve transformira pomoću „Adapt“ metode gdje navodimo u koji objekt ga želimo transformirati („<User>“). Pošto proslijeđeni model za zahtjeve ima polje s konkretnom lozinkom, dok model korisnika u bazi ima polje

„Password_hash“, bilo je potrebno izračunati „hash[14]“ nad proslijeđenom konkretnom lozinkom kako bi se na taj način osigurala njena sigurnost te kako se u bazi ne bi pohranjivale konkretne lozinke. Za to je korištena „EnhancedHashPassword“ funkcija iz „BCrypt“ paketa (paket za hashiranje lozinki) koja kreira hash nad proslijeđenom lozinkom pomoću SHA384 funkcije (generira znakovni niz od 60 znakova). Zatim se poziva asinkrona funkcija za dodavanje korisnika iz „DAL“ projekta s proslijeđenom konekcijom, transakcijom i modelom korisnika koji je prilagođen za bazu. Kako se prilikom dodavanja korisnika, odabiru i njegove role, odnosno uloge te su time modeli korisnika i rola zapravo spojeni u jedan model, bilo je potrebno i pohraniti korisnikove role u tablicu koja pohranjuje korisnike i njihove role („UserRoles“). Tu se je prvo iteriralo s „foreach“ petljom kroz sve dodane role za proslijeđenoga korisnika te se svaka od njih dodala u tablicu „UserRoles“ tako da se asinkrono poziva „AddUserRoleAsync“ iz datoteke „UserRole.cs“ iz „DAL“ projekta s proslijeđenom konekcijom, transakcijom te novo kreiranim unosom gdje se za Id_role pohranjuje ID role koji se dohvaća preko njenoga naziva i funkcije „GetValueFromDescription“, a za Id_user se uzima ID proslijeđenoga korisnika kako bi se kreirao novi unos s korisnikom i njegovim rolama (taj dio se ne vidi na slici). Na kraju se asinkrono izvršava transakcija nad bazom koristeći već gotovu funkciju „CommitAsync“ kako bi se sve promjene preslikale u bazu podataka. Funkcija vraća varijablu „userZaBazu“ koja predstavlja obični model korisnika koji se onda u Web API-u pretvara u model za odgovore te se vraća kao odgovor na zahtjev.

```
public static class Users
{
    2 references
    public static async Task<Models.Users.User> AddUserAsync(Models.Users.Requests.RequestUser user)
    {
        using (DbConnection cn = await DAL.DB.GetConnectionAsync())
        {
            using (DbTransaction tr = cn.BeginTransaction())
            {
                var userZaBazu = user.Adapt<User>();

                // izracunaj hash
                userZaBazu.Password_hash = BCrypt.Net.BCrypt.EnhancedHashPassword(user.Password);

                await DAL.Users.AddUserAsync(cn, tr, userZaBazu);

                // spremit role
                foreach (string role_name in user.Role)
                {
                    await DAL.UserRoles.AddUserRoleAsync(cn, tr, new Models.UserRoles.UserRole() { Id_role
                };

                await tr.CommitAsync();
                return userZaBazu;
            }
        }
    }
```

Slika 19 - Prikaz funkcije za dodavanje korisnika u "ML" projektu

[14] „Hash“ – rezultat funkcije „hashiranja“ kojom se vrijednost pretvara u niz znakova (Wikipedia, Hash function, 2023.).

Funkcija za dohvaćanje jednoga korisnika (slika 20) je nešto jednostavnija te ona također vraća obični model korisnika iz baze podataka te prima samo ID proslijeđenoga korisnika iz zahtjeva. Na početku se ponovno definiraju konekcija na bazu i transakcija jer se ponovno pristupa više tablica i izvodi više operacija. Nakon toga se u varijablu „user“ sprema konkretni dohvaćeni korisnik iz baze podataka, pozivom asinkrone funkcije „GetUserAsync“ iz „DAL“ projekta s proslijeđenom konekcijom i ID-om korisnika. Ako korisnik ne postoji vraća se „null“, a ako postoji, u varijablu „user_roles“ se dohvaćaju sve njegove role iz tablice „UserRoles“ pozivom asinkrone funkcije „GetUserRoleAsync“ iz „UserRoles.cs“ datoteke iz „DAL“ projekta koja dohvaća sve role korisnika čiji se ID proslijedi u upitu. Zatim se s „foreach“ petljom iterira kroz sve njegove dohvaćene role te se jedna po jedna dodaju u listu rola pomoću „Add“ metode koja dodaje vrijednost na kraj liste i kreirane metode „GetEnumDescription“ koja dohvaća konkretnu tekstualnu vrijednost koja je pridružena određenom ID-u. Na taj način se dohvaćaju tekstualni nazivi rola preko numeričkih vrijednosti definiranih u nabrojanom tipu podataka („enum“) koji se zatim dodaju u odgovarajuću listu rola (koja je lista znakovnih nizova, odnosno naziva rola) u modelu korisnika kako bi prilikom dohvaćanja određenoga korisnika, odmah dohvatili i njegove role, odnosno uloge. Na kraju se ponovno asinkrono izvršava transakcija pomoću „CommitAsync“ metode i vraća se dohvaćeni korisnik koji će se vratiti kao odgovor na zahtjev u Web API-u. Na isti način je implementirana i funkcija za dohvaćanje korisnika preko korisničkoga imena, samo je kao parametar korišten znakovni niz korisničkoga imena.

```
public static async Task<Models.Users.User?> GetUserAsync(string Id_user)
{
    using (DbConnection cn = await DAL.DB.GetConnectionAsync())
    {
        using (DbTransaction tr = cn.BeginTransaction())
        {
            var user = await DAL.Users.GetUserAsync(cn, null, Id_user);
            if (user == null) return null;
            var user_roles = await DAL.UserRoles.GetUserRoleAsync(cn, tr, user.Id_user);

            // po user roles izvadi iz tablice roles i to stavi na user.roles
            foreach (var role in user_roles)
            {
                user.Role.Add(Helpers.GetEnumDescription(role.Id_role));
            }
            await tr.CommitAsync();
            return user;
        }
    }
}
```

Slika 20 - Prikaz funkcije za dohvaćanje jednoga korisnika iz "ML" projekta

Na vrlo sličan način je implementirana i funkcija za dohvaćanje svih korisnika („GetUsersAsync“), samo je kod nje korišten drugačiji povratni tip te time ova funkcija vraća „Task“, asinkronu operaciju koja vraća kolekciju modela običnoga korisnika iz baze podataka te nema parametara (slika 21). Opet se, kao i kod svih funkcija u „ML“ projektu, najprije

definira konekcija na bazu i započinje transakcija kako bi mogli izvršiti više operacija i pristupa različitim tablicama kroz jednu transakciju. Tu se najprije u varijablu „users“ sprema rezultat izvođenja asinkrone funkcije „GetUsersAsync“ iz „DAL“ projekta kako bi se dohvatili svi konkretni korisnici iz baze. Iteracijom po dohvaćenim korisnicima pomoću „foreach“ petlje, za svakoga korisnika su prvo pronađene njegove role pomoću „GetUserRoleAsync“ datoteke iz „UserRoles.cs“ datoteke iz „DAL“ projekta te je onda svaka rola dodana u listu rola toga korisnika na isti način kao i kod prethodne funkcije kako bi se prilikom ispisa korisnika, koji u bazi nema pohranjeno polje rola, ispisale sve njegove role iz tablice „UserRoles“. Na kraju se ponovno asinkrono izvršava transakcija pomoću „CommitAsync“ te se vraćaju svi dohvaćeni korisnici koji se vraćaju kao odgovor na zahtjev u Web API-u.

```
1 reference
public static async Task<IEnumerable<Models.Users.User>> GetUsersAsync()
{
    using (DbConnection cn = await DAL.DB.GetConnectionAsync())
    {
        using (DbTransaction tr = cn.BeginTransaction())
        {
            var users = await DAL.Users.GetUsersAsync(cn, null);
            // for each user i stavi u user.role = njegove role
            foreach (var User in users)
            {
                var User_Roles = await DAL.UserRoles.GetUserRoleAsync(cn, tr, User.Id_user);
                foreach (var Role in User_Roles)
                {
                    User.Role.Add(Helpers.GetEnumDescription(Role.Id_role));
                }
            }
            await tr.CommitAsync();
            return users;
        }


    }
}
```

Slika 21 - Prikaz funkcije za dohvaćanje svih korisnika iz "ML" projekta

Funkcija za brisanje korisnika u „ML“ projektu („RemoveUserAsync“) ima, kao i kod Web API-a, najjednostavniju implementaciju (slika 22). Ona i ovdje vraća samo „Task“, odnosno asinkronu operaciju koja ne vraća niti jedan tip podataka te prima samo ID korisnika kojega treba obrisati i kojega je primila od proslijeđenoga zahtjeva iz Web API-a. Tu se na početku opet dohvaća konekcija na bazu te se pokreće transakcija, a zatim se prvo brišu korisnikove role, odnosno uloge pozivom asinkrone funkcije „RemoveUserRolesAsync“ iz datoteke „UserRoles.cs“ iz „DAL“ projekta, kojoj se prosljeđuju konekcija, transakcija i ID korisnika čije role treba obrisati. Na taj način se prvo obrišu korisnikove role iz tablice „UserRoles“ jer inače nam ne bi dalo da obrišemo korisnika jer je njegov ID korišten kao vanjski ključ u drugoj tablici, a u PostgreSQL bazi je na svim tablicama postavljena opcija „RESTRICT“ kod brisanja vanjskih ključeva. Tada se poziva i funkcija za brisanje korisnika „RemoveUserAsync“ iz „DAL“ projekta koja briše konkretne korisnike iz baze prema proslijeđenome ID-u te se na kraju asinkrono izvršava cijela transakcija.

```
1 reference
public static async Task RemoveUserAsync(string Id_user)
{
    using (DbConnection cn = await DAL.DB.GetConnectionAsync())
    {
        using (DbTransaction tr = cn.BeginTransaction())
        {
            await UserRoles.RemoveUserRolesAsync(cn, tr, Id_user);
            await DAL.Users.RemoveUserAsync(cn, tr, Id_user);
            await tr.CommitAsync();
        }
    }
}
```

Slika 22 - Prikaz funkcije za brisanje korisnika iz "ML" projekta

Na slici 23 prikazana je funkcija za ažuriranje korisnika („UpdateUserAsync“) koja je i ovdje nešto složenija, kao i kod Web API-a. Ova funkcija vraća asinkronu operaciju („Task“) koja vraća obični model korisnika iz baze podataka te prima model korisnika za zahtjeve koji je proslijeđen u zahtjevu iz Web API-a. Tu se ponovno prvo dohvaća konekcija na bazu podataka te se pokreće transakcija. Tada se prvo u varijablu koja predstavlja objekt običnoga modela korisnika dohvaća konkretni korisnik iz baze podataka, pozivom asinkrone funkcije „GetUserAsync“ iz „DAL“ projekta te se provjerava ako on postoji (ukoliko ne postoji vraća se „null“). Zatim se provjerava je li korisnik ažurirao lozinku, odnosno ako polje „Password“ u proslijeđenome modelu korisnika nije prazno kako bi se za novu lozinku mogao izračunati novi „hash“. Ako je korisnik ažurirao lozinku, u polje „Password_hash“ običnoga modela korisnika za bazu se sprema novo „hashirana“ lozinka pomoću već korištene funkcije „EnhancedHashPassword“ koja prima lozinku koju treba „hashirati“ (odnosno lozinku modela korisnika za zahtjeve iz proslijeđenoga zahtjeva iz Web API-a). Na taj način se konkretna lozinka iz modela korisnika za zahtjeve sprema kao njena „hash“ verzija u polje „Password_hash“ modela korisnika za bazu podataka. Pritom se odmah poziva i funkcija „UpdateUserAsync“ iz „DAL“ projekta s proslijeđenom konekcijom, transakcijom te dobivenim ažuriranim korisnikom kako bi se podaci ažurirali i u samoj bazi podataka. Ukoliko korisnik nije ažurirao lozinku, poziva se zasebna funkcija „UpdateUserWithoutPassword“ iz „DAL“ projekta koja ažurira sva polja korisnika u bazi podataka osim lozinke. Kako bi se ažurirale sve role korisnika, prvo se poziva asinkrona funkcija „RemoveUserRolesAsync“ iz „UserRoles.cs“ datoteke iz „DAL“ projekta koja briše konkretne podatke iz tablice „UserRoles“, odnosno konkretne role za korisnika čiji je ID proslijeđen kao parametar i zatim se ponovno dodaju sve role tako da se s „foreach“ petljom iterira po listi korisnikovih rola te se jedna po jedna dodaju kao i kod dodavanja korisnika pomoću „AddUserRoleAsync“ funkcije. Na kraju se asinkrono izvršava transakcija te se vraća ažurirani model običnoga korisnika koji se transformira u model korisnika za odgovore i koji se vraća kao odgovor u Web API-u.

```
public static async Task<Models.Users.User?> UpdateUserAsync(Models.Users.Requests.RequestUpdateUser us
{
    using (DbConnection cn = await DAL.DB.GetConnectionAsync())
    {
        using (DbTransaction tr = cn.BeginTransaction())
        {
            Models.Users.User UserZaBazu = await DAL.Users.GetUserAsync(cn, tr, user.Id_user);
            if (UserZaBazu == null) return null;
            UserZaBazu = user.Adapt<User>();
            if (user.Password != null)
            {
                UserZaBazu.Password_hash = BCrypt.Net.BCrypt.EnhancedHashPassword(user.Password);
                await DAL.Users.UpdateUserAsync(cn, tr, UserZaBazu);
            }
            else
            {
                await DAL.Users.UpdateUserWithoutPasswordAsync(cn, tr, UserZaBazu);
            }


            await DAL.UserRoles.RemoveUserRolesAsync(cn, tr, user.Id_user);

            foreach (var Role_name in user.Role)
            {
                await DAL.UserRoles.AddUserRoleAsync(cn, tr, new Models.UserRoles.UserRole() { Id_role
            };
            await tr.CommitAsync();

            return UserZaBazu;

        }
    }
}
```

