Web sustav za upravljanje korisnicima unutar telekomunikacijske tvrtke

Zorja, Alen

VELEUČILIŠTE U BJELOVARU

PREDDIPLOMSKI STRUČNI STUDIJ RAČUNARSTVO

Web sustav za upravljanje korisnicima unutar telekomunikacijske tvrtke

Završni rad br. 09/RAČ/2021

Alen Zorja

Bjelovar, rujan 2021.

obrazac ZR - 001

Veleučilište u Bjelovaru

Trg E. Kvaternika 4, Bjelovar

1. DEFINIRANJE TEME ZAVRŠNOG RADA I POVJERENSTVA

Kandidat: **Zorja Alen** Datum: 27.08.2021. Matični broj: 001998

JMBAG: 0314020049

Kolegij: **.NET PROGRAMIRANJE**

Naslov rada (tema): **Web sustav za upravljanje korisnicima unutar telekomunikacijske tvrtke**

Područje: **Tehničke znanosti** Polje: **Računarstvo**

Grana: **Programsko inženjerstvo**

Mentor: **Krunoslav Husak, dipl.ing.rač.** zvanje: **predavač**

Članovi Povjerenstva za ocjenjivanje i obranu završnog rada:

1. **Tomislav Adamović, mag.ing.el., predsjednik**
2. **Krunoslav Husak, dipl.ing.rač., mentor**
3. **Ante Javor, struč.spec.ing.comp., član**

2. ZADATAK ZAVRŠNOG RADA BROJ: 09/RAČ/2021

U radu je potrebno izraditi web sustav za upravljanje korisnicima u jednoj većoj telekomunikacijskoj tvrtki. Web sustav mora korisnicima omogućiti pregled vlastitih korisničkih podataka, usluga koje koriste ili žele koristiti te njihovu aktivaciju i deaktivaciju. U radu će se proći svi stadiji razvoja takvog sustava: od proučavanja i analize projektnog zadatka, procjenjivanje potrebno posla, izradu tehničke specifikacije za projekt do implementacije samog web sustava. Web sustav će se sastojati od klijentske aplikacije razvijene u modernom web razvojnom okviru, pozadinskog sučelja (API) u kojemu se provodi poslovna logika aplikacije, a podaci će biti spremljeni u bazi podataka otvorenog koda. Zbog pojednostavljenja uspostavljanja razvojne okoline te produkciju (cloud), svi dijelovi aplikacije će biti virtualizirani u kontejnere (engl. container) koristeći Docker.

Zadatak uručen: 27.08.2021. Mentor: **Krunoslav Husak, dipl.ing.rač.**

REPUBLIKA HRVATSKA 2 VELEUČILIŠTE U BJELOVARU

Sadržaj

Popis slika

1. UVOD

U ovom završnom radu obrađuje se tema razvoja *self-care* web sustava za jednu veću telekomunikacijsku tvrtku. U izradi završnog rada, prolazi se kroz sve stadije razvoja web aplikacije. Od proučavanja i analize projektnog zadatka, procjenjivanja vremena potrebnog za obradu svih dijelova aplikacije, te konačne implementacije samog web sustava.

Web sustav omogućava korisnicima pregled vlastitih korisničkih podataka, usluga koje imaju ili mogu imati, te aktivaciju ili deaktivaciju istih. Klijentska aplikacija izrađena je u .NET Core Web API tehnologiji. Baza podataka koja se koristi je PostgreSQL relacijska baza podataka. Korisničko sučelje izrađeno je u React JavaScript biblioteci.

Kompletna aplikacija (korisničko sučelje, *backend*, baza podataka) virtualizirana je u kontejnere koristeći Docker zbog pojednostavljenja uspostavljanja razvojne okoline.

Izradom ovog završnog rada obradila se tema zahtjeva za izradu Web aplikacije koja se može očekivati u stvarnom životu. Tehnologije za izradu ove aplikacije odabrane su zbog toga što se smatraju najboljim odabirom za izradu ove vrste Web aplikacije, te se stečeno znanje može upotrijebiti u poslovnim situacijama s kojima se moguće susresti u budućnosti.

2. ASP.NET CORE

ASP.NET [1] je popularan programski okvir (engl. *framework*) koji je kreiran za razvoj web aplikacija na .NET platformi. Ovom programskom okviru u glavnom fokusu su performanse. ASP.NET Core je verzija ASP.NET programskog okvira koja je prvi put objavljena 2016 godine. Ova verzija ASP.NET Core programskog okvira je redizajn prijašnje verzije ASP.NET programskog okvira koja je bila dizajnirana samo za upotrebu na Windows operativnim sustavima.[1]

Zadnja verzija ASP.NET Core programskog okvira objavljena je 2020 godine te joj je dodijeljen broj verzije 5.0. Na slici 2.1 nalazi se logotip .NET programskog okvira.

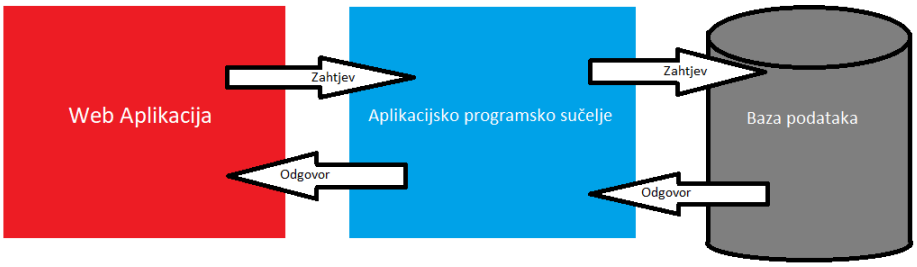
Slika 2.1. Logotip ASP.NET Core programskog okvira

ASP.NET Core pruža mogućnost postojanja više različitih verzija ASP.NET Core programskog okvira jedne uz drugu na istom serveru. Ovakav pristup omogućava jednoj ili više aplikacija da se nadograde na noviju verziju programskog okvira, a ostatak aplikacija na serveru se i dalje može pokretati na starijim verzijama programskog okvira.

2.1 Web programsko sučelje aplikacije

Aplikacijsko programsko sučelje [2, 3] (engl. *Application programming interface, API*) je posrednik između Web aplikacije i baze podataka kao što se može vidjeti na slici 2.2 Web API koristi HTTP (*HyperText Transfer Protocol*) protokol kako bih mogao uspostaviti komunikaciju između klijenata i Web stranica u svrhu dohvaćanja podataka.

[1] Operacijski sustav je skup mnoštva programa koji upravljaju računalnim sklopovljem.



Slika 2.2. Web API shema[2]

Web API je vrlo popularan zbog toga što što omogućuje pristup aplikaciji s više različitih uređaja time što odgovara na zahtjeve s JSON (engl. *JavaScript Object Notation*) formatom.


```
{
  "id": 1,
  "name": "Admin",
  "surname": "Admin",
  "oib": 99999999999,
  "emailAddress": "admin@gmail.com",
  "accounts": [
    {
      "id": 29,
      "connectionAddress": "Suhaja 123",
      "preferredContactType": 1,
      "temporaryExpulsion": false,
      "doNotCall": false,
      "phoneNumber": "0998413242",
      "userId": 1,
      "ministrations": [
        {
          "id": 65,
          "ministrationName": "Internet",
          "contractObligation": 5,
          "accountId": 29
        }
      ]
    }
  ],
  "role": "Admin"
}
```


Slika 2.3. JSON odgovor

Na slici 2.3 naveden je primjer JSON odgovora koji se koristi u ovom završnom radu. U ovom primjeru Web API šalje JSON odgovor s podatcima o korisniku iz Postgre relacijske baze podataka. Odgovor se sastoji od glavnog modela naziva *User*, *dijete* modela *Account*, te dijete modela *Ministration*. U ovom slučaju može se vidjeti kako korisnik imena Admin ima jedan dijete model *Account*, te on ima jedan dijete model *Ministration*. Roditelj

[2] Shema je pojednostavnjen grafički prikaz funkcioniranja nekog sustava.

model *User* može imati nebrojeno mnogo dijete modela *Accounta*, te *Account* model može imati nebrojeno mnogo *Ministration* dijete modela.

JSON format osmišljen je za razumljivu i lako čitljivu razmjenu podataka, te je odličan izbor za odgovore zbog toga što je lako čitljiv za ljude. Objekt u JSON formatu počinje s *{*, te završava s *}*. JSON format se sastoji od para imena (engl. *name*) i vrijednosti (engl. *value*). Ime i vrijednost dijeli dvotočka, a parovi su odvojeni sa zarezom.

2.1.1 Modeli

Modeli definiraju oblik podataka koji se spremaju u odabranu bazu podataka. Svaki model potrebno je odvojiti u posebnu klasu, te je svaku klasu potrebno odvojiti u posebnu datoteku, a sve datoteke modela moraju se kreirati u mapi naziva *Models*. Ovim načinom odvajanja koda, kod je lako čitljiv, te se vrlo lako mogu promijeniti potrebni dijelovi koda.

```
7   namespace Self_Care_app_azorja.Models
8   {
        17 references
9       public class User
10      {
11          [Key]
            4 references
12          public long Id { get; set; }
13          [Required]
            5 references
14          public string Name { get; set; }
15          [Required]
            5 references
16          public string Surname { get; set; }
17          [Required]
            5 references
18          public long OIB { get; set; }
19          [Required]
            7 references
20          public string EmailAddress { get; set; }
            4 references
21          public virtual List<Account> Accounts { get; set; }
            1 reference
22          public string Role { get; set; }
            4 references
23          public string Password { get; set; }
24      }
25  }
```

Slika 2.4. *User* model

Na slici 2.4 nalazi se *User* model koji se sastoji od informacija usko vezanih uz korisnika. *Id* je postavljen kao ključ (engl. *key*), to ga čini jedinstvenim (engl. *unique*), te se on koristi kao podatak koji služi za identificiranje jedinstvenog korisnika.

