# Web aplikacija za rezervaciju termina kod privatnih specijalista različitih grana medicine

**Fabijanić, Dora**

**VELEUČILIŠTE U VIROVITICI**

**Dora Fabijanić**

# Web aplikacija za rezervaciju termina kod privatnih specijalista različitih grana medicine

**ZAVRŠNI RAD**

**Virovitica, 2022.**

**VELEUČILIŠTE U VIROVITICI**

**Preddiplomski stručni studij Računarstva, smjer Programsko inženjerstvo**

Dora Fabijanić

# Web aplikacija za rezervaciju termina kod privatnih specijalista različitih grana medicine

**ZAVRŠNI RAD**

podnesen Veleučilištu u Virovitici
radi stjecanja akademskog zvanja
stručnog prvostupnika/stručne prvostupnice
elektrotehnike/računarstva

**Virovitica, 2022**

Veleučilište u Virovitici

*Preddiplomski stručni studij Računarstva - Smjer Programsko inženjerstvo*

**OBRAZAC 1b**

**ZADATAK ZAVRŠNOG RADA**

**Student/ica:** **DORA FABIJANIĆ** **JMBAG:** **0125161729**

**Imenovani mentor: Marko Hajba, mag. math., pred.**

**Imenovani komentor: -**

**Naslov rada:**

***Web aplikacija za rezervaciju termina kod privatnih specijalista različitih grana medicine***

**Puni tekst zadatka završnog rada:**

Izraditi web aplikaciju koja služi za rezervaciju termina kod privatnih specijalista različitih grana medicine.Za izradu aplikacije potrebno je koristiti.NET 6 s okvirom Blazor, a za rad s bazom podataka SQL Server. Bazu je potrebno napuniti testnim podacima. Potrebno je kreirati mikroservisnu arhitekturu i koristeći Docker demonstrirati interakciju mikroservisnih aplikacija.

Korisnici mogu biti liječnici, registrirani korisnici i posjetitelji. Web aplikacija treba imati različite funkcionalnosti, ovisno o tipu korisnika koji ju koristi:

- Omogućiti registraciju i prijavu korisnika.
- Administratori brinu o održavanju funkcionalnosti aplikacije, primaju prijedloge i rješavaju probleme koje mogu prijaviti ostali korisnici.
- Liječnici mogu dodavati i ažurirati svoje podatke o djelatnosti, lokaciji i sl. Kreiraju slobodne termine i ažuriraju status postojećih termina.
- Registrirani korisnici i posjetitelji mogu pregledavati dostupne liječnike i ordinacije po željenim kriterijima –npr. grani medicine, lokaciji, prosječnoj ocjeni itd.
- Registrirani korisnici mogu zakazati termin kod odabranog liječnika prema zadanoj proceduri, koja uključuje unos potrebnih informacija i dodavanje potrebnih dokumenata. Liječnik mora odobriti traženi termin. Registrirani korisnici mogu dodavati ocjene i komentare na uslugu i liječnika.
- Liječnici će imati uvid u osnovnu statistiku o svom poslovanju, npr. broj radnih sati u tjednu ili mjesecu, najtraženiji dan za termine i sl.

Osim programskog rješenja, u pisanom dijelu završnog rada opišite ukratko korištene tehnologije te detaljno opišite arhitekturu sustava i odabrane procese i/ili funkcije unutar

same aplikacije. Prilikom opisivanja, osim neformalnih koristite i neke od formalnih metoda koje poznajete.

---

**Datum uručenja zadatka studentu/ici**: 28.07.2022.
**Rok za predaju gotovog rada:** 09.09.2022.

Mentor:

**Marko Hajba, mag. math., pred.**

Marko Hajba

*Dostaviti:*
1. Studentu/ici
2. Povjerenstvu za završni rad - tajniku

**VELEUČILIŠTE U VIROVITICI** **Završni rad**

**Temeljna dokumentacijska kartica**

# Web aplikacija za rezervaciju termina kod privatnih specijalista različitih grana medicine


## Sažetak

Kako Internet postaje sve zastupljeniji u svakodnevnom životu, tako raste i uporaba web aplikacija. Povećanjem konkurencije na tržištu web aplikacija potaknut je razvoj brojnih okvira za razvoj web aplikacija. Oni omogućavaju sve brži, jednostavniji i fleksibilniji razvoj i održavanje aplikacija. Jedan od takvih okvira koji pruža brzi i fleksibilni razvoj je Blazor. On za razliku od većine okvira pruža mogućnost kompletne implementacije poslovne logike klijentskog i poslužiteljskog dijela web aplikacije pomoću C# programskog jezika. Komunikacija u stvarnom vremenu postignuta je SignalR bibliotekom, dok modeli izvođenja web aplikacije Blazor Server i Blazor WebAssembly omogućavaju developerima korištenje više različitih rješenja za potrebe web aplikacije. Osim Blazora, Docker *runtime* okruženje za kontejnerizaciju olakšava razvoj web aplikacije. Kako kontejneri sadrže sve potrebne izvršne datoteke, binarni kod, biblioteke i konfiguracijske datoteke za izvršavanje kontejnerizirane aplikacije developeri ne moraju voditi računa o verzijama okvira i biblioteka na svojim računalima nego jednostavno pokreću aplikacije. U radu će biti opisan klasični razvoj aplikacija, okvir Blazor, *runtime* okruženje Docker i sustav baze podataka, a na kraju i praktični dio u obliku aplikacije koja je razvijena pomoću opisanih tehnologija u teorijskom dijelu.



(39 stranica, 24 slika, 10 referenci)

**Ključne riječi:** Blazor, C#, Docker, SignalR, web aplikacija, WebAssembly

**Mentor:** Marko Hajba, mag.math., pred.

**VIROVITICA UNIVERSITY OF APPLIED SCIENCES** **Bachelor's Thesis**

**Basic documentation card**

## THESIS TITLE


### Abstract

As the Internet becomes more prevalent in everyday life, so does the use of web applications. Increasing competition in web application development has encouraged the development of numerous web application development frameworks. They enable faster, simpler and more flexible application development and maintenance. One such framework that provides fast and flexible development is Blazor. Unlike most frameworks, it provides the possibility of complete implementation of the business logic of the client and server part of the web application using the C# programming language. Real-time communication achieved with the SignalR library, Blazor Server and Blazor WebAssembly web application execution models allow developers to use several different solutions for web application needs. In addition to Blazor, the Docker runtime environment for containerizing provides simplification during web application development. As containers contain all the necessary executable files, binary code, libraries and configuration files to run a containerized application, developers do not have to take care of the versions of frameworks and libraries on their computers, but simply run the applications. The paper will describe classic application development, the Blazor framework, the Docker runtime environment and the database system, and finally the practical part in the form of an application developed using the technologies described in the theoretical part.



(39 pages, 24 figures, 10 references)

**Keywords:** Blazor, C#, Docker, SignalR, web application, WebAssembly

**Supervisor:** Marko Hajba, mag.math., lecturer

# Sadržaj

# 1. Uvod

Internet preglednik je jedna od osnovnih i svakodnevno korištenih aplikacija na računalu. Na početku razvoja Interneta, u listopadu 1994. godine objavljen je preglednik Netscape Navigator koji je zaslužan za širenje popularnosti Interneta u svijetu. Godinama, Microsoft je zanemarivao web i fokusirao se na vlastite platforme. Kako je web dobivao sve veći utjecaj, Microsoft je odlučio napraviti svoj web preglednik te je u kolovozu 1995. objavio svoj preglednik Internet Explorer. Prvi period "sukoba" različitih internetskih preglednika bio je obilježen nužnim povećavanjem kompatibilnosti s HTML-om (engl. *HyperText Markup Language)*.

Širenjem uporabe Interneta, razne tvrtke su izdavale svoje internetske preglednike, čime se konkurencija pojačavala. Kompatibilnost s HTML-om je otišla u zaborav uvođenjem HTML5 i modernih preglednika kao Google Chromea, Firefoxa, Safaria, Opere i Microsoft Edgea. HTML5 definira seriju standardnih HTML elemenata i pravila renderiranja istih. Razvoj programskog jezika ECMAScripta, odnosno JavaScripta, te zatim AJAX-a (engl. *Asynchronous JavaScript And XML*) koji opisuje set tehnologija koje koriste JavaScript za učitavanje podataka sa servera i korištenje tih podataka za ažuriranje HTML preglednika. Neke od prvih primjena AJAX-a su bile aplikacije Outlook Web Access i Google Maps. Takav razvoj mogle su si priuštiti samo velike tvrtke, zbog čega su uvedene JavaScript biblioteke jQuery i knockout.js.

Današnje web aplikacije su razvijene okvirima Angularom, Reactom i Vue.js-om koji koriste Javacript ili jezik više razine TypeScript. Kompatibilnost s JavaScript programskim programom postaje glavna odrednica daljnjeg razvoja internetskih preglednika, gdje se tvrtke koje nude internetske preglednike natječu u brzini izvođenja i razvojem vlastitih mehanizama. Ti mehanizmi koriste *Just-In-Time* kompilaciju gdje se Javascript konvertira u nativni kod preglednika.

Veliki nedostatak takvog pristupa je potreba preuzimanja Javascripta u preglednik, koji se zatim parsira, kompajlira i pretvara u nativni kod. Okvir Blazor ima prednost, jer omogućava pisanje programskog koda u programskom jeziku C# na klijentskoj i poslužiteljskoj aplikaciji, a pri tome nije potrebna uporaba programskog jezika JavaScript.

# 2. Klasičan razvoj web aplikacija

„Web development“ se odnosi na izradu i implementaciju stranice ili aplikacije kojoj mogu pristupiti korisnici preko Interneta te se bazira na modelu klijent-poslužitelj. Takva arhitektura se sastoji od komponenata podijeljenih u klijentske i poslužiteljske komponente. Klijentska strana je realizirana programom web-preglednikom, a poslužiteljska strana programom koji poslužuje web-stranice. Poslužiteljska komponenta neprestano osluškuje zahtjeve klijentskih komponenata te se njihova komunikacija odvija protokolom HTTP. Kada klijent traži uslugu od poslužitelja, poslužitelj prihvaća taj zahtjev te ga obrađuje i šalje povratni odgovor klijentu. Taj odgovor je sadržaj web-stranice koja obično referencira neko različitih datoteka u različitim formatima. [1]

## 2.1. Klijentska strana aplikacije

Razvoj klijentske strane aplikacije podrazumijeva implementaciju koda koju preglednik može interpretirati. Taj kod šalje se kao odgovor poslužitelja i sadrži informacije kako se internetska stranice treba prikazivati i kako treba korisnik komunicirati s elementima na stranici. Najzastupljenije tehnologije korištene za pisanje koda klijentske aplikacije su HTML, CSS i JavaScript. [1]

### 2.1.1. Jezik HTML

HTML je jezik za označavanje teksta. Osmišljen je za definiranje, prezentiranje i obradu teksta. On je zapravo jezik weba koji svi preglednici koriste za prikazivanje teksta, slika, videa i drugih materijala na web stranicama. Osnovna struktura HTML koda je sljedeća:

*Isječak programskog koda 1: Primjer HTML koda*

```
1.  <!DOCTYPE html>
2.  <html>
3.  <head>
4.  <title>Moj prvi html</title>
5.  </head>
6.  <body>
7.  <h1>Hello World!</h1>
8.  </body>
9.  </html>
```

Struktura HTML dokumenta je poput stabla. Prva razina je oznaka HTML koji je ujedno i korijenski element stranice koji se sastoji od zaglavlja (engl. *head*) i tijela (engl.

*body*). Zaglavlje je opcionalni dio sintakse koji služi za definiranje detalja stranice, dok je tijelo zapravo dio koji obuhvaća sadržaj koji će se prikazati na web stranici.

Oznake koje se koriste unutar HTML-a mogu se podijeliti u dvije glavne kategorije bazirane na predefiniranom ponašanju ili kako se prikazuju na web stranici su: *block-level* elementi i *inline* elementi. *Block-level* elementi su glavni kontejneri elemenata koji prikazuju sadržaj u novom retku, a primjeri su div, p, h1-h6. *Inline* elementi su kontejneri elemenata koji prikazuju sadržaj u istoj liniju u kojoj je i prethodni sadržaj kao što je oznaka span. [1]

### 2.1.2. CSS

HTML omogućuje dodavanje osnovnog sadržaja i izgradnju temeljne web stranice, ali bez stilova stranica izgleda sirovo. Sav sadržaj stranice potrebno je stilizirati kako bi bio privlačniji i intuitivniji krajnjim korisnicima. CSS (engl. *Cascading Style Sheets*) preuzima potpunu kontrolu nad stiliziranjem weba. Struktura CSS koda je vrlo jednostavna i sastoji se od skupova CSS pravila. CSS skup pravila se sastoji od selektora i deklaracijskog bloka. Selektor ukazuje na HTML element koji se želi stilizirati, a deklaracijski blok sadrži jednu ili više deklaracija koje će oblikovati HTML element.