Slika 23 - Prikaz funkcije za ažuriranje korisnika iz "ML" projekta

Na vrlo sličan i puno jednostavniji način su kreirane i datoteke za kompanije, prakse i role (s time da se datoteka za role nije koristila, ali je puštena ukoliko bi bilo potrebno uvesti upravljanje rolama) jer se kod kompanija nije trebalo vršiti pretvorbe između modela za zahtjeve i za odgovore te se nije trebalo koristiti transakcije pošto se je uvijek pristupalo jednoj operaciji i tablici u bazi podataka pa se u većini funkcija samo nalaze pozivi na funkcije iz „DAL“ projekta u kojemu se vrši konkretni pristup podacima (slika 24).

```
5 references
public static class Companies
{
    1 reference
    public static async Task<Models.Companies.Company> AddCompanyAsync(Models.Companies.Company company)
    {
        using (DbConnection cn =await  DAL.DB.GetConnectionAsync())
        {
            return await DAL.Companies.AddCompanyAsync(cn, null, company);
        }
    }

    1 reference
    public static async Task<Models.Companies.Company> GetCompanyAsync(string Id_company)
    {
        using (DbConnection cn =await  DAL.DB.GetConnectionAsync())
        {
            return await DAL.Companies.GetCompanyAsync(cn, null, Id_company);
        }
    }

    1 reference
    public static async Task<IEnumerable<Models.Companies.Company>> GetCompaniesAsync()
    {
        using (DbConnection cn =await  DAL.DB.GetConnectionAsync())
        {
            return await DAL.Companies.GetCompaniesAsync(cn, null);
        }
    }
```

Slika 24 - Prikaz funkcija za upravljanje kompanijama iz "ML" projekta

U „ML“ projektu se, osim datoteka za pojedine tablice iz baze podataka, nalazi i mapa „Lib“ u koju su smještene pomoćne datoteke za dohvaćanje numeričkih preko tekstualnih vrijednosti ili tekstualnih preko numeričkih vrijednosti nabrojanog tipa podataka rola („enum“) kako bi se moglo dohvaćati nazive rola preko njihove pridružene vrijednosti, ali i obratno, pridružene vrijednosti preko naziva. Funkcije iz te datoteke su korištene nekoliko puta u prethodno prikazanim opisima koda, a osim te datoteke u „Lib“ mapi se nalazi i „TokenService.cs“ datoteka koja je potrebna za generiranje tokena prijavljenoga korisnika kako bi se mogle pratiti njegove dozvole.

4.4 Sloj pristupa podacima (DAL projekt)

Unutar „DAL“ projekta definiran je sloj pristupa podacima kreirane web aplikacije gdje se odvijaju konkretni upiti nad podacima koji su pohranjeni u tablicama baze podataka. Tu su kreirane pojedine funkcije koje su se pozivale iz „ML“ projekta kako bi se pristupilo konkretnim podacima u bazi podataka. Unutar „DAL“ projekta, kreirane su zasebne .cs datoteke za korisnike, kompanije, prakse, sesije, korisnikove role te role (s time da Roles.cs datoteka nije korištena, ali je puštena zbog mogućnosti uvođenja upravljanja rolama).

U nastavku su prikazani glavni dijelovi sloja pristupa podacima za korisnike. I kod ovih datoteka kreirana je statična javna klasa kako bi se njene metode mogle koristiti bez kreiranja instance korisnika. Unutar klasa su opet korištene asinkrone statične funkcije u kojima su definirani konkretni upiti nad bazom podataka, a na samome početku klasa kreiran je statični znakovni niz „TableName“ koji je služio kao pohrana za naziv tablice kako se on ne bi morao svaki puta koristiti u navodnicima, već je bio dostupan pomoću te varijable. Prva funkcija, koja je prikazana na slici 25, predstavlja asinkronu statičnu funkciju za dohvaćanje korisnika iz baze prema korisničkome imenu („GetUserUsernameAsync“). Ta funkcija vraća asinkronu operaciju („Task“) koja vraća obični model korisnika iz baze podataka te prima sučelje za bilo koju vrstu otvorene konekcije na bazu podataka („IDbConnection“), sučelje s metodama za rad s transakcijama na različitim bazama podataka te korisničko ime koje je proslijeđeno u zahtjevu. Na taj način se omogućuje da „DAL“ projekt bude funkcionalan s bilo kojom bazom podataka jer se koriste sučelja za različite baze podataka, a ne konkretne konekcije i transakcije. Za korištenje konkretnih upita u bazi podataka, korišten je paket „Dapper“ koji služi kao proširenje „IDbConnection“ sučelja. „Dapper“ predstavlja jednostavnu biblioteku za izvođenje SQL upita i pohranjivanje njihovih rezultata u objekte u programskim jezicima. Time se omogućuje da se izvede konkretni upit nad bazom podataka i da se njegov rezultat pohrani u polja modela korisnika. Za odabir određenih redaka iz baze podataka korištena je metoda „QueryFirstOrDefaultAsync“ iz „Dapper“ paketa koja je pozivana nad konekcijom i koja vraća prvi redak koji zadovoljava uvjet. Za izvođenje određenih upita za unos, ažuriranje i brisanje korištena je „ExccuteAsync“ metoda iz „Dapper“ paketa koja je pozivana nad konekcijom kako bi se asinkrono izvršila određena naredba, a za vraćanje više redaka je korištena metoda „QueryAsync“ iz „Dapper“ paketa koja je pozivana nad konekcijom kako bi izvršila asinkroni upit i vratila sve retke koji zadovoljavaju uvjet. Time prva funkcija vraća rezultat asinkrone metode „QueryFirstOrDefault“ s definiranim tipom modela običnoga

korisnika kako bi se rezultat upita pohranio u njegova polja. Sam upit je definiran unutar navodnika, a znak $ ispred početka upita omogućuje da se unutar znakovnoga niza dohvati vrijednost varijable „TableName“ pomoću vitičastih zagrada. Za vrijednost korisničkoga imena unutar upita je korištena notacija sa znakom @ koji označava da tu treba doći vrijednost parametra koji je definiran u nastavku s ključnom riječi „new“. Tom parametru je vrijednost postavljena na proslijeđeno korisničko ime iz zahtjeva kako bi se omogućilo da možemo dinamički poslati korisničko ime u upit i kako bi spriječili SQL injekciju[15] jer se time vrijednosti tretiraju kao podaci, a ne kao SQL kod. Na kraju se još i prosljeđuje transakcija unutar koje bi se upit trebao izvršiti. Druga funkcija predstavlja asinkronu funkciju za implementaciju upita za dodavanje korisnika u bazu podataka (slika 25). Ona vraća isto što i prethodna funkcija te većinom prima iste argumente, samo se ovdje ne prosljeđuje korisničko ime, nego cijeli model korisnika. Tu se koristi „ExecuteAsync“ metoda za asinkrono izvođenje određene naredbe, točnije upita koji unosi podatke u tablicu korisnika („INSERT“), a kao parametar je proslijeđen cijeli korisnik, kako bi se za vrijednosti u upitu (označene s @) uzele vrijednosti proslijeđenoga korisnika, i dodatno transakcija unutar koje bi se upit trebao izvršiti. Na kraju te funkcije se vraća korisnik koji je dodan. Funkcija za dohvaćanje korisnika prema ID-u je ista kao i za dohvaćanje korisnika prema korisničkome imenu samo je kao parametar korišten ID korisnika (slika 25).

```
public static class Users
{
    private static string TableName = "users";

    1 reference
    public static async Task<Models.Users.User> GetUserUsernameAsync(IDbConnection cn, IDbTransaction? tr, string Username)
    {
        return await cn.QueryFirstOrDefaultAsync<Models.Users.User>($"select * from {TableName} where Username = @Username",
            new { Username = Username }, tr);
    }

    1 reference
    public static async Task<Models.Users.User> AddUserAsync(IDbConnection cn, IDbTransaction? tr, Models.Users.User user)
    {
        await cn.ExecuteAsync($"insert into {TableName} (Id_user, Username, Email, Password_hash, Name, Surname, JMBAG) " +
            $"values (@Id_user, @Username, @Email, @Password_hash, @Name, @Surname, @JMBAG)", user, tr);
        return user;
    }

    2 references
    public static async Task<Models.Users.User> GetUserAsync(IDbConnection cn, IDbTransaction? tr, string Id_user)
    {
        return await cn.QueryFirstOrDefaultAsync<Models.Users.User>($"select * from {TableName} where Id_user=@Id_user",
            new { Id_user = Id_user }, tr);
    }
```

Slika 25 - Prikaz funkcija za dohvaćanje i dodavanje korisnika unutar "DAL" projekta

Zatim je definirana asinkrona funkcija koja vraća asinkronu operaciju („Task“) koja vraća kolekciju korisnika, odnosno vraća rezultat izvođenja upita koji dohvaća sve korisnike (slika 26). Tu su kao parametri korišteni samo konekcija na bazu i transakcija jer u upitu nije bilo potrebno prosljeđivati određene parametre unutar „WHERE“ klauzule jer se dohvaćaju svi zapisi. Unutar funkcije se asinkrono poziva metoda „QueryAsync“ iz „Dapper“ paketa kako bi se asinkrono izvršio definirani upit u navodnicima, a kao tip podataka te metode unutar šiljastih zagrada je postavljen obični model korisnika kako bi se rezultati upita mapirali

[15] SQL injekcija – tehnika izvođenja hakerskoga napada pomoću ubrizgavanja zlonamjernoga koda u znakovne nizove koji predstavljaju upite kako bi taj zlonamjerni kod stigao do baze podataka i izvršio neželjene radnje (w3schools, SQL Injection, 2023.).

s poljima u modelu korisnika. Za vrijednost parametra je proslijeđen „null“ te je dodana i transakcija unutar koje se upit treba izvršiti. Ostatak funkcija je nešto drugačiji od ostalih jer vraćaju asinkronu operaciju („Task“) koja vraća „bool“ vrijednost ovisno o tome je li se upit uspješno izvršio nad određenim poljima u bazi (slika 26). Tako je definirana asinkrona funkcija za brisanje korisnika koja uz konekciju i transakciju prima i ID korisnika za brisanje kao parametre. Tu se koristi metoda „ExecuteAsync“ za izvođenje određene asinkrone naredbe koja vraća broj redaka koje je upit zahvatio te će zato cijela funkcija za brisanje korisnika vratiti istinu jedino ako upit zahvati određene retke (to je postignuto ispitivanjem uvjeta je li rezultat izvođenja funkcije „ExecuteAsync“ veći od nule). Unutar funkcije se osim upita („DELETE“), nalazi i parametar za definiranje vrijednosti ID-a korisnika koji je proslijeđen u zahtjevu te transakcija unutar koje bi se upit trebao izvršiti. Na sličan način radi i funkcija za implementaciju upita za ažuriranje korisnika, samo ona, osim konekcije i transakcije prima cijeli model korisnika te izvršava „UPDATE“ upit pomoću „ExecuteAsync“ metode iz „Dapper“ paketa. Tu se kao parametar, čije se vrijednosti dinamički koriste unutar upita s oznakom @, prosljeđuje cijeli model korisnika te se nadodaje i transakcija unutar koje se upit treba izvršiti. Na isti način je definirana i funkcija za ažuriranje korisnika bez lozinke, samo tu nije postavljeno da se ažurira polje lozinke.

```
2 references
public static async Task<IEnumerable<Models.Users.User>> GetUsersAsync(IDbConnection cn, IDbTransaction? tr)
{
    return await cn.QueryAsync<Models.Users.User>($"select * from {TableName}", null, tr);
}

1 reference
public static async Task<bool> RemoveUserAsync(IDbConnection cn, IDbTransaction? tr, string Id_user)
{
    return (await cn.ExecuteAsync($"delete from {TableName} where Id_user=@Id_user", new { Id_user = Id_user }, tr)) > 0;
}

1 reference
public static async Task<bool> UpdateUserAsync(IDbConnection cn, IDbTransaction? tr, Models.Users.User user)
{
    return (await cn.ExecuteAsync($"update {TableName} set Id_user=@Id_user, Username=@Username, Email=@Email, " +
        $"Password_hash=@Password_hash, Name=@Name, Surname=@Surname, JMBAG=@JMBAG where Id_user=@Id_user", user, tr)) > 0;
}

1 reference
public static async Task<bool> UpdateUserWithoutPasswordAsync(IDbConnection cn, IDbTransaction? tr, Models.Users.User user)
{
    return (await cn.ExecuteAsync($"update {TableName} set Id_user=@Id_user, Username=@Username, Email=@Email, " +
        $"Name=@Name, Surname=@Surname, JMBAG=@JMBAG where Id_user=@Id_user", user, tr)) > 0;
}
```

Slika 26 - Prikaz funkcija za dohvaćanje svih korisnika, brisanje i ažuriranje korisnika unutar "DAL" projekta