Linija 21 definira vezu modela *User* i modela *Account*. U ovom slučaju vrsta veze je *one to many*, što znači da jedan korisnik može imati više računa, ali pojedini račun može

imati samo jednog korisnika, stoga će *Account* model morati implementirati informaciju o identifikatoru kako bih se mogao referencirati na korisnika kojem pripada.

```
7   namespace Self_Care_app_azorja.Models
8   {
        18 references
9       public class Account
10      {
11          [Key]
            4 references
12          public long Id { get; set; }
13          [Required]
            5 references
14          public string ConnectionAddress { get; set; }
            5 references
15          public int PreferredContactType { get; set; }
            5 references
16          public bool TemporaryExpulsion { get; set; }
            5 references
17          public bool DoNotCall { get; set; }
18          [Required]
            5 references
19          public string PhoneNumber { get; set; }
            2 references
20          public long UserId { get; set; }
            0 references
21          public virtual User User { get; set; }
            5 references
22          public virtual List<Ministration> Ministrations { get; set; }
23      }
24  }
```

Slika 2.5. *Account* model

Na slici 2.5 nalazi se *Account* model koji je povezan s *User* modelom te *Ministration* modelom. Za ispravan rad *one to many* veze između *User* modela i *Account* modela, potrebno je postaviti svojstvo *UserId*, te postaviti objekt *Usera*.

Zadnja linija ovog modela također stvara *one to many* vezu poput one između *Usera* i *Accounta*, s *Ministration* modelom.

2.1.2 Servisi

Na slici 2.6 nalazi se jedan od mnogo servisa koji upravljaju podatcima u ovoj aplikaciji. *UserService* implementira *dbContext* te ga koristi za direktno upravljanje s podatcima u bazi.

U metodi *GetAll* može se vidjeti primjer korištenja *_dbContext* instance u direktnoj komunikaciji s bazom podataka. U *data* se spremaju svi korisnici, uključujući sve njihove račune i usluge, te se postavljaju u listu.

Nakon toga podatci se mapiraju pomoću *AutoMappera* na model koji je određen u *User DTO*[3]. Mapiranjem modela filtriraju se podatci koji se ne trebaju ili ne smiju prikazivati na korisničkom sučelju te se tek onda šalju na korisničko sučelje.

```
15      public class UserService : IUserService
16      {
17          //private readonly UserCreateValidator validator = new UserCreateValidator();
18          private readonly SelfCareAppDbContext _dbContext;
19          private readonly IMapper _mapper;
            0 references
20          public UserService(SelfCareAppDbContext dbContext, IMapper mapper)
21          {
22              _dbContext = dbContext;
23              _mapper = mapper;
24          }
25          //Get all users
            2 references
26          public IEnumerable<UserDto> GetAll()
27          {
28              var data = _dbContext.Users
29                  .Include(x => x.Accounts)
30                      .ThenInclude(account => account.Ministrations)
31                  .ToList();
32
33              var mappedResults = _mapper.Map<IEnumerable<UserDto>>(data);
34              return mappedResults;
35          }
```

Slika 2.6. Servis *UserService*

Na slici 2.7 nalazi se *Login* metoda koja se koristi kod prijave korisnika u sustav. Metodi se prosljeđuje email adresa i lozinka od korisnika.

```
112         //Login user
            2 references
113         public UserDto Login(string emailAddress, string password)
114         {
115             var user = _dbContext.Users
116                 .Where(x => x.EmailAddress == emailAddress)
117                 .SingleOrDefault();
118             if(user!=null && PasswordHasher.Verify(password, user.Password))
119             {
120                 var mappedResults = _mapper.Map<UserDto>(user);
121                 return mappedResults;
122             }
123             else
124             {
125                 return null;
126             }
127         }
```

Slika 2.7. *Login* metoda

[3] DTO (engl. *Data Transfer Object*) služi za promjenu oblika modela prije prikazivanja istog na korisničkom sučelju.

Pomoću *dbContexta* pokušava se dohvatiti korisnik s email adresom koja je unesena. Ukoliko adresa postoji u bazi podataka, dohvaća se vlasnik email adrese te se provodi validacija lozinke. Ukoliko su oba podatka točna, *Login* metoda vraća mapiranog korisnika na korisničko sučelje. U slučaju da jedan od podataka nije točan, odnosno da email adresa ne postoji u bazi podataka ili da lozinka nije validna, metoda će vratiti *null*.

2.2 Autentifikacija

U slučaju autentifikacije[4] za neku aplikaciju, korisnik unosi neke podatke koji se provjeravaju te se na temelju točnosti podataka utvrđuje identitet korisnika. U ovom Web sustavu usvojena je autentifikacija pomoću email adrese te pripadajuće lozinke za postojećeg korisnika s danom email adresom.

Lozinke u bazi podataka kriptirane su pomoću metode *PasswordHasher* kreirane za stvaranje *hasha* od lozinke unesene od strane administratora tijekom stvaranja novog korisnika. *PasswordHasher* kreira različit zapis u bazi tijekom svakog unosa nove lozinke, pa čak i kada su lozinke potpuno iste. Metoda za kriptiranje[5] lozinke također provjerava ne kriptiranu unesenu lozinku s kriptiranim zapisima u bazi podataka. Unesena lozinka prolazi kroz metodu za kriptiranje te se ta kriptirana verzija uspoređuje sa zapisima u bazi.

```
1 reference
49  public static bool Verify(string password, string hashedPassword)
50  {
51      if (!IsHashSupported(hashedPassword))
52      {
53          throw new NotSupportedException("The hashtype is not supported");
54      }
55      // Extract iteration and base64 string
56      var splittedHashString = hashedPassword.Replace(DefaultHashSignature, "").Split('$');
57      var iterations = int.Parse(splittedHashString[0]);
58      var base64Hash = splittedHashString[1];
59      // Get hash bytes
60      var hashBytes = Convert.FromBase64String(base64Hash);
61      // Get salt
62      var salt = new byte[SaltSize];
63      Array.Copy(hashBytes, 0, salt, 0, SaltSize);
64      // Create hash with given salt
65      var PasswordBasedKeyDerivationFunction = new Rfc2898DeriveBytes(password, salt, iterations);
66      byte[] hash = PasswordBasedKeyDerivationFunction.GetBytes(HashSize);
67      // Get result
68      for (var i = 0; i < HashSize; i++)
69      {
70          if (hashBytes[i + SaltSize] != hash[i])
71          {
72              return false;
73          }
74      }
75      return true;
76  }
```

Slika 2.8. Verifikacija lozinke

[4] Autentifikacija je proces u kojem se mora odrediti identitet korisnika.
[5] Kriptiranje je pretvaranje osjetljivih podataka koji se moraju zaštititi od krađe u nečitljiv oblik za sve one koji nemaju ključ koji ih može dekriptirati.

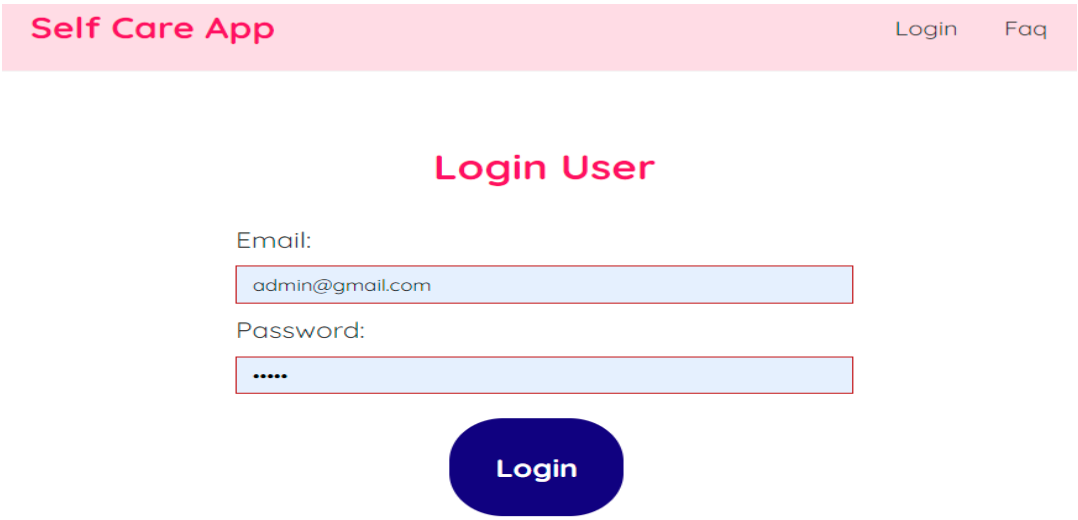
Na slici 2.8 nalazi se primjer metode za provjeru unesene lozinke s lozinkama koje su u bazi podataka korištene u ovom završnom radu.

Nakon uspješne autentifikacije, na temelju uloge identificiranog korisnika, korisnik se preusmjerava na popis svih korisnika sustava u slučaju da mu je dodijeljena uloga administratora ili na informacije o vlastitom korisničkom računu ukoliko mu je dodijeljena uloga običnog korisnika.

Na slici 2.9 nalazi se primjer lozinke „123“ provučene kroz metodu za kriptiranje lozinke, te spremljene u PostgreSQL bazu podataka.

korisnikkorisnic@gmail.com	MOJ_HASHV1$10000$tRWjbXjsvhV1fQv1i/ITSkZkrpLQwne/WS6Z5vy2tb1UO34r	User

Slika 2.9. Primjer kriptirane lozinke



Slika 2.10. Autentifikacija korisnika

Na slici 2.10 nalazi se ekran s dva polja za unos podataka. Korisnik mora unesti svoju email adresu te nakon toga ispravnu lozinku za navedenu email adresu. Nakon uspješne autentifikacije, prikazuje se ekran učitavanja te se korisnik prebacuje na URL određen vlastitom ulogom u aplikaciji.