Postoje tri načina uključivanja CSS-a u HTML dokumentu. Prvi *inline* stil je uvođenjem unikatnog stila koji se odnosi samo na jedan element. Atribut *style* je uključen u element kojem je potrebno pridodati određeni stil, ali taj način se ne preporuča. Drugi način je dodavanje internog koda u HTML dokumentu putem elementa *style* unutar *head* sekcije dokumenta. Zatim treći način je dodavanje odvojene datoteke koja sadrži CSS kod koji je ujedno i najfleksibilniji način. Isti stilovi mogu biti primijenjeni na više stranica dodavanjem reference na odvojenu datoteku koristeći link oznaku unutar zaglavlja HTML datoteke.

Primjenom višestrukih stilova na stranici pojavljuje se kaskadni efekt. To znači da će se primjenjivati prema određenom redoslijedu prioriteta. Najveći prioritet imaju stilovi pisani *inline* unutar HTML elementa, zatim interni ili odvojeni stilovi ovisno o poretku unutar zaglavlja i na kraju stilovi definirani unutar web preglednika.

Web preglednik prvo pristupa HTML dokumentu koji sadrži referencu na stil. Prvo se učita HTML-ov dokument koji se mora interpretirati kako bi se i prikazao. Ako unutar tog dokumenta ima referenca na vanjski CSS dokument, onda se šalje zahtjev za dohvaćanje tog dokumenta. Tek kada se dohvati vanjski dokument, onda kreće prikazivanje dokumenta u pregledniku. [1]

### 2.1.3. JavaScript

Produkt HTML-a i CSS-a je stilizirani sadržaj stranice, ono što je još potrebno kako bi korisnici mogli koristiti stranicu su interakcije. Dodavanjem interakcija statičnu stranicu pretvara u web aplikaciju. Kako bi definirali kako se stranica treba ponašati kada korisnik želi stvoriti interakciju s nekim elementima potreban je JavaScript. JavaScript je skriptni programski jezik koji je osmišljen kako bi mogli dodati logiku i definirati ponašanje web-a. Njegovim korištenjem web stranica postaje dinamična i interaktivna. Prednost JavaScripta leži u tome što se može koristiti na serverskoj i klijentskoj strani te se može izvesti u pregledniku ili na serveru.

Jedan način uključivanja unutar HTML dokumenta vrši se <script> oznakom koja direktno sadrži kod JavaScrtipta unutar zaglavlja ili tijela dokumenta. Drugi način je kod pisan u odvojenoj datoteci te ga se uključuje pomoću <script src="myJScript.js"></script> unutar zaglavlja ili tijela. Ovaj način se preporuča jer čini kod čišćim. Neki primjeri koje omogućuje JavaScript su klik na gumb, pomicanje miša, promjena atributa ili stilova pri interakciji, dodavanje *cookiea*. Osim toga, JavaScript putem AJAX-a uspostavlja komunikaciju s poslužiteljem [1].

## 2.2. Poslužiteljska strana aplikacije

Poslužiteljska strana je strana koju korisnici općenito ne vide i nemaju interakciju s njom. Koristi se za pohranu i upravljanje podacima. Neki od najčešćih jezika za pisanje poslužiteljskog koda su PHP, Java, Python, C#. Poslužitelji kada zaprime zahtjev od klijenta, obrađuju ga i šalju odgovor kao HTTP odgovor. Odgovor sadrži status o uspješnosti obrade zahtjeva, a tijelo odgovora može sadržavati očekivani sadržaj kao što je HTML stranica, slika ili neki drugi element prikaziv na web stranici.

Postoje statičke i dinamičke web stranice. Statička stranica je ona koja prikazuje uvijek isti  sadržaj na zahtjev, a šalje ga poslužitelj. To znači da web preglednik šalje HTTP „GET“ zahtjev sa određenim URL-om, a zatim poslužitelj vraća dokument ako je uspješno dohvatio traženi dokument. Za razliku od statičnih, kod dinamičkih stranica sadržaj odgovora se generira dinamički i to samo kada je potrebno. Uobičajen postupak izgradnje sadržaja ovakvih stranica je umetanje podataka iz baze u rezervirana mjesta u HTML predložak. Odgovor je ovisan o URL-u koji je temeljen na informacijama koje daje korisnik ili pohranjenih postavki. [1]

# 3. Blazor

Blazor je Microsoftov okvir za razvoj web aplikacija. Naziv Blazor je spoj riječi Preglednik (engl. *Browser*) i Razor. Umjesto izvršavanja Razor datoteka na poslužitelju kako bi se HTML prikazao na pregledniku, Blazor omogućava izvršavanje Razora i na klijentu. Dva su načina izvršavanja aplikacija pisanih u Blazoru. Može se izvršavati na poslužitelju i u tom slučaju se preglednik ponaša kao terminal ili se pokreće u samom pregledniku pomoću WebAssembly-a.

Uvijek je treba razmisliti o izboru tehnologija za razvoj aplikacija. Poželjno je proučiti kako okviri i alati utječu na performanse rada, koje su potrebe aplikacije koja će se razvijati te koje su prednosti i mane odabranih tehnologija. Revolucionarna promjena koju dobivamo korištenjem okvira Blazor je mogućnost pisanja klijentske strane aplikacije u cijelosti koristeći programski jezik C# bez potrebe za uključivanjem JavaScripta kao što je tipično za React i Angular. Zajedno s ASP.NET Core poslužiteljskim kodom stvoreno je .NET *end-to-end* okruženje. [9]

## 3.1. SignalR

SignalR je Microsoftova biblioteka koja omogućuje razvoj ASP.NET aplikacija u stvarnom vremenu. Primjenjuje se najviše u aplikacijama koje zahtijevaju *push* obavijesti od poslužitelja do klijenta i koje pružaju funkcionalnost chata. SignalR je pružao stalnu vezu između klijenta i poslužitelja. Koristi Hubs API za slanje obavijesti s poslužitelja na klijenta i podržava više kanala kao što su WebSocket, *long pooling* i SEE tehnologiju. Osim toga, podržava više klijenata u rasponu od C#/C++ do JavaScripta. Nadalje, svoju skalabilnost pokazuje i podržavanjem SQL Servera, Redisa, Azure Service Busa. Kako se razvojna paradigma pomicala prema ASP.NET Coreu, Microsoft je redizajnirao i ponovo izgradio SignalR kako bi podržao okvir ASP.NET Core. Redizajn i ponovni razvoj bio je potreban zbog značajki ASP.NET Corea kao što su HttpContext, konfiguracija i ovisnosti. [10]

## 3.2. Modeli izvođenja Blazor aplikacija

U ovisnosti načina izvođenja Blazor aplikacije nastala su dva modela izvođenja Blazor aplikacija. Blazor WebAssembly je model izvođenja aplikacije na klijentskoj strani, a Blazor Server model izvođenja na poslužiteljskoj strani. Oba modela koriste Razor za opisivanje

HTML sadržaja koji se prikazuje, ali se značajno razlikuju u načinu prikazivanja komponenti. [9]

### 3.2.1. Blazor WebAssembly

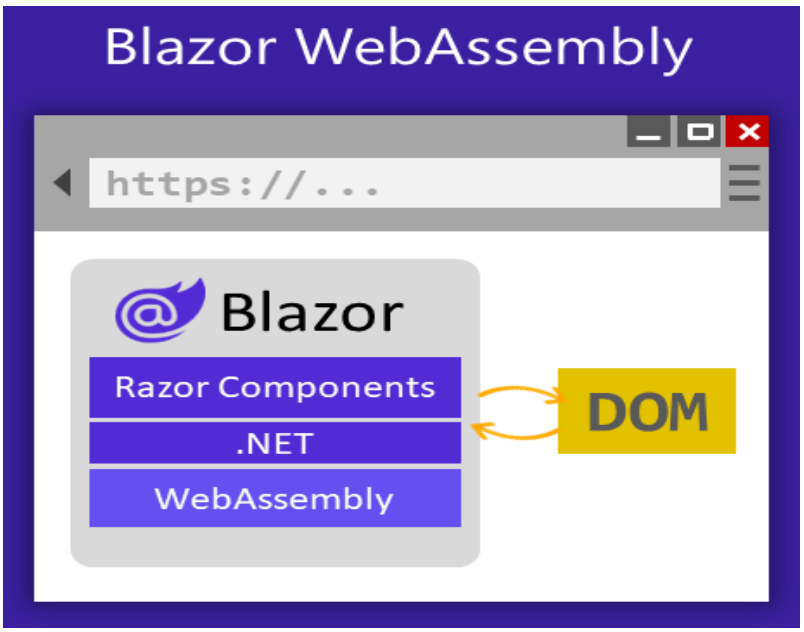
WebAssembly je jedno od rješenja za performanse rada aplikacije te je postao W3C standard s ciljem prevladavanja ograničenja JavaScripta i otvaranja vrata drugim jezicima u preglednicima. On omogućuje parsiranje i kompajliranje aplikacije na serveru, prije nego što ju korisnik otvori na pregledniku. Programski kod se kompajlira u formatu WASM, koji se zatim preuzme u pregledniku i *Just-In-Time* kompajlira u nativni kod.

Prema izvoru [3] Webassembly je binarni format instrukcija za virtualni stroj temeljen na stogu. WASM je dizajniran kao prijenosni cilj za kompilaciju jezika visoke razine kao što je C/C++/Rust, što omogućuje implementaciju na webu za klijentske i poslužiteljske aplikacije. Dakle, WebAssembly je novi optimiziran binarni format za izvođenje u pregledniku koji nije pisan u JavaScriptu. Ali kako je to novi standard, dizajniran je za rad s JavaScriptom, tako da je moguće pozivanje JavaScript skripte pomoću WebAssembly funkcije ili pozvati funkciju iz JavaScript skripte u WebAssembly. Takva interakcija može biti korisna u hibridnom scenariju i moćan je alat koji nudi najbolje iz oba svijeta.

Ujedno i današnji najpopularniji pristup Blazora je kompajliranje .NET Core runtime-a u WASM, a on se pokreće u pregledniku gdje se izvršava .NET Intemediate Language. Prednost takvog pristupa je jednostavnost pokretanja .NET sklopova bez potrebe da ih se prvo kompajlira u WASM. Osim toga, Blazor je temeljen na popularnom ASP.NET MVC pristupom za razvoj aplikacija koje se pokreću u preglednicima te se za generiranje HTML-a na serveru koristi Razor sintaksa. Izrađena Razor datoteka se kompajlira u .NET klasu koja se zatim izvršava u Blazor mehanizmu. Rezultat kompajlianja je struktura zvana „*render tree*“ koja se šalje JavaScriptu kako bi ažurirao DOM (engl. *The Document Object Model*). Takav model omogućuje i izradu progresivnih aplikacija. Osim toga postoje kompajleri koji prevode aplikacije pisane jezicima C++ i Rust u WASM te omogućuju njihovo pokretanje u preglednik te svi glavni preglednici podržavaju WebAssembly.

Ovisno o načinu razvoja Blazor WebAssembly aplikacije, ona se može izvoditi u potpunosti na izoliranom klijentu bez potrebe za poslužiteljem ili komunicirati s poslužiteljem samo kada su potrebni podaci od strane poslužitelja. Neke aplikacije mogu funkcionirati samostalno nakon preuzimanja aplikacije i njezinih ovisnosti u preglednik klijenta kao što je na primjer kalkulator. Još jedan način izvođenja je da aplikacija može raditi većinu vremena

izvan mreže. Kada su joj potrebni podaci s poslužitelja, komunicira s njim pomoću API poziva ili SignalR-a.




*Slika 1. Blazor WebAssembly [5]*

Nije potrebno instalirati nikakva proširenja na klijentu kako bi se Blazor WebAssembly izvršio u preglednik koji ga izvorno podržavaju, što je prednost rada sa WebAssemblyjem. Podrška za WebAssembly je zagarantirana u gotovo svim najnovijim verzijama preglednika, što je uobičajena praksa za W3C standard. [5] [6] [9]

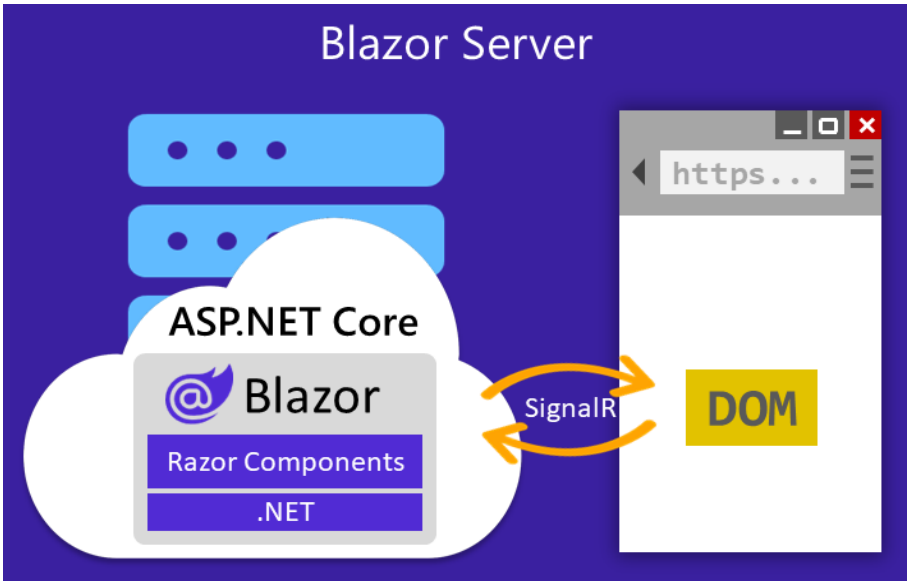
### 3.2.2. Blazor Server

Za razliku od klijentskog modela izvođenja WebAssembly, kod poslužiteljskog modela izvođenja Blazor Server aplikacija se izvršava na poslužitelju unutar ASP.NET Core aplikacije. Ažuriranja korisničkog sučelja se odvijaju preko SignalR veze. Takav model je prikazan na slici 2.