Slično su kreirane i datoteke za pristupe podacima kompanija, praksi i korisnikovih rola. Osim tih datoteka, u „DAL“ projektu se nalazi i mapa „Migrations“ u kojoj se nalazi datoteka „Migrations.cs“ za izvođenje migracija u bazi podataka. Također, definirana je i datoteka „Parameters.cs“ koja predstavlja tablicu u bazi podataka koja čuva i ažurira trenutnu verziju baze podataka, ali i datoteka „DB.cs“ u kojoj se definira i otvara konekcija na bazu podataka te se upravlja verzijama baze podataka (slika 27). Ovdje se koristio „Connection string“ koji sadrži glavne podatke o konekciji na bazu, poput servera, naziva baze, lozinke i sl. koji je onda bio korišten za kreiranje nove konekcije i njeno otvaranje kako bi se omogućio pristup bazi podataka (otvaranje konekcije je ostvareno pomoću „OpenAsync“ metode te „NpgsqlConnection“ konekcije za povezivanje na PostgreSQL bazu podataka). Vrijednost

„Connection string-a“ se definira u „appsetting.Development.json“ datoteci, a postavlja u „Program.cs“ datoteci.

```
public class DB
{
    const int VERSION = 1;
    public static string CNString = null!;

    28 references
    public static async Task<NpgsqlConnection> GetConnectionAsync()
    {
        var conn = new NpgsqlConnection (CNString);
        await conn.OpenAsync();
        return conn;
    }

    1 reference
    public async static Task Init(string cnString)
    {
        CNString = cnString;

        using (NpgsqlConnection cn = await GetConnectionAsync())
        {
            int currentDBVersion = await Version(cn); // izvlacimo trenutnu verziju baze

            if (currentDBVersion < DB.VERSION)
            {
                await RunMigrations(currentDBVersion, DB.VERSION, cn);
            }
        }
    }
```

Slika 27 - Prikaz dijela "DB.cs" datoteke gdje je definirana konekcija na bazu podataka u PostgreSQL-u

4.5 Prijava (Sesije)

Za implementaciju prijave i praćenja dozvola prijavljenoga korisnika korištene su sesije i tokeni. Kod sesija se je, u mapi „Sessions“ u modelima, kreiralo tri različita modela. Osim obične sesije koja je odgovarala tablici u bazi podataka te koja je pohranjivala „RefreshToken“ i ID korisnika koji trenutno koristi sesiju, u „request“ i „response“ mapama su kreirana još dva modela. Model sesija za zahtjeve je sadržavao samo „RefreshToken“, a model sesija za odgovore je sadržavao „RefreshToken“ i „AccessToken“. Glavna ideja cijele prijave je bila da se, nakon što se korisnik prijavi, generira nasumični znakovni niz („RefreshToken“) te se on zajedno s ID-om korisnika pohrani u bazu podataka. Zatim se pomoću tog „RefreshToken-a“, koji jedinstveno identificira sesiju, dobije „AccessToken“ koji sadržava sve dozvole prijavljenoga korisnika te koji ima vrlo kratko vrijeme valjanosti. Nakon isteka toga kratkoga vremena valjanosti se, ponovno pomoću jedinstvenoga „RefreshTokena“ iz baze, dobiva novi „AccessToken“. Na taj način je postignuta veća sigurnost jer kada bi postojali samo „AccessToken-i“ sa svim dozvolama korisnika, koji bi trajali duže vrijeme ili sve dok se korisnik ne odjavi, moglo bi se dogoditi da zlonamjerni korisnici dođu u posjed tome „AccesToken-u“ i time dobiju sve dozvole određenoga korisnika. No, koristeći „RefreshToken“ i kratko trajanje „AccessToken-a“ to je onemogućeno. Kada se korisnik odjavi, njegova sesija, odnosno redak s „RefreshToken-om“ i njegovim ID-om se briše iz baze podataka.

Time su i kontroleri u Web API-u te središnji sloj i sloj pristupa podacima nešto drugačiji za sesije. Unutar kontrolera, kreirana je funkcija za prijavu, odnosno dodavanje sesije u bazu podataka („AddSessionAsync“) iznad koje je korišten atribut

„[AllowAnonymous]“ kako bi se svatko mogao prijaviti te je postavljeno da obrađuje „POST“ zahtjeve na rutu „/Login“ (slika 28). To je također asinkrona funkcija pa vraća „Task“ i ActionResult“ kako bi se mogli vratiti statusni kodovi te model sesije za odgovore s „RefreshToken-om“ i „AccessToken-om“, a prima poseban model korisnika koji ima samo korisničko ime i lozinku. Unutar funkcije se prvo dohvaća prijavljeni korisnik preko „GetUserUsernameAsync“ funkcije iz „ML“ projekta koja je i ranije objašnjena. Nakon toga se provjeravaju, korisničko ime i lozinka (ona se provjerava pomoću „EnhancedVerify“ funkcije iz „BCrypt“ paketa koja provjerava odgovara li zaprimljeni znakovni niz zaprimljenome „hash-u“) te se vraća statusni kod 400 za pogrešan zahtjev i prigodne poruke pogreške u slučaju da korisnik s pružanim korisničkim imenom ne postoji ili da lozinka nije točna (korišten „ErrorResponse“). Ukoliko korisnik unese ispravno korisničko ime i lozinku, funkcija vraća te poziva asinkronu funkciju „AddSessionAsync“ iz „ML“ projekta kojoj su proslijeđeni model korisnika za prijavu i obični model korisnika.

```
[AllowAnonymous]
[HttpPost("Login")]
0 references
public async Task<ActionResult<Models.Sessions.Responses.ResponseSession>> AddSessionAsync(Models.Users.Requests.SessionRequestUser sessionRequestUser)
{

    var user = await ML.Users.GetUserUsernameAsync(sessionRequestUser.Username);



    if (user == null) return StatusCode(StatusCodes.Status400BadRequest,
        new ErrorResponse { ErrorMessage = "Nije pronađen korisnik sa ovim podacima. Molimo pokušajte ponovo." });

    bool PasswordValid = BCrypt.Net.BCrypt.EnhancedVerify(sessionRequestUser.Password, user.Password_hash);
    if(!PasswordValid) return StatusCode(StatusCodes.Status400BadRequest,
        new ErrorResponse { ErrorMessage = "Nije pronađen korisnik sa ovim podacima. Molimo pokušajte ponovo." });

    return await ML.Sessions.AddSessionAsync(sessionRequestUser, user);
}
```

Slika 28 - Prikaz funkcije za prijavu, odnosno dodavanje sesije u kontroleru sesija

Asinkrone funkcije za dobivanje novoga „AccessToken-a“ i za brisanje sesije (slika 29) su također dopuštene svima te obrađuju „POST“ zahtjeve na definirane rute. Funkcija za dobivanje novoga „AccessToken-a“ („CheckRTAsycn“) ima istu definiciju kao i funkcija za dodavanje sesije, samo kao argument prima model sesije za zahtjeve koji ima samo „RefreshToken“. Zatim se pomoću tog „RefreshToken-a“ dohvaća određena sesija pomoću „GetSessionRefreshTokenAsync“ funkcije iz „ML“ projekta kako bi se prvo provjerilo postoji li uopće ta sesija te ako postoji, poziva se funkcija za dobivanje novoga „AccessToken-a“ iz „ML“ projekta kojoj se prosljeđuje obični model sesije s „RefreshToken-om“ i ID-om prijavljenoga korisnika. Ukoliko sesija ne postoji, ponovno se vraća prigodna poruka pogreške pomoću „ErrorResponse“. Funkcija za brisanje sesije je nešto jednostavnija jer nema povratnoga tipa već vraća samo asinkronu operaciju „Task“ te prima model sesije za zahtjeve gdje je korišten „RefreshToken“ za poziv funkcije „RemoveSessionAsync“ iz „ML“ projekta.

```
[AllowAnonymous]
[HttpPost("refresh-token")]
0 references
public async Task<ActionResult<Models.Sessions.Responses.ResponseSession>> CheckRTAsync(Models.Sessions.Requests.RequestSession session)
{
    var CheckRTSession=await ML.Sessions.GetSessionRefreshTokenAsync(session.RefreshToken);
    if (CheckRTSession != null)
    {
        return await ML.Sessions.GetNewAccessTokenAsync(CheckRTSession);
    }
    else
    {
        return StatusCode(StatusCodes.Status400BadRequest,
            new ErrorResponse { ErrorMessage = "Nije pronađen korisnik sa ovim podacima. Molimo pokušajte ponovo." });
    }
}
[AllowAnonymous]
[HttpPost("Logout")]
0 references
public async Task RemoveSessionAsync(Models.Sessions.Requests.RequestSession session)
{
    await ML.Sessions.RemoveSessionAsync(session.RefreshToken);
}
```

Slika 29 - Prikaz funkcija za dobivanje novoga "AccessToken-a" i brisanje sesije u kontroleru sesija

Unutar „ML“ projekta za sesije je također korištena nešto drugačija poslovna logika. Na početku ove datoteke je najprije kreirana nova instanca token servisa s proslijeđenim glavnim parametrima čije su vrijednosti definirane u „appsettings.Development.json“ datoteci, a postavljene u „Program.cs“ datoteci. Ta instanca zapravo predstavlja .cs datoteku gdje je definirana logika kreacije tokena. Na slici 30 prikazana je asinkrona funkcija za dodavanje sesije u „ML“ projektu („AddSessionAsync“) koja se poziva na kraju funkcije za dodavanje sesije u Web API-u te koja najprije dohvaća konekciju na bazu podataka. Nakon toga korištena je funkcija za generiranje nasumičnoga znakovnoga niza „StringGenerator“ koja služi za definiranje „RefreshToken-a“. Tada se kreira nova instanca običnoga modela sesija u koju se spremaju generirani „RefreshToken“ koji služe za identificiranje sesije te ID korisnika koji je proslijeđen kroz parametar funkcije. Zatim se u varijablu „AccessToken“ sprema generirani „AccessToken“ pomoću funkcije „CreateToken“ iz token servisa koji sadrži sve dozvole prijavljenoga korisnika. Tu se onda kreira nova instanca modela sesije za odgovore, kojoj se prosljeđuju generirani „RefreshToken“ i generirani „AccessToken“, koja se onda i vraća kao rezultat funkcije, a time i kao odgovor na zahtjev iz Web API-a. Novo-kreirani obični model sesije se sprema u bazu, pozivom funkcije „AddSessionAsync“ iz „DAL“ projekta. Na taj način se, prilikom prijave korisnika, generira novi „RefreshToken“ koji služi za identificiranje sesije te koji se, zajedno s ID-om korisnika, sprema u bazu podataka. Također, odmah se i generira prvi „AccessToken“ za prijavljenoga korisnika kako bi on mogao koristiti svoje dostupne dozvole.

```
public static async Task<Models.Sessions.Responses.ResponseSession>
    1 reference
    AddSessionAsync(Models.Users.Requests.SessionRequestUser sessionUser, Models.Users.User user)
{
    using (DbConnection cn =await  DAL.DB.GetConnectionAsync())
    {

        const string characters = "ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789";

        string StringGenerator(int minLen, int maxLen)
        {
            var random = new Random();
            int len = random.Next(minLen, maxLen + 1);

            char[] chars = new char[len];
            for (int i = 0; i < len; i++)
            {
                chars[i] = characters[random.Next(characters.Length)];
            }
            return new string(chars);
        }


        string RefreshToken = StringGenerator(40, 100);
        Models.Sessions.Session sesija = new Models.Sessions.Session()
        {
            RefreshToken = RefreshToken,
            Id_user = user.Id_user,
        };
        var AccessToken = TokenService.CreateToken(user);
        Models.Sessions.Responses.ResponseSession ResponseSesija = new Models.Sessions.Responses.ResponseSession()
        {
            RefreshToken = RefreshToken,
            AccessToken = AccessToken
        };
        await DAL.Sessions.AddSessionAsync(cn, null, sesija);
        return ResponseSesija;
```

Slika 30 - Prikaz funkcije za dodavanje sesije unutar "ML" projekta

Kako bi korisnik, nakon isteka dobivenoga „AccessToken-a“, mogao dobiti novi, korištena je „GetNewAccessTokenAsync“ asinkrona funkcija (slika 31) koja se poziva unutar Web API-a te vraća model sesije za odgovore, a prima obični model sesije s „RefreshToken-om“ i ID-om prijavljenoga korisnika. Tu se na početku, nakon otvaranja konekcije, najprije dohvaća cijeli model prijavljenoga korisnika pomoću njegovoga ID-a i funkcije „GetUserAsync“ iz „Users.cs“ datoteke iz „ML“ projekta. To je potrebno kako bi generirali novi „AccessToken“ s funkcijom „CreateToken“ koja prima cijeli model korisnika. Tu se onda ponovno kreira nova instanca modela sesije za odgovore kojoj se, kao vrijednosti, postavljaju postojeći jedinstveni „RefreshToken“ i novi „AccessToken“ te se ta nova instanca modela sesije za odgovore vraća kao rezultat, odnosno time i kao odgovor na zahtjev u Web API-u. Tu su još kreirane i dvije funkcije za dohvaćanje sesije prema jedinstvenome „RefreshToken-u“ kako bi se moglo provjeriti postoji li sesija te funkcija za brisanje sesije preko jedinstvenoga „RefreshToken-a“ koji je proslijeđen u zahtjevu. One samo otvaraju konekciju na bazu te vraćaju ili pozivaju rezultate odgovarajućih funkcija iz „DAL“ projekta.

```
public static async Task<Models.Sessions.Responses.ResponseSession> GetNewAccessTokenAsync(Models.Sessions.Session session)
{
    using (DbConnection cn = await DAL.DB.GetConnectionAsync())
    {
        var user = await ML.Users.GetUserAsync(session.Id_user);

        var AccessToken = TokenService.CreateToken(user!);
        Models.Sessions.Responses.ResponseSession ResponseSesija = new Models.Sessions.Responses.ResponseSession()
        {
            RefreshToken = session.RefreshToken,
            AccessToken = AccessToken
        };
        return ResponseSesija;
    }
}
1 reference
public static async Task<Models.Sessions.Session> GetSessionRefreshTokenAsync(string RefreshToken)
{
    using (DbConnection cn = await DAL.DB.GetConnectionAsync())
    {
        return await DAL.Sessions.GetSessionRefreshToken(cn, null, RefreshToken);
    }
}
1 reference
public static async Task RemoveSessionAsync(string RefreshToken)
{
    using (DbConnection cn =await  DAL.DB.GetConnectionAsync())
    {
        await DAL.Sessions.RemoveSessionAsync(cn, null, RefreshToken);
    }
}
```