2.3 Validacija

Kod svih vrsta aplikacija, validacija[6] podataka je jedan od najvažnijih dijelova za pravilno funkcioniranje sustava. Svaki zahtjev se mora validirati prije nego počne procesiranje istog. Neke od čestih provjera koje se koriste tijekom unosa podataka od strane korisnika su provjera je li podatak *null*, premašuje li dužina niza znakova minimalnu ili maksimalnu dozvoljenu dužinu itd.

Na slici 2.11 nalazi se primjer korištenja *fluent* validacije unutar ovog završnog rada. *Fluent* validacija osmišljena je kako bih se validacijska pravila mogla odvojiti od modela te samim time da omogući lakše čitanje i razumijevanje validacijskih pravila koja su postavljena za svaki pojedini model.

```
2 references
public class UserCreateValidator : AbstractValidator<User>
{
    1 reference
    public UserCreateValidator()
    {
        RuleFor(user => user.Name)
            .Cascade(CascadeMode.StopOnFirstFailure)
            .NotEmpty()
            .Length(3,30).WithMessage("Ime mora biti više od 3 a manje od 30 znakova duljine");
        RuleFor(user => user.Surname)
            .NotEmpty()
            .Length(3,30).WithMessage("Prezime mora biti više od 3 a manje od 30 znakova duljine");
        RuleFor(user => user.OIB)
            .NotEmpty()
            .GreaterThan(9999999999).WithMessage("OIB mora biti 11 znamenaka")
            .LessThan(99999999999).WithMessage("OIB mora biti 11 znamenaka");
        RuleFor(user => user.EmailAddress)
            .NotEmpty()
            .Length(10,50).WithMessage("Email adresa mora biti najmanje 10 znakova duga");
        //RuleFor(user => user.Role).NotEmpty().When(x => x.Id != 0);
        RuleFor(user => user.Role)
            .NotNull();
        //RuleFor(user => user.Role).NotEmpty();
        RuleFor(user => user.Password)
            .NotEmpty()
            .MinimumLength(3);
    }
```

Slika 2.11. Validacija *User* modela

Na slici 2.12 nalazi se primjer korištenja *UserCreateValidator* validatora. Objekt korisnika provlači se kroz validator, te se na temelju valjanosti unesenih podataka vraća vrijednost *IsValid* koja može biti istina ili laž. U slučaju validnih podataka, *Create* metoda nastavlja, te poziva *User* servis i stvara novog korisnika. U suprotnom slučaju metoda će

[6] Validacija je postupak u kojem se utvrđuje ili provjerava valjanost nekog danog podatka.

vratiti *BadRequest*[7] s greškom koja pokazuje na podatak o korisniku koji ne zadovoljava postavljena pravila.

```
50          [HttpPost(nameof(Create))]
            1 reference
51          public IActionResult Create(User user)
52          {
53              UserCreateValidator validator = new UserCreateValidator();
54              ValidationResult result = validator.Validate(user);
55              if (result.IsValid)
56              {
57                  ////////////////////////////////////////////////////result.Errors
58                  _userService.Create(user);
59                  return Ok();
60              }
61              return BadRequest(result.Errors);
62          }
```

Slika 2.12. Primjer korištenja *UserCreateValidator* validatora unutar *Create* metode

[7] BadRequest je greška s kodom 400, koja se prikazuje u slučaju greške validacije podataka unesenih od strane korisnika.

3. BAZA PODATAKA

3.1 PostgreSQL relacijska baza podataka

U ovom završnom radu koristi se PostgreSQL relacijska baza podataka koja vuče korijene na modelu klijenta i poslužitelja.

Relacijska baza podataka ima strukturirane podatke spremljenje u obliku tablica, te svako polje u tablici može imati vezu s nekim drugim poljem u potpuno različitoj tablici. Kod nerelacijskih baza podataka tablice se ne koriste, već se koriste kolekcije.

PostgreSQL razvija se od strane PostgreSQL Global Development group, ali je otvorenog koda (engl. *open source*) što znači da je održavana od strane dobrovoljnih programera diljem svijeta. Na slici 3.1 nalazi se [4] logotip PostgreSQL relacijske baze podataka.

Slika 3.1. Logotip[8] PostgreSQL relacijske baze podataka

PostgreSQL baza podataka može komunicirati s aplikacijama stvorenim u mnoštvu programskih jezika zbog svojeg temelja na klijent-poslužitelj modelu.

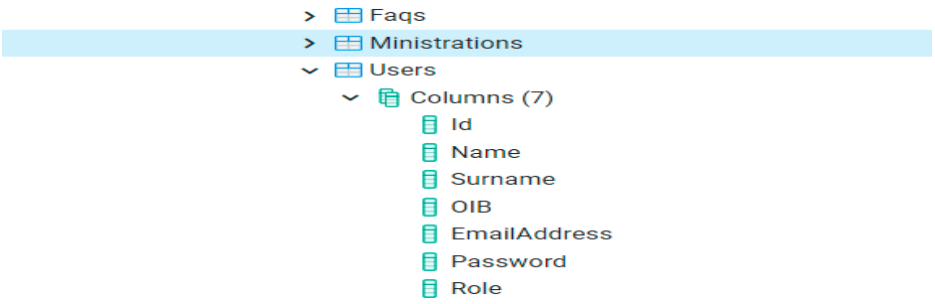
3.1.1 PgAdmin

Na slici 3.2 nalazi se početna stranica PgAdmin grafičkog korisničkog sučelja za prijavu korisnika u sustav. [5] PgAdmin je besplatno grafičko korisničko sučelje (engl. *Graphical user interface*) otvorenog koda, koje se koristi za komunikaciju i upravljanje s Postgre relacijskom bazom podataka i njenim servisima.

[8] Logotip je simbol koji predstavlja neki proizvod ili tvrtku

Slika 3.2. Prikaz prijave u pgAdmin GUI

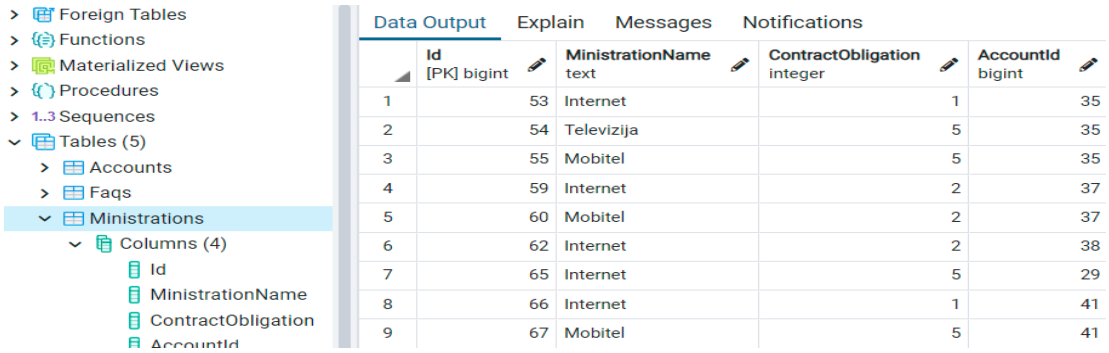
U PgAdminu se na vrlo lak način mogu dohvatiti potrebni podatci. Kada se proširi izbornik *Tables* kao na slici 3.3, moguće je na nekoliko načina prikazati podatke.



Slika 3.3. Primjer prikaza tablica u pgAdmin grafičkom korisničkom sučelju

Jedan od načina dohvaćanja podataka iz pojedinih tablica je da se odabere tablica, odabere *tools* u glavnom izborniku te se odabere *Query tool* iz ponuđenih opcija. Na ovaj način otvara se *Query Editor* u kojeg korisnik samostalno može upisivati upite te vrlo lako može upravljati Postgre bazom podataka.

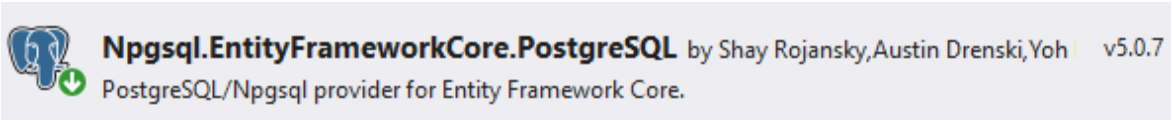
Drugi način je da se desnim klikom pritisne na željenu tablicu te se odabere *View/Edit data* i izabere se jedan od ponuđenih odabira (engl. *select*). Moguće je odabrati prvih 100 i zadnjih 100 redova te odabrati da se prikažu svi redovi. Primjer prikaza svih podataka nalazi se na slici 3.4.



Slika 3.4. Prikaz svih podataka od modela usluge

3.1.2 Povezivanje baze podataka i aplikacije

Kako bih se uspješno obavilo povezivanje PostgreSQL relacijske baze podataka i aplikacijskog programskog sučelja, potrebno je instalirati NuGet[9] paket koji se nalazi na slici 3.5, koji navedenu vezu čini ostvarivom.



Slika 3.5. Npsql.Entity.FrameworkCore.PostgreSQL NuGet paket

Na slici 3.6 nalazi se primjer konekcijskog niza znakova za spajanje na Postgre relacijsku bazu podataka korištenu u ovom završnom radu.

```
{
  "ConnectionStrings": {
    "Server": "db",
    "Port": "5432",
    "Database": "SelfCareAppDatabase",
    "UserId": "root",
    "Password": "root"
  },
```

Slika 3.6. Konekcijski niz znakova za PostgreSQL bazu podataka

[9] NuGet upravitelj paketa nudi mogućnost instaliranja, vraćanja te objavljivanja paketa.

Unutar *helpers* direktorija dodana je *DatabaseSettings* klasa koja prima svojstva potrebna za konekciju na bazu te nadjačava (engl. *override*) *ToString* metodu i vraća konekcijski niz znakova na temelju dobivenih svojstava.