Renderiranjem svake linije Razor koda u HTML u obliku teksta poslužitelj raspolaže instancom stranice uključujući bilo koje stanje proizvedeno. Kada se pojavi sljedeći zahtjev za tu stranicu, cijela stranica se ponovno renderira u HTML-u i šalje klijentu.

Zapravo, Blazor Server proizvodi grafikon komponenti za prikaz sličan HTML ili XML DOM koji uključuje stanje sadržano u svojstvima i poljima, a klijentima šalje binarnu reprezentaciju oznake na renderiranje. Uspostavom veze klijent-poslužitelj statički unaprijed renderirane komponente zamjenjuju se interaktivnim elementima. Interakcijom korisnika pokreću se ažuriranja korisničkog sučelja koja podrazumijevaju ponovno renderiranje

grafikona i izračunavanje razlika korisničkog sučelja. Ona predstavlja najmanji skup DOM izmjena potrebnih za ažuriranje koja se šalje klijentu u binarnom obliku te se primjenjuje u pregledniku. [9]




*Slika 2. Blazor Server[5]*

## 3.3. Komponente Blazora

Blazor aplikacija se zasniva na komponentama. Svaka klasa koja proizlazi iz klase ComponentBase je Blazor komponenta. Svaka Razor datoteka u Blazoru predstavlja komponentu i svaka komponenta se može napraviti od drugih komponenata te tako postati ugniježđene. Osim toga, komponenta u Blazoru može biti bilo koji element korisničkog sučelja od stranice, dijaloga ili forme. Sve te komponente definiraju logiku renderiranja fleksibilnog korisničkog sučelja i interakciju s korisnicima. Nadalje, mogu se dijeliti kao Nuget paketi ili Razor biblioteke klasa. [9]

### 3.3.1. Razor sintaksa

Blazor komponente koriste Razor sintaksu. Direktive i njihovi atributi su značajke Razora koje Blazor komponente koriste.

Razor direktive predstavljene su implicitno s ključnim riječima iza simbola @ te utječu na način na koji se analizira neki pogled ili omogućuje razne funkcije. Tako na primjer direktiva @code omogućuje komponenti dodavanje C# članova, metoda i polja. @attribute direktiva dodaje određeni atribut klasi generirane stranice ili prikaza. Zatim, @implements direktiva omogućava implementaciju sučelja za generiranu klasu, a @inherits pruža potpunu kontrolu nad klasom koju pogled nasljeđuje. To su samo neki od raznih primjera direktiva.

Usmjerivanje u Blazoru se postiže korištenjem direktive @page s predloškom rute dostupnoj svakoj komponenti u aplikaciji. Kompajliranjem stranice s @page direktivom generira se klasa sa RouteAttributom koja sadrži predložak rute. Tijekom rada usmjerivač prema traženom URL-u traži komponentu s atributom *RouteAttribute* te prikazuje komponentu koja ima predložak rute odgovarajućeg URL-u.

Razor atributi direktive su predstavljeni isto kao direktive ključnom riječi nakon simbola @, ali oni utječu na način kako se neki element parsira ili omogućava razne funkcionalnosti. Neki primjeri su @bind kojim se postiže povezivanje podataka u komponentama i @on{EVENT} koji pruža značajke rukovanja događajima za komponente. [5]

### 3.3.2. Struktura komponenta

Blazor komponente se definiraju koristeći Razor markup, C# i HTML. Kompajliranjem aplikacije, HTML markup i C# logika se pretvaraju u klasu komponente. Članovi klase komponente definirani su unutar jedne ili više direktiva @code. Također unutar njih su definirana stanja komponenta i navedene metode kako se njihova stanja mijenjaju.

*Isječak programskog koda 2. Primjer Blazor komponente*

```
1.  @page "/index"
2.
3.  <h1 style="font-style:@headingStyle">@headingText</h1>
4.
5.  @code {
6.      private string headingStyle = "oblique";
7.      private string headingText = "Naslov!";
8.  }
```

To je prvi način implementacije komponente s .razor ekstenzijom. Drugi način bi bio razdvajanje markupa u jednu .razor datoteku i C# koda u drugu istoimenu datoteku ekstenzije .razor.cs. [5]

## 3.4. Usporedba Blazora i JavaScript okvira

JavaScript je programski jezik najčešće korišten za razvoj klijentske strane aplikacije te se koristi za manipuliranje i kontrolu nad podacima. Brojni okviri koriste Javascript za razvoj web stranica i aplikacija, a najpoznatiji su React.js, Angular.js i Vue.js.

Javascript okviri nude samo jedan model izvođenja aplikacije, a kao što je već navedeno u poglavlju 2.1 Blazor nudi dva modela izvođenja aplikacije te za njega nije

potrebno instalirati nikakva proširenja u preglednik. Također, u okviru Blazor možemo dijeliti kod između klijentske i poslužiteljske strane, dok se Javascript koristi samo za razvoj klijentske strane. Ipak, treba uzeti u obzir da je Blazor zajednica programera relativno mala i da njezina baza znanja još uvijek sazrijeva.

Iako je teško usporediti React i Blazor okvir, za sada je React dobro uspostavljen i zaslužio je svoje poštovanje dokazanom reputacijom i snažnom zajednicom. Okruženje React-a puno je biblioteka i alata koji omogućavaju razvoj optimiziranih aplikacija. U usporedbi, Blazor je novi okvir, ali predstavlja dobru alternativu React-u. Bez obzira na starost Blazor-a donosi nekoliko naprednih značajki kao što su korištenje Nuget paketa, istih komponenti na klijentskoj i poslužiteljskoj strani aplikacije, ugrađena podrška za usmjeravanje, validaciju i rukovanje obrascima te slično Reactu, Blazor možemo implementirati kao statične datoteke.

Isto kao i kod usporedbe s React-om, Blazor je u ranoj fazi razvoja, dok Angular ima već dobro uspostavljene biblioteke i zajednicu. Prednost Blazora je spremanje stanja komponenata svakog klijenta na poslužitelju za razliku od okvira Angular, gdje svaki klijent mora imati aktivnu vezu. Nadalje, Angular je kompliciran za učenje.

# 4. Docker

Većina aplikacija je pokrenuta na poslužitelju i prije je bilo moguće pokrenuti samo jednu aplikaciju na pojedinom poslužitelju. Odnosno, Windows i Linux nisu imali tehnologije za sigurno pokretanje višestrukih aplikacija na istom serveru, a to znači da svaki puta kada je tvrtka htjela pokrenuti novu aplikaciju trebala je kupiti i novi server. To je dovelo do rasipanja kapitala tvrtki pri kupnji servera pogrešnih kapaciteta ili performansi. Revolucionarnu promjenu uvela je tvrtka VMware, Inc., koja na prijelazu u 21. stoljeće omogućava kreiranje virtualnih strojeva za sigurno pokretanje više poslovnih aplikacija na jednom poslužitelju. Uvođenjem virtualnih strojeva IT odjeli više nisu morali nabavljati novi poslužitelj pri svakom zahtjevu tvrtke za novom aplikacijom, nego su mogli iskoristiti već ionako slobodne kapacitete postojećih poslužitelja.

Virtualni strojevi su daleko od savršenih. Činjenica da svaki virtualni stroj zahtjeva vlastiti namjenski operativni sustav velika je mana. Svaki operativni sustav troši CPU, RAM i ostale resurse koji bi se inače mogli koristiti za pokretanje više aplikacija. Osim toga, virtualni strojevi se sporo pokreću i migracija i premještanje radnih opterećenja virtualnog stroja je kompliciranа.

Kao rješenje za te nedostatke, sve popularnije postaju tehnologije kontejnerizacije. U modelu kontejnera, kontejner je otprilike analogan virtualnom stroju, ali kontejneri ne zahtijevaju vlastiti potpuno razvijeni operacijski sustav. Svi kontejneri na istom *hostu* dijele njegov operacijski sustav. To rezultira smanjenjem opterećenja sistemskih resursa, kao što su CPU, RAM i pohrana. Također, za razliku od virtualnih strojeva, kontejneri se brzo pokreću i vrlo lako se prenose na primjer s računala na oblak. Konačni rezultat kontejnera je ušteda vremena, resursa i kapitala.

Prema [4] definicija kontejnera bi bila sljedeća: „Kontejneri su lagani paketi aplikacijskog koda zajedno s njegovim ovisnostima kao što su specifične verzije izvršavanja programskog koda i biblioteka potrebnih za pokretanje softverskih usluga“. Odnosno oni sadrže sve potrebe elemente da bi se softver pokrenuo u bilo kojoj okolini. Kontejneri vizualiziraju operativni sustav i mogu raditi na bilo kojem računalu, što je omogućilo razvojnim timovima brzu kretnju, učinkovitu implementaciju i rad u puno većem opsegu.

Oni se koriste za pokretanje svih aplikacija od malih mikroservisa ili softverskih procesa do velikih aplikacija. Sadrže sve potrebne izvršne datoteke, binarni kod, biblioteke i

konfiguracijske datoteke. Rezultat kontejnerizacije aplikacije je slika kontejnera koja se onda može pokrenuti na nekoj kontejnerskoj platformi.

Uzimamo poslužitelj koji pokreće jedan operativni sustav, a na kojemu su pokrenuta tri kontejnera (kontejnizirane aplikacije). Svi kontejneri tada međusobno dijele jezgru operativnog sustava. Ti dijelovi operativnog sustava koje kontejneri dijele međusobno su samo za čitanje, a svaki kontejner ima svoj vlastiti *mount* za pisanje. [4]

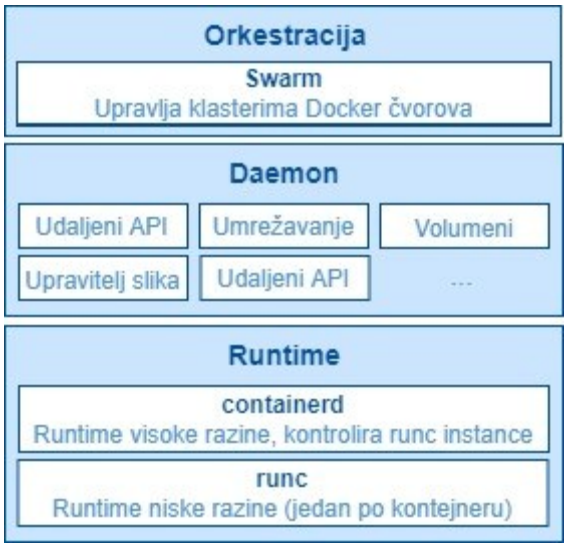
## 4.1. Struktura Docker-a

Docker je popularno *runtime* okruženje koje se koristi za stvaranje, izgradnju i upravljanje kontejnerima. Koristi *Docker slike* za implementaciju kontejnerskih aplikacija ili softvera u više okruženja, od razvoja do testiranja i puštanja u rad. Docker je izgrađen na otvorenim standardima i funkcijama u većini uobičajenih okruženja. Sastoji se od tri sloja: runtime, daemon i orkestrator.

Runtime radi na najnižoj razini i odgovorno je za pokretanje i zaustavljanje rada kontejnera. Docker implementira slojevitu runtime arhitekturu s runtimeovima visoke i niske razine koji rade zajedno. Runtime niske razine naziva se runc i referentna je implementacija OCI (engl. *Open Container Initiative*) runtime specifikacije. Njegov posao je pokretanje i zaustavljanje rada kontejnera. Svaki kontejner na Docker čvoru ima runc instancu koja njime upravlja. Runtime visoke razine naziva se containerd. On je odgovoran za više funkcija od runc-a, upravlja cijelim životnim ciklusom kontejnera što uključuje povlačenje slika, kreiranje mrežnih sučelja i upravljanje runc instancama niže razine. Tipična Docker instalacija ima jedan proces containerda koji kontrolira runc instance povezane sa svakim pokrenutim spremnikom.

Docker daemon je iznad containerda i njegovi su zadaci izlaganje API-ja, upravljanje slikama, volumenima, mrežama i slično. Njegova primarna namjena je osiguravanje standardnog sučelja jednostavnog za korištenje.