Slika 31 - Prikaz funkcije za dobivanje novoga "AccessToken-a" unutar "ML" projekta

Unutar „DAL“ projekta za sesije su se, kao i kod ostalih implementacija sloja pristupa podacima, kreirale funkcije za dodavanje, dohvaćanje i brisanje sesija (slika 32) pomoću već objašnjenih metoda i upita te proslijeđenih modela i vrijednosti sesija zaprimljenih iz „ML-a“.

```
public static class Sessions
{
    private static string TableName = "sessions";

    1 reference
    public static async Task<Models.Sessions.Session> AddSessionAsync(IDbConnection cn, IDbTransaction? tr, Models.Sessions.Session session)
    {
        await cn.ExecuteAsync($"insert into {TableName} (RefreshToken, Id_user) values (@RefreshToken, @Id_user)", session, tr);
        return session;
    }

    1 reference
    public static async Task<Models.Sessions.Session> GetSessionRefreshToken(IDbConnection cn, IDbTransaction? tr, string RefreshToken)
    {
        return await cn.QueryFirstOrDefaultAsync<Models.Sessions.Session>
            ($"select * from {TableName} where RefreshToken=@RefreshToken", new { RefreshToken = RefreshToken }, tr);
    }

    0 references
    public static async Task<bool> UpdateSessionAsync(IDbConnection cn, IDbTransaction? tr, Models.Sessions.Session session)
    {
        return (await cn.ExecuteAsync($"update {TableName} set  RefreshToken=@RefreshToken, Id_user=@Id_user", session, tr)) > 0;
    }

    1 reference
    public static async Task<bool> RemoveSessionAsync(IDbConnection cn, IDbTransaction? tr, string RefreshToken)
    {
        return (await cn.ExecuteAsync($"delete from {TableName} where RefreshToken=@RefreshToken", new { RefreshToken = RefreshToken }, tr)) > 0;
    }
}
```

Slika 32 - Prikaz funkcija za upravljanje sesijama unutar "DAL" projekta

Još jedna specifičnost za sesije je i generiranje „AccessToken-a“ koje je implementirano unutar „ML“ projekta u „Lib“ mapi gdje je kreirana „TokenService.cs“ datoteka. Unutar nje je definirana implementacija „AccessToken-a“ koji su zapravo JWT[16] tokeni. JWT tokeni se često koriste za autentifikaciju i autorizaciju jer predstavljaju vjerodajnice koje ograničavaju i determiniraju prava pristupa. Time se definira jedinstveni znakovni niz, odnosno sam token koji se dobije pomoću unaprijed određenih zaglavlja, korisnikovih informacija („claims“) i jedinstvenoga potpisa koji se onda može koristiti za identifikaciju korisnika i njegovih dozvola (Auth0, 2023.).

U datoteci „TokenService.cs“ prvo je definirana glavna funkcija za kreiranje JWT tokena za prosljeđenoga korisnika (slika 33). Za to su korištene već gotove biblioteke te se ovdje najprije deifinira konstruktor koji postavlja glavne vrijednosti tokena (tajni ključ, vrijeme valjanosti i sl. koje su definirane u „appsetting.Development.json“ datoteci te postavljene u „Program.cs datoteci“). U „CreateToken“ metodi, kojoj je prosljeđen korisnik koji se trenutno prijavljuje, definirana je glavna kreacija tokena te ona vraća znakovni niz koji predstavlja „AccessToken“ koji je dobiven pomoću „JwtSecurityTokenHandler-a“ i proslijeđenih parametara. Unutar nje se u varijablu „token“ sprema rezultat izvođenja „CreateJwtToken“ funkcije koja prima sve potrebne parametre za kreiranje „AccessToken-a“ poput korisnikovih zahjteva („claims“), jedinstvenog potpisa, vremena valjanosti te pomoću njih kreira novi JWT token uz dodatak „issuer“ i „audience“ parametara „JwtSecurityToken“ metode. Oni definiraju onoga tko izdaje i onoga za koga je namijenjen token i te vrijednosti često moraju biti jednake onima koje su postavljene prilikom konfiguracije kreacije JWT tokena (to je definirano u „Program.cs“ datoteci Web API-a gdje su definirane glavne konfiguracije autentifikacije, povezivanja na bazu i sl.).

```
public TokenService(string secretKey, string Issuer, string Audience, int ExpirationInSeconds)
{
    this.secretKey = secretKey;
    this.issuer = Issuer;
    this.audience = Audience;
    this.ExpirationInSeconds = ExpirationInSeconds;
}

2 references
public string CreateToken(Models.Users.User user)
{
    var expiration = DateTime.UtcNow.AddSeconds(this.ExpirationInSeconds);
    var token = CreateJwtToken(
        CreateClaims(user),
        CreateSigningCredentials(),
        expiration
    );
    var tokenHandler = new JwtSecurityTokenHandler();
    return tokenHandler.WriteToken(token);
}

1 reference
private JwtSecurityToken CreateJwtToken(List<Claim> claims, SigningCredentials credentials, DateTime expiration) =>
    new JwtSecurityToken(this.issuer, this.audience, claims, expires: expiration, signingCredentials: credentials);
```

Slika 33 - Prikaz glavne funkcije za generiranje JWT tokena

Za kreiranje liste zahtjeva („Claims“) koje se koriste prilikom kreacije JWT tokena korištena je funkcija „CreateClaims“ (slika 34) koja također prima model običnoga korisnika koji se prijavljuje u aplikaciju. Tu se stvara nova lista zahtjeva („Claims“) koji se onda koriste

[16] JWT – JSON Web Token (Auth0, 2023.).

prilikom kreacije tokena. Među njima spadaju i već definirani subjekt tokena, jedinstveni identifikator tokena te vrijeme nastajanja tokena, ali i oni ovisni o korisniku poput njegovoga ID-a, korisničkoga imena, e-maila te njegovih rola koje su, jedna po jedna, dodane u listu zahtjeva s „AddRange“ metodom koja služi za dodavanje elemenata na kraj liste. Na taj način nastaju tokeni koji su jedinstveni za svakoga korisnika te uzimaju u obzir njegove glavne podatke, uključujući i role kako bi se mogla provjeravati njegova prava pristupa.

```
private List<Claim> CreateClaims(Models.Users.User user)
{
    try
    {
        var claims = new List<Claim>
        {
             new Claim(JwtRegisteredClaimNames.Sub, this.issuer),
            new Claim(JwtRegisteredClaimNames.Jti, Guid.NewGuid().ToString()),
            new Claim(JwtRegisteredClaimNames.Iat, DateTime.UtcNow.ToString(CultureInfo.InvariantCulture)),
            new Claim("user_id", user.Id_user),
            new Claim("username", user.Username),
            new Claim("email", user.Email),
        };
        claims.AddRange(user.Role.Select(role => new Claim(ClaimTypes.Role, role)));
        return claims;
    }
    catch (Exception e)
    {
        Console.WriteLine(e);
        throw;
    }
}
```

Slika 34 - Prikaz funkcije za generiranje korisnikovih informacija ("Claims")

Na kraju datoteke se još definira i tajni ključ koji se koristi za potpisivanje i provjeru valjanosti tokena kako bismo bili sigurni da je korišteni token kreiran pomoću definiranoga tajnoga ključa. Na kraju se sva tri spomenuta dijela (zaglavlja, zahtjevi i tajni ključ) kodiraju s „base64[17]“ standardom i spajaju u jedan jedinstveni niz znakova, odnosno JWT token koji se onda može koristiti za autentifikaciju i praćenje dozvola određenih korisnika.

4.6 React (TypeScript)

Kod „React“ aplikacije, odnosno „ClientApp“ projekta su datoteke također bile raspodijeljene u mape kako bi projekt bio bolje organiziran. Tako su sve datoteke i mape dodane unutar „src“ mape gdje se definira glavni kod „React“ aplikacije pa time razlikujemo „api“ mapu gdje je definirana glavna konekcija na Web API, „components“ mapu gdje su definirane glavne komponente, odnosno pojedini dijelovi korisničkoga sučelja, „lib“ mapu s pomoćnim datotekama, „stores“ mapu gdje su definirana skladišta za upravljanje stanjem i podacima te „types“ mapu koja je vrlo slična modelima te koja služi za definiranje sučelja s glavnim atributima i njihovim tipovima podataka za svaki model. Također tu su većinom korištene .ts datoteke koje predstavljaju obične „TypeScript“ datoteke te .tsx datoteke koje predstavljaju „TypeScript“ datoteke koje koriste i „JSX“ sintaksu te time omogućuju definiranje pojedinih, konkretnih dijelova korisničkoga sučelja.

[17] Base64 – shema kodiranja koja pretvara binarne podatke u znakovnu reprezentaciju (Wikipedia, base64, 2023.).

4.6.1 Tipovi i „api“ mapa

Unutar „types“ mape, kreirane su zasebne „ts“ datoteke za svaki model. Time su i ovdje datoteke podijeljene na „request“ i „response“ mape te su u njima definirana sučelja s glavnim tipovima podataka. Tako je recimo za model korisnika za zahtjeve kreirano sučelje „RequestUpdateUser“, s njegovim glavnim atributima i tipovima podataka, koje je izvezeno s ključnom riječi „export“ kako bi se moglo koristiti kroz različite dijelove projekta (slika 35). Na taj način su kreirana i ostala sučelja.

```
export interface RequestUpdateUser {
    id_user: string;
    username: string;
    email: string;
    password: string;
    name: string;
    surname: string;
    is_admin: boolean;
    jmbag: string | null;
    role: string[];
}
```

Slika 35 - Prikaz sučelja za model korisnika za zahtjeve u "TypeScript-u"

U „api“ mapi su kreirane dvije obične „TypeScript“ datoteke, „agent.ts“ i „api.ts“ gdje su definirane glavne poveznice sa samim Web API-em, automatskim generiranjem „AccessToken-a“ te s obradom zahtjeva i odgovora. U „agent.ts“ (slika 36) korištena je „axios“ biblioteka koja omogućuje obradu HTTP zahtjeva sa strane klijenta na Web API. Tako je prvo kreirana varijabla „client“ u koju je pohranjena „axios“ instanca klijenta s definiranim temeljnim URL-om Web API-a na koji se šalju zahtjevi. Zatim se pomoću toga klijenta presreću zahtjevi i odgovori kako bi se dodavali i obnavljali „AccessToken-i“ u zahtjevima, ali i kako bi se upravljalo odgovorima servera i specifičnim pogreškama.

```
const client = axios.create({ baseURL: "https://localhost:7078", withCredentials: true });

client.interceptors.request.use(async (config: InternalAxiosRequestConfig) => {
    const token = store.currentUserStore.accessToken;
    if (token && config.headers != null) {
        if (isExpired(token) && !store.currentUserStore.isFetchingToken) {
            await FetchNewToken();
        }
        config.headers.authorization = `Bearer ${store.currentUserStore.accessToken}`
    }
    return config;
});

client.interceptors.response.use(async (response: any) => {
    try {
        //await sleep(1000);
        handleDates(response.data); // pretvaramo sve dates u js date objecte
        return response;
    } catch (e) {
        return Promise.reject(e);
    }
}, async (error: AxiosError) => {
    //await sleep(1000);
    if (error.response == null) {
        message.error({ content: "Server je trenutno nedostupan", duration: 4 });
        throw error;
    }
    const { data, status, config } = error.response as AxiosResponse;
    switch (status) {
        case 400:
            const serverError400 = (data as ErrorResponse);
            message.error({ content: serverError400.errorMessage, duration: 4 });
            break;
```

Slika 36 - Prikaz kreiranja klijenta za komunikaciju s Web API-em

Tu je onda još kreiran i „agent“ koji definira četiri generalizirane funkcije za različite HTTP zahtjeve gdje koristi kreiranoga klijenta za dohvaćanje, kreiranje, ažuriranje ili brisanje podataka (slika 37). Na kraju se taj agent izvozi kako bi se mogao koristiti u „api.ts“ datoteci.

```
const responsebody = <T>(response: AxiosResponse<T>) => response.data

const agent = {
    get: <T>(url: string) => client.get<T>(url).then(responsebody),
    post: <T>(url: string, data: any) => client.post<T>(url, data, { headers: { 'Content-Type': 'application/json; charset=utf-8' } })
        .then(responsebody),
    put: <T>(url: string, data: any) => client.put<T>(url, data, { headers: { 'Content-Type': 'application/json; charset=utf-8' } }),
    delete: <T>(url: string, data?: any) => client.delete<T>(url, { headers: { 'Content-Type': 'application/json; charset=utf-8' }, data })
        .then(responsebody),
}

export default agent
```

Slika 37 - Prikaz agenta koji primjenjuje glavne funkcije klijenta za obradu HTTP zahtjeva