DatabaseSettings klasa se dodaje kao svojstvo unutar *Startup* klase[10] te se koristi unutar *Startup*. Nakon toga *DatabaseConfig* metoda dohvaća sve podatke koji su povezani s konekcijskim nizom znakova, zatim te podatke postavlja u objekt *_databaseSettings*.

Nakon postavljanja podataka u objekt *_databaseSettings*, potrebno je stvoriti klasu *DbContext*. U ovom slučaju naziv *DbContext* klase je *SelfCareAppDbContext* koja se nalazi na slici 3.7, te ona sadrži 4 svojstva koja je moguće dohvatiti. U ovom slučaju koriste se modeli: *user*, *account*, *ministration* i *faq*.

```
8   namespace Self_Care_app_azorja.Data.Context
9   {
        19 references
10      public class SelfCareAppDbContext : DbContext
11      {
            8 references
12          public DbSet<User> Users { get; set; }
            6 references
13          public DbSet<Account> Accounts { get; set; }
            5 references
14          public DbSet<Ministration> Ministrations { get; set; }
            5 references
15          public DbSet<Faq> Faqs { get; set; }
```

Slika 3.7. *SelfCareAppDbContext* klasa

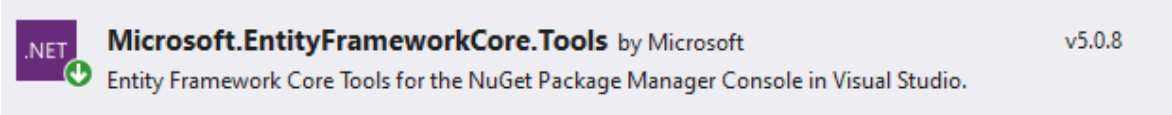
3.1.3 Migracije

U životnom ciklusu aplikacije, modeli podataka se mijenjaju zajedno s razvojem aplikacije i promjenom potreba klijenata. Shema unutar baze podataka mora se ažurirati zajedno s promjenom aplikacije kako bi bile sinkronizirane jedna s drugom.

[6] Migracije u ASP.NET Core programskom okviru omogućuju ažuriranje baze podataka tijekom životnog ciklusa aplikacije kako bih baza ostala sinkronizirana s aplikacijom. Pravilnom upotrebom migracija, podatci u bazi podataka se ne gube i ne mijenjaju.

[10] Startup klasa konfigurira servise aplikacije te uključuje metodu za konfiguriranje raznih zahtjeva.

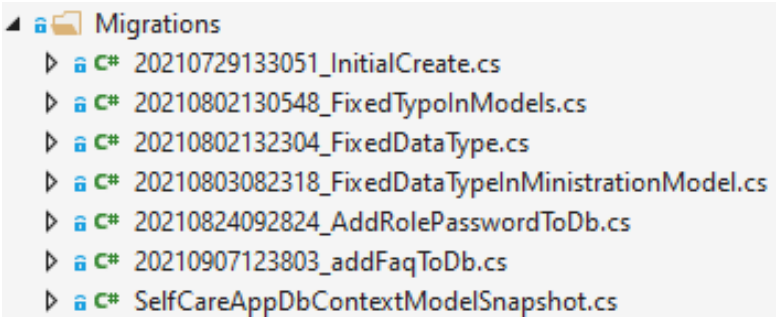
Kako bih se migracije mogle izvršiti, potrebno je instalirati Microsoft.EntityFrameworkCore.Tools NuGet paket prikazan na slici 3.8 koji omogućava korištenje naredbi za izvršavanje migracija u bazi podataka.



Slika 3.8. Microsoft.EntityFrameworkCore.Tools NuGet paket

Nakon uspješne instalacije NuGet paketa potrebnog za izvršavanje migracija, potrebno je unutar *package manager* konzole upisati naredbu *Add-Migration* te navesti željeno ime za migraciju.

Nakon uspješne migracije, EF Core će stvoriti direktorij naziva *Migrations* te u njega dodati migraciju s imenom koje je korisnik dodijelio migraciji.



Slika 3.9. *Migrations* direktorij sa svim migracijama

Na slici 3.9 nalazi se direktorij koji sadrži sve migracije koje su se dodale tijekom životnog ciklusa aplikacije. U pravilu, migracije se nebi trebale brisati jer se baza ne može vratiti na stanje prijašnje migracije.

Nakon stvaranja migracije potrebno je napraviti ažuriranje baze podataka. To se izvršava naredbom *Update-Database*. Za potrebe ovog projekta dodana je klasa [7] prikazana na slici 3.10, koja se bavi automatskim ažuriranjem promjena u bazi podataka.

```
public static class MigrationManager
{
        1 reference
        public static async Task<IHost> MigrateDatabase(this IHost host)
        {
            using (var scope = host.Services.CreateScope())
            {
                var services = scope.ServiceProvider;

                try
                {
                    var context = services.GetRequiredService<SelfCareAppDbContext>();

                    if (context.Database.IsNpgsql())
                    {
                        await context.Database.MigrateAsync();
                    }
                }
                catch (Exception ex)
                {
                    var logger = scope.ServiceProvider.GetRequiredService<ILogger<Program>>();
                    logger.LogError(ex, "An error occurred while migrating or seeding the database.");

                    throw;
                }
            }
            return host;
        }
```

Slika 3.10. *MigrationManager* klasa

Klasa za automatsko ažuriranje baze podataka vrlo je korisna zbog toga što se tijekom izrade projekta ažuriranje ne mora ručno izvršavati, već se nakon svakog pokretanja aplikacije ažuriranje samostalno izvrši. Potrebno je samo izvršiti naredbu *Add-Migration* te navesti proizvoljno ime migracije.

```
var host = CreateHostBuilder(args).Build();
if (Environment.GetEnvironmentVariable("ASPNETCORE_ENVIRONMENT") == Environments.Development)
{
    host.MigrateDatabase().Wait();
}

host.Run();
```

Slika 3.11. *Program.cs*[11]

Na slici 3.11 nalazi se primjer korištenja automatskog ažuriranja baze podataka koje se nalazi u *Program.cs* datoteci.

[11] Program.cs stvara instancu IWebHost-a koji *hosta* aplikaciju, te je mjesto gdje aplikacija započinje.

4. REACT

React je JavaScript biblioteka otvorenog koda (engl. *Open source*) razvijena za izradu Web aplikacija. Korisničko sučelje predstavlja skup ekrana, gumbova, tražilica, ikona i ostalog čime korisnik može ostvariti interakciju s aplikacijom.

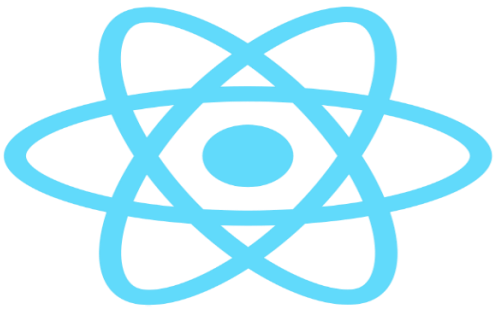
Sudeći prema ispitivanju [8] "State of JavaScript 2020", React je najpopularnija JavaScript biblioteka, a prema ispitivanju provedenom na Stack Overflow, React je drugo po redu najomiljenije razvojno okruženje za korisničko sučelje 2020. godine.

React je 2013. godine kreiran od strane inženjera u tvrtki Facebook[12], Jordana Walkea. Jordan Walke kreirao je React kako bi svim developerima mogao ubrzati razvoj korisničkog sučelja i povećati produktivnost u radu.

Za održavanje Reacta brine se Facebook te grupa individualnih developera. React je najčešće korišten za izradu jednostraničnih aplikacija (engl. *single-page application*).

React se koristi za izgradnju komponenti korisničkog sučelja (engl *User Interface, UI*). React aplikacija sastoji se od mnoštva komponenti. Svaka komponenta React aplikacije određuje svoj izgled pomoću HTML, Javascript i CSS koda . Tijekom osvježavanja stranice, React će ažurirati samo one dijelove ekrana koji su se promijenili što čini izradu aplikacija s React bibliotekom vrlo jednostavnom.

Održavanje React biblioteke obavlja Facebook te zajednica individualnih programera i kompanija.



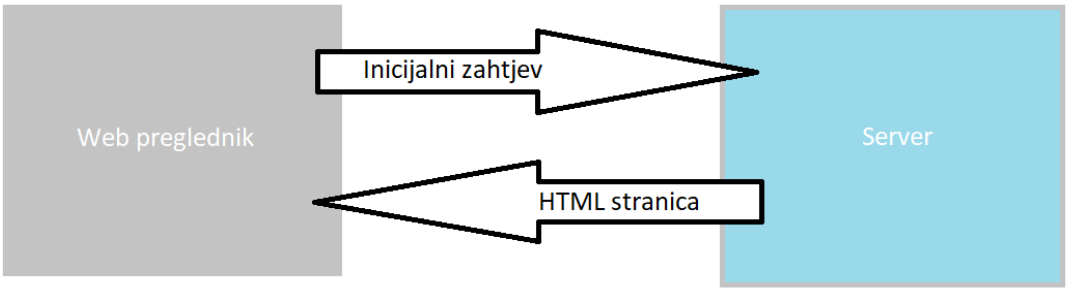
Slika 4.1. React logo

[12] Facebook je društvena mreža osnovana 2004. godine od strane Marka Zuckenberga.

Na slici 4.1 nalazi se [9] React logo. Kod kreiranja i pokretanja prve React aplikacije, na ekranu će se prikazati animacija React logoa.

4.1 React arhitektura

Kod arhitekture React biblioteke, server mora poslati samo jednu HTML stranicu Web pregledniku kako bih aplikaciju dalje preuzeo React te ju vodio u Web pregledniku kao što je prikazano na slici 4.2. React upravlja s podatcima Web stranice te aktivnostima korisnika kao što su *onClick* akcije, pa čak i prebacivanje sa stranice na stranicu.