Osim toga, Docker sadrži podršku za upravljanje odnosno orkestraciju klasterima koji pokreću Docker, a ta tehnologija se naziva Docker Swarm. Jednostavna je za korištenje, ali većina korisnika se odlučuje na korištenje Kubernetesa za orkestraciju zbog veće zajednice, pružanja mogučnosti upravljanja velikim arhitekturama i složenim opterećenjima, raznih integracija i ugrađenog nadzora. [8]




*Slika 3. Struktura Dockera [8]*

### 4.1.1. Docker slike

Docker slika je jedinica pakiranja koja sadrži sve što je potrebno za rad aplikacije, a to su kod aplikacije, njezine ovisnosti i konstrukcije operacijskog sustava. Slike se preuzimaju s registra slika, a najpoznatiji je Docker Hub. Slika se povlači s registra na lokalni *docker host* kako bi je Docker mogao koristiti za pokretanje jednog ili više kontejnera. Slike se sastoje od više slojeva koji su skupa reprezentirani kao jedan objekt uobičajeno male veličine. [8]

### 4.1.2. Dockerfile

Dockerfile je početna točka za kreiranje kontejnerskih slika, opisuje aplikaciju i govori Docker-u kako tu aplikaciju ugraditi u sliku. Direktorij koji sadrži aplikaciju i ovisnosti naziva se build context. Uobičajeno se u toj mapi/direktoriju nalazi i Dockerfile. Primjer Dockerfile-a sa izvora [4] za ASP.NET Core aplikaciju je sljedeći:

*Isječak programskog koda 3. Primjer Dockerfilea ASP.NET Core*

```
1.  # syntax=docker/dockerfile:1
2.  FROM mcr.microsoft.com/dotnet/sdk:6.0 AS build-env
3.  WORKDIR /app
4.
5.  # Copy csproj and restore as distinct layers
6.  COPY *.csproj ./
7.  RUN dotnet restore
8.
9.  # Copy everything else and build
10. COPY ../engine/examples ./
11. RUN dotnet publish -c Release -o out
12.
13. # Build runtime image
14. FROM mcr.microsoft.com/dotnet/aspnet:6.0
```

```
15. WORKDIR /app
16. COPY --from=build-env /app/out .
17. ENTRYPOINT ["dotnet", "aspnetapp.dll"]
```

Svi Dockerfileovi započinju naredbom FROM, koja predstavlja bazni dio slike, a ostatak aplikacije će biti dodan na bazu kao dodatni slojevi. Odabir baze ovisi i o vrsti aplikacije, za Linux aplikacije potrebna je slika bazirana na Linuxu, a za Windows aplikacije slika bazirana na Windowsu. Time je definiran bazni i trenutno jedini sloj slike. Instrukcija WORKDIR definira radni direktorij za ostale naredbe u datoteci. Naredba COPY *.csproj ./ kreira novi sloj i kopira projektu datoteku koja završava s .csproj. Nakon toga, potrebno je pozvati *dotnet restore* kako bi osigurali instalaciju svih potrebnih ovisnosti. Slijedi kopiranje ostalih datoteka aplikacija i buildanje aplikacije naredbom RUN dotnet publish –c Release –o out. Zatim, potrebno je ponovno definirati sliku i radni direktorij naredbama FROM i WORKDIR, ali razlika je što je ovoga puta potrebno kopirati built datoteke u app/out. Preostaje jedino pokretanje aplikacije, a to se izvršava naredbom ENTERYPOINT, koja prima niz koji se transformira u poziv naredbenog retka s argumentima. [6][8]

## 4.2. Deploy aplikacija koristeći Docker Compose

Većina suvremenih aplikacija koristi mikroservisnu arhitekturu, što podrazumijeva više manjih servisa koji međusobnom interakcijom i radom čine korisnu aplikaciju. Razvoj i upravljanje velikog broja servisa potencijalno može stvoriti razne probleme. Docker Compose olakšava rad s mikroservisima jer omogućuje opisivanje skripta za svaki mikroservis i korištenja puno docker naredba, Docker Compose omogućava opisivanje cijele aplikacije u jednoj konfiguracijskoj datoteci i deploy u jednoj liniji. Jednom kada je aplikacija deployana, upravljanje cijelim životnim ciklusom postaje vrlo jednostavno koristeći radani skup naredbi.

Za definiranje mikroservisnih aplikacija Docker Compose koristi YAML datoteke. Najčešće se nalazi u radnom direktoriju pod nazivom compose.yaml/compose.yml ili docker-compose.yaml/docker-compose.yml. YAML datoteka definira servise, mrežne postavke, volumene, konfiguracije i tajne. Ključ servisi je dio gdje se definiraju mikroservisi aplikacije, a Compose će deployati svaki servis kao zasebni kontejner. Ključ mreže govori Dockeru kako da kreira mrežne postavke. Docker Compose će prema zadanim postavkama stvoriti mrežu mostova. To su mreže s jednim *hostom* koje povezuju spremnike koji su na istom Docker *hostu*, ali mogu se definirati različite vrste mreža.

*Isječak programskog koda 4. Docker Compose datoteka*

```
1.  version: "3.9"
2.  services:
3.      web:
4.          build: .
5.          ports:
6.              - "8000:80"
7.          depends_on:
8.              - db
9.      db:
10.         image: "mcr.microsoft.com/mssql/server"
11.         environment:
12.             SA_PASSWORD: "Your_password123"
13.             ACCEPT_EULA: "Y"
```

U ovoj docker-compose.yml datoteci opisana su dva servisa odnosno kontejnera u aplikaciji web i db. Kontejner *web* opisan je naredbama *build*, *ports* i *depends_on*. Naredba *build:* . govori Dockeru da napravi sliku koristeći naredbe Dockerfilea u trenutnom direktoriju, *ports* u ovom primjeru određuje da se u kontejneru koristi port 80 i taj port prosljeđuje na port 8080 *hosta*, a *depends_on* određuje poredak pokretanja i zaustavljanja servisa što u ovome slučaju znači da će se prvo pokrenuti kontejner db. Definicija kontejnera db je jednostavnija. naredbom image Docker će stvoriti kontejner naziva db baziranog na slici mcr.microsoft.com/mssql/server. Naredbom docker-compose up pokrećemo deploy aplikacije kroz komandnu liniju. Očekivani naziv datoteke je docker-compose.yaml ili docker-compose.yml, ali moguće je specificirati datoteku različitog imena koristeći oznaku –f. [8]

# 5. Sustav baze podatka

Općenito sustav baze podataka je cjelokupna zbirka različitih komponenti softvera baze podataka i sastoji se od aplikacijskih programa za baze podataka, klijentskih komponenta, poslužitelja baze podataka i baze podataka. Aplikacijski program baze podataka je softver specijalne namjene koji dizajniraju i implementiraju korisnici ili tvrtke treće strane, a klijentske komponente su softveri baze podataka opće namjene koje osmišljavaju i implementiraju tvrtke baza podataka. Klijentske komponente omogućuju korisicima pristup spemljenim podacima na istom ili udaljenom računalu. Zadatak poslužitelja baze podataka je upravljanje pohranjenim podacima unutar baze. Komunikacija korisnika s bazom odvija se slanjem upita koji zatim poslužitelj obrađuje i vraća rezultat klijentu.

Sustavi baza podataka moraju omogućiti korisnička sučelja za kreiranje baza, dohvaćanje i izmjene podataka te komponente sustava za upravljanje pohranjenim podacima. Osim toga, treba pružati fizičku i logičku neovisnost podataka, kontrolu istovremenosti, integritet podataka, optimizaciju upita, sigurnosno kopiranje, oporavak i općenitu sigurnost baze podataka. [2]

## 5.1. Microsoft SQL Server

Komponenta Database Engine Microsoft SQL Servera je sustav relacijske baze podataka. Sustavi relacijskih baza podataka temelje se na modelu relacijskih podataka, a centar tog modela je relacija odnosno tablica. Relacijska baza podataka se sastoji od tablica koje se sastoje od jednog ili više stupaca i niti jedno ili više redova.

Jezik SQL Server relacijske baze podataka naziva se T-SQL (engl. *Transact-SQL*). T-SQL proizlazi iz danas najvažnijeg jezika baza podataka SQL (engl. *Structed Query Language*) koji je orijentiran na skupove odnosno zapise. To znači da SQL može postavljati upite za mnogo redaka iz jedne ili više tablica koristeći samo jednu naredbu. Kao neproceduralni jezik, on opisuje što korisnik želi, a sustav je odgovoran za pronalaženje načina za rješavanje tog zahtjeva. SQL se sastoji od dva podjezika, a to su jezik za definiranje podataka DDL (engl. *Data Definition Language*) i jezik za manipulaciju podacima DML (engl. *Data Manipulation Language*). DDL sadrži tri generičke SQL izjave CREATE, ALTER i DROP objekt. Te izjave stvaraju, mijenjaju i uklanjaju objekte baze podataka. S druge strane DML obuhvaća operacije manipuliranja podacima. Četiri generičke operacije za

manipulaciju su SELECT (dohvaćanje), INSERT (umetanje), DELETE (brisanje), UPDATE (ažuriranje). [2]

## 5.2. Dizajniranje baze podataka

Važna faza životnog ciklusa baze podataka je dizajniranje baze podataka. Bez dobro promišljenog dizajniranja baze, rezultirajuća baza najvjerovatnije neće zadovoljiti zahtjeve korisnika u pogledu perfomansi. Osim loših performansi, problem koji se još javlja pri lošem dizajniranju baze je redundancija podataka. Posljedica redundancije je postojanje anomalije podataka i korištenje nepotrebne količine prostora na disku. Ti problemi se uklanjaju procesima normalizacije i uklanjanja redundancije podataka.

Normalizacija podataka je proces testiranja postojećih tablica podataka kako bi se pronašle određene ovisnosti stupaca. U slučaju postojanja takve ovisnosti, rješenje je razdvajanje takve tablice u najčešće dvije s ciljem uklanjanja te ovisnosti. Ovaj proces se ponavlja sve dok se takve ovisnosti ne razrješe.

Proces uklanjanja redundacije temelji se na teoriji funkcionalnih ovisnosti. Funkcionalna ovisnost znači da se korištenjem poznate vrijednosti jednog stupca uvijek može jedinstveno odrediti odgovarajuća vrijednost drugog stupca. [2]

### 5.2.1. Model veza i entiteta

Model veza i entiteta ili kraće ER model (engl. *Entity-relationship model*) je konceptualna shema koja predstavlja apstrakciju realnog svijeta i njegov glavni cilj je uklanjanje redundantnosti podataka. Osnovni objekt ER modela je entitet, koji predstavlja objekt stvarnog svijeta. Atributi opisuju entitet i svaki entitet ima jedan ili više ključnih atributa odnosno svojstva koja ga opisuju i čije su vrijednosti jedinstvene za pojedini entitet. Ukoliko entitet ima više kandidata za ključ, tada se bira jedan od njih i proglašava primarnim ključem. Osim entiteta i atributa, veze su još jedan osnovi koncept ER modela. Veza entiteta postoji kada entitet ukazuje na jedan ili više drugih entiteta. Svaka veza između dva entiteta može biti tipa jedan-naprema-jedan (1:1), jedan-naprema-mnogo (1:N) ili mnogo-naprema-mnogo (M:N), te se ti omjeri nazivaju funkcionalnosti veze. Također veza može imati svoje atribute.

Za opisivanje ER modela koristi se grafička notacija ER dijagram. Entiteti se u ovom dijagramu prikazuju u pravokutnicima, s imenom unutar pravokutnika. Atributi prikazani u ovalnim oblicima i svaki atribut je pridružen određenom entitetu ili vezi ravnom linijom. Na

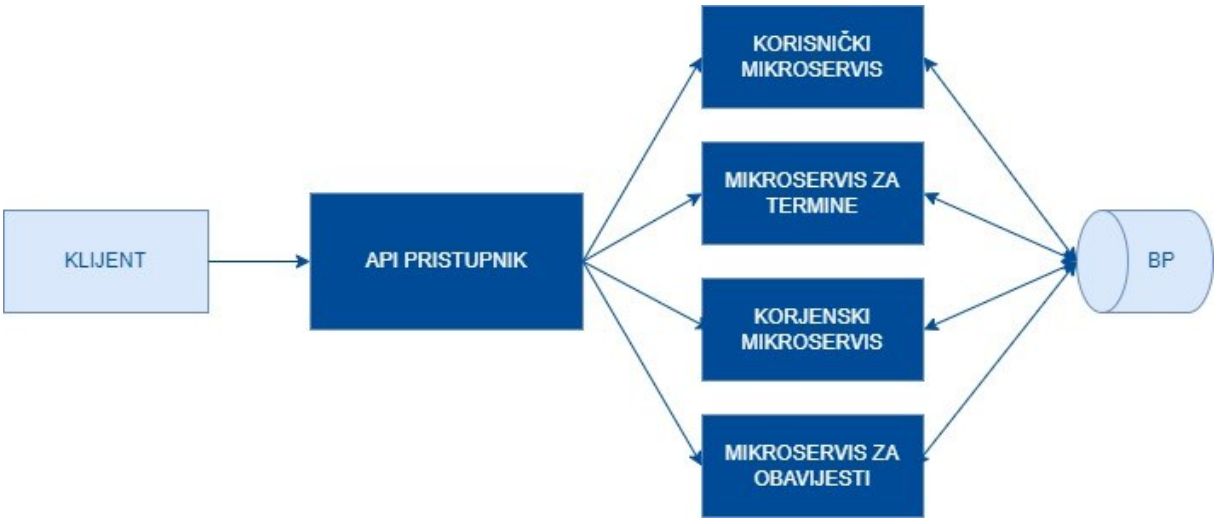
kraju, veze se modeliraju pomoću rombova te se na njih entiteti povezuju ravnom linijom. Funkcionalnost veze svakog entiteta ispisan je na odgovarajućoj liniji. [2]

# 6. Praktični dio

U ovom poglavlju opisuje se implementacija i realizacija aplikacije HealtHub, kojoj je svrha rezervacija termina kod specijalista različitih grana medicine. Tehnologije korištene pri izradi aplikacije su okvir Blazor WebAssembly, ASP.NET 6.0 za backend, DockerCompose za build, Apache Kafka za upravljanje porukama, ApiGateway za jednostavniju komunikaciju mikroservisa i SignalR koji omogućuje obavijesti u stvarnom vremenu.

