IMPLEMENTACIJA WEB APLIKACIJE ZA POVEZIVANJE POSLODAVACA I POSLOPRIMACA

Sabljić, Matija

VISOKO UČILIŠTE ALGEBRA

ZAVRŠNI RAD

IMPLEMENTACIJA WEB APLIKACIJE ZA POVEZIVANJE POSLODAVACA I POSLOPRIMACA

Matija Sabljić

Zagreb, veljača 2023.

„Pod punom odgovornošću pismeno potvrđujem da je ovo moj autorski rad čiji niti jedan dio nije nastao kopiranjem ili plagiranjem tuđeg sadržaja. Prilikom izrade rada koristio sam tuđe materijale navedene u popisu literature, ali nisam kopirao niti jedan njihov dio, osim citata za koje sam naveo autora i izvor, te ih jasno označio znakovima navodnika. U slučaju da se u bilo kojem trenutku dokaže suprotno, spreman sam snositi sve posljedice uključivo i poništenje javne isprave stečene dijelom i na temelju ovoga rada“.

U Zagrebu, 20.02.2023.

Matija Sabljic

Predgovor

Prije svega želio bih se zahvaliti svome profesoru i mentoru Aleksanderu Radovanu na vremenu i trudu uloženom prilikom izrade završnog rada. Također se želim zahvaliti svojoj obitelji, a posebno majci koji su mi bili stalna podrška tijekom školovanja.

Prilikom uvezivanja rada, Umjesto ove stranice ne zaboravite umetnuti original potvrde o prihvaćanju teme završnog rada kojeg ste preuzeli u studentskoj referadi

Sažetak

Rad opisuje implementaciju web aplikacije za potraživanje poslova. U radu su navedeni korisnički zahtjevi, korištene tehnologije, pregled sustava s navedenim mogućnostima, način izrade korisničkog sučelja, poslužitelja i baze podataka te načini komunikacije između navedenih sustava. Rezultat rada je implementirana i poslužena web aplikacija za povezivanje poslodavaca i posloprimaca. Za izradu rada korištene su tehnologije: ASP.NET Core Web API, SQL Server, Entity Framework Core, React, Material UI i Azure.

Ključne riječi: web aplikacija, potražnja poslova, ASP.NET Core, Web API, SQL Server, Entity Framework, React, Material UI, Azure.

Sadržaj

1. Uvod

Potraga za poslom može biti mukotrpan proces. Posloprimci se često moraju javiti na nekoliko pozicija prije nego što dobiju odgovor, dok se poslodavci znaju naći u situaciji da nemaju dovoljno podataka o osobama koje su se prijavile za oglas, što im znatno otežava izbor. Do samog zaposlenja dolazi i preko usmene preporuke, no ovaj rad je fokusiran na zaposlenja kroz web aplikacije. Najpopularnija rješenja za to u Hrvatskoj su trenutno: LinkedIn, MojPosao.hr i posao.hr. U procesu zaposlenja kroz web aplikacije, sam tijek procesa, neovisno o aplikaciji, uvijek dijeli nekoliko istih značajki:

- Poslodavac objavi oglas
- Posloprimac se prijavi na objavljeni oglas
- Poslodavac pregleda prijavu i napravi odluku

Jedan od čestih problema u navedenom procesu je neodlučnost. Poslodavci i posloprimci su često primorani praviti odluke na temelju ograničene količine informacija. Istovremeno broj potencijalnih pozicija posloprimcima može otežati izbor, dok u drugu ruku broj potencijalnih kandidata otežava izbor poslodavcima. Stoga je cilj ovoga rada izrada programskog rješenja koje bi olakšalo cjelokupan proces. Rezultat rada je web aplikacija koja nudi moguće rješenje na navedeni problem. Rješenje je po mnogočemu slično trenutno aktualnim proizvodima s istom svrhom, no razlikuje se po određenim značajkama koje su implementirane s ciljem da predstave moguće rješenje na navedeni problem.

U radu su prije svega detaljno opisani korisnički zahtjevi koje je bilo potrebno ispuniti za realizaciju rješenja. Naime, web aplikacija implementirana u sklopu ovog rada sastoji se od tri glavne komponente:

- Aplikacija koja predstavlja korisničko sučelje (engl. *client*)
- Aplikacija koja predstavlja poslužitelja (engl. *server*)
- Baza podataka

Rad opisuje implementaciju i međusobnu komunikaciju navedenih komponenti u svrhu ispunjenja korisničkih zahtjeva te skup tehnologija i paradigmi korištenih u razvojnom procesu.

2. Analiza postojećih aplikacija za pretragu oglasa za poslove

Na hrvatskom tržištu postoji nekoliko aplikacija za pretragu oglasa za poslove. Od njih se po broju korisnika kao najpoznatija izdvaja MojPosao[1]. Radi se o aplikaciji koja primarno služi za pronalazak zaposlenja, no aplikacija nudi i ostale mogućnosti, a samo neke od njih su *employer branding* i provođenje analize zadovoljstva zaposlenika. Ovaj rad se fokusira isključivo na značajke vezane uz pronalazak zaposlenja. Naime, jedna od glavnih specifičnosti aplikacije MojPosao.hr je to što se radi o aplikaciji koja koristi značajke nekoliko manjih aplikacija. Glavna rješenja koje MojPosao.hr koristi su:

- Hercul
- Baza profila

Hercul je alat za regrutaciju i selekciju kandidata koji pruža cjelovito rješenje provođenja selekcijskog procesa[2]. Neke od značajki Hercula koje su navedene na glavnoj stranici MojPosao.hr:

- Cjelokupni selekcijski proces se odvija kroz web aplikaciju
- Kvalitetnija selekcija kandidata prijavljenih na oglas
- Regrutiranje kandidata iz baze profile

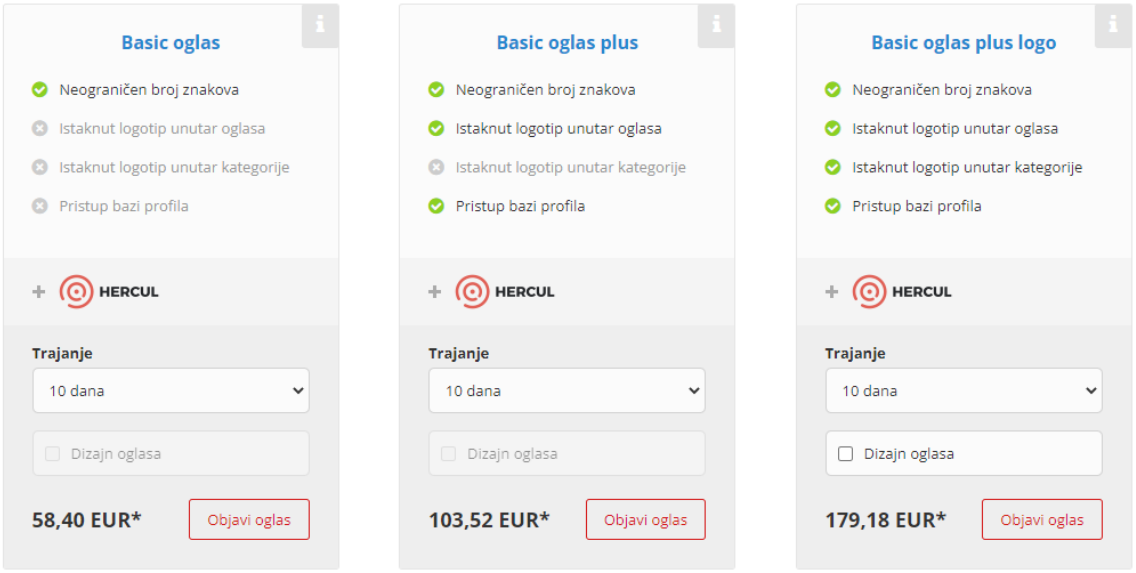
Baza profila je alat koji sadržava skup životopisa posloprimaca koje poslodavac može samostalno pretraživati prema ključnim riječima. Primarna svrha tog alata je ušteda vremena poslodavcima i poboljšanje selekcijskog procesa kroz odabir posloprimaca koji najviše zadovoljavaju određene kriterije. Glavne značajke baze profila su:

- Pronalazak posloprimaca u kratkom periodu
- Spremanje profila posloprimaca s ciljem bržeg pristupa
- Slanje personaliziranih pozivnica posloprimcima koji su se javili na određeni oglas

Nadalje, MojPosao.hr ima specifičan način monetizacije svojih usluga. Oglasi se naplaćuju ovisno o odabranom paketu, a svaki paket nudi određene mogućnosti na samom oglasu, dok

[1] https://www.moj-posao.net/
[2] https://www.moj-posao.net/hr/hercul/sale

se pristup Bazi profila naplaćuje kroz kredite[3]. Korisnik kredite kupuje ili prikuplja objavom oglasa za poslove.



Slika 1 Model monetizacije oglasa[4]

Slika 1 Model monetizacije oglasa prikazuje neke od mogućih oblika oglasa za posao koji se međusobno razlikuju prema mogućnostima koje nude poslodavcu. MojPosao trenutno nudi šest mogućih oblika oglasa za posao:

1. *Basic* oglas
2. *Basic* oglas plus
3. *Basic* oglas plus logo
4. *Standard* oglas
5. *Premium* oglas
6. *Exclusive* oglas

Navedeni oblici oglasa namijenjeni su specifičnim pozicijama koje se oglasom žele popuniti, ovisno o lakoći popunjavanja određene pozicije. Korisnici aplikacije su informirani da odaberu jedan od *basic* oblika oglasa ukoliko se radi o poziciji za koju smatraju da ju nije teško popuniti, odnosno da odaberu jedan od ostalih oblika oglasa ukoliko se radi o specijaliziranim pozicijama ili deficitarnim zanimanjima.

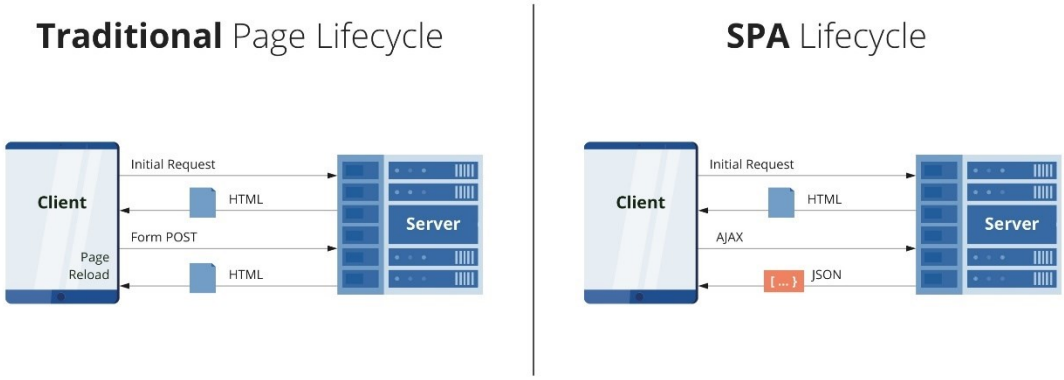
[3] https://www.moj-posao.net/Poslodavci/CVs/Offers/
[4] https://www.moj-posao.net/pregled-usluga/poslovi

3. Korištene tehnologije

Web aplikacija podijeljena je u tri glavne komponente:

- Baza podataka
- Poslužitelj
- Korisničko sučelje

Baza podataka služi za pohranjivanje podataka koje poslužitelj (engl. *server*) koristi i modificira te ih na kraju šalje prema korisničkom sučelju, tj. klijentu (engl. *client*), što je u ovom slučaju web aplikacija. Bitno je naznačiti da u konvencionalnom razvoju web aplikacija poslužitelj najčešće sadrži i aplikacijski kod potreban za generiranje korisničkog sučelja. To nije slučaj u aplikaciji koja je implementirana u ovom radu. Stoga nije krivo naglasiti da se rješenje realizirano u ovom radu sastoji od dvije web aplikacije i baze podataka. Jedna aplikacija predstavlja poslužitelja koji odgovara na zahtjeve od druge web aplikacije koja predstavlja klijenta, tj. korisničko sučelje. Aplikacija koja predstavlja korisničko sučelje napravljena je prema SPA (engl. *Single Page Application*) modelu.



Slika 2 SPA model[5]

SPA model se razlikuje od tradicionalnog pristupa komunikaciji između klijenta i poslužitelja po tome što u SPA modelu poslužitelj prema klijentu ne šalje HTML stranice koje će se prikazati u korisnikovom pregledniku, već klijent prilikom prvog zahtjeva prema poslužitelju u odgovoru dobiva jednu ili više JavaScript datoteka. Navedene JavaScript

[5] https://dzone.com/articles/the-comparison-of-single-page-and-multi-page-appli

datoteke služe za generiranje sadržaja u korisnikovom web pregledniku. Slika 2 SPA model prikazuje usporedbu komunikacije između klijenta i poslužitelja u tzv. tradicionalnom životnom ciklusu (engl. *lifecycle*) stranice i životnom ciklusu stranice u SPA modelu. Vidljivo je da je inicijalni zahtjev klijenta prema poslužitelju u tradicionalnom modelu i SPA modelu isti, dok se svaki idući zahtjev razlikuje. Naime, u tradicionalnom modelu svaki odgovor od poslužitelja sadrži HTML kojeg web preglednik zna prikazati, dok u SPA modelu poslužitelj na svaki zahtjev od klijenta osim inicijalnog, odgovara s podacima o zatraženom resursu u određenom formatu. Ključna komponenta realizacije SPA modela je korištenje JavaScript tehnologije. Nakon inicijalnog zahtjeva, upotrebom JavaScripta, web preglednik u SPA modelu ne iscrtava cjelokupnu stranicu u svrhu prikaza korisničkog sučelja, već pravi izmjene na dijelovima stranice ovisno o odgovoru od poslužitelja i sadržaju istog. SPA model je detaljnije opisan u poglavlju Korisničko sučelje.

Nadalje, za bazu podataka korištena je SQL Server baza podataka. Odabrana je relacijska baza podataka, jer kao takva čuva podatke u tablicama kroz retke i stupce te uz ograničenja (engl. *constraint*) kao što su primarni i strani ključevi omogućuje pravilno mapiranje veza između entiteta (tablica) koji sačinjavaju poslovnu logiku same aplikacije. Poslužitelj je napisan u ASP.NET radnom okviru u C# programskom jeziku. Microsoft nudi nekoliko opcija prilikom odabira tehnologije poslužitelja, a većinom ih razlikuje metodologija, tj. pravila kojih se treba pridržavati prilikom korištenja određene tehnologije. Među najkorištenijim tehnologijama mogu se izdvojiti:

- Web Forms
- MVC
- Web Pages (Razor)
- Web API

Za poslužitelja je odabrana Web API (engl. *application programming interface*) metodologija primarno iz razloga što takav pristup omogućuje separaciju logike i povećava skalabilnost. Korištenjem ove metodologije, u budućnosti može biti stvorena i mobilna verzija aplikacije bez da moraju biti pravljene modifikacije na samom poslužitelju. Konkretan naziv i verzija tehnologije korištene za razvoj poslužitelja je ASP.NET Core 6.0. Nadalje, postoji nekoliko načina za pristup bazi podataka iz poslužitelja. U tu svrhu upotrijebljen je alat Entity Framework Core. Radi se o ORM (engl. *object-relational mapping*) alatu koji nudi mogućnost prevođenja redaka i stupaca iz baze u objekte u

aplikaciji. Naposljetku, za korisničko sučelje izabrana je popularna JavaScript knjižnica (engl. *library*) React verzije 18.2.0.

4. Opis praktičnog rješenja

4.1. Korisnički zahtjevi

Potrebno je bilo stvoriti aplikaciju koja omogućava povezivanje posloprimaca i poslodavaca u svrhu potraživanja poslova. Iz tog razloga, definirano je nekoliko uloga:

- Administrator
- Moderator
- Posloprimac
- Poslodavac
  - Senior
  - Junior

Računi s ulogom Administrator zaduženi su za stvaranje novih računa uloge Moderator te zadržavaju mogućnosti računa s ulogom Moderator. Administratori su predefinirani i novi računi s tom ulogom se ne mogu napraviti kroz sučelje, dok su Moderatori zaduženi za potvrđivanje ili odbijanje novih prijava za organizacije od strane računa s ulogom Poslodavac.