Slika 4.2. React zahtjev i odgovor

Ovaj način upravljanja Web aplikacijom ubrzava responzivnost aplikacije i poboljšava iskustvo korisnika za razliku od prijašnjih načina upravljanja Web aplikacijama na kojima je svaki klik na link slao novi zahtjev serveru za HTML[13] stranicu.

4.2 Kreiranje React aplikacije

Postoji nekoliko načina kreiranja aplikacije u React biblioteci. U ovom završnom radu opisati će se način koji se smatra najbržim i najjednostavnijim. Za kreiranje React aplikacije, prvo je potrebno instalirati Node.js.[14] Nakon instalacije Node.js biblioteke, potrebno je otvoriti PowerShell te upisati naredbu *npx create-react-app imeAplikacije* za stvaranje React.js aplikacije

Nakon kreiranja aplikacije s proizvoljnim imenom, aplikacija se može pokrenuti naredbom *npm start*

[13] HTML (engl. HyperText Markup Language) je prezentacijski jezik koji se koristi za kreiranje web stranica.
[14] Node.js je *backend JavaScript okruženje otvorenog koda.*

Slika 4.3. Početni izgled novostvorene React aplikacije

Na slici 4.3 nalazi se početna web stranica novostvorene React aplikacije na adresi localhost:3000. Sve što se nalazi na početnoj web stranici dolazi iz *App* komponente koja se stvorila tijekom kreiranja okruženja za razvoj React aplikacije. Svaka promjena koda se automatski ažurira na korisničkom sučelju kada se promjena koda spremi.

U slučaju dohvaćanja nekog React projekta s github ili gitlab platforme, projekt vjerojatno neće raditi. Najčešći razlog tome je što programeri tijekom postavljanja React projekta na internetske stranice izuzimaju *node_modules* direktorij. U direktoriju *node_modules* nalaze se sve ovisnosti React projekta uključujući i sami React, stoga je količina prostora koju taj direktorij zauzima prevelika za postavljanje na github ili gitlab[15].

Rješenje tog problema je pokretanje naredbe *npm install*, u *root* direktoriju React projekta.

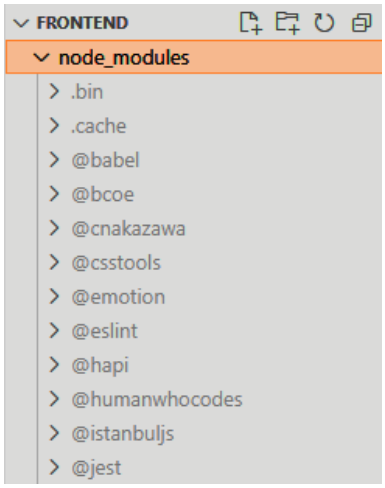
{} package.json > ...

```
{
  "name": "frontend",
  "version": "0.1.0",
  "private": true,
  "dependencies": {
    "@material-ui/core": "^4.12.3",
    "@material-ui/icons": "^4.11.2",
    "@testing-library/jest-dom": "^5.14.1",
    "@testing-library/react": "^11.2.7",
    "@testing-library/user-event": "^12.8.3",
    "axios": "^0.21.1",
    "react": "^17.0.2",
    "react-dom": "^17.0.2",
    "react-router-dom": "^5.2.0",
    "react-scripts": "4.0.3",
    "rsuite": "^4.10.2",
    "web-vitals": "^1.1.2"
  },
```

Slika 4.4. *Package.json* datoteka

[15] GitHub i GitLab su platforme za upravljanje Git direktorijima.

Pokretanjem te naredbe pokreće se provjera u *package.json* datoteci prikazanoj na slici 4.4, koja će proći kroz sve ovisnosti dohvaćenog projekta te će ih instalirati i stvoriti *node_modules* direktorij koji se nalazi na slici 4.5.



Slika 4.5. *Node_modules* direktorij unutar aplikacije ovog završnog rada

4.3 React komponente

Svaka komponenta u React aplikaciji mora implementirati metodu za vraćanje podataka koji će se prikazati na korisničkom sučelju.

src > JS index.js

```
1   import React from 'react';
2   import ReactDOM from 'react-dom';
3   import './index.css';
4   import App from './App';
5   import reportWebVitals from './reportWebVitals';
6
7   ReactDOM.render(
8     <React.StrictMode>
9       <App />
10    </React.StrictMode>,
11    document.getElementById('root')
12  );
13
14  // If you want to start measuring performance in your app, pass a function
15  // to log results (for example: reportWebVitals(console.log))
16  // or send to an analytics endpoint. Learn more: https://bit.ly/CRA-vitals
17  reportWebVitals();
```

Slika 4.6. *Index.js*

Na slici 4.6 nalazi se *Index.js* datoteka kojom započinje aplikacija u React biblioteci. *Index.js* je odgovoran za dohvaćanje svih stvorenih React komponenti i postavljanjem istih u DOM (engl. *Document Object Model*).[16]

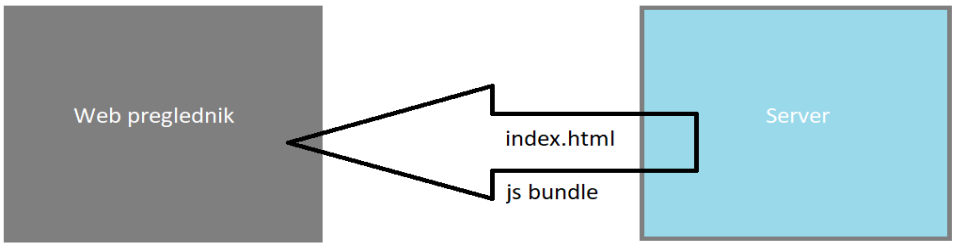
U ovom slučaju dohvaća *App* komponentu i renderira ju u DOM u div elementu naziva *root*. Na slici 4.7 nalazi se dio *App.js* komponente korištene u ovom završnom radu.

```
42    return (
43      <Router>
44        <MyContext.Provider value={{ loginUser, handleLogin, handleLogout, listOfCreateErrors, setListOfCreateErrors }}>
45          <div className="App">
46            <Navbar />
47            <ErrorsBar />
48            <div className="content">
49              <Switch>
50                <Route exact path="/">
51                  <Login />
52                </Route>
53                <Route exact path="/home">
54                  <Home />
55                </Route>
56                <Route exact path="/create">
57                  <Create />
58                </Route>
```

Slika 4.7. *App.js* komponenta ovog projekta

4.4 React Router

React Router je način na koji se u aplikaciju implementiraju različite stranice. *React Router* paket nije dio React biblioteke, već se mora ručno instalirati. Potrebno je pokrenuti naredbu *npm*[17] *install react-router-dom* unutar terminala u Visual Studio Code editoru. Nakon izvršenja navedene naredbe, unutar *package.json* datoteke pojavit će se „*react-router-dom*“ unutar aplikacijskih ovisnosti.



Slika 4.8. Server šalje HTML stranicu te ReactJavaScript datoteku

[16] DOM je struktura koja podsjeća na krošnju drveta koja sadrži sve elemente i svojstva web stranice.
[17] Npm (Nod Package Manager) je upravitelj paketa za JavaScript.

U slučaju React aplikacije, Web preglednik šalje zahtjev za promjenu stranice serveru, a server šalje odgovor koji sadrži HTML stranicu. Server zajedno s HTML stranicom šalje i prevedenu React JavaScript datoteku koja kontrolira React aplikaciju kao što je prikazano na slici 4.8.

HTML stranica na početku je prazna, nakon toga React će dinamički ubaciti komponente koje je programer kreirao u nju. Ovaj pristup ažuriranju DOM-a je najučinkovitiji zbog toga što se ne mora slati novi zahtjev serveru kod svake promjene stranice.

4.5 React Switch

React *Switch* komponenta uvezena je iz *react-router-dom,* te se ona mora pobrinuti da se na web stranici prikazuje samo jedna ruta u isto vrijeme. U slučaju komponente koja se uvijek mora prikazivati na web stranici, ona se mora postaviti izvan *switch* komponente. Primjer toga može se vidjeti na slici 4.9, koja prikazuje komponente *Navbar* i *ErrorsBar* van *Switch* komponente. Na ovaj način, te dvije komponente će se prikazivati tijekom cijelog životnog ciklusa aplikacije, bez obzira na ono što se nalazi u *Switch* komponenti.

```
45      <div className="App">
46        <Navbar />
47        <ErrorsBar />
48        <div className="content">
49          <Switch>
50            <Route exact path="/">
51              <Login />
52            </Route>
53            <Route exact path="/home">
54              <Home />
55            </Route>
56            <Route exact path="/create">
57              <Create />
58            </Route>
```

Slika 4.9. Primjer *Switch* komponente

4.6 React Hooks

React je u verziji 16.8 predstavio novu mogućnost Reacta [10] naziva *Hooks*. React *Hooks* dozvoljavaju programerima da koriste *use state* i druge komponente Reacta bez da pišu potpuno novu klasu. Najjednostavniji primjer React *Hooka* je *use state hook*. Predstavljanjem *Hooksa,* React ne planira ukloniti klase iz React biblioteke, te se

implementacijom istih ne gubi znanje o osnovnim konceptima Reacta koje su programeri stekli do sada.

React *Hooks* rješavaju mnoštvo problema na koje su korisnici React biblioteke naišli tijekom svih ovih godina postojanja Reacta. *Hook* pomaže programerima da izuzmu logiku o stanju konstanti u komponenti te ju ponovo koriste bez mjenjanja same komponente

4.6.1 UseState Hook

UseState Hook vraća vrijednost o trenutnom stanje i funkciju koja se poziva u trenutku kada se stanje konstante želi promijeniti. Tijekom prvog *renderiranja*, vrijednost konstante je ista kao vrijednost koja joj je proslijeđena nakon znaka jednakosti.

```
24  export default function App() {
25
26    const [loginUser, setLoginUser] = useState('');
27    const [listOfCreateErrors, setListOfCreateErrors] = useState('');
28
29    const handleLogin = (userData) => {
30      console.log('handleLogin', userData);
31      setLoginUser(userData);
32    }
```

Slika 4.10. Primjer *UseState Hooka*

Na slici 4.10 nalaze se primjeri jednih od mnogo korištenih *useState hookova* u ovom završnom radu. *LoginUser* konstanta postavljena je na prazan niz znakova tijekom prvog *renderiranja*. Nakon toga u metodi *handleLogin* može se vidjeti kako se vrijednost konstante *loginUser* pomoću *setLoginUser* metode postavlja na vrijednost podataka koji se prosljeđuju unutar *userData* objekta.