Unutar „api.ts“ datoteke, kreirani su objekti za sesije, korisnike, kompanije i prakse u kojima su definirane funkcije koje implementiraju funkcije iz agenta samo su ovdje navedeni konkretni tipovi podataka (sučelja koja su ranije objašnjena) i URL-ovi na koje treba slati zahtjeve u Web API. Na slici 38 je prikazan primjer za korisnike.

```
const users = {
    GetAll: () => agent.get<ResponseUser[]>("/users"),
    GetUser: (id_user: string) => agent.get<ResponseUser>(`/users/${id_user}`),
    AddUser: (req: RequestUser) => agent.post<ResponseUser>('/users', req),
    UpdateUser: (id_user: string, req: RequestUpdateUser) => agent.put<ResponseUser>(`/users/${id_user}`, req),
    RemoveUser: (id_user: string) => agent.delete<null>(`/users/${id_user}`)
}
```

Slika 38 - Prikaz implementacije različitih HTTP zahtjeva za korisnike u React aplikaciji

4.6.2 Skladišta („stores“ mapa)

Unutar „stores“ mape kreirane su obične .ts datoteke za implementaciju skladišta korisnika, kompanija, praksi, prijavljenoga korisnika te same aplikacije. Skladišta služe za lakše upravljanje stanjima i samim podacima koji se trebaju mijenjati i prikazivati na korisničkome sučelju. Za sva skladišta je korištena „MobX“ JavaScript biblioteka za upravljanje stanjem koja pruža mogućnosti kreiranja uočljivih („Observable“) podataka kako bismo mogli pratiti i ažurirati njihove promjene, akcija („Actions“) koje predstavljaju funkcije koje ažuriraju uočljive podatke te reakcija („Reactions“) za automatsko pokretanje prilikom promjene ovisnih podataka (MobX, 2023.). Reakcije su korištene za upravljanje „AccessToken-ima“ prilikom njihove automatske promjene. Osim skladišta aplikacije gdje se upravlja njenim pojedinim dijelovima te skladišta prijavljenoga korisnika gdje su implementirane funkcije za dobivanje njegovih informacija te za upravljanje samom prijavom, odjavom i generiranjem JWT tokena, kreirana su i skladišta za korisnike, prakse i kompanije koja su implementirana na vrlo slične načine. Time su u nastavku objašnjeni samo pojedini dijelovi skladišta za korisnike koji su onda na isti način implementirani i za kompanije i prakse.

Na početku je definirana klasa „KorisniciStore“ sa svim potrebnim varijablama (slika 39). Tu je korišten „korisniciRegistry“ s „Map“ tipom podataka koji pohranjuje parove ključeva i vrijednosti. Taj registar je zapravo korišten za pohranu podataka koji dolaze iz Web API-a. Osim definicije svih varijabli, kreiran je i konstruktor koji koristi metodu „makeAutoObservable“ iz „MobX“ biblioteke kako bi sve varijable postale uočljive

(„Observable“) i kako bi se sve funkcije automatski prepoznale kao akcije te se time eliminira potreba za ručnim korištenjem ključnih riječi „observable“ i „action“ prilikom definiranja varijabli i funkcija.

```
export default class KorisniciStore {
    korisniciRegistry = new Map<string, ResponseUser>();
    fetchedKorisnici: boolean = false;
    selectedKorisnik: ResponseUser | undefined = undefined;
    isloading: boolean = false;
    isFetchingUser: boolean = false;


    1 reference
    constructor() {
        makeAutoObservable(this);
    }
```

Slika 39 - Prikaz početka skladišta za korisnike

Tada su kreirane određene akcije, odnosno funkcije za upravljanje samim korisnicima unutar registra gdje se spremaju rezultati izvođenja zahtjeva iz Web API-a pa su tako kreirane funkcije za dohvaćanje svih korisnika iz registra, postavljanje odabranoga korisnika, stvaranje novoga korisnika koja postavlja odabranoga korisnika na nedefinirano i prosljeđuje na rutu za stvaranje novoga korisnika, za dodavanje, ažuriranje i brisanje korisnika koje koriste metode dostupne nad „Map“ tipom podataka za dodavanje ili brisanje iz registra („set“, „delete“) i slične funkcije. Na slici 40 su prikazane samo neke.

```
noviKorisnik() {
    this.setSelectedKorisnik(undefined);
    router.navigate("/korisnici/novi");
}

private setirajKorisnike(korisnici: ResponseUser[]) {
    this.korisniciRegistry.clear();
    korisnici.forEach((korisnik) => {
        this.korisniciRegistry.set(korisnik.id_user, korisnik);
    });
}
```

```
private dodajKorisnika(korisnik: ResponseUser) {
    this.korisniciRegistry.set(korisnik.id_user, korisnik);
}

private updateKorisnika(korisnik: ResponseUser) {
    this.korisniciRegistry.set(korisnik.id_user, korisnik);
}

private deleteKorisnika(id_user: string) {
    this.korisniciRegistry.delete(id_user);
}
```

Slika 40 - Prikaz akcija unutar skladišta za korisnike

Zatim su kreirane konkretne asinkrone funkcije, odnosno akcije koje koriste prethodno prikazane akcije za upravljanje podacima u registru te pozivaju određene metode iz „api.ts“ datoteke kako bi se zahtjevi stvarno proslijedili samome Web API-u. Tako su kreirane akcije za dohvaćanje svih korisnika, odabir jednoga korisnika, dodavanje korisnika, ažuriranje i brisanje korisnika te su u nastavku prikazane samo neke od njih jer su sve bazirane na istim temeljima. Unutar svih akcija su korišteni try-catch blokovi kako bi se mogle hvatati pogreške i vraćati prigodni odgovori te je kod poziva akcija i varijabli korištena ključna riječ „this“ kako bi se one koristile s ispravnom trenutnom instancom skladišta ili varijable.

Na slici 41 prikazana je asinkrona akcija za dohvaćanje svih korisnika („GetAll“) gdje se unutar „try“ bloka pozivaju prethodno kreirane akcije za postavljanje učitavanja i za postavljanje korisnika. Ovdje se u varijablu „data“ asinkrono čeka rezultat poziva konkretnog zahtjeva za dohvaćanje svih korisnika iz Web API-a te se zatim ti korisnici prosljeđuju akciji „setirajKorisnike“ koja postavlja korisnike u registar. U slučaju pogreške, ona se ispisuje i na kraju se zaustavlja stanje učitavanja.

```
async GetAll() {
    try {
        this.setIsLoading(true);
        this.setirajKorisnike([]);
        const data = await api.users.GetAll();
        this.setirajKorisnike(data);
        this.setFetchedKorisnici(true);
    } catch (e) {
        console.log(e);
    } finally {
        this.setIsLoading(false);
    }
};
```

Slika 41 - Prikaz funkcije za dohvaćanje svih korisnika u skladištu korisnika

Tada je i implementirana asinkrona funkcija za dohvaćanje jednoga korisnika („SelectKorisnik“) koja kao argument prima ID korisnika (slika 42). Tu se prvo unutar „try“ bloka, u varijablu korisnik, pronalazi korisnik iz niza „korisnici“ koji je nastao pretvaranjem registra korisnika u niz s „Array.from“ metodom. Tu se koristi „find“ metoda i lambda funkcija kojom se prolazi preko svih korisnika dok se ne pronađe onaj s istim proslijeđenim ID-om. Ukoliko se korisnik ne može pronaći unutar niza, mijenja se stanje akcije „setIsFetchingUser“ te se poziva konkretna metoda Web API-a za dohvaćanje korisnika prema ID-u iz baze podataka. Zatim se dohvaćeni korisnik prosljeđuje u metodu „setSelectedKorisnik“ kako bi se on odabrao. U slučaju pogreške ona se ispisuje, a nakon dohvaćanja korisnika mijenja se stanje akcije „setIsFetchingUser“.

```
async SelectKorisnik(id_user: string) {
    try {
        let korisnik = this.korisnici.find((x) => x.id_user === id_user);
        if (!korisnik) {
            this.setIsFetchingUser(true);
            korisnik = await api.users.GetUser(id_user);
        }
        this.setSelectedKorisnik(korisnik);
    } catch (e) {
        console.log(e);
    } finally {
        this.setIsFetchingUser(false);
    }
}
```

Slika 42 - Prikaz funkcije za dohvaćanje korisnika u skladištu korisnika

Na slici 43 prikazana je funkcija za dodavanje korisnika („DodajKorisnika“) koja je nešto drugačija jer, osim što prima cijeloga korisnika za zahtjeve, vraća „Promise“ koji u JavaScript programskom jeziku označava isto što i „Task“ u C# programskom jeziku, odnosno označava asinkronu operaciju (njezin rezultat). U ovom slučaju akcija vraća „bool“ vrijednost ovisno o tome je li korisnik uspješno dodan. Tu se prvo proslijeđenom korisniku postavlja ID na „UUID[18]“ pomoću „uuid“ metode kako bi ID korisnika bio jedinstven. Postavlja se stanje akcije „setIsLoading“ na „true“ te se u varijablu „noviKorisnik“ asinkrono čeka rezultat dodavanja korisnika unutar Web API-a. Ukoliko je novi korisnik uspješno dodan, poziva se akcija za dodavanje novoga korisnika u registar i vraća se „true“, a inače se vraća „false“. U slučaju pogreške, ona se ovdje „baca“ kako bi se

[18] „UUID“ – Universally Unique Identifier, jedinstveni znakovni niz od 36 znakova koji služi za jedinstvenu identifikaciju (Gillis, 2021.).

konkretna pogreška mogla obraditi od strane definiranog globalnog upravljanja pogreškama i na kraju se mijenja stanje akcije „setIsLoading“.

```
async DodajKorisnika(korisnik: RequestUser): Promise<boolean> {
    try {
        korisnik.id_user = uuid();
        this.setIsLoading(true);
        const noviKorisnik = await api.users.AddUser(korisnik);
        if (noviKorisnik) {
            this.dodajKorisnika(noviKorisnik);
            return true;
        }
        return false;
    } catch (e) {
        throw e;
    } finally {
        this.setIsLoading(false);
    }
};
```

Slika 43 - Prikaz funkcije za dodavanje korisnika u skladištu korisnika

Asinkrona funkcija za ažuriranje korisnika („UpdateKorisnika“) je vrlo slična dodavanju samo ne postavlja ID korisnika na „UUID“, uz korisnika prima i njegov ID i poziva konkretnu funkciju za ažuriranje korisnika iz „api.ts“ datoteke s proslijeđenim ID-om i cijelim korisnikom. Asinkrona funkcija za brisanje korisnika („DeleteKorisnika“) prima samo ID korisnika za brisanje te unutar „try“ bloka postavlja stanje akcije „setIsLoading“ na „true“, asinkrono čeka rezultat izvođenja konkretnoga brisanja korisnika iz Web API-a te poziva akciju za brisanje korisnika iz registra. U slučaju pogreške, ona se ispisuje i na kraju se mijenja stanje akcije „setIsLoading“ na „false“ (slika 44).

```
async DeleteKorisnika(id_user: string) {
    try {
        this.setIsLoading(true);
        await api.users.RemoveUser(id_user);
        this.deleteKorisnika(id_user);
    } catch (e) {
        console.log(e);
    } finally {
        this.setIsLoading(false);
    }
};
```

Slika 44 - Prikaz funkcije za brisanje korisnika u skladištu korisnika

Na isti način su kreirane i ostale funkcije u skladištu praksi i kompanija. Skladište prijavljenoga korisnika je nešto drugačije jer se u njemu nalaze funkcije za prijavu i odjavu te upravljanje „AccessToken-ima“ (slika 45).

```
async login(user:
    { username: string; password: string, rememberMe: boolean }) {
    try {
        this.setIsLoggingIn(true);
        const res = await api.sessions.login(
            { username: user.username, password: user.password });
        this.setRememberMe(user.rememberMe);
        this.setRefreshToken(res.refreshToken);
        this.setAccessToken(res.accessToken);
        this.setKorisnik(JWTParse(this.accessToken));
    } catch (error) {
        console.log(error);
    } finally {
        this.setIsLoggingIn(false);
    }
};
```

```
async logout() {
    try {
        store.appStore.setIsLoading(true, "Odjava u tijeku...");
        await api.sessions.logout(
            { refreshToken: this.refreshToken as string });
        this.clearAll();
    } catch (error) {
        const e = error as Error | AxiosError;
        if (axios.isAxiosError(e)) {
            if ((e as AxiosError).response?.status === 404) {
                //ako ga API nema ne vazi..
                this.clearAll();
            }
        }
    } finally {
        store.appStore.setIsLoading(false);
    }
};
```

Slika 45 - Prikaz funkcija za prijavu i odjavu u skladištu prijavljenoga korisnika