4.1.1. Posloprimac

Aplikacija ima dva sučelja za stvaranje novih računa. Jedno sučelje je namijenjeno posloprimcima, a drugo poslodavcima. Ukoliko korisnik odluči predstavljati organizaciju te popuni registracijski obrazac, u aplikaciji je stvoren račun s ulogom Posloprimac. Posloprimci zadržavaju sljedeće mogućnosti:

- Pregled obavijesti
- Pregled oglasa za posao i popunjavanje prijava za posao
- Praćenje dosadašnjih prijava za posao
- Pregled svih profila u aplikaciji
- Upravljanje referencama

Prilikom određenih korisničkih akcija unutar aplikacije, korisnici dobivaju obavijesti. Tipovi obavijesti koje su trenutno definirane u aplikaciji su:

- Obavijest o novoj objavi organizacije

- Obavijest o novom oglasu
- Obavijest o novoj prijavi za posao
- Obavijest o novoj poruci na prijavi za posao

Bitno je naznačiti da računi s ulogom Posloprimac mogu primati sve tipove obavijesti osim obavijesti o novoj prijavi za posao, koja je rezervirana za račune s ulogom Poslodavac. Obavijesti su ključan dio realizacije socijalnog aspekta aplikacije. Kako bi korisnik primio određene obavijesti, mora pratiti profil organizacije za koju želi primati obavijesti. Iz tog razloga, obavijesti imaju dvije svrhe: informirati korisnika o određenoj akciji i povećati interakciju korisnika s aplikacijom, tj. angažirati korisnika. Potrebno je naznačiti da korisnik u svakom trenutku zadržava mogućnost da prestane pratiti određenu organizaciju.

Nadalje, glavna funkcionalnost aplikacije je potražnja poslova. Korisnici imaju jasan prikaz svih oglasa za posao i mogućnost prijave na svaki od prikazanih oglasa. Oglase za posao također mogu i filtrirati prema određenim kriterijima. Kako bi se prijavili za oglas, korisnici moraju otvoriti detalje samog oglasa te popuniti prijavu za posao. Ukoliko korisnik odluči poslati prijavu za posao, aplikacija generira obavijest za sve korisnike koji predstavljaju određenu organizaciju za koju je definiran oglas.

Osim obavijesti, socijalni aspekt aplikacije vidljiv je i u funkcionalnosti pregleda svih profila u aplikaciji. Korisnik ima mogućnost pretraživanja korisničkih profila ili profila organizacija unutar aplikacije.

4.1.2. Poslodavac

Naime, ukoliko korisnik aplikacije odluči da želi predstavljati određenu organizaciju i predavati oglase za poslove, tada se korisnik mora registrirati kao račun s ulogom Poslodavac. U tom slučaju, korisnik popunjava prijavu za takav račun. Prijavu sačinjavaju osnovni podaci o organizaciji koju korisnik predstavlja, a cilj procesa prijave je potvrditi da osoba koja je popunila prijavu zaista predstavlja određenu organizaciju. Sve do trenutka do kad prijava ne bude prihvaćena, Poslodavac zadržava jednaka prava unutar aplikacije kao i račun s ulogom Posloprimac. Ukoliko korisnik s ulogom Moderator odluči prihvatiti takvu prijavu, u aplikaciji će biti stvoren novi račun s ulogom Poslodavac. Tek kada je prijava potvrđena od strane Moderatora ili Administratora, Poslodavac može predavati oglase. Ukoliko je takav račun prvi put stvoren, dodijeljena mu je uloga Senior. Razlika između poslodavaca s ulogama Senior i Junior je ta da računi s ulogom Senior imaju pregled svih

računa koji mogu uređivati podatke o njihovoj organizaciji te mogu slati pozivnice budućim korisnicima aplikacije sa svrhom da postanu dio njihove organizacije. Bitno je naznačiti da računi s ulogom Poslodavac, i nakon potvrđene prijave, zadržavaju sve mogućnosti računa s ulogom Posloprimac: mogu pregledavati oglase ostalih organizacija, mogu se prijavljivati na oglase i pratiti vlastite prijave.

Organizacije podržavaju određen broj računa koji predstavljaju tu organizaciju. Maksimalan broj korisničkih računa zaduženih za određenu organizaciju definiran je aplikacijskom postavkom. Predstavnici organizacija, nakon što im je prihvaćena prijava, dobivaju sljedeće mogućnosti:

- Uređivanje profila organizacije
- Objavljivanje novosti vezanih uz organizaciju i pregled dosadašnjih objava
- Objavljivanje novih oglasa za poslove i pregled dosadašnjih oglasa
- Pregled prijava za posao
- Pregled izvještaja organizacije

4.1.3. Profil organizacije

Svaka organizacija ima svoj javni profil kojeg korisnici mogu posjetiti. Podatke prikazane na profilu određene organizacije mogu uređivati samo računi koji su zaduženi za određenu organizaciju. Organizacije su stvorene na temelju podataka iz prijave te se iz tog razloga ne mogu svi podatci uređivati. Podatci koji se mogu uređivati nakon potvrđene prijave su sljedeći:

- Web stranica
- Izjava o misiji
- Adresa

Web stranica organizacije je navedena na profilu sa svrhom da korisnik ima direktan pristup za dodatno prikupljanje informacija o određenoj organizaciji.

Profilu organizacije korisnici mogu pristupiti kroz tražilicu svih profila u aplikaciji.

4.1.4. Objave

Predstavnici organizacija mogu pisati objave o novostima vezanima uz organizaciju. Objave su vidljive na samom profilu organizacije. Korisnici koji prate određenu organizaciju su

obaviješteni o novim objavama za navedenu organizaciju te na početnoj stranici aplikacije imaju direktne poveznice na objave o kojima su obaviješteni. Objava je definirana tekstom, imenom korisnika koji je stvorio određenu objavu i datumom stvaranja objave.

4.1.5. Oglasi

Poslodavci imaju pregled svih dosadašnjih oglasa za svoju organizaciju kao i mogućnost stvaranja novih oglasa. Kako bi stvorili novi oglas, Poslodavci moraju popuniti obrazac koji sadrži osnovne podatke o poziciji, a to su:

- Naziv pozicije
- Radno vrijeme
- Postavke vezane uz udaljen (engl. *remote*) rad
- Opis radnog mjesta
- Datum roka prijave
- Adresa

Nakon što je obrazac popunjen i podnesen, oglas još nije vidljiv ostalim korisnicima aplikacije. Poslodavac u ovom trenutku i dalje zadržava mogućnost izmjene podataka oglasa. Ukoliko je zadovoljan s oglasom, može ga objaviti akcijom promjene statusa. Svaki oglas se može nalaziti u jednom od tri moguća statusa:

- Na čekanju
- Otvoren
- Zatvoren

Nakon što je novi oglas stvoren, njegov inicijalni status je „Na čekanju“. U ovom statusu su dozvoljene promjene na oglasu. Kako bi korisnik objavio oglas potrebno je izvršiti akciju promjene statusa na sučelju. Oglasi koji su na čekanju mogu biti promijenjeni jedino u status „Otvoren“. Posloprimci mogu slati prijave jedino za oglase koji se nalaze u otvorenom statusu. Ukoliko je na oglasu postavljen podatak o datumu roka prijave, status oglasa se na navedeni datum automatski mijenja u „Zatvoren“. Oglase koji su u zatvorenom statusu nije moguće pretraživati od strane posloprimaca, ali su vidljivi od strane poslodavaca. Oglasi u zatvorenom statusu služe samo za pregled te mogu jedino biti obrisani. Obrisani oglasi se ne prikazuju nijednom korisničkom računu.

4.1.6. Prijave za posao

Korisnik se prilikom pregleda detalja oglasa za posao može i prijaviti za isti. Proces prijave za posao iniciran je kada korisnik akcijom na korisničkom sučelju naznači da želi poslati obrazac za prijavu. U obrascu također ima mogućnost poslati poruku poslodavcu. Prijavu mogu pregledavati jedino korisnik koji je podnio prijavu i predstavnici organizacije za koju je oglas objavljen. Svaka prijava, kao i oglas za posao, ima status u kojem se trenutno nalazi. Status prijave služi tome da pruža dodatnu informaciju korisniku o stanju prijave u bilo kojem vremenu. Jedino računi koji predstavljaju određenu organizaciju mogu mijenjati statuse prijava. Prijava se može nalaziti u jednom od četiri moguća statusa:

- Poslano
- U tijeku
- Prihvaćeno
- Odbijeno

Inicijalni status u kojem se prijava nalazi nakon što je poslana je „Poslano“. U tom trenutku svi računi zaduženi za određenu organizaciju dobivaju notifikaciju koja ih obavještava da je zaprimljena nova prijava za oglas za njihovu organizaciju. Poslodavac u ovom trenutku može pregledati profil korisnika u svrhu prikupljanja općenitih informacija o mogućem posloprimcu. Ukoliko odluči da nije zadovoljan s korisnikom, može promijeniti status prijave u „Odbijeno“, no ako smatra da želi prikupiti više informacija o korisniku ili ga želi pozvati na intervju, može prebaciti status prijave u „U tijeku“. U ovom statusu Poslodavac također ima mogućnost pregleda referenci korisnika. Ukoliko poslodavac odluči da želi zaposliti određenog korisnika, može promijeniti status prijave u „Prihvaćeno“. Tada se korisniku koji se prijavio za oglas šalje obavijest o uspješnoj prijavi. U protivnom, ako poslodavac odluči da ne želi nastaviti selekcijski proces, može promijeniti status prijave u „Odbijeno“. Poslodavac i posloprimac također imaju mogućnost komunikacije kroz tekstualne poruke na svakoj prijavi. Komunikacija između korisnika je moguća jedino u trenutku kada je prijava u statusu „U tijeku“.

4.1.7. Pregled profila

Posloprimci i poslodavci kroz opciju pregleda profila imaju mogućnost lakog povezivanja i prikupljanja informacija jedni o drugima. Naime, nakon uspješnog stvaranja novog računa

u aplikaciji, korisnik je obavješten s porukom dobrodošlice i pozvan da uredi svoj profil. Profil, za razliku od korisničkog računa, predstavlja sve javne informacije o korisniku. Profil je definiran kroz nekoliko podataka, a nijedan od navedenih nije obavezan:

- Opis profila
- Adresa stanovanja
- Radno iskustvo
- Edukacija
- Certifikati
- Poveznice na socijalne mreže

Svi navedeni podaci su vidljivi ostalim korisnicima aplikacije prilikom posjeta profila drugih korisnika. Bitno je naznačiti da se podaci i korisničko sučelje razlikuju u slučaju pregleda profila organizacije i u slučaju pregleda korisničkih profila. Korisnici u svakom trenutku mogu pristupiti svom profilu i ažurirati podatke istog. Kako bi potencijalnim poslodavcima olakšali odluku, korisnici na svoj profil mogu unijeti podatke o svojoj edukaciji, radnom iskustvu i certifikatima. Ukoliko imaju socijalne mreže, mogu unijeti podatke o istima, a socijalne mreže koje aplikacija trenutno podržava su: LinkedIn, Instagram i Facebook.

4.1.8. Reference

Osim obavijesti i pregleda profila, jedan od načina realizacije aspekta socijalnih mreža su i reference. Poslodavci se ponekad nalaze u situaciji gdje ne mogu s lakoćom odlučiti između kandidata za posao te reference predstavljaju moguće rješenje upravo tog problema. Reference služe kao povratne informacije o zajedničkoj suradnji između samih korisnika aplikacije s ciljem da olakšaju poslodavcima izbor.

Kako bi stvorili referencu, korisnik mora iskoristiti tražilicu profila kako bi našao profil za koji želi napisati referencu. Korisnici imaju prikaz svih referenci stvorenima za njih, kao i referencama koje su sami stvorili za druge korisnike. Reference također mogu biti stvorene i od strane korisnika koji nemaju račun u aplikaciji. U tom slučaju korisnici koji zahtijevaju referencu šalju generiranu poveznicu korisnicima od kojih zahtijevaju referencu. Poveznica vodi na stranicu koja sadrži obrazac za popunjavanje podataka o samoj referenci, a za pristup poveznici nije potrebno imati račun u aplikaciji.

Poslane reference se primateljima nalaze u sandučiću. Ukoliko korisnik odluči da je zadovoljan s referencom, može ju poslati iz sandučića u svoju karticu za menadžment. U

menadžment kartici, korisnik može mijenjati vidljivost reference. Kako bi reference bile prikazane potencijalnim poslodavcima, korisnici mogu mijenjati vidljivost referenci. Referenca koja se nalazi u kartici za menadžment može biti vidljiva ili sakrivena. Svaki profil može imati određen maksimalan broj vidljivih referenci, a sam broj definiran je aplikacijskom postavkom. Navedena postavka se provjerava prilikom dodavanja novih referenci te ukoliko je korisnik dosegao maksimalan broj vidljivih referenci, svi naknadni zahtjevi za dodavanje vidljivih referenci su odbijeni.

Reference koje su označene kao vidljive se prikazuju predstavnicima organizacija prilikom prijave za posao. Kako bi poslodavac imao uvid u referencu posloprimca, prijava se mora nalaziti u statusu „U tijeku“ te poslodavac mora klikom gumba na korisničkom sučelju aktivirati akciju prikaza referenci.

4.2. Sloj za pristup bazi podataka

Baza podataka je ključan dio mrežnih aplikacija čija implementacija zahtjeva pohranu korisničkih podataka. U .NET programskom okviru postoji nekoliko načina za pristup bazi podataka, no jedan od sve češćih načina je korištenje ORM-a. Glavna svrha ORM alata je korištenje razreda definiranih u kodu aplikacije za pristup bazi umjesto pisanja upita u programskom jeziku specifičnom za odabranu bazu. Takav razvoj znatno smanjuje vrijeme potrebno za pisanje koda te samim time povećava produktivnost razvojnih timova. ORM alat korišten u aplikaciji je Entity Framework Core. Ovaj alat je odabran iz razloga što dolazi od Microsofta i stoga ga je lako integrirati i koristiti u .NET aplikacijama. Entity Framework Core za pristup bazi koristi modele. Model je razred sačinjen od razreda koji predstavljaju tablice u bazi podataka i konteksta koji predstavlja sesiju s bazom[6]. Kontekstom se smatra bilo koji razred koji nasljeđuje razred *DbContext* iz *Microsoft.EntityFrameworkCore* imenskog prostora (engl. *namespace*).

Odabrani ORM alat podržava nekoliko pristupa razvoju modela potrebnog za povezivanje s bazom podataka. Mogući pristupi i njihove prednosti i mane navedeni su u poglavlju Razvoj modela. U svrhu ispunjavanja pravila poslovne logike, na bazi podataka stvorene su tablice

[6] https://learn.microsoft.com/en-us/ef/core/

korištenjem upita na bazi, a za povezivanje na bazu podataka stvorene su definicije razreda čija svojstva i tipovi direktno odgovaraju stupcima svake tablice.

```
public class OdeaContext : DbContext, IOdeaContext
{
      public OdeaContext(DbContextOptions<OdeaContext>
      options) : base(options) {}

      public DbSet<Account> Account { get; set; }
      public DbSet<Role> Role { get; set; }
      public DbSet<Profile> Profile { get; set; }
      public DbSet<EmploymentHistory> EmploymentHistory
      { get; set; }
      public DbSet<EducationHistory> EducationHistory
      { get; set; }
      public DbSet<CertificationHistory> CertificationHistory
      { get; set; }
      public DbSet<Country> Country { get; set; }
      public DbSet<JobField> JobField { get; set; }
      public DbSet<JobAdStatus> JobAdStatus { get; set; }
      public DbSet<JobAd> JobAd { get; set; }
      public DbSet<JobApplication> JobApplication
      { get; set; }
      public DbSet<JobApplicationHistory>
      JobApplicationHistory { get; set; }
      public DbSet<BusinessSize> BusinessSize { get; set; }
      public DbSet<SocialMediaType> SocialMediaType
      { get; set; }
      public DbSet<EmployerApplication> EmployerApplication
      { get; set; }
      public DbSet<EmployerApplicationStatus>
      EmployerApplicationStatus { get; set; }
      public DbSet<EmployerApplicationHistory>
      EmployerApplicationHistory { get; set; }
      public DbSet<Organization> Organization { get; set; }
      public DbSet<OrganizationAccount> OrganizationAccount
      { get; set; }
      public DbSet<OrganizationAccountInvite>
      OrganizationAccountInvite { get; set; }
      public DbSet<OrganizationFollowing>
      OrganizationFollowing { get; set; }
```

```
        public DbSet<NotificationType> NotificationType
        { get; set; }
        public DbSet<Notification> Notification { get; set; }
        public DbSet<Post> Post { get; set; }
        public DbSet<ProfileSocialMediaLink>
        ProfileSocialMediaLink { get; set; }
        public DbSet<Reference> Reference { get; set; }
        public DbSet<ModeratorAccountInvite>
        ModeratorAccountInvite { get; set; }
}
```

Kod 4.1 Kontekst aplikacije

Svaka tablica u bazi podataka predstavljena je razredom u aplikacijskom kodu. EF Core mora znati koji razred u aplikaciji korespondira određenoj tablici u bazi podataka. Jedan od načina za postizanje opisanog ponašanja u je nazivanje razreda istim nazivima kao što su nazvane tablice u bazi podataka. Također je bitno naznačiti da svako svojstvo (engl. *property*) u kontekstu koje predstavlja tablicu u bazi, mora biti generički parametar razreda *DbSet*. Nadalje, odabrani ORM alat radi na način da autor aplikacijskog koda piše tzv. LINQ (engl. *Language Integrated Query*) upite, a alat ih prilikom izvođenja upita prevodi u jezik razumljiv bazi podataka, u ovom slučaju T-SQL (engl. *Transact-SQL*).

```
public async Task<IEnumerable<Country>> GetAsync()
{
    return await _odeaContext.Country.ToListAsync();
}
```

Kod 4.2 Dohvaćanje podataka korištenjem ORM alata

Metoda iz primjera Kod 4.2 Dohvaćanje podataka korištenjem ORM alata prikazuje dohvat svih podataka o zemljama iz baze podataka za potrebe postavljanja adrese na korisničkom profilu. Uzimajući u obzir da EF Core prevodi LINQ upite u T-SQL upite, korištenjem određenih aplikacijskih postavki u ASP.NET Core programskom okviru, može se vidjeti konkretan prijevod aplikacijskog koda iz prethodnog primjera u T-SQL upit razumljiv bazi podataka.

```
info: Microsoft.EntityFrameworkCore.Database.Command[20101]
      Executed DbCommand (53ms) [Parameters=[], CommandType='Text', CommandTimeout='30']
      SELECT [c].[Id], [c].[Name]
      FROM [Country] AS [c]
```

Slika 3 Rezultat prijevoda LINQ upita

Prethodna slika prikazuje T-SQL upit koji je stigao na bazu podataka prevođenjem upita iz primjera Kod 4.2 Dohvaćanje podataka korištenjem ORM alata. Nadalje, metoda *GetAsync* navedena u prethodnom primjeru, je specifična po tome što, kao i ostali razredi u imenskom prostoru ovog razreda, predstavlja implementaciju oblikovnog obrasca repozitorij. Korišteni oblikovni obrasci su opisani u poglavlju Implementacija oblikovnih obrazaca.