4.6.2 UseEffect Hook

UseEffect Hook pokreće se kod svakog novog renderiranja komponente. Komponenta se *renderira* kod prvog učitavanja iste, ali i nakon svake promjene stanja (engl. *state*).

UseEffect mora se uvesti iz React biblioteke isto kao i *useState Hook*. *UseEffect* mora primati funkciju koja se pokreće svaki put kada se dogodi novo *renderiranje* komponente ili promjena stanja konstanti.

Tijekom korištenja *UseEffect Hooka* potrebno je obraćati pozornost na mijenjanje stanja (engl. *state*). Mijenjanjem stanja unutar *UseEffect Hooka* vrlo lako može doći do beskonačne petlje. *UseEffect Hook* se pokreće kada se stanje promijeni, ukoliko se stanje ponovo promijeni unutar *Hooka*, *UseEffect Hook* će se ponovo pokrenuti i tako će nastaviti u beskonačnu petlju.

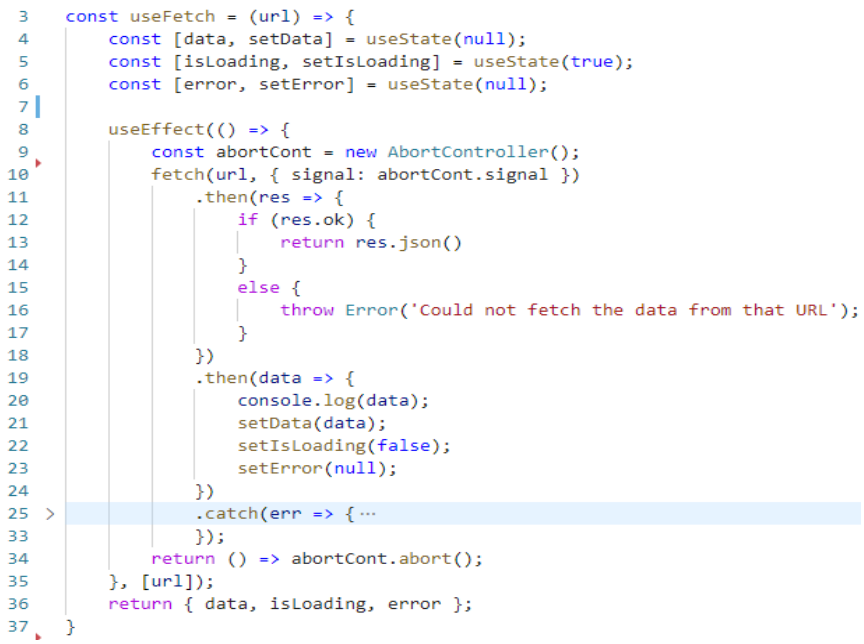
```
38  React.useEffect(() => {
39    console.log('UseEffect: loginUser: ', loginUser);
40  }, [loginUser]);
41
```

Slika 4.11. *UseEffect Hook*

Na slici 4.11 nalazi se jednostavan primjer korištenja *UseEffect Hooka* u ovom završnom radu. U ovom primjeru nije potrebno pokrenuti kod unutar *hooka* tijekom svakog *renderiranja* komponente. Stoga se u ovom primjeru koristi polje u kojem se navode ovisnosti. To znači da će se *UseEffect* pokrenuti samo kod prvog *rendera* te kod svake od navedenih ovisnosti, u ovom slučaju konstante *loginUser*.

4.6.3 UseFetch hook

UseFetch hook prikazan na slici 4.12 je *custom hook* kreiran za potrebe ovog završnog rada. Ovaj *custom hook* prima URL te ne osnovu njega dohvaća podatke s *backenda*. Na osnovu uspješnosti dohvaćanja podataka s *backenda* vraća iste, eventualnu grešku te podatak je li dohvaćanje završilo ili je u tijeku. Kreiranje *custom hook* komponenti znatno smanjuje količinu napisanog koda i olakšava daljnje programiranje.



```
const useFetch = (url) => {
    const [data, setData] = useState(null);
    const [isLoading, setIsLoading] = useState(true);
    const [error, setError] = useState(null);

    useEffect(() => {
        const abortCont = new AbortController();
        fetch(url, { signal: abortCont.signal })
            .then(res => {
                if (res.ok) {
                    return res.json()
                }
                else {
                    throw Error('Could not fetch the data from that URL');
                }
            })
            .then(data => {
                console.log(data);
                setData(data);
                setIsLoading(false);
                setError(null);
            })
            .catch(err => {···
            });
        return () => abortCont.abort();
    }, [url]);
    return { data, isLoading, error };
}
```

Slika 4.12. *UseFetch custom hook*

4.7 React Context

Context [11] u Reactu pruža mogućnost proslijeđivanja podataka kroz cijelo stablo komponenata bez potrebe za proslijeđivanjem *propsa* kroz svaku granu stabla pa sve do komponente u kojem te podatke programer može zatrebati.

Proslijeđivanje *propsa* kroz stablo nije loš izbor kada se u projektu nalazi samo nekoliko komponenti te svaka od njih ili velika većina ima koristi od istih.

U slučaju velikog projekta koji ima mnoštvo komponenata, proslijeđivanje *propsa* postaje mukotrpan posao. Osim što se troši vrijeme na proslijeđivanje *propsa*, većina komponenata ih niti ne koriste, ali ih moraju dohvatiti i proslijediti kako bih se komponente niže u stablu mogle koristiti njima.

Ponekad se pojedini podatci trebaju koristiti duboko u aplikaciji te se tada koristi *context* koji se bavi tim problemom. Kreiranje *context* metode ima nekoliko jednostavnih koraka. Prvo je potrebno kreirati *context* koristeći metodu *createContext* kao na slici 4.13.

```
export const MyContext = React.createContext();
```

Slika 4.13. React *createContext* metoda

Nakon toga se sa stvorenom *context* metodom obmotava (engl. *wrap*) stablo komponenata u kojemu se želi koristiti *context*. Postavljanje vrijednosti koje će se koristiti u *context* metodi obavlja se postavljanjem u *value*. Na slici 4.14 može se vidjeti primjer korištenja *context* metode u ovom završnom radu.

```
42    return (
43      <Router>
44        <MyContext.Provider value={{ loginUser, handleLogin, handleLogout, listOfCreateErrors, setListOfCreateErrors }}>
45          <div className="App">
46            <Navbar />
47            <ErrorsBar />
48            <div className="content">
```

Slika 4.14. Korištenje *contexta* u *App.js* komponenti

Zadnji korak je dohvaćanje podataka u komponentama u kojima je pojedini podatak potreban. Podatci se iz *contexta* dohvaćaju korištenjem *useContext* metode kao na slici 4.15 unutar *UserDetails* komponente.

```
7    const UserDetails = () => {
8        const { loginUser } = useContext(MyContext);
```

Slika 4.15. Dohvaćanje *loginUser* objekta iz *MyContext*

4.8 Kreiranje korisnika

4.8.1 Metoda za kreiranje korisnika

Na slici 4.16 nalazi se *handleSubmit* metoda koja se nalazi unutar *Create* komponente koja služi za kreiranje korisnika. Metoda *handle submit* poziva se odabirom gumba *Add User*. Metoda dohvaća podatke potrebne za kreiranje korisnika s *backenda* pomoću *fetcha*.

U ovom slučaju šalje se POST zahtjev (engl. request) na *localhost*, *port* 5000, /api/User/Create. *LocalHostUrl* korišten u *Create* komponenti može se naći u datoteci Url.js.

Localhost url odvojen je u posebnu datoteku zbog toga što se koristi na više mjesta u projektu te se u slučaju promjene *porta* mora promjeniti samo na jednom mjestu.

Nakon dohvaćanja podataka u *res*, provjerava se jesu li podatci validni. U slučaju da su podatci validni, stvara se novi korisnik te se administrator preusmjerava na *home* stranicu. Kada se provjerom utvrdi da uneseni podatci nisu validni, greške se spremaju te se ispisuju

na ekranu kako bih administrator mogao promjeniti podatke koji krše validacijska pravila stvaranja novog korisnika.

```
17      const handleSubmit = async (e) => {
18          e.preventDefault();
19          //debugger;
20          const user = { name, surname, oib, emailAddress, role, password };
21          setIsLoading(true);
22
23          try {
24              const res = await fetch(localhostUrl + '/api/User/Create', {
25                  method: 'POST',
26                  headers: {
27                      'Content-Type': 'application/json',
28                      'Accept': 'application/json'
29                  },
30                  body: JSON.stringify(user)
31              });
32              if (res.ok) {
33                  console.log("no errors");
34                  console.log({ user });
35                  setListOfCreateErrors(null);
36                  //history.go(-1);
37                  history.push('/home');//vracam se na Home Page
38              }
39              if (!res.ok) {
40                  const errors = await res.json();
41                  const errror = errors.map(err => err.errorMessage);
42                  console.log("dobio grešku:" + errror);
43                  setListOfCreateErrors(errror);
44              }
```

Slika 4.16. *HandleSubmit* metoda unutar *Create* komponente za korisnika

Abort Controller je potrebno koristiti u slučaju brzog prijelaza između dva ili više ekrana korisničkog sučelja. Kada se za vrijeme učitavanja komponente te dohvaćanja podataka s *backenda* odabere neki drugi ekran koji će montirati novu komponentu, dolazi do greške prikazane na slici 4.17.

```
▶Warning: Can't perform a React state update on an unmounted component. This is a no-op, but it indicates a memory leak in your application. To fix, cancel all subscriptions and asynchronous tasks in a useEffect cleanup function.
```
index.js:1

Slika 4.17. Greška uzrokovana neučitanom komponentom

Do greške dolazi zbog toga što *fetch* nakon završetka dohvaćanja podataka s *backenda* pokušava ažurirati komponentu koja ga je pozvala. Međutim ta komponenta više nije učitana na ekran već je učitana druga komponenta koja je odabrana u vrijeme dohvaćanja podataka te zbog toga dolazi do konflikta.