4.6.3 Komponente („components“ mapa)

Unutar „components“ mape, kreirane su glavne komponente koje predstavljaju pojedine dijelove korisničkoga sučelja. Tu su korištene .tsx datoteke koje omogućuju kreiranje komponenti, odnosno TypeScript koda koji vraća vrstu jezika za označavanje („JSX“ sintaksa) pomoću kojega se onda kreiraju pojedini dijelovi korisničkoga sučelja. Te datoteke su i ovdje, radi bolje organizacije, podijeljene na mape koje predstavljaju pojedine dijelove sučelja, pa tako imamo mape „auth“ za kreiranje ekrana za prijavu, „kompanije“ za kreiranje dijelova sučelja vezanih za kompanije, „main“ za definiranje glavnih dijelova web aplikacije poput ruta, navigacije, glavnoga menija i sl. Mape za korisnike, kompanije i prakse temeljile su se na tri glavne .tsx datoteke, odnosno komponente koje su predstavljale općeniti izgled i raspored ekrana i tablice, definiciju tablice s popisima te formu za dodavanje, odnosno ažuriranje podataka. Time su u nastavku prikazani pojedini dijelovi sučelja za korisnike, a dijelovi za prakse i kompanije su implementirani na iste načine samo su korištene druge varijable i tipovi podatka. Tako je svaka komponenta, pa i ona za općeniti raspored ekrana za korisnike (slika 46), zapravo bila funkcija koja je na početku imala definirane potrebne varijable, funkcije i kuke („Hook“), a nakon toga je u „return“ dijelu definirala JSX sintaksu koja je vraćala konkretne dijelove sučelja i na kraju bi se ta komponenta izvezla uz ključnu riječ „observable“ kako bi se automatski mijenjala zajedno s promjenama u skladištu. Kod ove komponente se na početku definira korištenje skladišta korisnika i trenutnoga korisnika. Varijable se nalaze unutar vitičastih zagrada kako bi u njih pohranili vrijednost useStore().korisniciStore te tako omogućili pozivanje metoda iz skladišta direktno kroz varijablu. Tada je definirana „useEffect“ kuka („Hook“) za obavljanje dodatnih zadataka, u ovome slučaju dohvaćanja svih korisnika preko akcije iz skladišta. Na kraju kuke, u uglatim se zagradama navode ovisne varijable i nizovi koji određuju o čemu komponenta ovisi i da se kuka ponovno pokrene kada se promijene ovisne varijable i nizovi, u ovome slučaju skladište korisnika.

```
function Korisnici() {
    const { korisniciStore } = useStore();
    const { currentUserStore } = useStore();

    useEffect(() => {
        if (!korisniciStore.fetchedKorisnici) {
            korisniciStore.GetAll();
        }
    }, [korisniciStore]);
```

Slika 46 - Prikaz početka komponente za raspored ekrana s popisom korisnika

Unutar „return“ dijela svake komponente korištene su pojedine komponente iz „Ant Design“ biblioteke koja predstavlja popularnu biblioteku za korisnička sučelja namijenjenu za „React“ koja sadrži brojne komponente za definiranje stupaca, redaka, formi, gumba i ostalih dijelova korisničkoga sučelja (AntGroup, 2022.). Tako se ovdje koriste „Col“ i „Row“ komponente za definiranje rasporeda stupaca, redaka i njihove širine, ali i „Button“ komponente gdje se koristi „onClick“ opcija za definiranje događaja koji gumb

pokreće, odnosno u ovom slučaju, pozivanja određene akcije iz skladišta korisnika (kod dodavanja novog korisnika se poziva „noviKorisnik“ metoda koja nas vodi na rutu „korisnici/novi“). Tu je također definirano i da se kroz sve stupce prikaže tablica s popisom korisnika tako da se poziva komponenta „KorisniciDataTable“ gdje se njeni podaci pretvaraju u JavaScript niz. Kod poziva komponenta se postavljaju i određena svojstva („Props“), a sami pozivi se implementiraju koristeći sintaksu „<ImeKomponente/>“. Na kraju se komponenta „Korisnici“ izvozi uz ključnu riječ „observer“ kako bi se njeni podaci automatski ažurirali prilikom promjene podataka u skladištu (slika 47).

```
    return (
        <Row>
            <Col span={24}>
                <Row style={{ marginBottom: "40px" }}>
                    <Col span={24}>
                        {currentUserStore.korisnik?.role.includes(Rola.Administrator) && (
                            <Button disabled={korisniciStore.isloading} onClick={
                                () => korisniciStore.noviKorisnik()}>Novi korisnik</Button>
                        )}
                        <Button style={{ marginLeft: "10px" }}
                            disabled={korisniciStore.isloading} onClick={(e) => korisniciStore.GetAll()}>Osvježi</Button>
                    </Col>
                </Row>
                <Row>
                    <Col span={24}>
                        <KorisniciDataTable data={toJS(korisniciStore.korisnici)} isLoading={korisniciStore.isloading} />
                    </Col>
                </Row>
            </Col>
        </Row>
    )
}

export default observer(Korisnici);
```

Slika 47 - Prikaz rezultata komponente za raspored ekrana s popisom korisnika

Komponenta „KorisniciDataTable“, koja je prikazana na slici 48, je nešto složenija jer ima proslijeđena svojstva te složeniji rezultat. Tako su ovdje, prilikom kreacije komponente, definirana i njena svojstva („Props“) koja se mogu koristiti kroz različite komponente, što je i prikazano kod komponente „Korisnici“. Ovdje se opet na početku definiraju potrebne varijable i skladišta te se koristi i „useState“ kuka („Hook“) za kreiranje varijable stanja „upit“ i metode za ažuriranje toga stanja „postaviUpit“ koje se kasnije koriste za implementaciju polja za pretragu. Tada su u varijablu „columns“, kao niz, kreirani stupci tablice koristeći „ColumnsType“ komponentu s tipom podataka korisnika za odgovore kako bi definirali da stupci i njihova konfiguracija pripadaju tipu korisnika za odgovore. „ColumnsType“ nam omogućuje da za svaki stupac navedemo i njegove postavke, poput naslova stupca, indeksa koji mora odgovarati nazivu atributa iz tipova kako bi se znalo da se podaci iz toga atributa trebaju ispisati u tom stupcu, sortiranja, prikazivanja („render“) i responzivnosti gdje se unutar uglatih zagrada navodi na kojim ekranima bi stupac trebao ostati vidljiv. Na taj način su kreirani i ostali stupci za tablicu korisnika, no stupac za ispis korisnikovih rola je bio kreiran na nešto drugačiji način jer je još koristio i „align“ opciju za definiranje položaja ispisa, ali i „render“ opciju kako bi se korisnikove role ispisale pomoću „Tag“ komponente koja omogućuje dodavanje oznake s određenom pozadinskom bojom.

```
type KorisniciDataTableProps = {
    data: ResponseUser[],
    isLoading: boolean
}

3 references
function KorisniciDataTable({ data, isLoading }: KorisniciDataTableProps) {
    const { korisniciStore } = useStore();
    const { currentUserStore } = useStore();

    const student = currentUserStore.korisnik;

    const [upit, postaviUpit] = useState('');


    const columns: ColumnsType<ResponseUser> = [
        {
            title: 'Korisničko ime',
            dataIndex: 'username',
            sorter: (a: ResponseUser, b: ResponseUser) => defaultSort(a.username, b.username),
            responsive: ["xs", "sm"]
        },
        {
            title: 'Email',
            dataIndex: 'email',
            sorter: (a: ResponseUser, b: ResponseUser) => defaultSort(a.email, b.email),
            responsive: ["md"]

        },
```

Slika 48 - Prikaz komponente za prikaz tablice korisnika

Zadnji stupac svake tablice imao je naslov „Akcija“ te je imao vlastitu funkciju za prikazivanje (slika 49). Unutar nje su se na početku definirale varijable koje su korištene za provjeru prikazivanja gumba za uređivanje i brisanje. Tako se varijablom „studentId“ provjerilo je li ID prijavljenoga korisnika jednak ID-u korisnika u određenome retku, varijablom „isAdministrator“ se provjerilo ima li trenutno prijavljeni korisnik rolu administratora, a varijablom „isProfesor“ ima li rolu profesora. Te varijable su se kasnije koristile za uvjetno prikazivanje gumba. Kao rezultat „render“ funkcije vratile su se „BlueButton“ i „RedButton“ komponente koje definiraju različite vrste gumba za uređivanje i brisanje retka. Gumb „Uredi“ je pomoću „NavLink“ komponente postavljen da vodi na rutu za uređivanje određenoga korisnika, dok je gumb „Izbriši“ postavljen unutar „PopConfirm“ komponente iz „Ant Design“ biblioteke kako bi se njegovim odabirom pojavio lebdeći okvir iznad gumba s potvrdom brisanja. Tu su se definirale određene postavke poput teksta koji se pojavljuje na tom lebdećem okviru, metode iz skladišta korisnika koja se poziva u slučaju potvrde te tekstova gumba za odustajanje ili potvrdu brisanja. Iznad prikazivanja pojedinih gumba korištene su provjere za role, odnosno uloge prijavljenoga korisnika kako bi se ti gumbi prikazivali samo određenim korisnicima te samo na određenim redovima (recimo za studenta samo na njegovom retku kako bi mogao uređivati samo sebe).

```
{
    title: 'Akcija',
    key: 'action',
    align: "center",
    render: (text, record) => {
        const studentId = student?.user_id === record.id_user;
        const isAdministrator = student?.role.includes(Rola.Administrator);
        const isProfesor = student?.role.includes(Rola.Profesor);

            return (
                <Row>
                    <Col span={24}>
                        {(studentId || isAdministrator || (isProfesor && (!record.role.includes(Rola.Administrator)
                            && !record.role.includes(Rola.Profesor)))) && (
                            <NavLink to={`/korisnici/${record.id_user}`}>
                                <BlueButton style={{ width: "70px", margin: "2.5px" }}>Uredi</BlueButton>
                            </NavLink>
                        )}
                        {((isAdministrator || (isProfesor && (!record.role.includes(Rola.Administrator)
                            && !record.role.includes(Rola.Profesor)))) &&
                            <Popconfirm
                                title="Jeste li sigurni da želite izbrisati korisnika?"
                                onConfirm={() => korisniciStore.DeleteKorisnika(record.id_user)}
                                okText="Potvrdi"
                                cancelText="Odustani"
                            >
                                <RedButton style={{ width: "70px", margin: "2.5px" }}>Izbriši</RedButton>
                            </Popconfirm>
                        )}
                    </Col>
                </Row>
            );
        }
},
```

Slika 49 - Prikaz zadnjega stupca unutar komponente za prikaz tablice korisnika

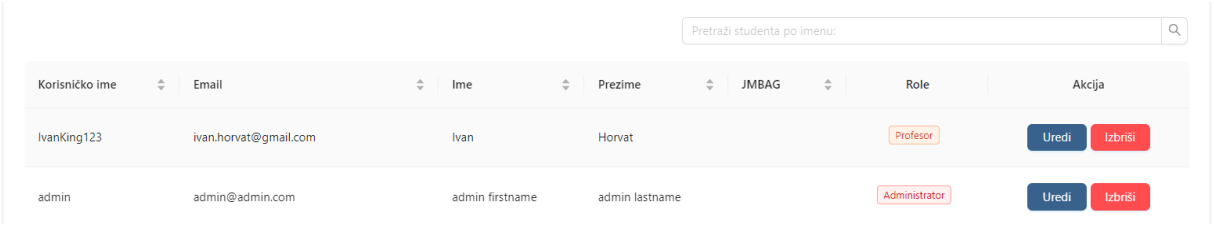
Prije samoga „return“ dijela ove komponente, koji je prikazan na slici 50, kreirana je varijabla „filtrirano“ koja je služila za pohranu filtriranoga popisa korisnika, ovisno o unesenome imenu u polje za pretragu. Tako se pomoću „filter“ metode za filtriranje niza prema uvjetu prvo pronašao korisnik iz skladišta korisnika čiji ID odgovara ID-u unesenoga korisnika pomoću „find“ metode, a zatim se u varijablu ime, ovisno o postojanju korisnika, pohranilo njegovo ime i prezime. Na kraju, funkcija provjerava odgovara li ime određenoga korisnika varijabli stanja, odnosno znakovnom nizu kojega korisnik unese u polje za pretragu, pomoću „.includes“ metode koja provjerava sadrži li znakovni niz proslijeđeni znakovni niz i ovisno o tome vraća „true“ ili „false“. Time se korisnici filtriraju tako da se dohvaćaju samo oni koji zadovoljavaju navedeni uvjet. Na kraju, u „return“ dijelu ove komponente, najprije se, unutar „ “ oznaka kreira polje za pretragu pomoću „Input.Search“ komponente iz „Ant Design“ biblioteke. Unutar „Input.Search“ komponente definiran je glavni početni tekst koji se pojavljuje u polju, vrijednost mu se dinamički postavlja ovisno o varijabli stanja „upit“, čija je promjena definirana u „onChange“ dijelu gdje se poziva „postaviUpit“ metoda za promjenu varijable stanja ovisno o korisnikovom unosu. Na kraju se još definira veličina polja za pretragu pomoću „style“ oznake. Ispod polja za pretragu prikazuje se i tablica pomoću „Table“ komponente iz „Ant Design“ biblioteke te se tu glavne postavke tablice dinamički postavljaju unutar vitičastih zagrada. Tako se za primarni ključ tablice uzima polje „user_id“, stupci se postavljaju na definirane stupce u varijablu „columns“, sami podaci koji popunjavaju tablicu su postavljeni na varijablu „filtrirano“ kako bi se prikazivali svi podaci, ali i oni koji su filtrirani po pretrazi. Na kraju se još komponenta izvozi s „observer“ ključnom riječi. Izgled početka kreirane tablice za korisnike prikazan je na slici 51.

```
    const filtrirano = data.filter((unos) => {
        const korisnik = korisniciStore.korisnici.find((korisnik) => korisnik.id_user === unos.id_user);
        const ime = korisnik ? `${korisnik.name} ${korisnik.surname}` : '';
        return ime.toLowerCase().includes(upit.toLowerCase());
    });

    return (
        <>
        <div style={{display:'flex', justifyContent:'flex-end'} }>
            <Input.Search
            placeholder="Pretraži studenta po imenu:"
            value={upit}
            onChange={(e) => postaviUpit(e.target.value)}
            style={{ marginBottom: '20px', width: '600px' }}
                />
        </div>
            <Table rowKey="user_id" columns={columns} dataSource={filtrirano} loading={isLoading} />
        </>
    )
}

export default observer(KorisniciDataTable);
```