4.3. Poslužitelj korisničkih podataka - .NET

Svaka web aplikacija koja radi s podacima mora imati ostvarenu komunikaciju s poslužiteljem. Svrha poslužitelja je odgovoriti na zahtjeve od klijenta izvođenjem aplikacijskog koda zaduženog za rad s podacima. Poslužitelj aplikacije napravljen je koristeći Web API metodologiju. Ovakav pristup ima nekoliko benefita, a skalabilnost je jedan od njih. Mogućnost skalabilnosti je vidljiva u tome da ukoliko se ukaže potreba za razvojem mobilne ili desktop aplikacije, nije potrebno raditi promjene na poslužitelju. Mobilna aplikacija bi ostvarivala komunikaciju s poslužiteljem na isti način na koji ju ostvaruje i klijent koristeći web preglednik. Kako bi klijent i poslužitelj ostvarili komunikaciju, prema definiciji API-ja, poslužitelj je dužan definirati krajnje točke (engl. *endpoint*) iza kojih se izvršavaju određene akcije nad podacima.

4.3.1. Krajnje točke

Sa svrhom postizanja interoperabilnosti između klijenta i poslužitelja, poslužitelj prati REST (engl. *Representational state transfer*) smjernice. Web servisi ili aplikacije koje prate REST smjernice kolokvijalno se nazivaju *RESTful* servisima. *RESTful* servis izlaže informacije o svojim mogućnostima u obliku informacija o svojim resursima te na taj način omogućuje klijentu da poduzme određene akcije na navedenim resursima[7]. Resurs je podatak nad kojim se izvode navedene akcije. U poslužitelju implementiranom u sklopu rada, resurs predstavlja tablicu na bazi podataka. Najčešće akcije koje se *RESTful* servisima poduzimaju su:

- Dohvat resursa
- Stvaranje novog resursa
- Ažuriranje podataka resursa

[7]https://medium.com/extend/what-is-rest-a-simple-explanation-for-beginners-part-1-introduction-b4a072f8740f

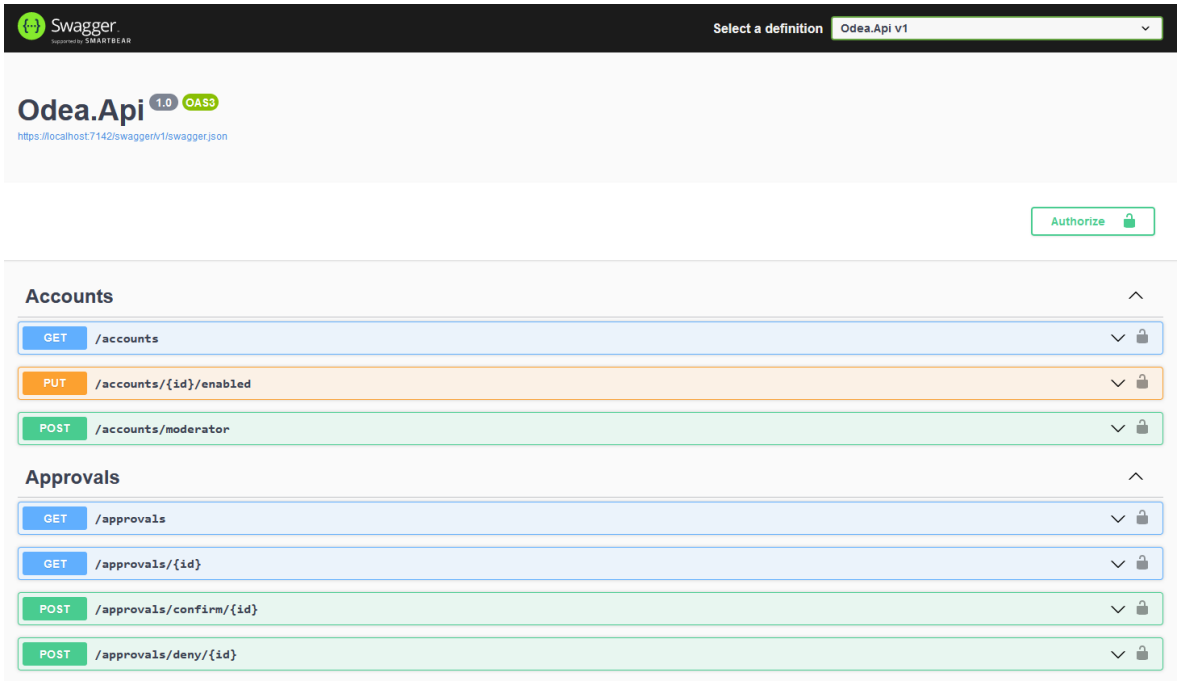
- Brisanje resursa

Uzimajući u obzir da je HTTP (engl. *Hypertext Transfer Protocol*) najčešći protokol za implementaciju ovakvih servisa, metode protokola se koriste kao identifikacije akcija nad resursom.

HTTP metoda	Opis
GET	Dohvaća resurs.
POST	Stvara novi resurs.
PUT	Mijenja resurs.
DELETE	Briše resurs.

Tablica 1 Najčešće HTTP metode u razvoju *RESTful* servisa

Jedan od standarda stvaranja ovakvih servisa definiran je OAS (engl. *OpenAPI Specification*) formatom. U .NET programskom okviru postoji nekoliko načina za implementaciju navedenog formata, a jedan od njih je korištenjem knjižnice Swashbuckle. Swashbuckle omogućuje generiranje navedene specifikacije te prikaz iste kroz korisničko sučelje.



Slika 4 *OpenAPI* specifikacija aplikacije

Slika 4 *OpenAPI* specifikacija aplikacije prikazuje definirane resurse, a svaki od resursa označen je rutom na kojoj se nalazi i HTTP metodom koja ja potrebna za pozivanje istog.

4.3.2. Autentikacija i autorizacija

Za pravilnu identifikaciju svakog korisnika aplikacije potrebno je implementirati autentikacijski proces. Aplikacija u tu svrhu koristi JWT (engl. *JSON Web Token*) predloženi[8] standard. JWT definira način sigurnog prenošenja informacija kao JSON objekt[9]. Kako bi uspješno izvršio proces registracije korisnik mora popuniti registracijski obrazac. Ukoliko su podaci u obrascu validni, poslužitelj stvara novi JWT token i dodjeljuje ga korisniku.

```
var token = new JwtSecurityToken(
                claims: claims,
                expires: DateTime.UtcNow.AddDays(1),
                signingCredentials: signingCredentials);

var jwt = new JwtSecurityTokenHandler().WriteToken(token);
```

Kod 4.3 Primjer stvaranja JWT tokena u aplikaciji

Varijabla `claims` predstavlja listu sigurnosnih tvrdnji kojima je korisnik u aplikaciji opisan, dok varijabla `signingCredentials` predstavlja instancu razreda *SigningCredentials* koja definira sigurnosni ključ i sigurnosni algoritam za stvaranje tokena. Poslužitelj odgovara na klijentski zahtjev za registraciju sa tokenom. U tom trenutku klijent je dužan spremiti taj token i koristiti ga za sve sljedeće komunikacije s poslužiteljem. Svaki sljedeći zahtjev od klijenta mora sadržavati HTTP zaglavlje (engl. *header*) naziva *Authorization* čija je vrijednost dobiveni token. Upravo iz tog razloga samo dvije rute na poslužitelju ne provjeravaju ovaj token, a to su registracijska ruta i ruta za prijavu. Nadalje, autorizacija je u aplikaciji implementirana kroz podatkovne atribute u C# programskom jeziku na razini razreda koji predstavljaju skupove krajnjih točaka. U web API projektima .NET programskog okvira, to su razredi koji nasljeđuju razred *ControllerBase* i kolokvijalno se nazivaju kontrolerima. Atribut koji omogućuje autorizaciju na razini kontrolera je ima naziv *Authorize*.

[8] https://www.rfc-editor.org/info/rfc7519
[9] https://jwt.io/introduction

```
[Authorize(Roles =
"SeniorEmployerRepresentative,JuniorEmployerRepresentative")]
[Route("[controller]")]
public class ReportsController : OdeaBaseController
{
      private readonly IReportsService _reportsService;

      public ReportsController(
      IReportsService reportsService)
      {
            _reportsService = reportsService;
      }

      [HttpGet("organization/{id}")]
      public async Task<IActionResult> FilterAsync(
      [FromRoute] int id, [FromQuery] int year)
      {
            return Ok(await
            _reportsService.GetOrganizationReportAsync(
            id, year, User));
      }
}
```

Kod 4.4 Primjer razreda koji zahtjeva autorizaciju

Navedeni primjer prikazuje razred s nekoliko atributa. *Authorize* atribut na razini razreda zabranjuje pristup navedenom resursu, u ovom slučaju izvještajima, ukoliko korisnik nije prethodno autoriziran. Konkretno, za akcije nad resursima označene s navedenim atributom, poslužitelj provjerava vrijednost HTTP zaglavlja naziva *Authorization*. Ukoliko vrijednost navedenog zaglavlja na HTTP zahtjevu nije postavljeno, poslužitelj na takav zahtjev odgovara s HTTP status kodom 401. *Authorize* atribut je specifičan po tome da može biti postavljen na razred, što znači da su svi resursi u navedenom razredu zaštićeni, no može biti postavljen i na pojedinačne resurse, što znači da su samo određeni resursi zaštićeni.

4.3.3. Preusmjeravanje

Nadalje, u primjeru Kod 4.4 Primjer razreda koji zahtjeva autorizaciju su osim navedenog *Authorize* atributa, vidljivi i sljedeći atributi: *Route*, *HttpGet, FromRoute* i *FromQuery*. *Route* atribut služi za pravilno preusmjeravanje klijentskih zahtjeva. Preusmjeravanje je

način na koji Web API usklađuje URI (engl. *Uniform Resource Identifier*) s akcijom na određenom resursu[10]. U navedenom primjeru, koristeći *Route* atribut, poslužitelj će za HTTP zahtjeve čija putanja nakon naziva domene sadrži vrijednost „/reports“ izvršavati aplikacijski kod razreda ReportsController. Preusmjeravanje korištenjem atributa je također omogućeno i drugim atributom iz primjera Kod 4.4 Primjer razreda koji zahtjeva autorizaciju, *HttpGet*. Ovaj atribut označava da je za navedeni resurs potrebno koristiti HTTP GET metodu te također definira dio URI-ja. Korištenjem atributa *FromRoute* i *FromQuery* poslužitelju je rečeno na koji način da postavi vrijednosti parametara metode *FilterAsync*. Naime, navedeni atributi, u ASP.NET Core programskom okviru, služe autoru aplikacijskog koda da piše manje koda vezanog uz raščlanjivanje (engl. *parsing*) vrijednosti navedenih parametara iz HTTP zahtjeva koristeći tehnologiju naziva *Model binding*. *Model binding* je funkcionalnost koja omogućuje autoru da dohvaća podatke u željenim C# tipovima bez manualnog konvertiranja istih[11]. Bez korištenja navedene tehnologije, tipovi parametara u navedenom primjeru morali bi biti tipa *string*, dok je u ovom slučaju konverzija tipova, zbog postojanja navedenih atributa, u ASP.NET Core okviru bila odrađena automatski.

4.3.4. CORS

Poslužitelji prilikom HTTP zahtjeva ujedno i provjeravaju s koje domene je određeni zahtjev stigao. Iz sigurnosnih razloga, web preglednici ne dozvoljavaju pravljenje HTTP zahtjeva na domene koje nisu poslužile određenu web stranicu. Takva restrikcija se naziva *same-origin policy*[12]. U ASP.NET Core programskom okviru, svi zahtjevi koji nisu stigli s iste domene na kojoj se poslužitelj nalazi su automatski odbijeni. Uzimajući u obzir da je poslužitelj web API koji prima pozive od aplikacije koja predstavlja korisničko sučelje te da su obje aplikacije poslužene na različitim domenama i portovima, potrebno je omogućiti komunikaciju između istih. U tu svrhu na poslužitelju mora biti omogućen CORS. CORS (engl. *Cross-Origin Resource Sharing*) je mehanizam baziran na HTTP zaglavljima koja

[10] https://learn.microsoft.com/en-us/aspnet/web-api/overview/web-api-routing-and-actions/attribute-routing-in-web-api-2
[11] https://learn.microsoft.com/en-us/aspnet/core/mvc/models/model-binding?view=aspnetcore-7.0
[12] https://learn.microsoft.com/en-us/aspnet/core/security/cors

omogućuju da poslužitelj naznači da prihvaća zahtjeve s podrijetlom (engl. *origin*) koji nije isti od poslužiteljevog[13].

```
builder.Services.AddCors(options =>
{
      options.AddPolicy(name:
      ClientApplicationPermissionPolicyName,
      policy =>
      {
            policy.AllowAnyOrigin()
            .AllowAnyHeader()
            .AllowAnyMethod();
      });
});
```

Kod 4.5 Omogućavanje CORS-a u ASP.NET Core Web API projektu

Primjer Kod 4.5 Omogućavanje CORS-a u ASP.NET Core Web API projektu prikazuje način na koji je CORS omogućen na poslužitelju. U ASP.NET programskom okviru, poslužitelja je preferirano konfigurirati korištenjem ekstenzijskih metoda iz *Microsoft.Extensions.DependencyInjection* NuGet paketa. Jedna od takvih metoda je *AddCors*. Pozivom navedene metode, na poslužitelju su dodani aplikacijski servisi potrebni za CORS. Konkretne postavke CORS-a dodane su kroz poziv *AddPolicy* metode. Rezultate ove metode je dodavanje nove sigurnosne politike. Svaka politika definirana je nazivom i postavkama. Naziv politike je u primjeru vrijednost varijable `ClientApplicationPermissionPolicyName`, a postavke su navedene kroz pozive ekstenzijskih metoda, koje u navedenom primjeru rezultiraju time da je poslužitelj konfiguriran da prihvaća zahtjeve s bilo koje domene, s bilo kojim vrijednostima HTTP zaglavlja te s bilo kojom HTTP metodom. Ovakva CORS politika iz sigurnosnih razloga nije primjerena za rad u produkcijskom okruženju, već je samo korisna u testnim okruženjima. Nadalje, u ASP.NET Core programskom okviru, pozivom *AddCors* metode, aplikacijski servisi su spremni za korištenje, no potreban je poziv dodatne metode kako bi navedeni servisi bili automatski korišteni od strane poslužitelja. Za tu svrhu potrebno je pozvati `UseCors` metodu s nazivom politike iz primjera.