Rješenje ovog problema je korištenje *Abort Controllera* unutar *useEffect hooka* koji će spriječiti dohvaćanje podataka u slučaju brze izmjene ekrana i učitavanja nove komponente na ekran. Primjer korištenja *Abort Controllera* nalazi se na slici 4.18.

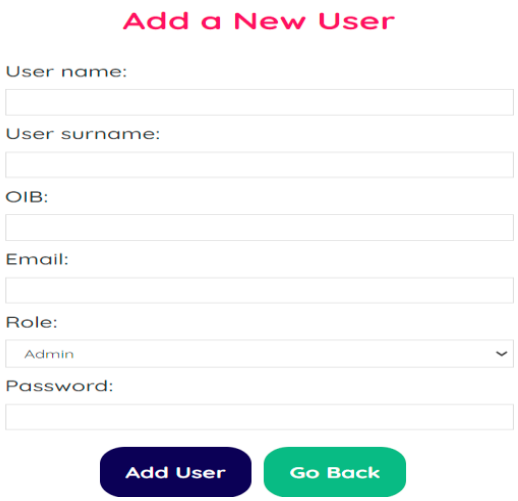
```
56    useEffect(() => {
57        const abortCont = new AbortController();
58        //Dodan Abort controller iz useFetch hook-a
59        return () => abortCont.abort();
60    }, [isLoading, listOfCreateErrors])
```

Slika 4.18. *Abort Controller* unutar useEffect *hooka*

4.8.2 Korisničko sučelje za kreiranje novog korisnika

Na slici 4.19 nalazi se izgled korisničkog sučelja kreiranog za dodavanje novog korisnika u ovaj projekt. Sučelje se sastoji od 6 polja za unos podataka. Sva polja zahtijevaju unos podataka ručnim unosom, osim polja s oznakom *Role*, na kojem administrator ne može unijeti podatak, već mora odabrati između 2 ponuđene mogućnosti.

Ažuriranje podataka korisnika obavlja se preko sličnog sučelja, ali su polja uloge korisnika te lozinka izuzeti zbog sigurnosti sustava.



Slika 4.19. Korisničko sučelje za kreiranje korisnika

Na slici 4.20 nalazi se [12] JSX kod korišten za kreiranje izgleda *Create* komponente koja se koristi kod kreiranja novog korisnika. JavaScript kod se može ubaciti unutar JSX koda korištenjem vitičastih zagrada. JSX je više nalik JavaScriptu nego HTML kodu, pa se

zbog toga koristi *camelCase*[18] način imenovanja. Primjer ovog načina imenovanja je klasa koja iz *class* prelazi u *className*.

```
return (
    <div className="create">
        <h2>Add a New User</h2>
        <form onSubmit={handleSubmit}>
            <label>User name:</label>
            <input type="text" required value={name} onChange={(e) => setName(e.target.value)}></input>
            <label>User surname:</label>
            <input type="text" required value={surname} onChange={(e) => setSurname(e.target.value)}></input>
            <label>OIB:</label>
            <input type="text" required value={oib} onChange={(e) => setOib(e.target.value)}></input>
            <label>Email:</label>
            <input type="text" required value={emailAddress} onChange={(e) => setEmailAddress(e.target.value)}></input>
            <label>Role:</label>
            <select type="text" required value={role} onChange={(e) => setRole(e.target.value)}>
                <option value="Admin">Admin</option>
                <option value="User">User</option>
            </select>
            <label>Password:</label>
            <input type="text" required value={password} onChange={(e) => setPassword(e.target.value)}></input>

            {!isLoading && <button type="submit" id="AddButton">Add User</button>}
            {isLoading && <button disabled>Adding user</button>}
            <button type="button" id="cancelButton" onClick={() => { setListOfCreateErrors(); history.go(-1) }}>Go Back</button>
            <p>{name} {surname}</p>
        </form>
    </div>
);
```

Slika 4.20. JSX kod za *Create* komponentu

[18] CamelCase je konvencija pisanja imena varijabli, objekata i sl. na način da su riječi spojene, počinju prvim malim slovom te je svako slovo slijedeće riječi veliko slovo.

5. DOCKER

5.1 Docker kontejnerizacija

Docker [13] je platforma otvorenog koda osmišljena za kontejnerizaciju aplikacija. Platforma Dockera omogućuje programerima kontejnerizaciju aplikacija u svrhu lakšeg prijenosa i pokretanja aplikacije na bilo kojem sustavu.

Docker kontejneri[19] sadrže samo ono što je potrebno kako bih se aplikacija mogla pokrenuti bilo gdje. Kontejneri su vrlo kompaktni, što znači da ne sadrže cijele operativne sustave u sebi, već sadrže aplikaciju sa svim potrebnim bibliotekama i komponentama koje su potrebne za pokretanje aplikacije te datoteke za konfiguraciju.

Veličina kontejnera mjeri se u megabajtima, u odnosu na virtualne mašine koje svoju veličinu mjere u nekoliko gigabajta zauzetog prostora.

Kontejneri su optimizirani te time efikasnije iskorištavaju sposobnosti *hardwarea* te imaju brže vrijeme pokretanja. Na slici 5.1 nalazi se [14] Logotip Docker platforme.

Slika 5.1. Logotip Docker platforme

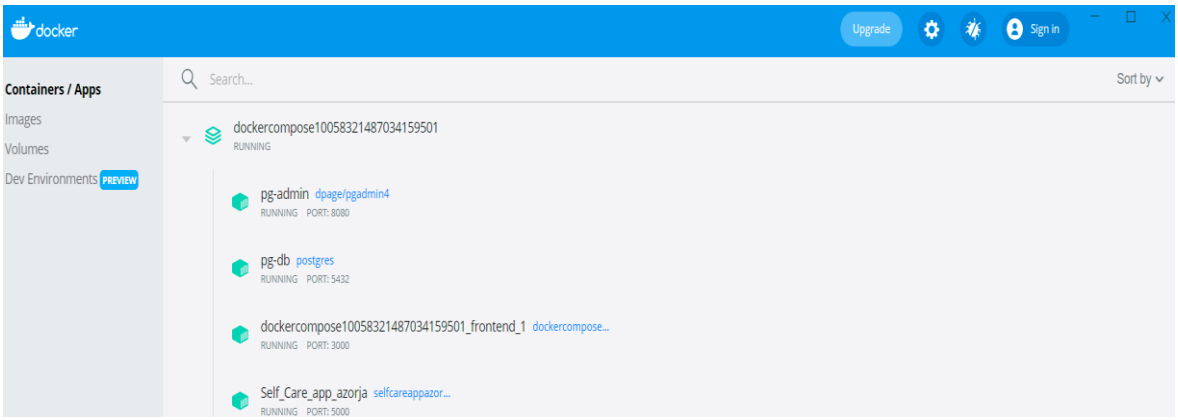
Docker platforma u osnovi je alat kojim programer s lakoćom može izraditi, pokrenuti i zaustaviti kontejnere koristeći jednostavne naredbe. Docker ima mogućnost automatske izrade kontejnera na temelju koda napisanog u aplikaciji. Kontejneri u Dockeru mogu se ponovo iskoristiti na način da se postojeći kontejner iskoristi kao slika (engl. *image*), odnosno kao predložak za izgradnju novog kontejnera.

[19] Kontejner je držač svega onog što je potrebno za pokretanje aplikacije.

5.2 Docker Desktop

Docker Desktop je aplikacija s vrlo jednostavnim načinom instalacije koja služi za izgradnju, pokretanje, pauziranje, zaustavljanje i dijeljenje kontejneriziranih aplikacija na brz i učinkovit način. Aplikaciju je moguće instalirati na Windows operativnom sustavu ili Mac-u.

Aplikacija Docker Desktop sadrži Docker Engine, Docker CLI klijent, Docker Compose i mnoge druge.



Slika 5.2.Docker Desktop aplikacija

Na slici 5.2 nalazi se aplikacija Docker Desktop na primjeru *Self-care* aplikacije koja se obrađuje u ovom završnom radu. Slika prikazuje četiri pokrenuta kontejnera. Svaki od kontejnera predstavlja jedan dio *Self-care* web sustava.

5.3 Docker Compose

U slučaju aplikacija koje se sastoje od više kontejnera koji se nalaze na istom poslužitelju, potrebno je koristiti *Docker Compose* kako bih se moglo upravljati aplikacijskom arhitekturom.

Docker Compose stvara YAML datoteku. YAML datoteka određuje servise koji su uključeni u aplikaciju te se pomoću *docker-compose up* naredbe mogu pokrenuti svi kontejneri.

Slika 5.3. docker-compose.yml datoteka

Na slici 5.3 nalazi se primjer *Docker Compose* YAML datoteke. Na slici se prikazuju dva od četiri servisa koja se kreiraju u ovom projektu, a to su *pg-admin* koji se koristi za komunikaciju i upravljanje s Postgre relacijskom bazom podataka i njenim servisima te *frontend* koji sadrži React kod.

5.4 Docker File

Docker kontejneri se stvaraju od jednostavnih instrukcija koje se sastoje od mnoštva uputa koje opisuju kako izgraditi Docker Image. Taj proces se može olakšati automatizacijom koju u ovom slučaju obavlja Docker File koji sadrži instrukcije koje Docker Engine mora obaviti kako bi sastavio Docker Image.