## 6.1. Arhitektura aplikacije

HealtHub aplikacija je razvijena prema mikroservisnoj arhitekturi. U ovom slučaju to znači da se aplikacija sastoji od četiri neovisna mikroservisa pri čemu svaki služi određenoj svrsi. Svaki mikroservis ima zaseban API projekt specifičan potrebama krajnih točaka mikroservisa. Tako je omogućeno postavljanje mikrousluga bilo gdje na webu bez da moraju ići uz glavnu aplikaciju. Na taj način je postignuto logičko i fizičko razdvajanje aplikacije na različite dijelove. Kao što je vidljivo na slici 4. ova aplikacija je razdvojena na korisnički mikroservis, mikroservis za termine, korjenski mikroservis i mikroservis za obavijesti. Klijent u ovom dijagramu označava glavnu aplikaciju HealthHub, a između klijenta i mikroservisa se nalazi API pristupnik (engl. *ApiGateway*). API pristupnik preusmjerava sve krajnje točke API-ja i objedinjuje ih u jednu domenu kako klijent ne bi morao znati lokacije svih mikroservisa. Klijent predstavlja Blazor WebAssembly aplikaciju koja se izvršava u internet pregledniku. Svaki poslužiteljski dio mikroservisa se pokreće zasebno, dok se klijentski dio mikroservisa pokreće kao dio Blazor WebAssembly aplikacije.




*Slika 4. HealthHub arhitektura*

## 6.2. Mikroservisi

Svaki mikroservis unutar ovog projekta je iste strukture. Oni sadrže klijentski projekt s pripadajućim stranicama za prikaz, poslužiteljski projekt sa kontrolerima, koji su potrebni klijentu kako bi dohvatili ili kako bi manipulirali podacima u bazi i zajedničku klasnu biblioteku koja sadži DTO-ove (engl. *Data Transfer Object*) potrebne klijentskoj i poslužiteljskoj strani mikroservisa.

### 6.2.1. Korisnički mikroservis

Korisnički mikroservis upravlja korisnicima. Poslužiteljski projekt mikroservisa sadrži jedan kontroler naziva *UserController* koji sadrži metode za registraciju, prijavu, dodavanja rola i tokena korisnicima. Registracijom korisnika i odabirom željene role u aplikaciji korisniku se kreira te mu se dodaje odabrana rola. Uspješna prijava postojećeg korisnika podrazumijeva dodjeljivanje JWT (engl. *JSON Web Token*) tokena korisniku koji omogućuje sigurni prijenos informacija na webu. Osim toga, poslužitelj sadrži konfiguraciju za JWT, kontekst ApplicationDbContext, migracije i servise. Unutar klijentskog dijela mikroservisa nalaze se stranice za prijavu i registraciju s pripadajućim razor.cs i razor.css datotekama. Unutar projekta Shared smješteni su DTO-ovi *Login*, *Registration* i *Response* koji su korisničkom mikroservisu potrebni.

### 6.2.2. Mikroservis za termine

Posužiteljska strana mikroservisa za termine sadrži samo jedan kontroler naziva *AppointmentController* koji sadrži metode za dohvaćanje, dodavanje i ažuriranje termina za korisnike s ulogom pacijent i liječnik, ažuriranje i brisanje termina kod korisnika s ulogom liječnik te statistiku termina za liječnike. Zatim, poslužitelj sadrži i kontekst *AppointmentServiceDbContext*, migracije, servise i profile za mapiranje podataka. Klijentska strana sadrži stranice i komponente potrebne za upravljanje tereminima kod pacijenta i liječnika, a projekt Shared sadrži DTO-ove *AppointmentDTO* i *StatisticsDTO* te enum *AppointmentStatus*.

### 6.2.3. Korjenski mikroservis

Korjenski mikroservis je najveći mikroservis u aplikaciji HealtHub. Poslužitelj sadrži šest kontrolera *BugReportController*, *DoctorController*, *ReviewController*, *ServiceController*, *SpecializationController i WorkHoursController*, pripadajuće servise, migracije, profile za mapiranje i kontekst *MainServiceDbContext*. Osim toga, sadrži i klasu za pretvorbu podataka

tipa TimeOnly u TimeSpan i obrnuto te usporebu vremena, entitete potrebne poslužitelju i mapu za pohranu statičkih datoteke. Klijentska strana sadrži komponente i stranice potrebe za korisnike uloga admin, pacijent i liječnik. U Shared projektu se nalaze DTO-ovi podjeljeni u mape *BugReport*, *Doctor*, *Review*, *Service*, *Specialization*, *WorkHourse* te enum *Raiting.*

### 6.2.4. Mikroservis za obavijesti

Kako bi bila moguće obavještavanje u stvarnom vremenu SignalR je korišten u mikroservisu za obavijesti. Poslužitelj sadrži kontroler *NotificationController* s metodama za dohvačanje i upravljanje obavijestima, migracije, servise i entitet *Notification.* Klijentska aplikacija sadrži stranicu za pregled svih obavijesti, a projekt Shared *NotificationDTO*.

### 6.2.5. Upravljanje tokovima porukama između mikroservisa

Upravljanje tokovima poruka je praksa snimanja podataka u stvarnom vremenu iz izvora događaja, spremanja tokova, obrada tokova poruka u stvarnom vremenu ili retrospektivno i preusmjerivanje poruka na druge lokacije po potrebi. Kafka je distribuirani sustav koji se sastoji od proizvođača i pretplatnika. Proizvođači su klijentske aplikacije koje objavljuju poruke u određenu temu u Kafki, a s druge strane pretplatnici su oni koji su pretplaćeni na određene teme, čitaju te poruke i po potrebi ih dalje obrađuju. Poruke su organizirane i pohranjene u teme te uvijek imaju više proizvođača i više pretplatnika. One se ne brišu nakon uporabe pa ih je moguće čitati i više puta ovisno o potrebi. Moguće je definirati unutar postavki Kafke koliko je potrebno dugo zadržavanje poruka u određenoj temi. Teme su podijeljene u particije, što znači da je tema raspodijeljena na više Kafka brokera. Mogućnost klijentske aplikacije istovremenog čitanja i pisanja podataka sa više brokera pridonosi skalabilnosti. Zadaća brokera je primanje poruka, pohranjivanje na disk i markiranje jedinstvenim ofsetom. Zatim, na osnovu teme, patricije i ofseta omogućuje potrošačima pristup porukama. [7]

Primjer korištenja Kafke u HealthHubu je registracija korisnika. U korisničkom servisu stvaramo korisnika u tablici *Users*, ali da bi korjenski mikroservis imao pristup podacima o korisniku koji je npr. liječnik i kako bi korjenski mikroservis mogao spremiti više podataka o tom korisniku bitne samo njemu potrebno je stvoriti korisnika i u tablici *Doctors* s istim ID-om korisnika u tablici *Users.* Rješenje ovoga problema je kreiranje teme *Users*, implementacija proizvođača *UserEventProducerService* u korisničkom mikroservisu i implementaciju pretplatnika *UserEventConsumerHostedService.* Kafka se još koristi i u slučaju prikazivanja notifikacija pri dodavanju termina između pacijenta i liječnika. U tom

slučaju mikroservis za obavijesti predstavlja pretplatnika, a mikroservis za termine proizvođača.

## 6.3. API pristupnik

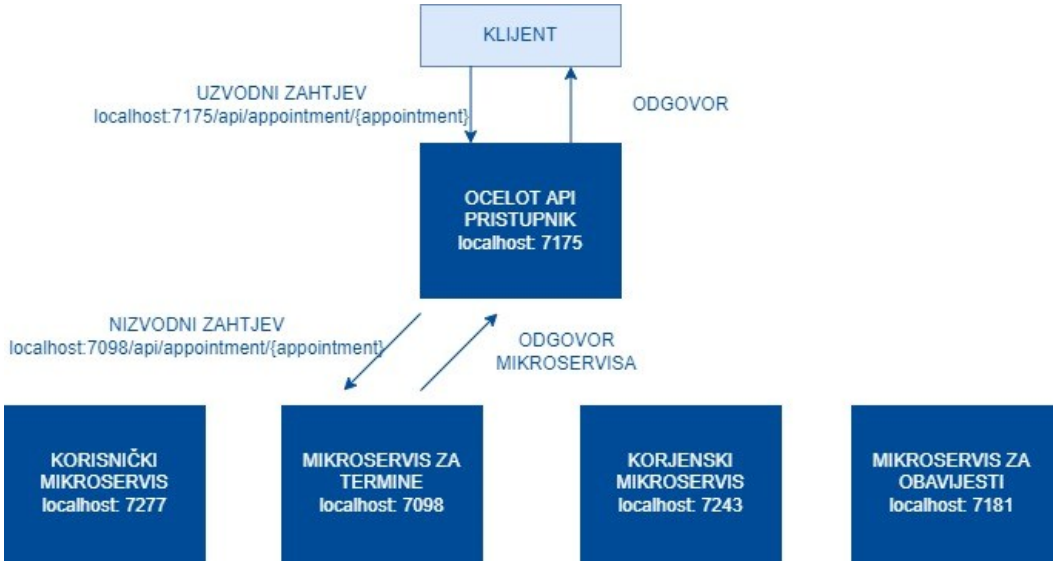
Ocelot je API pristupnik otvorenog koda za .NET platformu te je korišten u aplikaciji HealthHub. Njegova zadaća je objedinjenje mikroservisa kako klijent ne bi morao brinuti o lokacijama svakog pojedinog mikroservisa. Kod API pristupnika postoje uzvodni i nizvodni zahtjevi. Ocelot API pristupnik pretvara dolazni HTTP zahtjev klijenta ili uzvodni zahtjev i prosljeđuje ga odgovarajućem mikroservisu odnosno Ocelot šalje nizvodni zahtjev mikroservisu. Kako bi Ocelot znao lokaciju mikroservisa potrebno je konfigurirati sve rute unutar JSON konfiguracijske datoteke *ocelot.json*. Primjer konfiguracije za korisnički mikroservis prikazan isječkom programskog koda 5. Ocelot prima polje objekta rute. *DownstreamPathTemplate* označava rutu krajnje točke korisničkog mikroservisa *api/user/{anything}*. *DownstreamScheme* je shema mikroservisa i u ovom slučaju je to https. *DownstreamHostAndPorts* definira lokaciju mikroservisa (*host* i broj porta), što znači da se korisnički mikroservis nalazi na localhostu i broju porta 7277. *UpstreamPathTemplate* je putanja na kojoj će klijent zatražiti Ocelot API pristupnik. Na kraju imamo *UpstreamHTTPMethod* gdje su definirane podržane HTTP metode za API pristupnik. Na temelju dolazne metode, Ocelot šalje zahtjev HTTP metode mikroservisu.

*Isječak programskog koda 5: konfiguracija rute za korisnički mikroservis*

```
{
      "DownstreamPathTemplate": "/api/user/{anything}",
      "DownstreamScheme": "https",
      "DownstreamHostAndPorts": [
        {
          "Host": "localhost",
          "Port": 7277
        }
      ],
      "UpstreamPathTemplate": "/api/user/{anything}",
      "UpstreamHttpMethod": [ "Post", "Get", "Put" ]
}
```

Na slici 5 možemo vidjeti primjer jednog toka zahtjeva od klijenta do mikroservisa za termine. U aplikaciji HealthHub Ocelot API pristupnik nalazi se na portu 7175, a mikroservis za termine na portu 7098. Klijent nema direktni pristup portu 7098 nego samo portu 7175 odnosno Ocelotu te šalje uzvodni zahtjev *localhost:7175/api/appointment/{appointment}* kako bi spremio novi termin. Zatim, Ocelot šalje novi HTTP zahtjev odnosno nizvodni

zahtjev mikroservisu *localhost:7098/api/appointment/{appointment}* i dobiva određeni odgovor. Nakon izvršene akcije Ocelot šalje odgovor klijentu. Ostali portovi mikroservisa su također prikazani na slici 5.