[13] https://developer.mozilla.org/en-US/docs/Web/HTTP/CORS

4.3.5. Predmemoriranje

U bazi podataka postoje određene tablice čiji se podaci neće često mijenjati. Primjer takve tablice u aplikaciji je tablica *Country* koja drži podatke o zemljama. Kada klijent zatraži podatke o zemljama, u svrhu prikazivanja istih na određenim obrascima, poslužitelj mora praviti upit na bazu podataka. Takvi upiti na bazu podataka su redundantni, upravo iz razloga što se podaci ne mijenjaju ili se mijenjaju jako rijetko. Iz tog razloga, na poslužitelju je implementiran mehanizam predmemorije (engl. *caching*). ASP.NET Core programski okvir ima potporu za nekoliko različitih mehanizama predmemorije. Najjednostavniji mehanizam je baziran na sučelju *IMemoryCache*[14]. Upravo taj mehanizam implementiran je na poslužitelju. Naime, ukoliko klijent napravi HTTP zahtjev prema resursu koji služi za dohvaćanje svih zemalja, poslužitelj prvo provjerava da li se navedeni podatak nalazi u predmemoriji. Ukoliko se podatak nalazi u predmemoriji, poslužitelj u tom slučaju ne pravi upit na bazu već vraća podatak iz predmemorije. Svaka implementacija predmemorije mora imati *fallback* mehanizam. *Fallback* mehanizam u ovom slučaju je pravljenje upita na bazu. To znači da ukoliko se podatak o zemljama ne nalazi u predmemoriji, poslužitelj će napraviti upit na bazu. Koristeći *IMemoryCache* interface, sve tablice na bazi podataka čiji podaci nisu skloni čestim promjenama, nalaze se u predmemoriji aplikacije. Mehanizam predmemorije na poslužitelju je implementiran kroz pozadinski zadatak (engl. *background task*) koji se periodično izvršava tijekom životnog ciklusa aplikacije. U ASP.NET Core programskom okviru, jedan od načina pravljena pozadinski zadataka je implementacijom *IHostedService* sučelja. Za implementaciju navedenog sučelja, potrebno je implementirati dvije metode: *StartAsync*, koja se poziva kada je aplikacija spremna za pokretanje pozadinskog zadatka i *StopAsync*, koja se poziva prilikom zaustavljanja pozadinskog zadatka.

```
private void ExecuteMemoryCachingTask()
{
      using (IServiceScope scope = Services.CreateScope())
      {
            ICachingService cachingService =
            scope.ServiceProvider
            .GetRequiredService<ICachingService>();
```

[14] https://learn.microsoft.com/en-us/aspnet/core/performance/caching/memory

```
            bool memoryCacheEnabled =
            _featureManager.IsEnabledAsync(
            nameof(Features.MemoryCacheEnabled))
            .GetAwaiter().GetResult();

            if (memoryCacheEnabled)
            {
                  Task.Run(async () => await cachingService
                  .AddUpdateMemoryCacheTablesAsync())
                  .GetAwaiter().GetResult();
            }
            else
            {
                  Task.Run(async () => await cachingService
                  .RemoveTablesFromMemoryCacheAsync());
            }
        }
    }
```

Kod 4.6 Metoda koja se koristi za punjenje i pražnjenje predmemorije

Bitno je naznačiti da se pozadinski zadatak izvršava periodično u vremenskom intervalu definiranom kroz aplikacijsku postavku te da se odvojen aplikacijski servis brine o konkretnom dohvatu podataka iz baze i spremanju istih u predmemoriju. Navedeni aplikacijski servis je u primjeru Kod 4.6 Metoda koja se koristi za punjenje i pražnjenje predmemorije referenciran varijablom `cachingService`.

```
private async Task GetAndSaveTablesToMemoryCacheAsync()
{
      _memoryCache.Set(nameof(CacheKeys.Country),
      new CacheObject<Country>(
      await _countryDbRepository.GetAsync()),
      _memoryCacheEntryOptions);

      _memoryCache.Set(nameof(CacheKeys.BusinessSize),
      new CacheObject<BusinessSize>(
      await _businessSizeDbRepository.GetAsync()),
      _memoryCacheEntryOptions);

      _memoryCache.Set(
```

```
            nameof(CacheKeys.EmployerApplicationStatus),
            new CacheObject<EmployerApplicationStatus>(
            await
            _employerApplicationStatusDbRepository.GetAsync()),
            _memoryCacheEntryOptions);

            _memoryCache.Set(nameof(CacheKeys.JobField),
            new CacheObject<JobField>(await
            _jobFieldDbRepository.GetAsync()),
            _memoryCacheEntryOptions);

            _memoryCache.Set(nameof(CacheKeys.NotificationType),
            new CacheObject<NotificationType>(
            await _notificationTypeDbRepository.GetAsync()),
            _memoryCacheEntryOptions);

            _memoryCache.Set(nameof(CacheKeys.Role),
            new CacheObject<Role>(
            await _roleDbRepository.GetAsync()),
            _memoryCacheEntryOptions);

            _memoryCache.Set(nameof(CacheKeys.SocialMediaType),
            new CacheObject<SocialMediaType>(
            await _socialMediaTypeDbRepository.GetAsync()),
            _memoryCacheEntryOptions);
        }
```

Kod 4.7 Implementacija servisa za punjenje predmemorije

Za svaku tablicu u bazi podataka čiji se podaci trebaju nalaziti u predmemoriji, aplikacijski servis pravi upite na bazu. Rezultat svakog od upita sprema se u predmemoriju koristeći jedinstvene ključeve. Iz primjera je također vidljivo da su vrijednosti ključeva iskazane imenima tablica čije podatke predstavljaju. Nadalje, kako bi se određeni posao odvijao periodički, na poslužitelju se koristi razred *Timer*. *Timer* razred koji predstavlja poseban mehanizam za izvođenje metoda u specifičnim vremenskim intervalima[15]. Na poslužitelju se koristi vremenski interval od pola sata, što znači da svakih trideset minuta poslužitelj napravi određen broj upita na bazu te rezultate sprema u predmemoriju. Kada klijent napravi

[15] https://learn.microsoft.com/en-us/dotnet/api/system.threading.timer

zahtjev za podacima koji se nalazi u predmemoriji, poslužitelj dohvaća podatke iz predmemorije po istom ključu na kojem su i spremljeni.

```
public Task<IEnumerable<Country>> GetAsync()
{
      _memoryCache.TryGetValue(nameof(CacheKeys.Country),
      out CacheObject<Country> countriesCacheObject);

      return countriesCacheObject?.CachedItemsCount !=
      default ? Task.FromResult(countriesCacheObject.Items)
      : _countryDbRepository.GetAsync();
}
```

Kod 4.8 Dohvat zemalja iz predmemorije

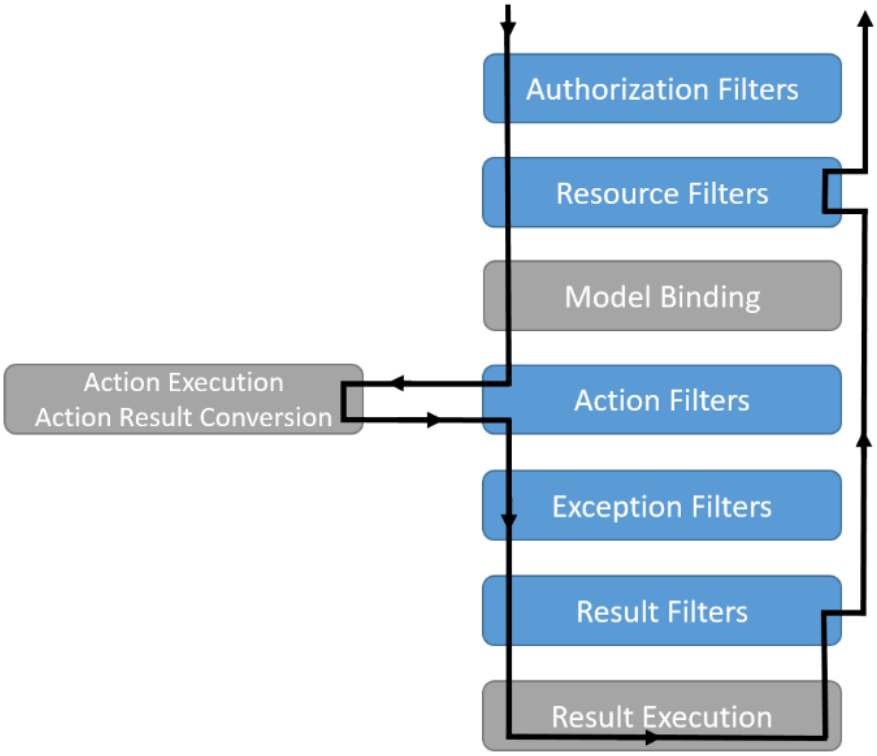
Iz primjera Kod 4.8 Dohvat zemalja iz predmemorije vidljivo je da aplikacijski servis prvo dohvaća podatke iz predmemorije korištenjem ključa *Country* te nakon toga provjerava vrijednost istog. Ukoliko na navedenom ključu u predmemoriji ne postoji vrijednost, servis će dohvatiti navedeni podatak iz baze podataka pozivom *GetAsync* metode na varijabli `_countryDbRepository`. Ovo je primjer *fallback* mehanizma koji je važan za pravilnu implementaciju predmemorije.

4.3.6. Rukovanje iznimkama

Iznimke su neizbježan dio aplikacijskog koda, a mogu rezultirati neočekivanim ponašanjem aplikacije. Pravilno rukovanje iznimkama prije svega podrazumijeva stvaranje rješenja koje omogućuje jednostavno raspoznavanje uzroka iznimke te smanjenje posljedica uzrokovanih određenom iznimkom. U svrhu raspoznavanja uzroka neke iznimke, potrebno je isti zabilježiti. Nadalje, u kontekstu web aplikacija, pravilnim rukovanjem iznimkama smatra se pravovremena notifikacija korisnika. To znači da je potrebno informirati korisnika o akciji koja nije mogla biti izvršena zbog iznimke. S ciljem stvaranja jednoznačnog rješenja za rukovanje iznimkama, na poslužitelju se koriste filteri. Filteri su specifični ASP.NET Core programskom okviru, a omogućuju izvršavanje aplikacijskog koda prije ili poslije specifičnih faza u toku procesiranja zahtjeva[16]. Postoji nekoliko tipova filtera koji se

[16] https://learn.microsoft.com/en-us/aspnet/core/mvc/controllers/filters

izvršavaju u određenim fazama procesiranja zahtjeva. Sljedeća slika prikazuje tipove filtera i njihov poredak izvršavanja.



Slika 5 Tipovi filtera i njihov poredak izvršavanja

Tip filtera korištenog u implementaciji poslužitelja je filter iznimaka. Naime, programski okvir poslužitelja sadrži nekoliko načina za implementaciju navedenog filtera. Jedan od načina implementacije filtera je korištenjem atributa koji predstavlja određeni filter. U tu svrhu na poslužitelju se koristi atribut definiran apstraktnim razredom *ExceptionFilterAttribute*. Ovaj razred je specifičan po tome što sadrži dvije metode, *OnExceptionAsync* i *OnException*, a potrebno je implementirati samo jednu od njih. Metoda *OnExceptionAsync* predstavlja asinkronu verziju metode, dok metoda *OnException* predstavlja sinkronu verziju metode. Programski okvir prilikom izvođenja aplikacijskog koda prvo provjerava da li je na određenom filteru implementirana asinkrona verzija metode, te ukoliko jest, ta verzija metoda se izvršava. Ukoliko asinkrona verzija metode nije implementirana, provjerava se da li je sinkrona verzija metode implementirana, te ukoliko jest, onda će se ta verzija i izvršiti. Poslužitelj implementiran u sklopu ovog rada koristi sinkronu verziju metode.

```
internal sealed class ApiExceptionFilterAttribute :
ExceptionFilterAttribute
```

```
{
    private readonly ILogger<ApiExceptionFilterAttribute>
    _logger;

    public ApiExceptionFilterAttribute(
    ILogger<ApiExceptionFilterAttribute> logger)
    { _logger = logger; }

    public override void OnException(
    ExceptionContext context)
    {
        _logger.LogError(
        "An unexpected error has occurred: ",
        context.Exception.Message);

        if (context.Exception is
        CustomHttpException customHttpException)
        {
            context.HttpContext.Response.StatusCode
            = (int)customHttpException.StatusCode
            .Value;

            var errorObject = new
            THttpResponse<string>()
            {
                Message = customHttpException.Message
            };

            context.Result = new
            ObjectResult(errorObject);
        }
        else if (context.Exception is
        DatabaseException databaseException)
        {
            context.HttpContext.Response.StatusCode =
            (int)HttpStatusCode.InternalServerError;

            var errorObject = new
            THttpResponse<string>()
            {
```

```
                    Message =
                    "An internal error has occurred."
                };

                context.Result = new
                ObjectResult(errorObject);
            }

            base.OnException(context);
        }
    }
```

Kod 4.9 Implementacija filtera iznimaka

Prethodni isječak koda prikazuje implementaciju filtera iznimaka stvaranjem razreda koji implementira sinkronu verziju metode koja je definirana u razredu *ExceptionFilterAttribute*. Zadaća ovog razreda je predstavljati jednoznačno rješenje za rukovanje neočekivanim ponašanjem aplikacije. Navedeni razred predstavlja jednoznačno mjesto unutar aplikacije u kojem su naznačeni mogući odgovori od poslužitelja na HTTP zahtjeve od klijentske aplikacije. Iz isječka Kod 4.9 Implementacija filtera iznimaka također je vidljivo da se uzrok svake iznimke bilježi korištenjem varijable `_logger`. Nakon što je uzrok iznimke zabilježen, koristi se tzv. kratki spoj (engl. *short-circuit*) postavljanjem vrijednosti svojstva *Result*. Korištenjem kratkog spoja, na strani poslužitelja se ne izvršava aplikacijski kod ostalih filtera (vidljivih na slici Slika 5 Tipovi filtera i njihov poredak izvršavanja) tijekom procesiranja klijentskog zahtjeva, već se odmah vraća odgovor na HTTP zahtjev.

Razred *ApiExceptionFilterAttribute* čija je implementacija vidljiva u primjeru Kod 4.9 Implementacija filtera iznimaka predstavlja atribut. Atributi se u ASP.NET Core programskom okviru koriste na način da dekoriraju određene objekte programskog koda. Na poslužitelju je potrebno navedeni atribut postaviti na sve razrede koji predstavljaju kontrolere za koje je potrebno imati istu logiku rukovanja iznimkama. S ciljem izbjegavanja suvišnosti aplikacijskog koda, na poslužitelju je definiran odvojen razred koji obuhvaća sve funkcionalnosti koje su zajedničke svim kontrolerima na poslužitelju.

```
[ApiController]
[ServiceFilter(typeof(ApiExceptionFilterAttribute))]
public class OdeaBaseController : ControllerBase
{
```

```
}
```

Kod 4.10 Definicija bazičnog razreda kontrolera

Iz prethodnog isječka koda vidljivo je da su funkcionalnosti dodijeljene bazičnom razredu kroz dva atributa, *ApiController* i *ServiceFilter*. Atribut *ApiController* omogućuje nekoliko funkcionalnosti, a potrebno je izdvojiti dvije najkorisnije. Korištenje ovog atributa zahtjeva korištenje atributa *Route* na razini svih razreda koji predstavljaju kontrolere. Takav pristup omogućuje jednoznačnost u definiciji istih. Druga bitna funkcionalnost dobivena kroz *ApiController* atribut su automatski odgovori od poslužitelja u slučaju neuspješne validacije parametara određenog zahtjeva. To znači da ukoliko programski okvir nije uspio postaviti vrijednosti parametara određene krajnje točke, poslužitelj automatski vraća odgovor na HTTP zahtjev sa status kodom 400, što direktno smanjuje aplikacijski kod potreban za validaciju parametara. Drugi korišten atribut je i *ServiceFilter*. Ovaj atribut omogućuje korištenje oblikovnog obrasca injekcije ovisnosti za omogućavanje funkcionalnosti atributa *ApiExceptionFilterAttribute*. Navedeni oblikovni obrazac i njegovo korištenje objašnjeni su u poglavlju Implementacija oblikovnih obrazaca.

Nadalje, primjer Kod 4.10 Definicija bazičnog razreda kontrolera prikazuje i nasljeđivanje razreda *ControllerBase*. Kroz ovaj razred omogućene su sve moguće funkcionalnosti potrebne za ostvarivanje komunikacije s pozivateljem krajnjih točaka definiranih na poslužitelju. Također je bitno razlikovati ovaj razred od razreda *Controller*, koji dodatno podržava rad s web stranicama, dok *ControllerBase* podržava samo rad s web API zahtjevima. Iz tog razloga poslužitelji koji predstavljaju web API u ASP.NET Core programskom okviru sadrže razrede koji nasljeđuju od razreda *ControllerBase,* a ne *Controller.* Kao što razred iz prethodnog primjera svoju funkcionalnost nasljeđuje kroz razred *ControllerBase*, tako i svi razredi koji predstavljaju kontrolere poslužitelja implementiranog u sklopu ovog rada, svoje funkcionalnosti nasljeđuju od razreda *OdeaBaseController*.