Kada se Docker File jednom konstruira, Docker Image se iz njega može izraditi nebrojeno mnogo puta, bez potrebe da se kroz proces mora prolaziti ručno. Na slici 5.4 nalazi se Dockerfile koji se koristi u ovom završnom radu.

```
FROM mcr.microsoft.com/dotnet/aspnet:5.0 AS base
RUN apt-get update && apt-get install
RUN apt-get install -y wget
RUN apt-get install -y apt-transport-https
RUN wget https://packages.microsoft.com/config/ubuntu/20.10/packages-microsoft-prod.deb -O packages-microsoft-prod.deb
RUN dpkg -i packages-microsoft-prod.deb
RUN apt-get update
RUN apt-get install -y dotnet-sdk-5.0
RUN dotnet tool install --global dotnet-ef
#RUN curl -sL https://deb.nodesource.com/setup_10.x |  bash -
#RUN apt-get install -y nodejs
ENV PATH $PATH:/root/.dotnet/tools
WORKDIR /app
#EXPOSE 80


FROM mcr.microsoft.com/dotnet/sdk:5.0 AS build
WORKDIR /src
COPY ["Self_Care_app_azorja.csproj", "."]
RUN dotnet restore "Self_Care_app_azorja.csproj"
COPY . .
WORKDIR "/src/"
RUN dotnet build "Self_Care_app_azorja.csproj" -c Release -o /app/build

FROM build AS publish
RUN dotnet publish "Self_Care_app_azorja.csproj" -c Release -o /app/publish

FROM base AS final
WORKDIR /app
COPY --from=publish /app/publish .
ENTRYPOINT ["dotnet", "Self_Care_app_azorja.dll"]
```

Slika 5.4. Dockerfile

5.4.1 Docker Image

Docker Image sadrži izvršni kod aplikacije zajedno sa svim alatima, bibliotekama te zahtjevima koji su potrebni aplikacijskom kodu kako bih se mogao pokrenuti u obliku kontejnera.

Docker Image sastoji se od više različitih slojeva. Kod promjene izvršnog koda, stvara se novi nad sloj koji mjenja prijašnji sloj. Na taj način se stvaraju verzije Docker Imagea. Svi prijašnji slojevi se spremaju kako bih bilo moguće vratiti se na prijašnju verziju *Imagea* u slučaju pogreške.

6. ZAKLJUČAK

Najvažniji cilj ovog završnog rada bilo je kreirati web sustav koji bih bio koristan za jednu veću telekomunikacijsku tvrtku. Kreiranje takvog sustav ostvareno je koristeći ASP.NET Core programski okvir, PostgreSQL relacijsku bazu podataka, React JavaScript biblioteku, te Docker platformu za kontejnerizaciju aplikacija.

Za ostvarenje funkcionalnog sustava, bilo je potrebno steći osnovno znanje o Docker platformi te srednju razinu znanja o React biblioteci.

U ovom radu opisani su najjednostavniji i najučinkovitiji načini kreiranja koda za *backend* u Web aplikacijskom programskom sučelju. Prikazano je stvaranje modela, pripadajućih servisa za njega te autentifikacija i validacija svih podataka korištenih u sustavu.

Kod React biblioteke opisani su osnovni dijelovi stvaranja ove aplikacije, kao što je stvaranje Routera sa switch komponentom koja na jednostavan način preusmjerava korisnika između ruta koje se koriste u aplikaciji, razni *hookovi* kao što su *useState* i *useEffect* te primjer *custom hooka useFetch* koji je napisan posebno za ovaj završni rad te se koristi kroz cijelu hijerarhiju komponenata koje tvore aplikaciju.

Na kraju je cijeli sustav postavljen u četiri kontejnera koristeći Docker. Docker sustav funkcionira vrlo brzo i efikasno te se s lakoćom pomoću njega može pokrenuti kompletni sustav koristeći samo jednu naredbu ili pritisak gumba koji pokreće Docker Compose.

Sustav besprijekorno funkcionira koristeći virtualizaciju od Docker platforme, ali se prednosti korištenja Dockera ne ističu u potpunosti zbog toga što je ovo relativno mali sustav s četiri kontejnera.

7. LITERATURA

1) What is ASP.NET Core? [Online]. 2021. Dostupno na: https://dotnet.microsoft.com/learn/aspnet/what-is-aspnet-core (14.9.2021)
2) ASP.NET Core 5.0 Web API [Online]. 2021 Dostupno na: https://www.c-sharpcorner.com/article/asp-net-core-5-0-web-api/ (14.9.2021)
3) .NET documentation [Online]. 2021 Dostupno na: https://docs.microsoft.com/hr-hr/dotnet/?WT.mc_id=blog-blog-lbugnion (14.9.2021)
4) Optimizing Storage and Managing Cleanup in PostGreSQL [Online]. 2019 Dostupno na: https://medium.com/coding-blocks/optimizing-storage-and-managing-cleanup-in-postgresql-c2fe56d4cf5 (14.9.2021)
5) An overview of PGAdmin - PostgreSQL Managment Tool. [Online] 2021. Dostupno na: https://www.sqlshack.com/an-overview-of-pgadmin-postgresql-management-tool/ (15.9.2021)
6) Migrations Overview [Online]. 2020. Dostupno na: https://docs.microsoft.com/en-us/ef/core/managing-schemas/migrations/?tabs=vs (15.9.2021)
7) ASP.NET Core - Automatic EF Core Migrations to SQL Database on Startup [Online]. 2019. Dostupno na: https://jasonwatmore.com/post/2019/12/27/aspnet-core-automatic-ef-core-migrations-to-sql-database-on-startup (15.9.2021)
8) What Is React Used For? [Online]. 2021. Dostupno na: https://www.telerik.com/blogs/what-is-react-used-for (16.9.2021)
9) An intro to React and React Native [Online]. 2021 Dostupno na: https://blog.wildix.com/an-intro-to-react-and-react-native/ (16.9.2021)
10) Introducing Hooks [Online]. 2021. Dostupno na: https://reactjs.org/docs/hooks-intro.html (17.9.2021)
11) Context [Online]. 2021. Dostupno na: https://reactjs.org/docs/context.html (20.9.2021)
12) Introducing JSX [Online]. 2021. Dostupno na: https://reactjs.org/docs/introducing-jsx.html (21.9.2021)

13) What is Docker? [Online]. 2021. Dostupno na: https://www.ibm.com/cloud/learn/docker (16.9.2021)
14) Docker Fundementals training [Online]. 2021 Dostupno na: https://tmcalm.nl/docker-fundamentals-training/ (16.9.2021)

8. OZNAKE I KRATICE

API – aplikacijsko programsko sučelje (engl. *Application Programming Interface)*

HTTP – Hypertext Transfer Protocol

URL – Uniform Resource Locator

GUI – Graphical User Interface

DOM – Document Object Model

UI – User Interface

9. SAŽETAK

Naslov: Web sustav za upravljanje korisnicima unutar telekomunikacijske tvrtke

Cilj ovog rada je kreiranje web sustava za jednu veću telekomunikacijsku tvrtku koji omogućava korisnicima pregled vlastitih korisničkih podataka, usluga koje imaju ili mogu imati, te aktivaciju ili deaktivaciju istih. U izradi sustava korišten je ASP.NET Core programski okvir, React JavaScript biblioteka, PostgreSQL baza podataka te Docker platforma. Frontend ovog sustava pisan je u React JavaScript biblioteci, posrednik između baze podataka i React aplikacije je ASP.NET Core Web API, te su svi dijelovi uključujući Postgre bazu podataka virtualizirani koristeći Docker platformu.

Ključne riječi: ASP, .NET, React, PostgreSQL, Docker

10. ABSTRACT

Title: Web system for managing users inside a telecommunication company

The main goal of this Undergraduate thesis is to create a web system for a bigger telecommunication company which allows users to view their account data, ministrations which they have or can have, and activation or deactivation of those ministrations. In the making of this system, ASP.NET Core framework, React JavaScript library, PostgreSQL database and Docker platform were used. Frontend of this system was written in React JavaScript library, ASP.NET Core Web API was the mediator between the database and the React application. Every part of the application, including the Postgre database was virtualized using Docker platform

Key Words: ASP, .NET, React, PostgreSQL, Docker

IZJAVA O AUTORSTVU ZAVRŠNOG RADA

Pod punom odgovornošću izjavljujem da sam ovaj rad izradio/la samostalno, poštujući načela akademske čestitosti, pravila struke te pravila i norme standardnog hrvatskog jezika. Rad je moje autorsko djelo i svi su preuzeti citati i parafraze u njemu primjereno označeni.

Mjesto i datum	Ime i prezime studenta/ice	Potpis studenta/ice
U Bjelovaru, 19.10.2021	Alen Žorja	Alen Žorja

Prema Odluci Veleučilišta u Bjelovaru, a u skladu sa Zakonom o znanstvenoj djelatnosti i visokom obrazovanju, elektroničke inačice završnih radova studenata Veleučilišta u Bjelovaru bit će pohranjene i javno dostupne u internetskoj bazi Nacionalne i sveučilišne knjižnice u Zagrebu. Ukoliko ste suglasni da tekst Vašeg završnog rada u cijelosti bude javno objavljen, molimo Vas da to potvrdite potpisom.

Suglasnost za objavljivanje elektroničke inačice završnog rada u javno dostupnom nacionalnom repozitoriju

Alen Zorja

ime i prezime studenta/ice

Dajem suglasnost da se radi promicanja otvorenog i slobodnog pristupa znanju i informacijama cjeloviti tekst mojeg završnog rada pohrani u repozitorij Nacionalne i sveučilišne knjižnice u Zagrebu i time učini javno dostupnim.

Svojim potpisom potvrđujem istovjetnost tiskane i elektroničke inačice završnog rada.

U Bjelovaru, 19.10.2021

Alen Zorja

potpis studenta/ice