Slika 50 - Prikaz "return" dijela "KorisniciDataTable" komponente



Slika 51 - Izgled opisanih komponenti za tablicu korisnika u korisničkome sučelju web aplikacije

I zadnja bitna komponenta je komponenta „Korisnik“ koja služi za prikazivanje forme za unos, odnosno ažuriranje korisnika. Tu se na početku, prije same „Korisnik“ komponente, kreira manja komponenta „ColumnKorisnik“ koja definira veličine stupaca za responzivnost i različite veličine ekrana. Unutar komponente „Korisnik“ se najprije definiraju potrebne varijable pomoću određenih kuka („Hooks“), pa se tako definira skladište koje se koristi pomoću „useStore()“ kuke, korištenje formi pomoću „useForm()“ kuke, mogućnost navigacije na različite rute pomoću „useNavigate()“ kuke, izvlačenje ID-a korisnika iz URL-a pomoću „useParams“ kuke te korištenje validacija iz Web API-a pomoću „useServerValidations“ kuke kojoj je proslijeđena forma (slika 52).

```
function Korisnik() {
    const { korisniciStore } = useStore();
    const [form] = Form.useForm();
    const navigate = useNavigate();
    const { id } = useParams<{ id: string }>();
    const [handleValidationErrors] = useServerValidations(form);
    const { currentUserStore } = useStore();
```

Slika 52 - Prikaz početka komponente za prikaz forme za unos/ažuriranje

Zatim se ovdje koriste dodatne funkcije i kuke za definiranje dodatnih radnji unutar komponente (slika 53). Tako se na početku koriste dvije „useEffect“ kuke, jedna za odabiranje određenoga korisnika pomoću metode iz skladišta korisnika, ukoliko je ID iz URL-a definiran te koja je ovisna o ID-u korisnika i samome skladištu korisnika pa se

ponovno pokreće svaki puta kada se te vrijednosti promijene te druga koja resetira polja u formi koristeći „resetFileds“ metodu i koja ovisi o samoj formi i odabranome korisniku pa se ponovno pokreće svaki puta kada se te vrijednosti promijene. Ovdje je definirana i „onSubmit“ asinkrona funkcija koja poziva „CreateOrUpdate“ metodu iz skladišta korisnika koja poziva funkciju za dodavanje ili ažuriranje korisnika, ovisno o tome je li ID korisnika definiran. Ukoliko se uspješno kreira ili ažurira korisnik, vraća nas se na rutu „/korisnici“ koja vodi na tablicu s popisom korisnika, a inače se ispisuju pogreške definirane kod validacija. Tu je također definirana i „onOdustani“ metoda koja postavlja da odabrani korisnik nije definiran te nas vraća na prethodnu stranicu, odnosno stranicu s tablicom popisa korisnika.

```
// na loadu selectamo korisnika
useEffect(() => {
    if (id) korisniciStore.SelectKorisnik(id);
}, [id, korisniciStore]);


// kada se selectedKorisnik promijeni resetamo form fieldove
useEffect(() => {
    form.resetFields()
}, [form, korisniciStore.selectedKorisnik]);

// form submit i odustani
const onSubmit = async (values: any) => {
    try {
        const res = await korisniciStore.CreateOrUpdate(id, values);
        if (res) {
            router.navigate("/korisnici");
        }
    } catch (e) {
        handleValidationErrors(e);
    }
}

const onOdustani = () => {
    korisniciStore.setSelectedKorisnik(undefined);
    navigate(-1);
}
```

Slika 53 - Prikaz kuka i funkcija unutar "Korisnik" komponente

Ovdje se također vrši provjera je li dohvaćen korisnik iz skladišta te ako nije, vraća se komponenta „Loading“ s prigodnim tekstom „Dohvaćamo podatke“. Tu je kreirana i metoda „onValuesChange“ koja provjerava je li određenom ključu pridružen prazan znakovni niz kako bi ga pretvorila u „null“ vrijednost radi osiguravanja konzistentnosti kod praznih vrijednosti u formi (slika 54).

```
//render
if (korisniciStore.isFetchingUser && korisniciStore.selectedKorisnik === undefined) return <Loading text="Dohvaćamo podatke..." />

const onValuesChange = (changedValues: any) => {
    Object.keys(changedValues).forEach(key => {
        if (changedValues[key] === '') {
            form.setFieldsValue({ [key]: null });
        }
    }
    );
}
```

Slika 54 - Prikaz funkcije za promjenu vrijednosti unutar forme kod korisnika

Ova komponenta vraća cijelu formu koja se prikazuje na korisničkome sučelju te koja je postavljena unutar „<Fragment>“ oznake koja je dostupna unutar „React-a“ kako bi se grupiralo više elemenata unutar jednoga ispisa (slika 55). Tu se onda koristila komponenta „<Form>“ iz „Ant Design“ biblioteke za kreiranje forme i definiranja njenih glavnih postavki poput same instance forme, veličine pojedinih polja na različitim ekranima, metode koja se poziva po završetku ispunjavanja forme (tu se poziva opisana „onSubmit“ metoda), rasporeda forme, inicijalnih vrijednosti koje su postavljene na odabranoga korisnika iz skladišta te ponašanja forme prilikom ažuriranja vrijednosti. Sva polja forme su smještena unutar jedne „<Row>“ oznake, a na vrhu ekrana se kreira komponenta za prikazivanje gumba za povratak s naslovom koji se mijenja ovisno o tome je li ID korisnika definiran, odnosno dodaje li se novi korisnik ili se ažurira postojeći. Time je i kod ažuriranja i dodavanja zapravo korištena ista forma.

```
return (
    <Fragment>
        <Form
            form={form}
            labelCol={{ sm: { span: 7 }, md: { span: 7 }, lg: { span: 8 }, xl: { span: 7 }, xxl: { span: 7 } }}
            onFinish={async (values: any) => await onSubmit(values)}
            autoComplete="off"
            layout="horizontal"
            initialValues={korisniciStore.selectedKorisnik}
            onValuesChange={onValuesChange}
        >
            <Row>
                <Col span={24} style={{ marginBottom: "20px" }}>
                    <TitleWithBackButton title={id ? "Izmjena korisnika" : "Novi korisnik"} />
                </Col>
```

Slika 55 - Prikaz početka "return" dijela "Korisnik" komponente za formu za unos/ažuriranje korisnika

Sva polja forme su postavljena unutar „<ColumnKorisnik>“ komponente u kojoj su definirana ponašanja stupaca na različitim veličinama ekrana. Svako polje unutar forme je označeno s „<Form.Item>“ oznakom unutar koje se definira oznaka polja, jedinstveno ime polja te pravila („rules“), odnosno lista s validacijama za svako polje gdje se definira i prigodna poruka (slika 56). Tu je osim „required“ validacije, korištena i „type“ validacija kod e-maila te „pattern“ validacija kod JMBAG-a kako bi se te dvije navedene validacije prikazivale prilikom unosa u polje, a ne prilikom pokušaja dodavanja. Unutar svakoga polja forme korištena je određena vrsta unosa, tako je većinom svugdje korištena „<Input>“ oznaka za definiranje običnog tekstualnoga unosa s početnim pomoćnim tekstom i jedinstvenim nazivom koji služi za povezivanje sa samim poljem forme (Form.Item) (slika 56). Osim tog unosa, korišteni su i „<Input.Password>“ za unos skrivene lozinke, „<Select>“ i „<Option>“ oznake za definiranje padajuće liste s dostupnim studentima i kompanijama kod unosa praksi, „<Radio.Group>“ za definiranje radio gumba kod odabira studija kod praksi, „<DatePicker>“ za odabir datuma kod praksi, „<InputNumber>“ za definiranje unosa brojeva te „<Input.TextArea>“ za definiranje unosa dužega teksta kod opisa poslova kod prakse. Na taj način su definirane različite vrste unosa u formi kako bi se korisniku olakšalo unošenje ili ažuriranje podataka.

```
<ColumnKorisnik>
    <Form.Item
        label="Korisničko ime"
        name="username"
        rules={[
            { required: true, message: 'Molimo unesite korisničko ime' },
        ]}
    >
        <Input
            placeholder="Korisničko ime" name="username"

        />
    </Form.Item>
    <Form.Item
        label="E-mail"
        name="email"
        rules={[
            { required: true, message: 'Molimo unesite E-mail korisnika' },
            { type: 'email', message: 'Molimo unesite ispravnu E-mail adresu' },
        ]}
    >
        <Input
            placeholder="Email korisnika" name="email"
        />
```

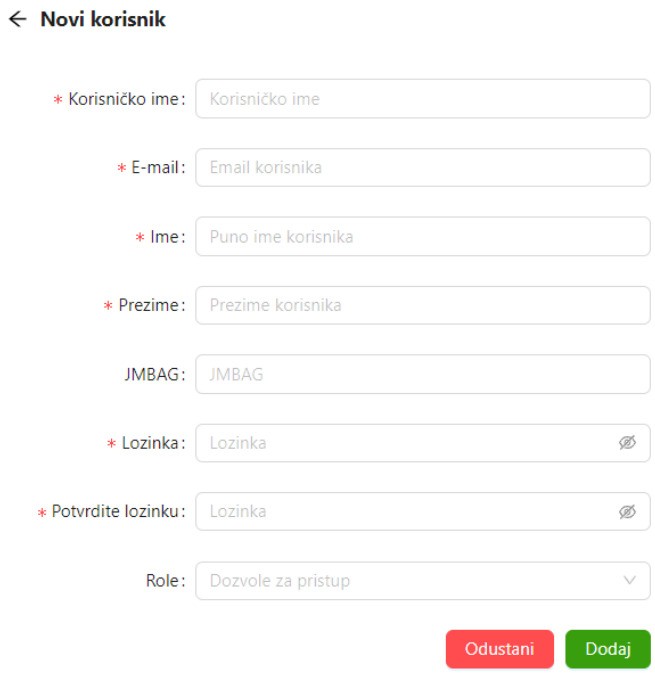
Slika 56 - Prikaz polja unutar formi za unos/ažuriranje korisnika

Kod unosa korisnika je još specifičan odabir rola, koji je implementiran kao zasebna komponenta u kojoj se kreira padajuća lista. Tu se prvo provjerava je li prijavljeni korisnik administrator te se ovisno o tome šalju drugačije vrijednosti u svojstvo „disabled“ „RoleSelect“ komponente kako bi taj unos bio onemogućen korisnicima koji nisu administratori jer ne bi bilo dobro kada bi svaki korisnik mogao sebi postaviti željenu rolu (slika 57). Na kraju se još kreira jedan redak u kojemu se kreiraju gumb za odustajanje koji poziva „onOdustani“ metodu objašnjenu na početku ove komponente te gumb za dodavanje ili izmjenu, ovisno o tome je li ID korisnika definiran, koji pokreće „onFinish“ metodu također objašnjenu na početku (slika 57). Na samome kraju se i ova komponenta izvozi s ključnom riječi „observer“.

```
            {!currentUserStore.korisnik?.role.includes(Rola.Administrator) ? (
            <RoleSelect disabled = {true}/>):(<RoleSelect disabled={false}/>)}
        </ColumnKorisnik>
    </Row>
    <Row>
        <ColumnKorisnik style={{ display: "flex", justifyContent: "flex-end" }}>
            <Form.Item>
                <RedButton disabled={korisniciStore.isloading} onClick={
                    () => onOdustani()} style={{ marginRight: "10px" }}>Odustani</RedButton>
                <GreenButton loading={korisniciStore.isloading} htmlType="submit">{id ? "Izmijeni" : "Dodaj"}</GreenButton>
            </Form.Item>
        </ColumnKorisnik>
    </Row>
</Form>
```

Slika 57 - Prikaz kraja ispisa forme za unos/ažuriranje korisnika

Izgled forme za unos/ažuriranje korisnika prikazan je na slici 58.