4.4. Organizacija strukture projekta

Web aplikacija implementirana u sklopu ovog rada sastoji se od dvije aplikacije i baze podataka. Aplikacije predstavljaju klijenta i poslužitelja te su odvojeno poslužene. Arhitektura poslužitelja prati tzv. *n-tier* arhitekturni obrazac (engl. *architecture pattern*). Neke literature navedenu arhitekturu nazivaju i višeslojnom arhitekturom (engl. *layered*

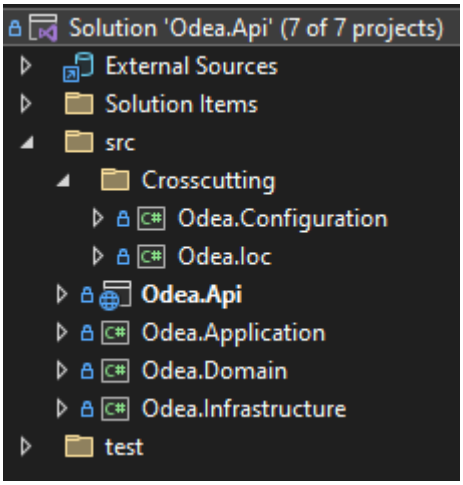
architecture pattern). Naime, radi se o jednoj od najpoznatijih i najčešćih arhitekturnih pristupa u razvoju web aplikacija.

4.4.1. Arhitektura poslužitelja

Višeslojna arhitektura bazira se na komponentama organiziranim u horizontalne slojeve, a svaki sloj izvršava specifičnu ulogu unutar aplikacije[17]. U navedenom pristupu ne postoji pisano pravilo koliko horizontalnih slojeva treba postojati, no većina projekata baziranih na ovom pristupu sastoje se od četiri glavna sloja:

- Prezentacijski sloj
- Sloj poslovne logike
- Sloj za pristup bazi podataka
- Baza podataka

Ovakva separacija komponenti je u aplikaciji postignuta kroz stvaranje nekoliko različitih projekata u istom rješenju (engl. *solution*). Svi slojevi, osim prezentacijskog, predstavljeni su knjižnicama (engl. *library*), dok je prezentacijski sloja predstavljen konzolnom aplikacijom. Razlog tome je taj što projekti koji su tipa knjižnica nemaju tzv. *entry point*, tj. mjesto početka izvođenja aplikacijskog koda. Jedna od prednosti višeslojne arhitekture je činjenica da je poželjno imati implementacije navedenih slojeva, ali dodatna separacija u više slojeva nego što većina literature navodi je također dozvoljena.



Slika 6 Organizacija strukture poslužitelja

[17] M. Richards; Software Architecture Patterns; O'Reilly Media, Inc.; (2015);

Nadalje, Slika 6 Organizacija strukture poslužitelja prikazuje implementaciju višeslojne arhitekture gdje je svaki sloj predstavljen jednim ili više projektom u ASP.NET Core programskom okviru.

Naziv projekta	Sloj
Odea.Configuration	Konfiguracijski sloj
Odea.IoC	Konfiguracijski sloj
Odea.Api	Prezentacijski sloj
Odea.Application	Sloj poslovne logike
Odea.Domain	Domenski sloj
Odea.Infrastructure	Sloj za pristup bazi podataka

Tablica 2 Nazivi projekata i njihovi odgovarajući slojevi u višeslojnoj arhitekturi

Konfiguracijski sloj aplikacije sastoji se od dva projekta, *Odea.Configuration* i *Odea.IoC*. Projekt *Odea.Configuration* sadržava definicije razreda koji predstavljaju aplikacijske postavke, dok *Odea.IoC* predstavlja implementaciju *dependency injection* oblikovnog obrasca. Dodatak definiciji višeslojne arhitekture, uz konfiguracijski sloj, je i domenski sloj. Domenski sloj sadrži razrede koji predstavljaju tablice u bazi podataka, a koriste se u sloju ispod, sloju za pristup bazi podataka. Ovaj sloj također sadrži i sučelja (engl. *interface*) čije se implementacije isto nalaze u sloju za pristup bazi podataka. Prezentacijski sloj u okviru razvoja web API-ja predstavlja skupine krajnjih točaka, tj. način na koji ostvaruje komunikaciju s klijentom. Ne sadrži nikakvu poslovnu logiku, već delegira rad nad podacima sloju ispod prezentacijskog, sloju poslovne logike.

```
[HttpGet("organization/{id}")]
[ProducesResponseType(StatusCodes.Status200OK, Type =
typeof(GetOrganizationReportsResponse))]
[ProducesResponseType(StatusCodes.Status400BadRequest, Type =
typeof(string))]
[ProducesResponseType(
StatusCodes.Status500InternalServerError,Type =
typeof(string))]
public async Task<IActionResult> FilterAsync(
[FromRoute] int id, [FromQuery][Required] int year)
```

```
{
        return Ok(
        await _reportsService.GetOrganizationReportAsync(
        id, year, User));
}
```

Kod 4.11 Krajnja točka za dohvat organizacijskih izvještaja

Iz primjera Kod 4.11 Krajnja točka za dohvat organizacijskih izvještaja vidljivo je da su u prezentacijskom sloju definirani izgled rute, koristeći *HttpGet* atribut i mogući odgovori od poslužitelji, koristeći *ProducesResponseType* atribut. Sva poslovna logika za dohvaćanje podataka vezanih uz organizacijske izvještaje je delegirana sloju za poslovnu logiku. Varijabla *_reportsService* definirana je razredom *ReportsService* koji je dio poslovne logike aplikacije i nalazi se u projektu *Odea.Application*.

4.4.2. Implementacija oblikovnih obrazaca

Oblikovni obrasci predstavljaju dokazana rješenja na ponavljajuće probleme u dizajnu softvera. Svrha obrasca je opisati određeni problem i njegovo rješenje te predstaviti pravilno okruženje u kojem se koristi određeni obrazac i posljedice korištenja istog[18]. Na strani poslužitelja implementirano je nekoliko oblikovnih obrazaca, no potrebno je izdvojiti dva obrasca čija implementacija sveobuhvatna. To su obrasci repozitorij i injekcija ovisnosti. Naime, sam programski okvir u kojem je poslužitelj izveden sadrži implementacije određenih oblikovnih obrazaca te se korištenjem istih smatra najboljom praksom rada u navedenom programskom okviru. Injekcija ovisnosti je primjer upravo takvog obrasca. Ovaj oblikovni obrazac je ugrađen u sam ASP.NET Core programski okvir, a predstavlja tehniku postizanja inverzije kontrole između razreda[19]. Odnos u kojem određeni razred ovisi o drugom razredu naziva se ovisnost između razreda. Stoga je svrha inverzije kontrole pisanje aplikacijskog koda na način da implementacija određenog razreda ne ovisi o implementaciji drugog razreda, tj. da direkcija ovisnosti u aplikaciji ide u smjeru apstrakcije, a ne u smjeru implementacijskih detalja.

[18] E. Gamma, R. Helm, R. Johnson, J. Vlissides; Design Patterns: Elements of Reusable Object-Oriented Software; Addison-Wesley Professional; (1994); 0201633612
[19] https://learn.microsoft.com/en-us/dotnet/core/extensions/dependency-injection

Za implementaciju injekcije ovisnosti potreban je tzv. kontejner servisa u kojeg se stavljaju i iz kojeg se razlažu (engl. *resolve*) i koriste konkretne implementacije razreda sa svrhom da nijedan razred prilikom pisanja koda ne koristi konkretnu implementaciju drugog razreda, već se koriste sučelja koja diktiraju definicije razreda koji ih implementiraju. Poslužitelj implementiran u sklopu ovog rada koristi kontejner servisa ugrađen u ASP.NET Core programski okvir predstavljen sučeljem *IServiceProvider*. Servisi se najčešće registriraju prilikom inicijalnog pokretanja aplikacije te se dodaju u kolekciju servisa definiranu sučeljem *IServiceCollection*. Kada su svi servisi dodani, pozivom određene metode stvara se kontejner servisa.

```
private static IServiceCollection RegisterRepositories(this
IServiceCollection services)
{
      services.AddScoped<IAuthService, AuthService>();
}
```

Kod 4.12 Primjer registracije servisa

Prethodni primjer prikazuje registraciju servisa definiranog sučeljem *IAuthService*. Korištenjem ugrađenog mehanizma injekcije ovisnosti, ASP.NET Core programski okvir će za bilo koji razred kojemu je potrebna implementacija *IAuthService* sučelja vratiti instancu razreda *AuthService*, upravo iz razloga što je taj razred registriran kao implementacija tog sučelja. Korištenjem ovog pristupa u aplikacijskom kodu nije potrebno manualno stvarati nove instance razreda *AuthService*, već ih programski okvir stvara automatski. Nadalje, način na koji se u programskom okviru poslužitelja naznačuje da je potrebno iskoristiti prethodno registriranu implementaciju određenog razreda naziva se injekcija kroz konstruktor.

```
public class AuthController : ControllerBase
{
      private readonly IAuthService _authService;
      public AuthController(IAuthService authService)
      {
            _authService = authService;
      }

      [HttpPost("register")]
      public async Task<IActionResult>
      RegisterAsync(RegisterRequest request)
```

```
        {
            return Ok(await
            _authService.RegisterAsync(request));
        }
    }
```

Kod 4.13 Primjer injekcije kroz konstruktor

Kod 4.13 Primjer injekcije kroz konstruktor prikazuje korištenje razreda koji implementira sučelje *IAuthService* bez potrebe manualnog (korištenjem ključne riječi `new`) stvaranja nove instance istog. Sam programski okvir stvara novu instancu razreda ukoliko se sučelje nalazi u prethodno spomenutoj kolekciji servisa. Svaki servis definiran u kolekciji servisa također ima i životni vijek. Servisi koji koriste kontejner koji je ugrađen u ASP.NET Core programskom okvir mogu imati jedan od tri moguća životna vijeka: *Transient*, *Scoped* i *Singleton*[20]. Za servise registrirane sa životnim tijekom *transient* stvaraju se nove instance svaki put kada su takvi servisi zatraženi iz kontejnera, dok se za *scoped* servise nove instance stvaraju po klijentskom zahtjevu. Primjer takve registracije servisa postiže se pozivom metode *AddScoped* vidljive u primjeru Kod 4.12 Primjer registracije servisa. Nadalje, *singleton* servisi su specifično po što se stvaraju prvi put kada su zatraženi te svi razredi koji koriste takav servis, koriste istu instancu razreda.

Kolekcija registriranih servisa koja se nalazi u kontejneru se tipično naziva stablo ovisnosti (engl. *dependency tree*), graf ovisnosti (engl. *dependency graph*) ili graf objekata (engl. *object graph*). Graf se sastavlja prilikom pokretanja aplikacije, a ukoliko se radi o kompleksnoj aplikaciji s puno servisa složenost grafa može znatno porasti. Iz tog razloga poslužitelj implementiran u sklopu ovog rada sadrži odvojen projekt čija je jedina svrha predstavljati jedinstveno mjesto u aplikacijskom kodu za kompoziciju razreda u stablu ovisnosti, tj. registraciju servisa u kontejneru. Projekt *Odea.IoC* spomenut u tablici Tablica 2 Nazivi projekata i njihovi odgovarajući slojevi u višeslojnoj arhitekturi, sastoji se od jednog razreda, a služi upravo tome.

Injekcija ovisnosti je samo jedan od oblikovnih obrazaca korištenih u implementaciji poslužitelja. U svrhu apstrakcije implementacije načina pristupa bazi podataka korišten je drugi oblikovni obrazac naziva repozitorij. Repozitorijima se smatraju razredi i komponente

[20] https://learn.microsoft.com/en-us/dotnet/core/extensions/dependency-injection#service-lifetimes

koje sadržavaju logiku potrebnu za pristup bazi podataka[21]. Naime, svrha ovog oblikovnog obrasca je postići razdvojenost (engl. *decoupling*) logike između one koja služi za pristup podacima i one koja radi s istima. U objektno orijentiranim jezicima, kao što je C#, ovakvo ponašanje postiže se korištenjem sučelja. Sučelja se koriste s ciljem da predstavljaju spojnice između slojeva. Određen sloj ne pristupa konkretnoj implementaciji drugog sloja, već pristupa metodama definiranima na sučeljima koja se nalaze u drugom sloju. Ukoliko se u budućnosti pokaže potreba za korištenjem druge baze podataka za određene tablice, potrebno je izmijeniti samo razrede definirane u sloju koji služi za pristup na bazu podataka, dok sloj koji sadrži poslovnu logiku aplikacije ostaje nepromijenjen.

4.5. Baza podataka

Baza podataka sadrži sve potrebne tablice i relacije između istih za ispunjavanje pravila poslovne logike. Sljedeća poglavlja opisuju veze između tablica reprezentirane ER modelom, kao i definiciju modela potrebnog za ostvarivanje komunikacije s bazom podataka na strani poslužitelja.

4.5.1. ER model

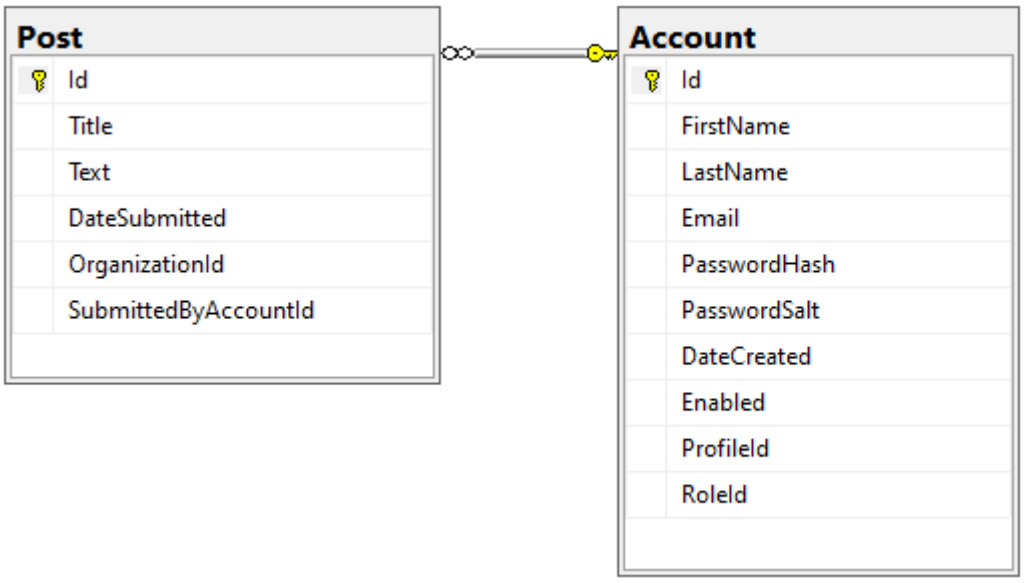
ER (engl. *entity-relationship*) model je vizualna reprezentacija strukture tablica i veza između logički povezanih tablica[22]. Koristeći ER model, vidljiva su određena pravila poslovne logike koja su prije svega definirana relacijama među tablicama. Potrebno je napomenuti da ER modeli prikazani u radu ne sadrže sve tablice na bazi podataka, već samo one koje sačinjavaju jezgru poslovne logike aplikacije. Nadalje, u relacijskim bazama podataka postoji nekoliko tipova relacija:

- Jedan naprema jedan (engl. *one-to-one*)
- Jedan naprema više (engl. *one-to-many*)
- Više naprema više (engl. *many-to-many*)

Naziv svake od navedenih relacija predstavlja broj zapisa, tj. redaka u tablici koji referenciraju retke u nekoj drugoj tablici. Primjerice, relacija jedan naprema jedan označava

[21]https://learn.microsoft.com/en-us/dotnet/architecture/microservices/microservice-ddd-cqrs-patterns/infrastructure-persistence-layer-design
[22] https://hasura.io/learn/database/microsoft-sql-server/er-modeling

odnos u kojemu jedan redak u tablici referencira jedan ili nijedan redak u drugoj tablici. Ovakva relacija može se ostvariti na nekoliko načina, a u bazi podataka dizajniranoj u sklopu ovog rada, navedena relacija kao i ostale relacije, postiže se kroz strane ključeve.