*Slika 5. Primjer toka zahtjeva preko Ocelot API pristupnika*

## 6.4. Baza podataka

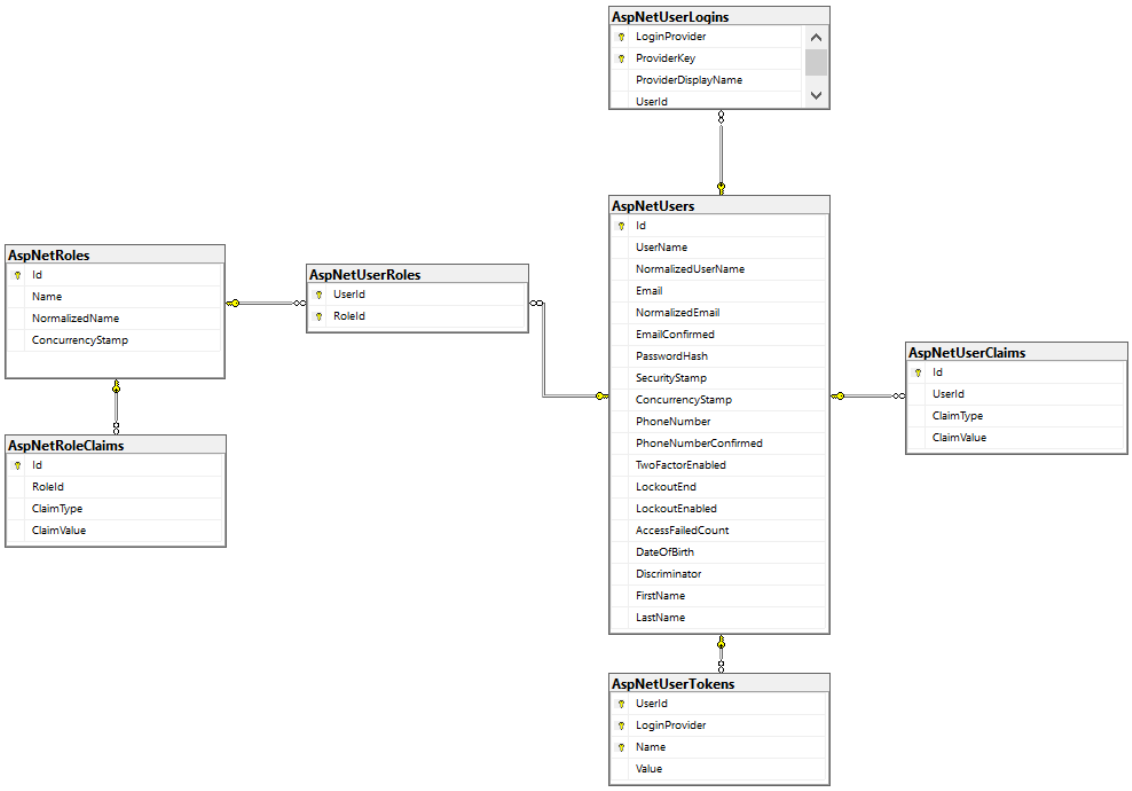
Mikroservisi su dovoljno slobodni da bi mogli koristiti zajedničku bazu podataka ili da imaju više baza podataka namijenjenih svojim preferencijama. U HealthHub aplikaciji koristi se baza podataka SQL Server te svaki mikroservis posjeduje svoje tablice u bazi koje samo taj mikroservis koristi. Tablice u bazi podataka su stvorene principom *code first* pomoću EF Corea (engl. *Entity Framework Core*). Pristup *code first* omogućuje definiranje modela entiteta u kodu koristeći C# klase, stvaranje baze podataka iz modela, a zatim dodavanje podataka u bazu podataka. Dodatna konfiguracije se može vršiti korištenjem atributa u klasama ili korištenjem API-ja.

Kako bi mogli slati upite i spremati podatke potrebno je definirati kontekst koji predstavlja sesiju s bazom podataka i izlaže definirane DbSet<TEntity> za svaku klasu u modeli. Za to je potreban *EntityFramework Nuget* paket. Svaki mikroservis u ovoj aplikaciji raspolaže vlastitim kontekstom definiranom unutar API-ja pojedinog mikroservisa. Na primjer za korjenski mikroservis kontekst možemo vidjeti na slici 7. iz koje je vidljivo da sadrži šest različitih DbSetova od kojih svaki predstavlja jednu tablicu u bazi. Svaki taj kontekst razrađuje klase koje je potrebno uključiti u model prema svojstvima DbSetova koja su definirana. Koristeći pristup *code first* određuje nazive tablica, stupaca, određuje vrste

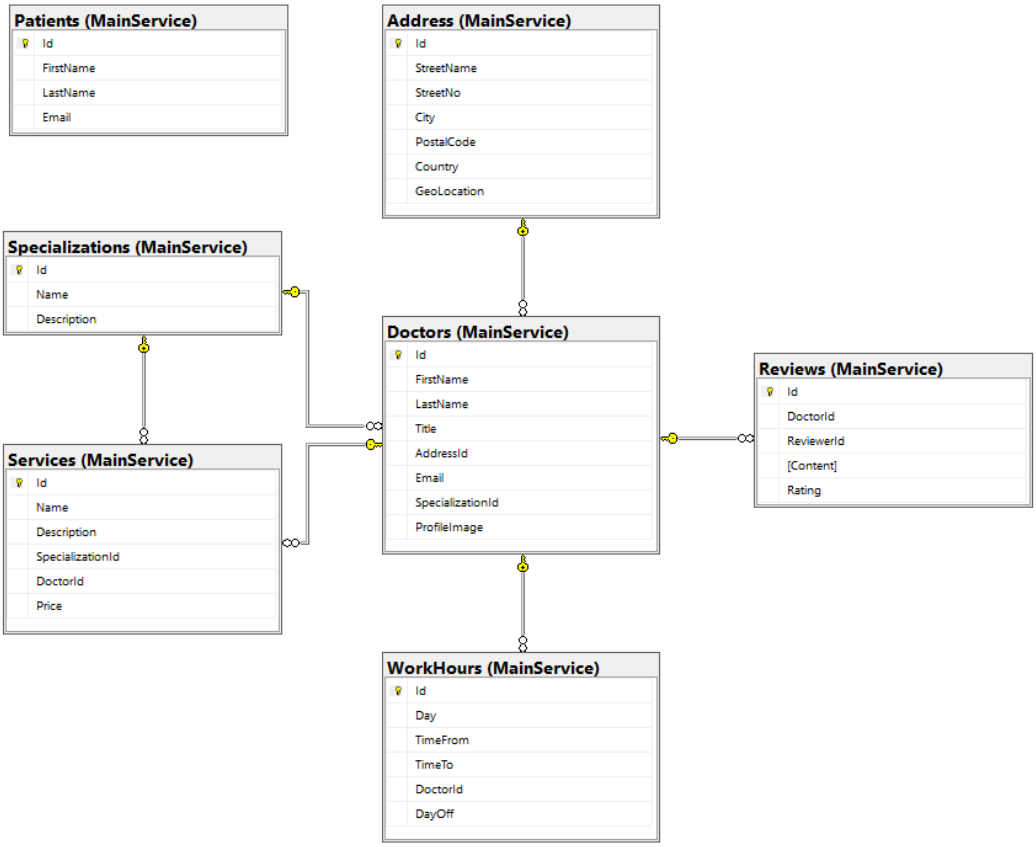
podataka, primarne ključeve i tako dalje. Dodatne konfiguracije poput primarnog ključa možemo definirati pomoću *DataAnnotations*.

Promjene u modelima zahtijevaju i ažuriranje sheme baze podataka. One se primjenjuju principom migracija (engl. *Code First Migrations*). Migracije omogućuju opisivanje ažuriranja sheme baze podataka kroz uređeni skup podataka. Svaki korak sadrži kod koji opisuje promjene koje je potrebno primijeniti na bazu podataka. Ažuriranje se vrši u dva koraka. Prvi korak je upisivanjem linije *Add-Migration Naziv* u Package Manager konzolu, a zatim ako su ispravne promjene upisivanjem linije *Update-Database*.

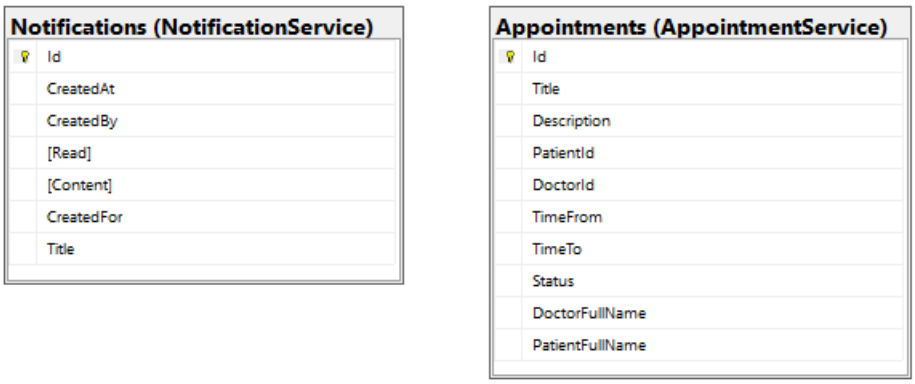
Koristeći četiri konteksta unutar aplikacije HealthHub MainServiceDbContext, AppointmentServiceDbContext, NotificationServiceDbContext i ApplicationDbContext stvorena je shema baze podataka prikazana na slikama 5., 6. i 7.




*Slika 6. Dio sheme baze za korisnički mikroservis*




*Slika 7. Dio sheme baze za korjenski mikroservis*




*Slika 8. Dio sheme baze za  mikroservis za termine i za obavijesti*

## 6.5. Kontejnerizacija aplikacije

Kako bi aplikacija *HealtHub* mogla biti pokrenuta koristeći *Docker Compose* bilo je potrebno napraviti Dockerfilove za svaki mikroservis, glavni server i API pristupnik. Primjer Dockerfilea za korjenski mikroservis prikazan je programskim isječkom koda  6.

*Isječak programskog koda 6: Dockerfile korjenskog mikroservisa*

```
1.  FROM mcr.microsoft.com/dotnet/aspnet:6.0 AS base
2.  WORKDIR /app
3.  EXPOSE 80
4.  EXPOSE 443
5.
6.  FROM mcr.microsoft.com/dotnet/sdk:6.0 AS build
7.  WORKDIR /src
8.  COPY
    ["HealthHub/MainService/HealthHub.MainService.Server/HealthHub.MainService.Server.csproj"
    , "HealthHub/MainService/HealthHub.MainService.Server/"]
9.  COPY
    ["HealthHub/MainService/HealthHub.MainService.Shared/HealthHub.MainService.Shared.csproj"
    , "HealthHub/MainService/HealthHub.MainService.Shared/"]
10. COPY ["HealthHub/Shared/HealthHub.Shared.csproj", "HealthHub/Shared/"]
11. RUN dotnet restore
    "HealthHub/MainService/HealthHub.MainService.Server/HealthHub.MainService.Server.csproj"
12. COPY . .
13. WORKDIR "/src/HealthHub/MainService/HealthHub.MainService.Server"
14. RUN dotnet build "HealthHub.MainService.Server.csproj" -c Release -o /app/build
15.
16. FROM build AS publish
17. RUN dotnet publish "HealthHub.MainService.Server.csproj" -c Release -o /app/publish
18.
19. FROM base AS final
20. WORKDIR /app
21. COPY --from=publish /app/publish .
22. ENTRYPOINT ["dotnet", "HealthHub.MainService.Server.dll"]
```

Zatim, potrebno je bilo kreirati i docker-compose.yaml izvršnu datoteke. U ovoj datoteci su definirani svi dijelovi aplikacije koji se moraju pokrenuti s definiranim portovima svakog servisa. Primjer definiranja korjenskog mikroservisa unutar docker-compose.yaml prikazan je isječkom programskog koda 7.

*Isječak programskog koda 7: docker-compose.yaml*

```
1.    healthhub.mainservice.server:
2.      container_name: mainservice
3.      environment:
4.        - ASPNETCORE_ENVIRONMENT=Development
5.      ports:
6.        - "8001:80"
7.      volumes:
8.        - ${APPDATA}/Microsoft/UserSecrets:/root/.microsoft/usersecrets:ro
9.        - ${APPDATA}/ASP.NET/Https:/root/.aspnet/https:ro
10.       - ${APPDATA}/ASP.NET/Https:/root/.aspnet/https:ro
```

Osim toga, potrebne je bila prilagodba Ocelot API pristupnika kako bi znao lokacije mikroservisa. Dio nove konfiguracijske datoteke ocelot.json prikazana je programskim isječkom koda 8 gdje je vidljiva promjena *hosta* koji je definiran u *docker-compose.yamlu* i *porta* na kojemu je pokrenut mikroservis.