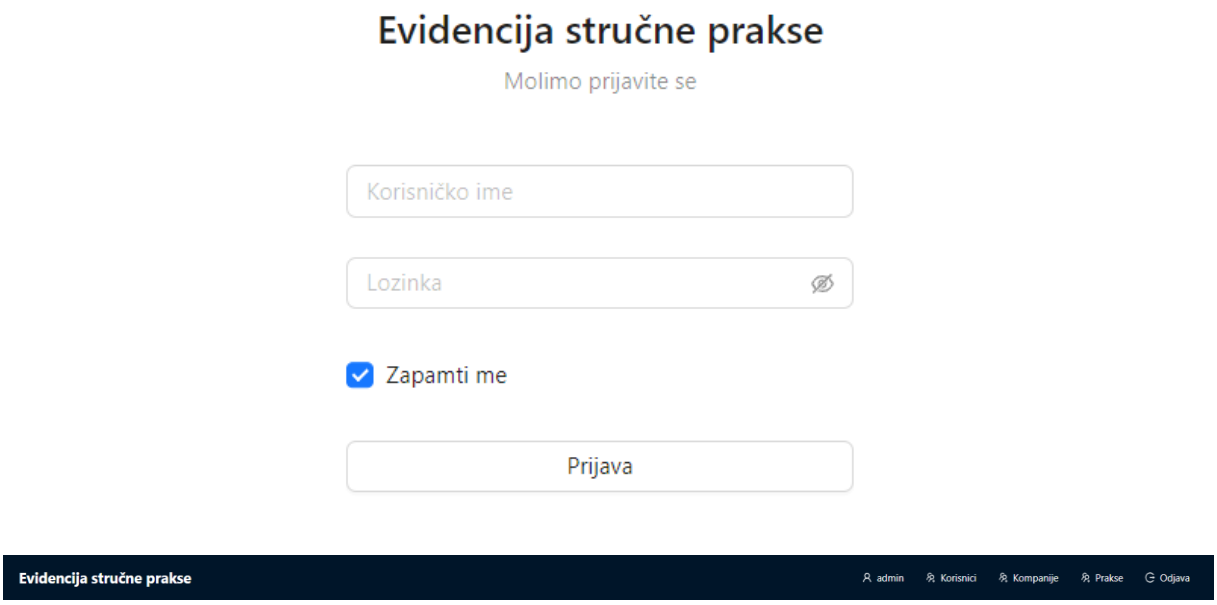
Slika 58 - Izgled forme za unos/ažuriranje korisnika

Još jedan od drugačijih i bitnih dijelova web aplikacije je definiran u „main“ mapi u komponentama, a to je „Routes.tsx“ datoteka u kojoj je definirana glavna logika upravljanja rutama i pozivanja određenih kreiranih komponenti (slika 59). Tu se stvara niz ruta tipa „RouteObject“ koji predstavlja sučelje za upravljanje pozivima komponenti prilikom pristupanju određenim rutama. Tako je ovdje definirano da se kod putanje „/“ prikazuje „App“ komponenta s definiranim glavnim dijelovima poput glavnoga menija, sekundarne navigacije i sl. Ostale rute prikazuju odgovarajuće komponente koje se pozivaju kao djeca „<ProtectedRoute>“ komponente gdje se provjerava je li korisnik prijavljen i ima li potrebnu dozvolu za pristup.

```
export const routes: RouteObject[] = [
    {
        path: '/',
        element: <App />,
        children: [
            {
                path: 'login',
                element: <LoginForm />
            },
            {
                path: 'profil',
                element:
                <ProtectedRoute requiredRole={[Rola.Administrator, Rola.Asistent, Rola.Profesor, Rola.Student]}>
                        <UlogiraniKorisnik />
                </ProtectedRoute>
            },
            {
                path: 'korisnici',
                element:
                    <ProtectedRoute requiredRole={[Rola.Administrator, Rola.Asistent, Rola.Profesor, Rola.Student]} >
                        <Korisnici />
                    </ProtectedRoute>
            },
```

Slika 59 - Prikaz ruta u "Routes.tsx" datoteci

Kod korisnika je još bitna jednostavna komponenta „UlogiraniKorisnik“ gdje je implementirano ispisivanje podataka trenutno prijavljenoga korisnika, a komponente za prakse i kompanije su kreirane na vrlo slične načine kao i spomenute tri najbitnije komponente za korisnika. Ipak komponente za prakse imaju određene specifičnosti pa se tako kod forme za unos/ažuriranje nude padajuće liste za odabire dostupnih studenata i kompanija prema njihovim nazivima, a kod tablice praksi se ne prikazuju vanjski ključevi na stupcima studenata i kompanija kao i u bazi podataka, već njihova imena i nazivi. Tu se kod prikaza imena i prezimena studenata u tablici s popisom praksi koristila posebna funkcija s „useMemo“ kukom kako bi se optimizirao vlastiti definirani ispis (pošto se koristio vlastito definirani ispis s imenom i prezimenom, bez te kuke je trebalo dosta vremena da se korisnici učitaju, a ovako ostaju pohranjeni u memoriji). Također kod praksi se je, opet pomoću varijable stanja, implementirao i filter po akademskim godinama koji je, uz filtriranje po pretrazi, dodatno filtrirao prakse i po akademskim godinama koje su bile sortirane te su se prikazivale samo jedinstvene vrijednosti (korišten „Set“ tip podataka i metoda „sort“). Filter je implementiran kao padajuća lista pomoću „Select“ komponente te je pozicioniran pored polja za pretragu. U mapi „components“ su pomoću sličnih komponenti definirani i ekran, odnosno forma za prijavu unutar „auth“ mape, glavni raspored ekrana web aplikacije gdje se pozivaju komponente za glavni meni (koja prikazuje kategorije koje vode na odgovarajuće rute) i sekundarnu navigaciju unutar „main“ mape te razne pomoćne komponente za prikazivanje gumba, ekrana za pogreške ili za učitavanje unutar „lib“ mape. Izgled forme za prijavu te glavne navigacije prikazan je na slici 60.



Slika 60 - Prikaz forme za prijavu i glavne navigacije

5. Zaključak

U ovome radu opisana je izrada cjelokupne web aplikacije za vođenje evidencije stručne prakse na fakultetima. Time su izrađeni, ali i opisani svi ključni dijelovi web aplikacije, od baze podataka, središnjega sloja, sloja pristupa podacima do samoga korisničkoga sučelja. Za izradu web aplikacije su korištene vrlo aktualne tehnologije poput PostgreSQL baze podataka, .NET razvojnoga okvira te React biblioteke.

U radu su opisani i glavni temelji i principi rada web aplikacija te njihove prednosti i nedostaci te su detaljno opisane glavne mogućnosti kreirane web aplikacije. Kako bi se prikazale glavne prednosti i razlozi korištenja, detaljno su opisane već spomenute korištene tehnologije te su ponegdje prikazani i pojedini koraci njihove implementacije. Također, opisani su korišteni programski jezici i uređivač koda Visual Studio 2022 te su detaljno prikazani i objašnjeni pojedini, najvažniji dijelovi koda kreirane web aplikacije.

Na kraju je, korištenjem glavnih principa web aplikacija, ali i spomenutih tehnologija, kreirana web aplikacija koja služi fakultetima i njihovim studentima za lakše i efikasnije vođenje evidencije stručne prakse. Kreirana web aplikacija ima brojne funkcionalnosti, uključujući četiri glavne role, odnosno uloge koje imaju različite dozvole i prava pristupa. Također omogućuje i upravljanje podacima korisnika, kompanija i stručnih praksi kako bi svi ti podaci bili na jednome, organiziranome mjestu.

Na samome kraju, cjelokupni zaključak ovoga rada je da kreirana web aplikacija za vođenje evidencije stručne prakse može biti vrlo korisna za fakultete, ali i njihove studente. Svojim jednostavnim sustavom unosa i pohranjivanja podataka o studentima, kompanijama i stručnim praksama, kreirana web aplikacija fakultetima može služiti kao jedinstveno mjesto za pohranu i vođenje evidencije stručnih praksi svojih studenata, a studentima može predstavljati sustav za praćenje vlastitoga napretka, ali i pisanja određenog dnevnika stručne prakse tijekom njenoga odrađivanja.

6. Popis slika

7. Popis literature

Altynpara, E., & Bestaieva, D. (10. Lipanj 2023.). *Web Applications Architectures: Components, Layers, and Types*. Preuzeto 25. Lipanj 2023. iz Cleveroad: https://www.cleveroad.com/blog/web-application-architecture/

AntGroup. (18. Studeni 2022.). *Ant Design - The world's second most popular React UI library*. Preuzeto 4. Srpanj 2023. iz Ant Design: https://ant.design/

Auth0. (2023.). *JSON Web Tokens - jwt.io*. Preuzeto 3. Srpanj 2023. iz JWT: https://jwt.io/

Gillis, A. (Kolovoz 2021.). *UUID (Universal Unique Identifier)*. Preuzeto 6. Srpanj 2023. iz TechTarget: https://www.techtarget.com/searchapparchitecture/definition/UUID-Universal-Unique-Identifier

InterviewBit. (17. Lipanj 2022.). *Web Application Architecture - Detailed Explanation*. Preuzeto 24. Lipanj 2023. iz InterviewBit: https://www.interviewbit.com/blog/web-application-architecture/

Microsoft. (15. Srpanj 2022.). *What is the .NET SDK?* Preuzeto 4. Srpanj 2023. iz Microsoft: https://learn.microsoft.com/en-us/dotnet/core/sdk

Microsoft. (4. Svibanj 2023.). *A tour of the C# language*. Preuzeto 28. Lipanj 2023. iz Microsoft: https://learn.microsoft.com/en-us/dotnet/csharp/

Microsoft. (13. Veljača 2023.). *Asynchronous programming with async and await*. Preuzeto 1. Srpanj 2023. iz Microsoft: https://learn.microsoft.com/en-us/dotnet/csharp/asynchronous-programming/

Microsoft. (3. Ožujak 2023.). *Primary and Foreign Key Constraints*. Preuzeto 3. Srpanj 2023. iz Microsoft: https://learn.microsoft.com/en-us/sql/relational-databases/tables/primary-and-foreign-key-constraints?view=sql-server-ver16

Microsoft. (2023.). *TypeScript: JavaScript With Syntax For Types*. Preuzeto 29. Lipanj 2023. iz TypeScript: https://www.typescriptlang.org/

Microsoft. (2023.). *Visual Studio: IDE and Code Editor for Software Developers*. Preuzeto 28. Lipanj 2023. iz Microsoft: https://visualstudio.microsoft.com/

MobX. (2023.). *About MobX - MobX*. Preuzeto 4. Srpanj 2023. iz MobX UA: https://mobx.js.org/README.html

Murugesan, S. (2008.). Web Application Development: Challenges And The Role Of Web Engineering. U S. Murugesan, *Human-Computer Interaction Series* (str. 7-32). London: Springer.

pgAdmin. (31. Svibanj 2023.). Preuzeto 26. Lipanj 2023. iz pgAdmin: https://www.pgadmin.org/

PGDG. (25. Svibanj 2023.). *PostgreSQL: The World's Most Advanced Open Source Relational Database*. Preuzeto 26. Lipanj 2023. iz PostgreSQL: https://www.postgresql.org/

ReactTeam. (3. Svibanj 2023.). *React The library for web and native user interfaces*. Preuzeto 27. Lipanj 2023. iz React: https://react.dev/

Roth, D., Anderson, R., & Luttin, S. (15. Studeni 2022.). *Overview of ASP.NET Core*. Preuzeto 27. Lipanj 2023.¸ iz Microsoft: https://learn.microsoft.com/en-us/aspnet/core/introduction-to-aspnet-core?view=aspnetcore-7.0

Sharma, J. (2015). *Web Application Development.* Gujarat: Knowledge Management and <R>esearch Organization Pune.

Volle, A. (6. Listopad 2022.). *Web application*. Preuzeto 23. Lipanj 2023. iz Britannica: https://www.britannica.com/topic/Web-application

w3schools. (2023.). *HTML tutorial*. Preuzeto 4. Srpanj 2023. iz w3schools: https://www.w3schools.com/html/

w3schools. (2023.). *React JSX*. Preuzeto 5. Srpanj 2023. iz w3schools: https://www.w3schools.com/react/react_jsx.asp

w3schools. (2023.). *SQL Injection*. Preuzeto 5. Srpanj 2023. iz w3schools: https://www.w3schools.com/sql/sql_injection.asp

w3schools. (2023.). *SQL Tutorial*. Preuzeto 3. Srpanj 2023. iz w3schools: https://www.w3schools.com/sql/

Wikipedia. (1. Siječanj 2022.). *Hash tablica*. Preuzeto 4. Srpanj 2023. iz Wikipedia The Free Encyclopedia: https://hr.wikipedia.org/wiki/Hash_tablica

Wikipedia. (21. Srpanj 2022.). *Sustav za upravljanje bazom podataka*. Preuzeto 2. Srpanj 2023. iz Wikipedia The Free Encyclopedia: https://hr.wikipedia.org/wiki/Sustav_za_upravljanje_bazom_podataka

Wikipedia. (13. Srpanj 2022.). *URL*. Preuzeto 1. Srpanj 2023. iz Wikipedia The Free Encyclopedia: https://hr.wikipedia.org/wiki/URL

Wikipedia. (5. Svibanj 2023.). *base64*. Preuzeto 6. Srpanj 2023. iz Wikipedia The Free Encyclopedia: https://en.wikipedia.org/wiki/Base64

Wikipedia. (8. Lipanj 2023.). *Frontend and backend*. Preuzeto 30. Lipanj 2023. iz Wikipedia The Free Encyclopedia: https://en.wikipedia.org/wiki/Frontend_and_backend

Wikipedia. (10. Lipanj 2023.). *Front-end web development*. Preuzeto 30. Lipanj 2023. iz Wikipedia The Free Encyclopedia: https://en.wikipedia.org/wiki/Front-end_web_development

Wikipedia. (4. Srpanj 2023.). *Hash function*. Preuzeto 5. Srpanj 2023. iz Wikipedia The Free Encyclopedia: https://en.wikipedia.org/wiki/Hash_function

Wikipedia. (29. Lipanj 2023.). *UTF-8*. Preuzeto 4. Srpanj 2023. iz Wikipedia The Free Encyclopedia: https://en.wikipedia.org/wiki/UTF-8

Wikipedia. (4. Ožujak 2023.). *Web API*. Preuzeto 30. Lipanj 2023. iz Wikipedia The Free Encyclopedia: https://en.wikipedia.org/wiki/Web_API

8. Prilozi

Cijeli projekt i web aplikacija se nalaze u GitHub repozitoriju gdje se nalaze detaljne upute sa svim potrebnim zahtjevima koji su potrebni za njeno pokretanje. Također, prikazane su korak po korak napisane upute kako osposobiti web aplikaciju nakon njenoga preuzimanja s GitHub-a. Nakon odrađivanja svih potrebnih koraka i pokretanja web aplikacije, automatski će se pokrenuti migracije u PostgreSQL bazu podataka i kreirati će se račun administratora s unaprijed određenim podacima.

Poveznica: https://github.com/T30M47/web-app-for-proffesional-practice.git