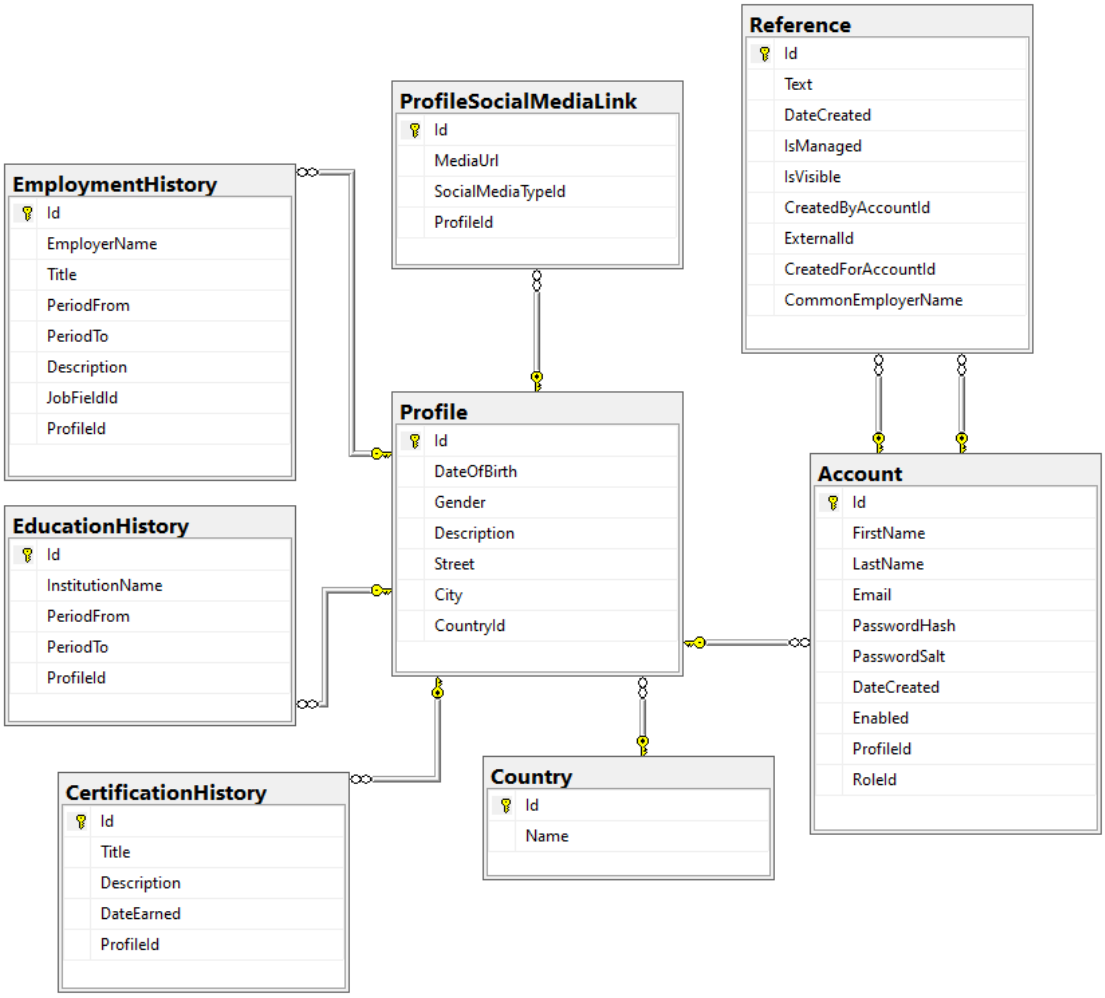
Slika 7 Primjer relacije jedan naprema jedan

Relacija jedan naprema jedan vidljiva je na prethodnoj slici. Jedan od korisničkih zahtjeva navedenih u poglavlju Korisnički zahtjevi definira pravilo da korisnici koji predstavljaju organizacije mogu objavljivati novosti o svojim organizacijama. Uzimajući u obzir da u aplikaciji samo jedan korisnik može objaviti novost, potrebno je prevesti takav zahtjev u relaciju na bazi podataka. U tu svrhu stvorena je tablica *Post*, koja sadrži stupac *SubmittedByAccountId*. Strani ključevi referenciju primarne ključeve u tablicama na koje se vežu te se stoga navedeni ključ referencira na primarni ključ tablice *Account* predstavljen stupcem *Id*. Na ovaj način se pohranjuje podatak o korisniku koji je objavio određenu novost. Slična funkcionalnost vidljiva je i u korisničkom zahtjevu da se novosti objavljuju za organizacije, što znači da svaka novost pripada određenoj organizaciji. U tu svrhu, tablica *Post* sadrži drugi strani ključ naziva *OrganizationId*. Ovaj stupac se referencira na primarni ključ naziva *Id* tablice *Organization*, a vidljiv je na slici Slika 9 Dijagram tablica vezanih uz organizacije.

Drugi tip veze je jedan naprema više u kojoj jedan redak u tablici referencira nula, jedan ili više redaka u drugoj tablici. Primjer ovakve relacije vidljiv je u određenom korisničkom zahtjevu koji određuje da svaki korisnik može unijeti adresu na svoj profil koja će biti

vidljiva korisnicima koji posjete taj profil. Adresa je određena ulicom, gradom i zemljom. Podaci o zemljama svijeta pohranjeni su u tablici *Country*. Stoga tablica Profile koja sadrži podatke o korisničkim profilima sadržava stupac *CountryId*, koji predstavlja strani ključ koji referencira primarni ključ tablice *Country*. Na ovaj način, ukoliko u aplikaciji postoji više korisnika iz iste države, redci u tablici Profile imaju istu vrijednost stupca *CountryId*, iz razloga što referenciraju isti redak, tj. istu zemlju iz tablice *Country*. Navedeni primjer relacije vidljiv je na slici Slika 8 Dijagram tablica vezanih uz korisnički račun. Relacija jedan naprema više vidljiva je i u zahtjevu koji tvrdi da korisnici na svojim profilima mogu dodavati i uređivati poveznice na svoje socijalne mreže. Uzimajući u obzir da jedan profil može sadržavati nekoliko poveznica, u aplikaciji je stvorena tablica *ProfileSocialMediaLink* koja sadržava podatke o poveznicama i profilu za koji su vezane. Poveznica na profil na koji se određena poveznica odnosi iskazan je stupcem *ProfileId* koji je strani ključ na tablicu *Profile*. Na ovaj način može postojati više poveznica za isti korisnički profil, ali ne i više korisničkih profila s istom poveznicom.

Posljednji tip relacije u relacijskim bazama podataka je više naprema više. Ova relacija naznačuje da jedan ili više redaka u jednoj tablici može referencirati nula, jedan ili više redaka u drugoj tablici. Realizacija ovakve relacije zahtjeva stvaranje dodatne tablice koja se u relacijskim bazama podataka najčešće naziva veznom tablicom (engl. *linking table*). Vezne tablice, kao što im naziv govori, služe za povezivanje redaka iz jedne tablice s redcima druge tablice te iz tog razloga najčešće sadrže samo stupce koji predstavljaju strane ključeve na obje tablice. Navedena relacija bila je potrebna za realizaciju funkcionalnosti praćenja organizacija koja predstavlja ključni dio aspekta socijalnih mreža. Naime, korisnički zahtjevi definiraju pravilo koje tvrdi da korisnici mogu pratiti organizacije i na taj način su obaviješteni o novostima organizacija koje prate. Za realizaciju ovakvog zahtjeva stvorena je vezna tablica *OrganizationFollowing*. Ova tablica je ključna za realizaciju navedene relacije iz razloga što predstavlja poveznicu između tablica *Organization* i *Account*, a vidljiva je na slici Slika 9 Dijagram tablica vezanih uz organizacije. Tablica *OrganizationFollowing* sadrži stupce *OrganizationId* i *AccountId* koji predstavljaju strane ključeve koji referenciraju primarne ključeve istih tablica. Tablica također sadrži i podatak o datumu stvaranja određenog zapisa koji je predstavljen stupcem *DateCreated*.



Slika 8 Dijagram tablica vezanih uz korisnički račun

Prethodni primjer predstavlja tablice potrebne za pohranu osnovnih podataka o korisniku kao i podataka potrebnih za ostvarivanje značajki socijalnih mreža. Tablica *Account* je ključna tablica za pohranu osnovnih korisničkih podataka, dok ostale tablice, izuzevši tablicu *Reference*, sadržavaju stupce potrebne za prikaz dodatnih informacija o korisniku. Bitno je naznačiti da je tablica *Profile* ključna u ispunjavanju aspekta socijalnih mreža. Glavni razlog stvaranja navedene tablice je potreba za prikazom podataka o korisničkim profilima prilikom posjećivanja istih u aplikaciji. Potrebno je bilo napraviti distinkciju između javnih i sigurnosnih podataka o korisniku aplikacije. Shodno tome, napravljena je separacija između osnovnih korisničkih podataka pohranjenih u tablici *Account* i javnih korisničkih podataka pohranjenih u tablici *Profile*.

Slika 9 Dijagram tablica vezanih uz organizacije

Jedan od korisničkih zahtjeva navodi i potrebu za pohranu specifičnih podataka o organizacijama. U tu svrhu stvorene su tablice prikazane na slici Slika 9 Dijagram tablica vezanih uz organizacije. U prethodnom poglavlju navedeno je da se nova organizacija stvara jedino kada prijava za određenu organizaciju bude potvrđena. Ova funkcionalnost predstavljena je tablicama *EmployerApplication* i *Organization*. Naime, podaci o novoj organizaciji koju određeni korisnik želi predstavljati unutar aplikacije pohranjeni su u *EmployerApplication* tablici. Ukoliko podaci u tablici budu potvrđeni od strane korisničkog računa s određenom ulogom, isti podaci se koriste, tj. kopiraju, u svrhu stvaranje nove organizacije. Iz tog razloga se određeni redci pojavljuju u obje tablice. Svi korisnički računi koji mogu upravljati podacima organizacija pohranjeni su u tablici *OrganizationAccount*, a pozivnice za stvaranje takvih računa pohranjene su u tablici *OrganizationAccountInvite*.

4.5.2. Razvoj modela

Kako bi poslužitelj mogao raditi s podacima, navedene podatke je potrebno pohraniti. U prethodnom poglavlju spomenuto je da je za pristup na bazu podataka korišten ORM naziva Entity Framework Core te da je za pristup na bazu podataka potreban model. Navedeni alat ima nekoliko načina za razvoj modela[23]:

- Generiranje modela iz postojeće baze podataka
- Ručno sastavljanje modela koji odgovara tablicama u bazi podataka
- Stvaranje modela i korištenje migracijskih skripti za stvaranje tablica u bazi podataka

Model koji aplikacija koristi je ručno sastavljen na način da svi tipovi u određenom razredu budu jednaki tipovima stupaca određenih tablica na bazi podataka koje navedeni razredi predstavljaju. Uzimajući za primjer tablicu *EducationHistory*, čija je svrha čuvanje podataka o povijesti edukacije korisnika aplikacije, nekoliko stupaca je definirano:

- *Id*
- *InstitutionName*
- *PeriodFrom*
- *PeriodTo*
- *ProfileId*

Za uspješno stvaranje modela, potrebno je obratiti pozornost na SQL Server tip podatka svakog od navedenih stupaca te isti mapirati u odgovarajući tip podatka u C# programskom jeziku. Nadalje, također je važno napraviti distinkciju između stupaca koji zahtijevaju vrijednosti i stupaca koji ne zahtijevaju vrijednosti.

```
create table EducationHistory (
      Id int primary key identity not null,
      InstitutionName nvarchar(200) not null,
      PeriodFrom datetime not null,
      PeriodTo datetime null,
      ProfileId int foreign key references [Profile](Id)
      not null
)
```

Kod 4.14 T-SQL naredbe za definiciju tablice

[23] https://learn.microsoft.com/en-us/ef/core

Prethodni kod prikazuje definiciju tablice *EducationHistory*. Iz koda je vidljivo da je stupac *PeriodTo* jedini stupac čija je vrijednost opcionalna. Zbog toga, definicija modela na razini poslužitelja koji predstavlja navedenu tablicu na bazi podataka je sljedeća:

```
public class EducationHistory
{
      public int Id { get; set; }
      public string InstitutionName { get; set; }
      public DateTime PeriodFrom { get; set; }
      public DateTime? PeriodTo { get; set; }
      public int ProfileId { get; set; }
}
```

Kod 4.15 Model koji predstavlja tablicu *EducationHistory*

U prikazanom modelu potrebno je obratiti pozornost na tip podatka svojstva *PeriodTo*. Iz razloga što je vrijednost stupca *PeriodTo* na bazi podataka opcionalna, isti stupac je u modelu predstavljen tipom podatka koji ne zahtjeva vrijednost. Nedostatak odabranog pristupa je činjenica da produžuje vrijeme potrebno za razvoj aplikacije, no jedna prednosti ovakvog pristupa je to što aplikacijski kod ne sadrži nijednu automatski generiranu datoteku potrebnu za povezivanje na bazu podataka što smanjuje kompleksnost projekta.

Baza podataka i tablice stvorene su koristeći T-SQL naredbe.

4.6. Korisničko sučelje

Korisničko sučelje je razvijeno kroz izdvojenu aplikaciju koja predstavlja klijenta. Najveća korist izdvajanja aplikacijskog koda koji predstavlja klijenta od koda koji predstavlja poslužitelja je separacija odgovornosti. Odgovornost klijentske aplikacije je klijentski prikaz web stranice, dok je odgovornost poslužitelja rad s podacima. Nadalje, u posljednje vrijeme sve veću popularnost dobivaju jednostranične aplikacije - SPA. SPA model i njegove specifičnosti spomenute su u poglavlju u kojem su opisane korištene tehnologije. Ključna komponenta realizacije SPA modela je AJAX (*Asynchronous JavaScript and XML*). AJAX nije tehnologija, već predstavlja pristup korištenju nekoliko postojećih tehnologija zajedno, a samo neke od njih su HTML, CSS i JavaScript[24]. Jedan od najbitnijih dijelova AJAX-a je

[24] https://developer.mozilla.org/en-US/docs/Web/Guide/AJAX

XMLHttpRequest (XHR) objekt. XHR objekti se koriste za interakciju s poslužiteljima, a omogućuju dohvat podataka s određenog URL-a bez osvježavanja kompletne stranice u web pregledniku[25]. Kombinacijom navedenih tehnologija u AJAX modelu, web aplikacije imaju mogućnost pravljenja brzih, inkrementalnih izmjena na korisničkom sučelju bez ponovnog učitavanja kompletne stranice web preglednika. Ovakav pristup omogućava veću responzivnost aplikacije što samim time rezultira kvalitetnijim korisničkim iskustvom (engl. *user experience*). Cijena za navedene prednosti SPA modela je činjenica da je prvi zahtjev znatno sporiji od ostalih iz razloga što web preglednik prilikom prvog zahtjeva u odgovoru dobiva cjelokupnu aplikaciju, tj. u odgovoru od poslužitelja dobiva jednu ili više JavaScript datoteka potrebnih za pravilan prikaz cijele stranice.

Pristupi razvoju korisničkog sučelja shodni su čestim promjenama. Jedna od takvih promjena je korištenje *Fetch* API-ja za pravljenje asinkronih poziva prema poslužitelju. Ovakva funkcionalnost je prethodno postignuta koristeći XHR[26], a navedeni API predstavlja jednostavan, logički način asinkronog dohvaćanja podataka o resursu preko mreže.

Za razvoj klijentske aplikacije korištena je popularna JavaScript knjižnica React.

4.6.1. Kontekst

U Reactu postoji nekoliko specifičnosti na koje je potrebno obratiti pozornost prilikom korištenja knjižnice. Jedna od tih specifičnosti je činjenica da React radi na način da komponente međusobno komuniciraju kroz svojstva (engl. *props*)[27] te da se sve komponente moraju nalaziti u hijerarhiji. To znači da nije rijedak slučaj naći se u poziciji da se određen podatak mora proslijediti s vrha hijerarhije na nekoliko mjesta na dnu hijerarhije. Takav način rada može dovesti do situacije koja se ponekad zove „*prop drilling*“[28], što rezultira povećanjem kompleksnosti aplikacijskog koda. Jedna od alternativa rješavanja navedenog problema je kontekst. Kontekst pruža način prosljeđivanja podataka kroz hijerarhijsko stablo komponenata bez manualnog prosljeđivanja svojstava na svakoj razini[29]. Aplikacija napravljena u sklopu ovog rada koristi kontekst za nekoliko informacija, a jedna od njih je i informacija o trenutnom korisniku aplikacije. Za korištenje konteksta, potrebno je definirati

[25] https://developer.mozilla.org/en-US/docs/Web/API/XMLHttpRequest
[26] https://developer.mozilla.org/en-US/docs/Web/API/Fetch_API/Using_Fetch
[27] https://beta.reactjs.org/learn/passing-props-to-a-component
[28] https://beta.reactjs.org/learn/passing-data-deeply-with-context
[29] https://reactjs.org/docs/context.html

komponentu koja će posluživati vrijednosti konteksta i komponentu koja će služiti za dohvaćanje istih. U aplikaciji napravljenoj u sklopu ovog rada, komponente koje služe za postavljanje i čitanje informacija o korisniku nazivaju se *UserContext* i *UserContextProvider*.

```
const UserContext = React.createContext(null);

const UserContextProvider = ({ children }) => {
    const [user, setUser] = React.useState(null);
    return (
        <UserContext.Provider value={{ user, setUser }}>
            {children}
        </UserContext.Provider>
    );
};

export { UserContext, UserContextProvider };
```

Kod 4.16 Definicija *UserContext* i *UserContextProvider* komponenti

Inicijalna vrijednost konteksta je `null`, a koristeći svojstvo `value`, u kontekst su stavljene dvije vrijednosti: podaci o trenutnom korisniku aplikacije predstavljeni varijablom `user` i funkcija za postavljanje podataka o trenutnom korisniku predstavljena varijablom `setUser`. Kontekst koji čuva informacije o korisniku aplikacije je specifičan po tome što je takva informacija potrebna na nekoliko mjesta te je iz tog razloga komponenta *UserContextProvider* postavljena na vrh hijerarhijskog stabla.

```
<UserContextProvider>
  <BrowserRouter>
    <Routes>
      <Route path="/" element={<Landing />} />
    </Routes>
  </BrowserRouter>
</UserContextProvider>
```

Kod 4.17 Primjer korištenja *UserContextProvider* komponente

U svrhu spremanja i čitanja vrijednosti konteksta, potrebno je organizirati kod na način da se u hijerarhijskom stablu komponente koje koriste kontekst nalaze ispod komponenti koje pružaju informacije o kontekstu. Ovakva hijerarhija komponenti vidljiva je u primjeru Kod 4.17 Primjer korištenja *UserContextProvider* komponente. U navedenom primjeru,

komponenta *Landing* ima mogućnost postavljanja i čitanja vrijednosti trenutno prijavljenog korisnika aplikacije.