*Isječak programskog koda 8: dio ocelot.json datoteke*

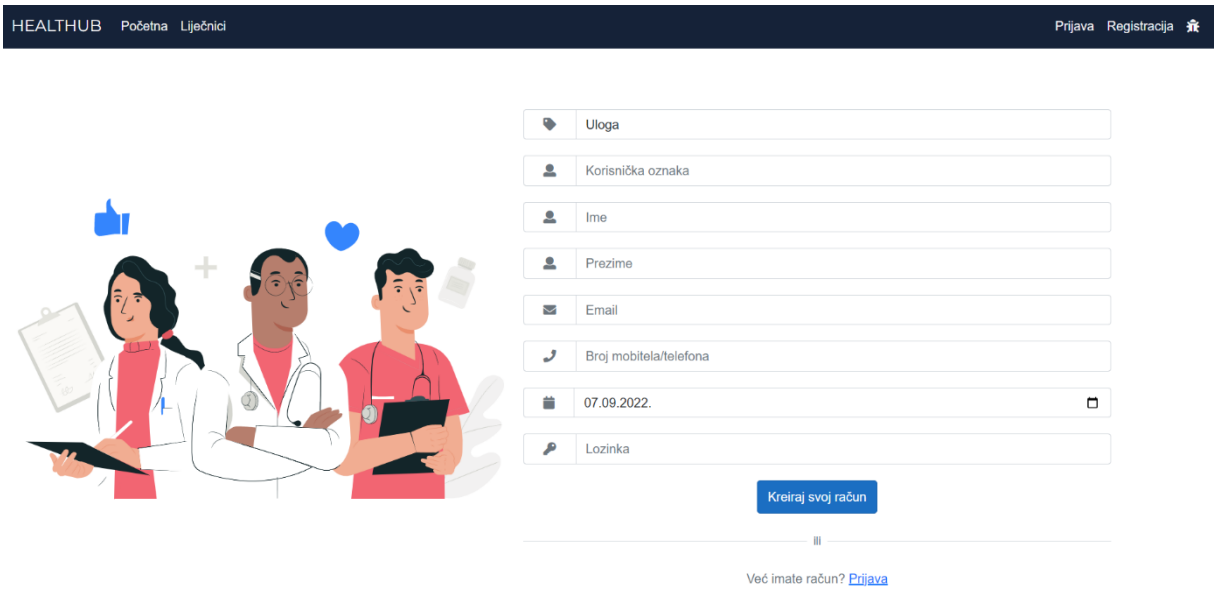
```
{
      "DownstreamPathTemplate": "/api/appointment/{anything}",
      "DownstreamScheme": "http",
      "DownstreamHostAndPorts": [
        {
          "Host": "appointmentservice",
          "Port": 80
        }
      ],
      "UpstreamPathTemplate": "/api/appointment/{anything}",
      "UpstreamHttpMethod": [ "Post", "Get", "Delete" ]
}
```

## 6.6. Funkcionalnosti aplikacije

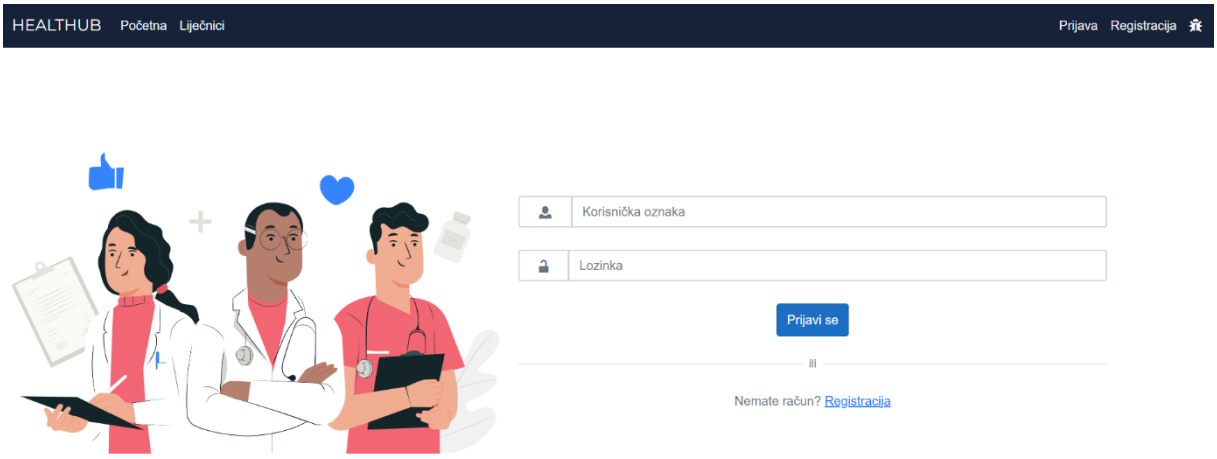
Primarna funkcionalnost aplikacije HealthHub je rezervacija termina kod liječnika. Funkcionalnosti omogućene za korisnike uloge pacijenta su pretraga liječnika, pregled detalja liječnika od općih podatka do pregleda zauzetosti kalendara, slanje zahtjeva za određeni termin liječniku i pregled vlastitih zakazanih termina. Kod korisnika uloge liječnika funkcionalnosti su pregled statistike termina, pregled zahtjeva za termine u listi čekanja i kalendara s trenutno potvrđenim terminima, dodavanje termina i premještanje termina iz liste čekanja u kalendar, potvrđivanje i odbacivanje termina, uređivanje vlastitog profila. Nakon općih korisnika aplikacije, postoji još uloga administratora čija je svrha održavanje aplikacije. Administrator može dodavati nove specijalizacije i uređivati stare specijalizacije te prima obavijesti od korisnika koji su prijavili poteškoće pri radu aplikacije ili prijedlog za promijene i dodavanje novih funkcionalnosti.

### 6.6.1. Registracija i prijava korisnika

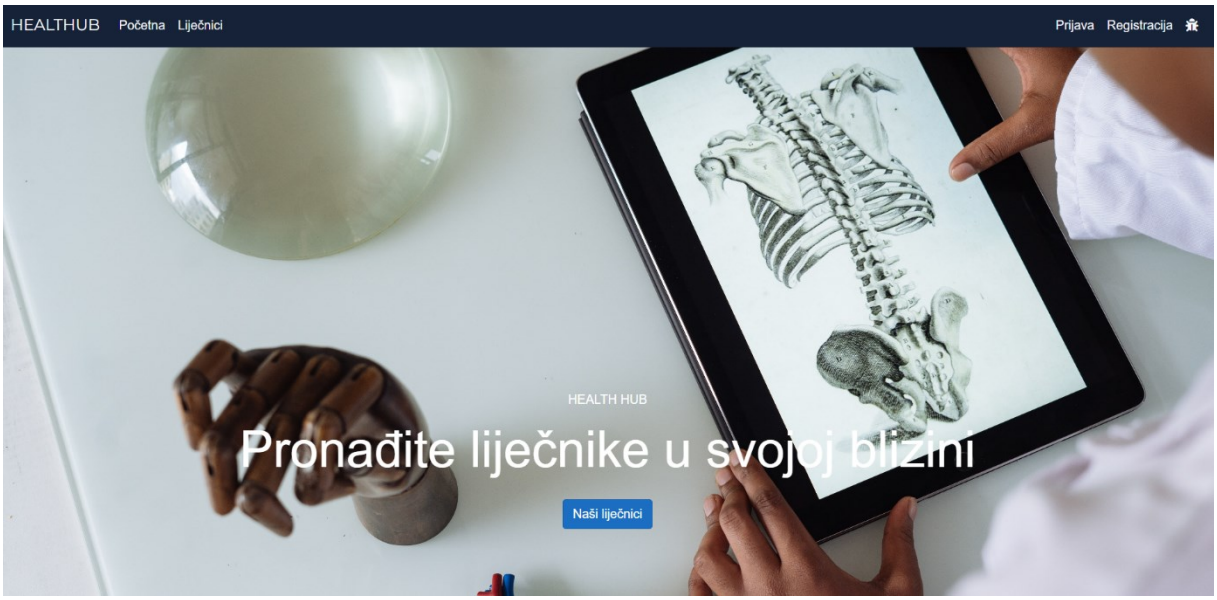
Stranica za registraciju korisnika prikazana je na slici 9. Prilikom registracije korisnik ima mogućnost biranja između uloge pacijenta i liječnika. Unosom ispravnih podataka unutar traženih polja korisnik se prebacuje na stranicu za prijavu koja je prikazana na slici 10. Unosom ispravne korisničke oznake i lozinke korisnika se prebacuje na početnu stranicu ovisno o njegovoj ulozi u aplikaciji. Prijaviti se mogu postojeći korisnici uloge pacijent, liječnik ili administrator. Početna stranica pacijenta ujedno je i početna stranica koja je dostupna javno prikazana na slici 11., dok liječnika uspješna prijava preusmjerava na stranicu sa vlastitom statistikom termina.




*Slika 9. Registracija korisnika*



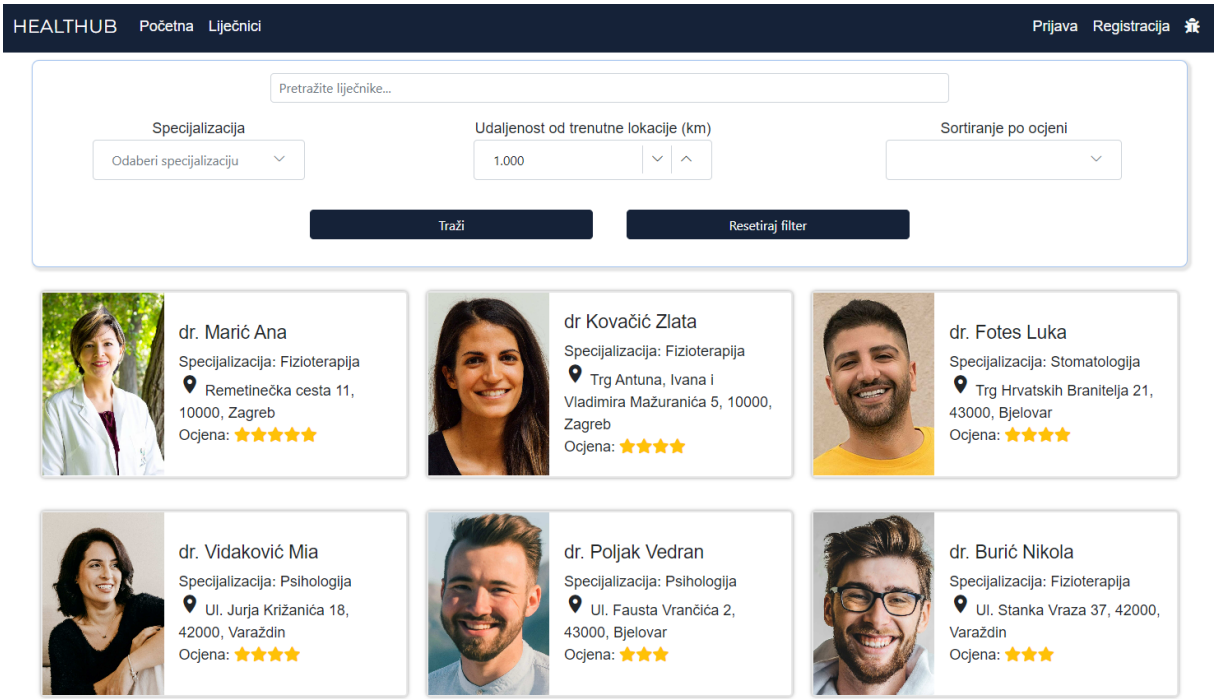

*Slika 10. Prijava korisnika*




*Slika 11. Javna početna stranica*

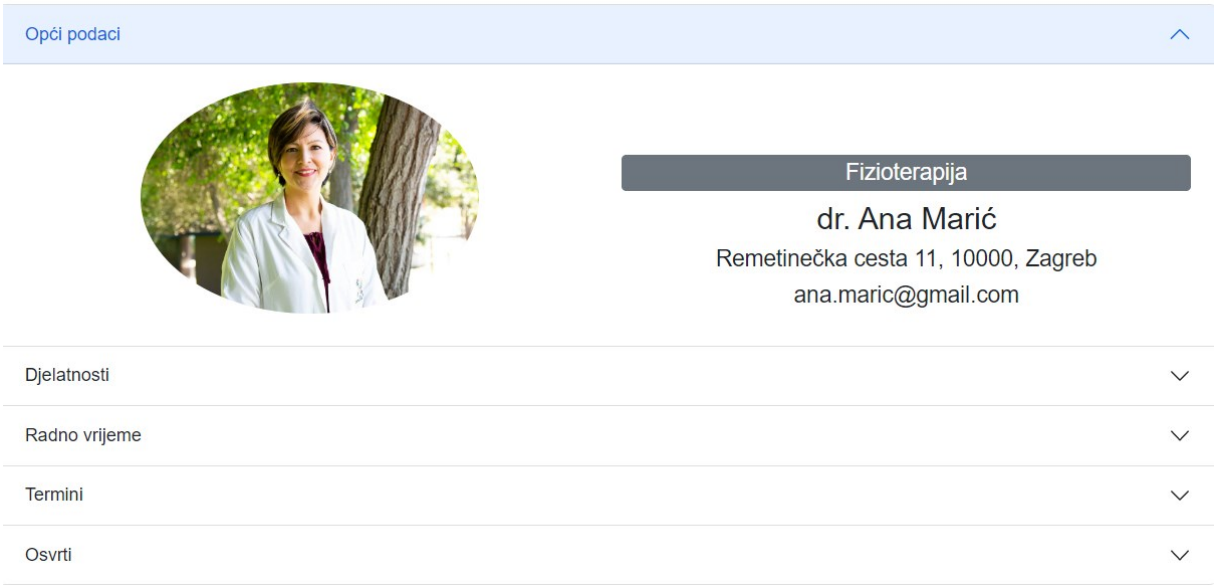
### 6.6.2. Funkcionalnosti korisnika uloge pacijenta

Funkcionalnosti korisnika u ulozi pacijenta su pregled svih raspoloživih liječnika prikazanih na stranici *Liječnici* - slika 12. Osim toga, na toj stranici pacijent ima mogućnost pretražiti liječnike i filtrirati prema željenoj udaljenosti od vlastite lokacije, prema odabranoj specijalizaciji liječnika i filtrirati prema ocjenama.