4.6.2. Preusmjeravanje na strani klijenta

U razvoju jednostraničnih aplikacija, preusmjeravanje se više ne odvija na poslužitelju, već u web pregledniku klijenta. Takav princip naziva se preusmjeravanjem na strani klijenta (engl. *client-side routing*). U konvencionalnom razvoju web aplikacija, zadaća poslužitelja je poslužiti web stranici relativnu zatraženom URL-u, dok kod jednostraničnih aplikacija to postaje zadaća klijenta. React knjižnica sama po sebi nema podršku za pravilno rukovanje s rutama. Iz tog razloga klijentska aplikacija koristi dodatnu JavaScript knjižnicu čiji je zadatak rješavanje upravo ovog problema, React Router. React Router nudi nekoliko JavaScript React komponenti[30] za ostvarivanje opisanog ponašanja, a one koje klijentska aplikacija koristi su: *BrowserRouter*, *Routes* i *Route*.

```
function App() {
  return (
    <BrowserRouter>
      <Routes>
        <Route path="/home" element={<HomePage />} />
        <Route path="*" element={<PageNotFound />} />
      </Routes>
    </BrowserRouter>
  );
}

export default App;
```

Kod 4.18 Primjer korištenja React Router knjižnice

Kod 4.18 Primjer korištenja React Router knjižnice prikazuje pravilno korištenje komponenti React Router knjižnice za definiranje klijentskih ruta. Navedeni primjer definira ponašanje za dvije moguće rute. Ukoliko korisnik posjeti rutu „/home“, prikazana će mu biti *HomePage* komponenta, dok će mu za bilo koju drugu rutu biti prikazana *PageNotFound* komponenta. Koristeći navedeni princip, s puno više ruta i kompleksnosti komponenti, postignuto je preusmjeravanje na strani klijenta.

[30] https://reactjs.org/docs/components-and-props.html

Ovakav način preusmjeravanja donosi određene izazove, a jedan od njih je zabrana pristupa stranicama koje su namijenjene korisničkim računima sa specifičnim ulogama. Primjerice, vrijednost rute za pristup organizacijskim izvještajima je „organization/reports“. Pristup ovoj ruti potrebno je zabraniti za sve korisničke račune čija uloga nije Poslodavac. U poslužiteljima koji ne predstavljaju *RESTful* web API-je za postizanje ovakvog ponašanja u ASP.NET Core programskom okviru koriste se razne metode za preusmjeravanje korisničkih zahtjeva, no za postizanje navedenog rezultata u aplikacijama koje podržavaju preusmjeravanje samo na strani klijenta potrebno je na klijentskoj strani implementirati logiku za preusmjeravanje korisnika. Korištenjem React Router knjižnicu, ovakvo ponašanje je u aplikaciji postignuto kroz definiranje React komponenti čija je glavna svrha provjeriti ulogu korisničkog računa i prikazati korisniku određenu stranicu ukoliko uloga računa smije pristupiti određenoj ruti, odnosno preusmjeriti korisnički zahtjev na rutu kojoj smije pristupiti ili informirati korisnika o pristupu na zabranjenu rutu. U tu svrhu u klijentskoj aplikaciji su definirane dvije React komponente: *PublicRoute* i *ProtectedRoute*. Zadaća *PublicRoute* komponente je zabraniti pristup stranicama unutar aplikacije koje su namijenjene korištenju od strane neprijavljenih korisnika. Stranica koja sadrži obrazac za prijavu korisnika je primjer rute koja ne bi trebala biti posjećena od strane korisnika koji su se uspješno prijavili u aplikaciju.

```
<BrowserRouter>
  <Routes>
    <Route path="/signin" element={
      <PublicRoute>
            <SignIn />
      </PublicRoute>
      } />
  </Routes>
</BrowserRouter>
```

Kod 4.19 Definicija rute koristeći *PublicRoute* komponentu

Primjer Kod 4.19 Definicija rute koristeći *PublicRoute* komponentu prikazuje implementaciju ograničavanja pristupa na razini klijentske aplikacije. Kada korisnik posjeti rutu čija je vrijednost „/signin“, zbog vrijednosti svojstva (engl. *property*) *element* definiranog na komponenti *Route*, u aplikaciji će biti prikazana React komponenta `<PublicRoute><SignIn/></PublicRoute>`. Shodno tome, u `PublicRoute`

komponenti nalazi se logika za preusmjeravanje korisnika ukoliko je prijavljen, odnosno prikaz vrijednosti *SignIn* komponente ukoliko korisnik nije prijavljen.

```
export default function PublicRoute(props) {
   const { user } = React.useContext(UserContext);

   if (user) {
      return <Navigate to="/home" />;
   } else {
      return props.children;
   }
};
```

Kod 4.20 Definicija *PublicRoute* komponente

Iz prethodnog primjera vidljivo je da se u *PublicRoute* komponenti provjerava vrijednost varijable `user`. Naime, prilikom uspješne prijave ili registracije, postavlja se vrijednost navedene varijable, a u prethodnom poglavlju spomenuto je da se ista vrijednost tada sprema u kontekst. Posljedično tome, u komponenti *PublicRoute* provjerava se vrijednost varijable `user` te ukoliko je njena vrijednost *truthy*[31], korisnika se preusmjerava na rutu „/home" koristeći React Router komponentu *Navigate*. U protivnom, ukoliko korisnik nije prijavljen, biti će mu prikazana komponenta koja se u hijerarhijskom stablu nalazi direktno ispod komponente *PublicRoute*. što znači da ukoliko je hijerarhijsko stablo organizirano kao u primjeru Kod 4.20 Definicija *PublicRoute* komponente, korisniku će biti prikazana vrijednost *SignIn* komponente.

Komponenta *ProtectedRoute* radi na sličan način, samo što ima drugačiju svrhu od komponente *PublicRoute*. Zadaća *ProtectedRoute* komponente je zabraniti pristup rutama koje zahtijevaju da je korisnik uspješno prijavljen u aplikaciji i zabraniti pristup rutama koje su namijenjene samo računima s određenim korisničkim ulogama. To znači da ukoliko korisnik nije prijavljen u aplikaciju te preko adresne trake pokuša posjetiti URL koji zahtjeva da korisnik bude prijavljen, komponenta će ga korištenjem React Router knjižnice preusmjeriti na rutu za prijavu u aplikaciju. Drugi mogući scenarij je da je korisnik prijavljen u aplikaciju s korisničkim računom određene uloge, no pokušava pristupiti stranici koja je namijenjena korisničkim računima s drugačijom ulogom. Ukoliko pokuša posjetiti navedenu

[31] https://developer.mozilla.org/en-US/docs/Glossary/Truthy

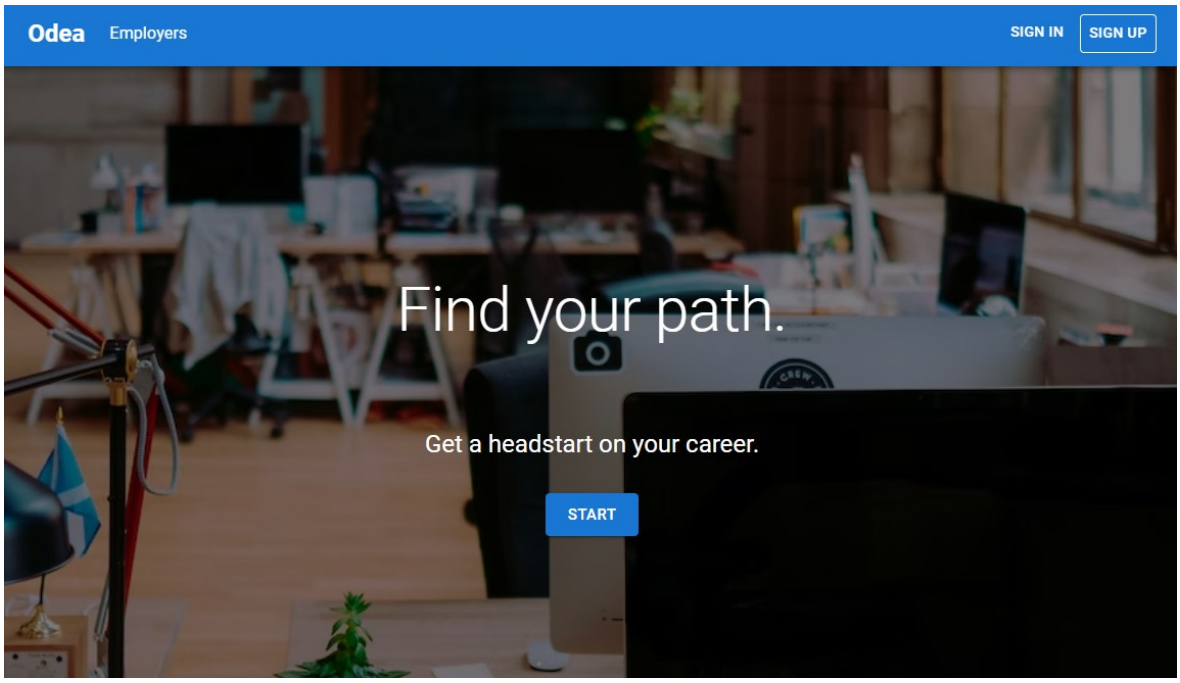
stranicu, biti će mu prikazana komponenta koja služi za informiranje korisnika o pristupu zabranjenoj ruti. Na ovaj način je postignuto preusmjeravanje na strani klijenta. Nadalje, sigurnosni mehanizmi zabrane pristupa opisani u ovom poglavlju nisu dovoljni iz razloga što se cijeli aplikacija i dalje izvodi u web pregledniku korisnika. Stoga je potrebno sve sigurnosne mehanizme implementirati i na strani poslužitelja.

4.6.3. Dizajn sučelja

Dizajn korisničkog sučelja (engl. *user interface*, UI) se fokusira na iščekivanje akcija koje korisnici mogu odraditi kroz stvaranje sučelja koje sadrži elemente koji su pristupačni, razumljivi te se koriste za olakšavanje izvođenja navedenih akcija[32]. Za realizaciju dizajna korištena je JavaScript knjižnica Material UI. Knjižnica se sastoji od React komponenti koje implementiraju Googleov Material Design[33] sustav. Material UI je samo jedno od mogućih rješenja za dizajn. Jedan od glavnih razloga korištenja knjižnica za dizajn je smanjenje vremena potrebnog za razvoj korisničkog sučelja korištenjem React komponenti dostupnih u samim knjižnicama. Bez korištenja ovakvog alata, izgled i ponašanje svakog elementa korisničkog sučelja morali bi biti manualno definirani te bi samim time bila povećana veličina i kompleksnost projekta.

U prethodnom poglavlju navedeno je da klijentska aplikacija sadrži distinkciju između javnih i zaštićenih ruta, odnosno ruta koje zahtijevaju da korisnik nije prijavljen u aplikaciji i ruta koje zahtijevaju uspješnu prijavu korisnika. Iz tog razloga komponente korištene za izradu korisničkog sučelja javnih ruta se znatno razliku od onih korištenih za izradu sučelja zaštićenih ruta.

[32] https://www.usability.gov/what-and-why/user-interface-design.html
[33] https://mui.com/material-ui/getting-started/overview/



Slika 10 Početna stranica aplikacije

Prethodno spomenuti fokus na korisničke akcije vidljiv je na početnoj stranici aplikacije prikazanoj na slici Slika 10 Početna stranica aplikacije. Stranica sadrži alatnu traku koja omogućava pristup sučelju za registraciju novih korisnika ili prijavu postojećih korisnika. Korisnicima koji predstavljaju organizacije se također nudi posebna stranica a pozvani su ju posjetiti prelaskom miša na riječ *Employers*, koja je vidljiva na slici. Prateći trenutne trendove u razvoju web aplikacija, početna stranica također sadrži pozadinsku sliku čiji je cilj simbolizirati glavnu tematiku aplikacije – potražnju poslova. Iščekivanje određene korisničke akcije također je iskazano gumbom s tekstom „Start“ koji je u središtu korisnikove pozornosti s ciljem angažiranja korisnika.

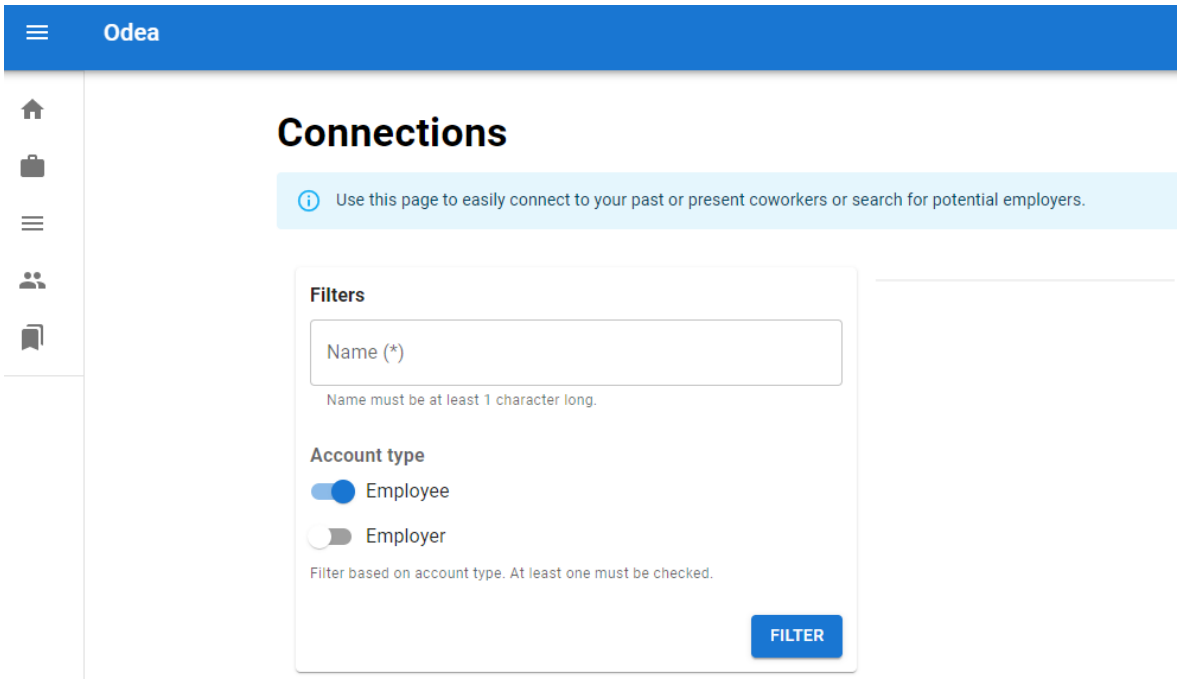
Uzimajući u obzir akcije koje korisnik može poduzeti u aplikaciji nakon uspješne prijave, definirane su komponente koje su zajedničke većini stranica unutar aplikacije. Iz tog razloga, potrebno je izdvojiti komponentu *Layout* čija je glavna zadaća stvaranje konzistentnog korisničkog sučelja.

```
export default function Layout(props) {
    return (
        <Box>
            <Toolbar />
            <Container>
                <PageTitle title={props.pageTitle} />
                {props.children}
```

```
                </Container>
            </Box>
        );
    }
```

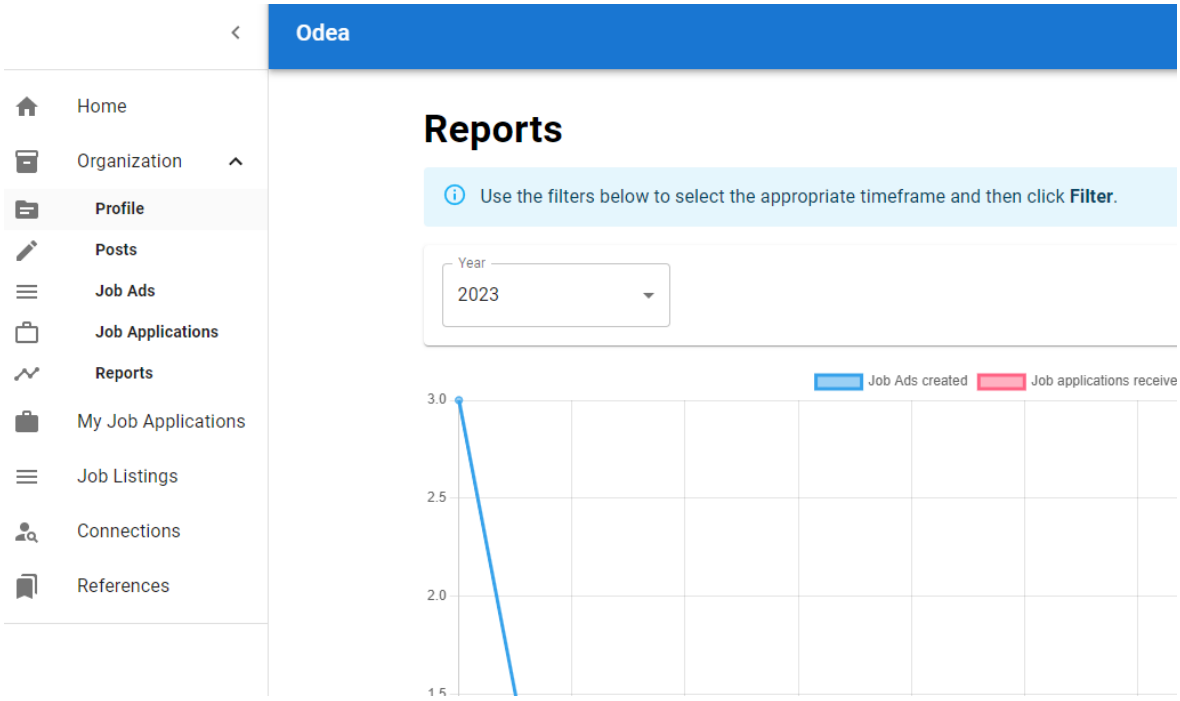
Kod 4.21 Definicija *Layout* komponente

Bitno je naznačiti da navedena komponenta znatno utječe ne samo na konzistentnost dizajna već i na ponovnu iskoristivost koda. Naime, nakon uspješne prijave u aplikaciju, svaka prikazana stranica koristi navedenu komponentu sa zadatkom postizanja ujednačenosti među stranicama i smanjenjem kompleksnosti koda. Navedena komponenta također omogućuje kontrolu toka korisničkih akcija korištenjem komponente *Toolbar* koja predstavlja glavni aplikacijski izbornik. Komponenta *Layout* se u aplikaciji koristi kao omotač (engl. *wrapper*) oko samog dizajna određenih stranica te se na taj način i koristi: `<Layout pageTitle="Reports"><OrganizationReports/></Layout>`. Ovakav dizajn korisničkog sučelja, organizacijom React komponenti u manje, ponovno iskoristive komponente, znatno je smanjio količinu aplikacijskog koda potrebnog za realizaciju korisničkih zahtjeva.



Slika 11 Prikaz korisničkog sučelja aplikacije s korisničkom ulogom Posloprimac

Prethodna slika prikazuje stranicu unutar aplikacije koja se koristi za povezivanje korisnika. Na slici je vidljiva i prethodno spomenuta *Toolbar* komponenta koja se nalazi na lijevoj strani ekrana. Korištenjem CSS animacija, navedena komponenta se u svakom trenutku može nalaziti u otvorenom ili zatvorenom stanju. U zatvorenom stanju na komponenti su prikazane samo ikone koje predstavljaju određene akcije, dok su u otvorenom stanju vidljiva i imena određenih akcija. Potrebno je naznačiti da akcije koje se nalaze u izborniku ovise o ulozi korisničkog računa. Razlika u akcijama ovisno o ulozi korisničkog računa trenutno prijavljenog korisnika vidljiva je i na sljedećoj slici.



Slika 12 Prikaz korisničkog sučelja aplikacije s korisničkom ulogom Poslodavac

Slika 12 Prikaz korisničkog sučelja aplikacije s korisničkom ulogom Poslodavac prikazuje dodatne akcije vidljive korisnicima s ulogom Poslodavac. Na slici je također vidljiva i prethodno spomenuta komponenta *Toolbar* u otvorenom stanju. U skladu s korisničkim zahtjevima popisanim u poglavlju Posloprimac, bilo je potrebno omogućiti pristup podacima o organizacijama. Iz prethodnog primjera, vidljivo je da je navedeni zahtjev ispunjen dodatnim izbornikom koji je označen akcijom naziva *Organization*, a nalazi se u glavnom izborniku aplikacije. Na taj način korisnici koji predstavljaju organizacije imaju jednostavan način pristupa navedenim podacima.

5. Postavljanje u rad i održavanje aplikacije

Kako bi korisnici mogli posjetiti aplikaciju stvorenu u sklopu ovog rada koristeći web preglednik, istu je potrebno postaviti u rad, tj. omogućiti posluživanje aplikacijskog koda opisanog u radu na udaljenom poslužitelju. U tu svrhu korištena je Microsoftova Azure platforma.

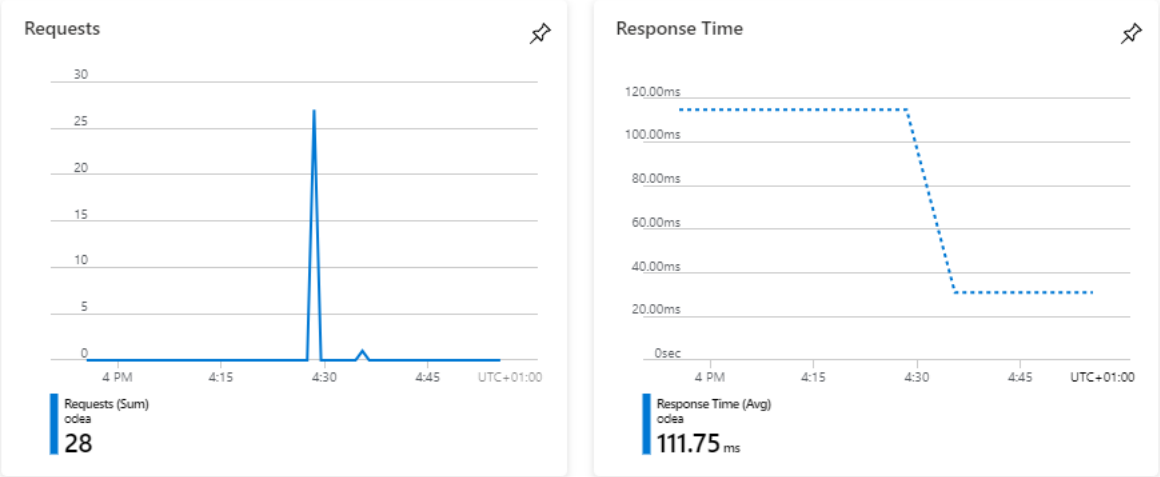
Azure je javna *cloud* platforma koja nudi širok izbor servisa i resursa za omogućavanje ubrzanog razvoja aplikacija[34]. Azure servisi koji su korišteni za posluživanje klijentske i poslužiteljske aplikacije te baze podataka su:

- Azure App Service
- Azure SQL

Azure App Service je naziv za platformu koja služi za posluživanje web aplikacija i API-ja na Azuru[35]. Korištenjem ovog servisa klijentska i poslužiteljska aplikacija posluženi su na udaljenim virtualnim mašinama u visoko dostupnom okruženju. Okruženje u kojem su poslužene aplikacije opisane u radu se razlikuje od okruženja koje je korišteno za razvoj. Okruženje korišteno za razvoj često se naziva lokalno okruženje, dok se okruženje kojem korisnici pristupaju aplikaciji naziva produkcijskim okruženjem. Kako bi poslužitelj u produkcijskom okruženju mogao pristupiti bazi podataka, istu je potrebno postaviti u rad kao i aplikacije. U tu svrhu koristi se poseban Azure servis koji predstavlja SQL Server bazu podataka na *cloudu* naziva Azure SQL. Korištenjem navedenog servisa, stvorena je baza podataka i izvršene su SQL skripte potrebne za stvaranje tablica na bazi podataka.

Korištenjem Azure platforme znatno je olakšan proces održavanja aplikacije. Naime, Azure je zadužen za upravljanje operacijskim sustavima virtualnih mašina na kojima su aplikacije poslužene, a upravljanje i praćenje stanja aplikacije omogućeno je kroz web aplikaciju naziva Azure Portal. Navedena aplikacija pruža korisničko sučelje za upravljanje resursima alociranima unutar platforme. Azure Portal također sadrži mnoge značajke koje olakšavaju nadgledanje (engl. *monitoring*) aplikacija te pruža uvid u razne statističke podatke vezane uz stanje virtualnih mašina na kojima se izvršava aplikacijski kod.

[34] https://learn.microsoft.com/en-us/dotnet/azure/intro
[35] https://learn.microsoft.com/en-us/dotnet/azure/key-azure-services



Slika 13 Primjer metrika klijentske aplikacije

Slika 13 Primjer metrika klijentske aplikacije prikazuje samo neke od metrika dostupnih u Azure Portalu. Navedene metrike prikazuju broj HTTP zahtjeva prema klijentskoj aplikaciji i prosječno vrijeme potrebno za odgovor na HTTP zahtjev.

Nadalje, održavanje aplikacije također podrazumijeva i dodavanje novih značajki. Proces dodavanja novih značajki na Azuru je podržan kroz kontinuiranu integraciju. Kontinuirana integracija je proces automatizacije integracije promjena u aplikacijskom kodu[36]. Kontinuirana integracija smanjuje manualan rad potreban za uvođenje novih promjena u aplikacijskom kodu u produkcijsko okruženje.

[36] https://www.atlassian.com/continuous-delivery/continuous-integration

Zaključak

Rezultata rada je proizvod čija je svrha olakšati povezivanje poslodavaca i posloprimaca s ciljem zapošljavanja. Proizvod je implementiran web aplikacijom. Primjenom Microsoftove tehnologije i praćenjem ustaljenih praksi u programiranju stvorena je web aplikacija koju je lako moguće nadograditi s dodatnim značajkama. Mogućnost uvođenja novih značajki u aplikaciju prije svega je podržana odlukom o separaciji rješenja u dvije odvojene aplikacije – aplikaciju koja predstavlja klijenta i aplikaciju koja predstavlja poslužitelja.

Ovakav pristup omogućio je izdvajanje aplikacijskog koda vezanog uz razvoj korisničkog sučelja od aplikacijskog koda koji sadrži poslovnu logiku poslužitelja, a implementacijom ovog pristupa znatno je olakšan proces dodavanja novih značajki. Prilikom implementacije, korišteni su razni principi i oblikovni obrasci s ciljem stvaranja aplikacijskog koda koji je podložan čestim promjenama. Navedeni način razvoja omogućio je stvaranje proizvoda sa znatnim mogućnostima skalabilnosti. Nadalje, korištenje *cloud* platforme za postavljanje aplikacije u rad uvelike je utjecalo na mogućnosti skalabilnosti iste. Upotrebom Azure platforme, olakšani su kritični infrastrukturni procesi koji dolaze uz implementaciju web aplikacije, a aplikacija s lakoćom može biti poslužena na više lokacija s više resursa.

Popis kratica

AJAX	*Asynchronous JavaScript and XML*	Asinkroni JavaScript i XML
API	*Application programming interface*	Aplikacijsko programsko sučelje
CORS	*Cross-Origin Resource Sharing*	Dijeljenje resursa unakrsnog podrijetla
CSS	*Cascading Style Sheets*	Kaskadni stilizacijski list
ER	*Entity relationship*	Veza entiteta
HTTP	*Hypertext Transfer Protocol*	Hipertekst prijenosni protokol
JWT	*JSON Web Token*	JSON mrežni znak
LINQ	*Language-Integrated Query*	Jezično integrirani upit
OAS	*OpenAPI Specification*	OpenAPI specifikacija
ORM	*Object-relational mapper*	Objektno relacijski maper
REST	*Representational state transfer*	Prijenos reprezentacijskog stanja
SPA	*Single page application*	Jednostranična aplikacija
T-SQL	*Transact-Structured Query Language*	Transakcijsko strukturirani upitni jezik
UI	*User interface*	Korisničko sučelje
URI	*Uniform Resource Identifier*	Jedinstveni identifikator resursa
URL	*Uniform Resource Locator*	Jedinstveni lokator resursa
XML	*Extensible Markup Language*	Jezik za označavanje podataka

Popis slika

Popis tablica

Popis kôdova

Literatura

[1] *MojPosao*, https://www.moj-posao.net/, siječanj 2023.

[2] *MojPosao*, https://www.moj-posao.net/hr/hercul/sale, siječanj 2023.

[3] *MojPosao*, https://www.moj-posao.net/Poslodavci/CVs/Offers/, siječanj 2023.

[4] *MojPosao*, https://www.moj-posao.net/pregled-usluga/poslovi, siječanj 2023.

[5] *DZone*, SERGEY VALUY, https://dzone.com/articles/the-comparison-of-single-page-and-multi-page-appli, siječanj 2023.

[6] *Microsoft*, https://learn.microsoft.com/en-us/ef/core/, siječanj 2023.

[7] *Medium*, SHIF BEN AVRAHAM, https://medium.com/extend/what-is-rest-a-simple-explanation-for-beginners-part-1-introduction-b4a072f8740f, siječanj 2023.

[8] JONES M., BRADLEY J., and N. SAKIMURA, JSON Web Token (JWT), RFC 7519, DOI 10.17487/RFC7519, 2015, https://www.rfc-editor.org/info/rfc7519, siječanj 2023.

[9] *Auth0*, https://jwt.io/introduction, siječanj 2023.

[10] *Microsoft*, https://learn.microsoft.com/en-us/aspnet/web-api/overview/web-api-routing-and-actions/attribute-routing-in-web-api-2, siječanj 2023.

[11] *Microsoft*, https://learn.microsoft.com/en-us/aspnet/core/mvc/models/model-binding?view=aspnetcore-7.0, siječanj 2023.

[12] *Microsoft*, https://learn.microsoft.com/en-us/aspnet/core/security/cors, siječanj 2023.

[13] *MDN Web Docs*, https://developer.mozilla.org/en-US/docs/Web/HTTP/CORS, veljača 2023.

[14] *Microsoft*, https://learn.microsoft.com/en-us/aspnet/core/performance/caching/memory, veljača 2023.

[15] *Microsoft*, https://learn.microsoft.com/en-us/dotnet/api/system.threading.timer, veljača 2023.

[16] *Microsoft*, https://learn.microsoft.com/en-us/aspnet/core/mvc/controllers/filters, veljača 2023.

[17] RICHARDS M. *Software Architecture Patterns.* O'Reilly Media Inc, 2015

[18] GAMMA E., HELM R., JOHNSON R., VLISSIDES J. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional, 1994

[19] *Microsoft*, https://learn.microsoft.com/en-us/dotnet/core/extensions/dependency-injection, veljača 2023.

[20] *Microsoft*, https://learn.microsoft.com/en-us/dotnet/core/extensions/dependency-injection#service-lifetimes, veljača 2023.

[21] *Microsoft*, https://learn.microsoft.com/en-us/dotnet/architecture/microservices/microservice-ddd-cqrs-patterns/infrastructure-persistence-layer-design, veljača 2023.

[22] *Hasura*, https://hasura.io/learn/database/microsoft-sql-server/er-modeling, veljača 2023.

[23] *Microsoft*, https://learn.microsoft.com/en-us/ef/core, veljača 2023.

[24] *MDN Web Docs*, https://developer.mozilla.org/en-US/docs/Web/Guide/AJAX, veljača 2023.

[25] *MDN Web Docs*, https://developer.mozilla.org/en-US/docs/Web/API/XMLHttpRequest, veljača 2023.

[26] *MDN Web Docs*, https://developer.mozilla.org/en-US/docs/Web/API/Fetch_API/Using_Fetch, veljača 2023.

[27] *Meta*, https://beta.reactjs.org/learn/passing-props-to-a-component, veljača 2023.

[28] *Meta*, https://beta.reactjs.org/learn/passing-data-deeply-with-context, veljača 2023.

[29] *Meta*, https://reactjs.org/docs/context.html, veljača 2023.

[30] *Meta*, https://reactjs.org/docs/components-and-props.html, veljača 2023.

[31] *MDN Web Docs*, https://developer.mozilla.org/en-US/docs/Glossary/Truthy, veljača 2023.

[32] *User Interface Design Basics*, https://www.usability.gov/what-and-why/user-interface-design.html, veljača 2023.

[33] *Material UI*, https://mui.com/material-ui/getting-started/overview/, veljača 2023.

[34] *Microsoft*, https://learn.microsoft.com/en-us/dotnet/azure/intro, veljača 2023.

[35] *Microsoft*, https://learn.microsoft.com/en-us/dotnet/azure/key-azure-services, veljača 2023.

[36] *Atlassian*, https://www.atlassian.com/continuous-delivery/continuous-integration, veljača 2023.