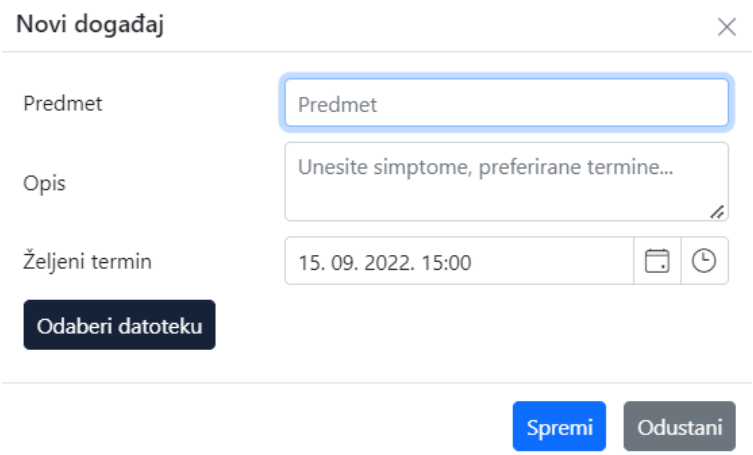

*Slika 12. Pregled liječnika*

Također, klikom na željenog liječnika može pregledati detalje o liječniku na slici 13 koji uključuju opće podatke, djelatnosti, radno vrijeme, zauzetost termina u obliku kalendara i pregledati osvrte koje su ostavili ostali pacijenti.




*Slika 13. Detalji liječnika*

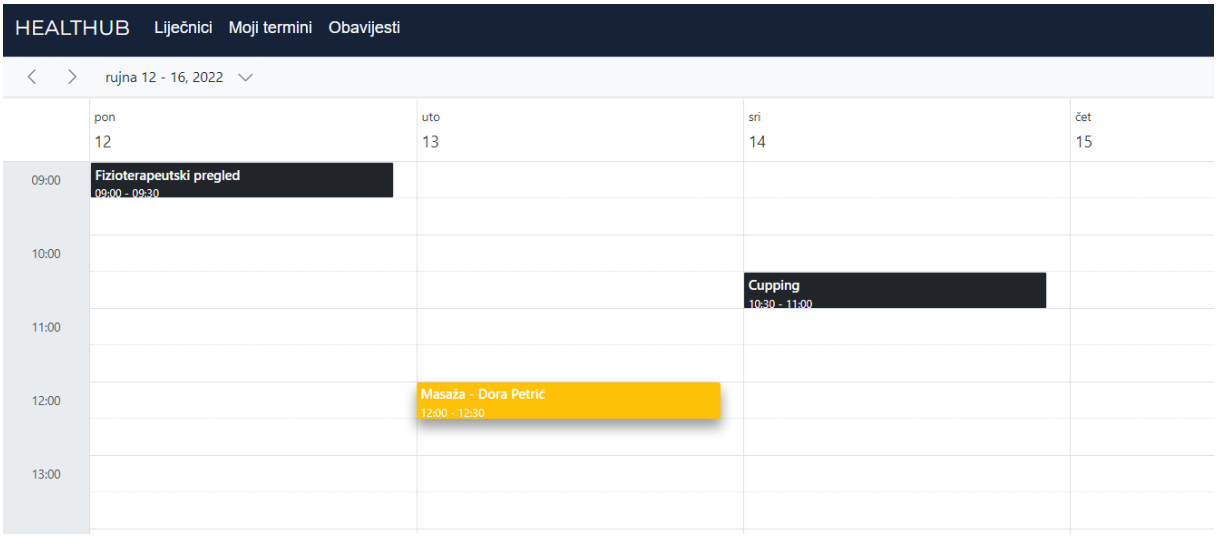
Akcije koje može izvršiti na detaljima o liječniku su rezervacija termina i ostaviti vlastiti osvrt. Klikom na gumb *Rezerviraj termin* prebacuje se na stranicu s popunjenim kalendarom, a sama rezervacija termina se odvija popunjavanjem forme prikazane na slici 14. Unosom predmeta, željenog termina, opisa problema i opcionalno prilaganjem dokumenta pacijent šalje liječniku zahtjev za termin. Važno je napomenuti da termine nije moguće dodati izvan radnog vremena liječnika.




*Slika 14: Forma za popunjavanje zahtjeva termina*

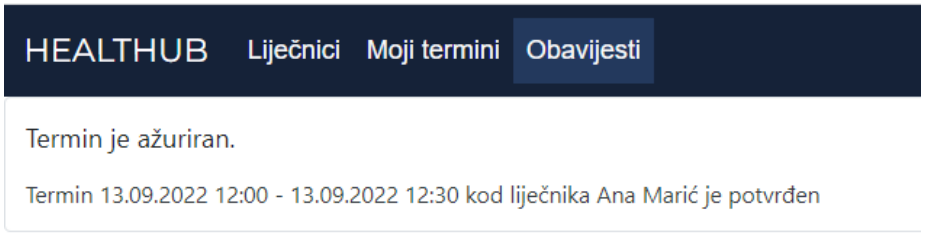
Pacijentu je omogućeno i pisanje vlastitog osvrta i ocjenjivanje liječnika nakon što termin završi.

Stranica *Moji termini* (slika 15.) prikazuje sve termine za koje je pacijent poslao zahtjev i vidjeti status u kojem se nalaze. Mogući statusi zahtjeva su u obradi, potvrđen ili otkazan.




*Slika 15. Stranica Moji termini pacijenta*

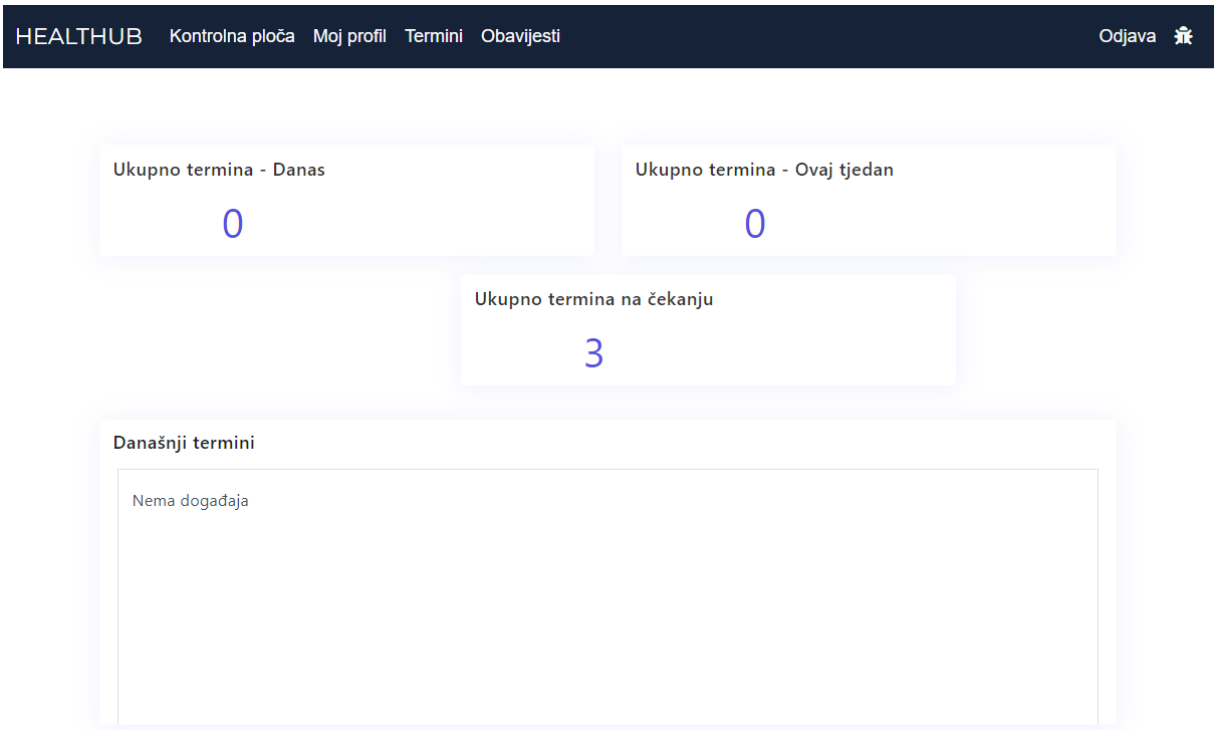
Na slici 16. prikazana je stranica *Obavijesti*. Ovdje pacijent može vidjeti sve obavijesti vezanih za termine kao što su potvrde termina ili otkazivanja.




*Slika 16. Stranica Obavijesti pacijenta*

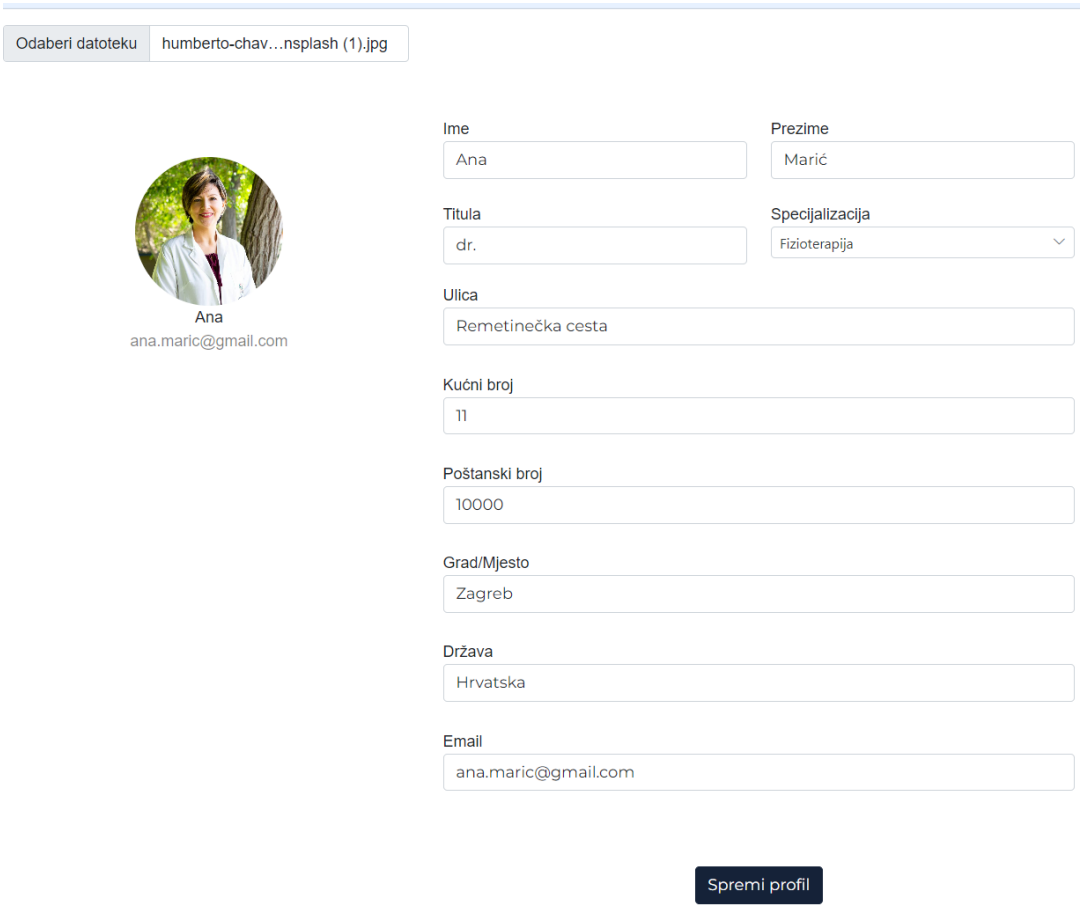
### 6.6.3. Funkcionalnosti korisnika uloge liječnika

Početna stranica liječnika prikazana na slici 17. sadrži statistiku termina liječnika koja uključuje broj potvrđenih termina u trenutnom danu i tjednu, broj termina u listi čekanja te tablicu današnjih termina.

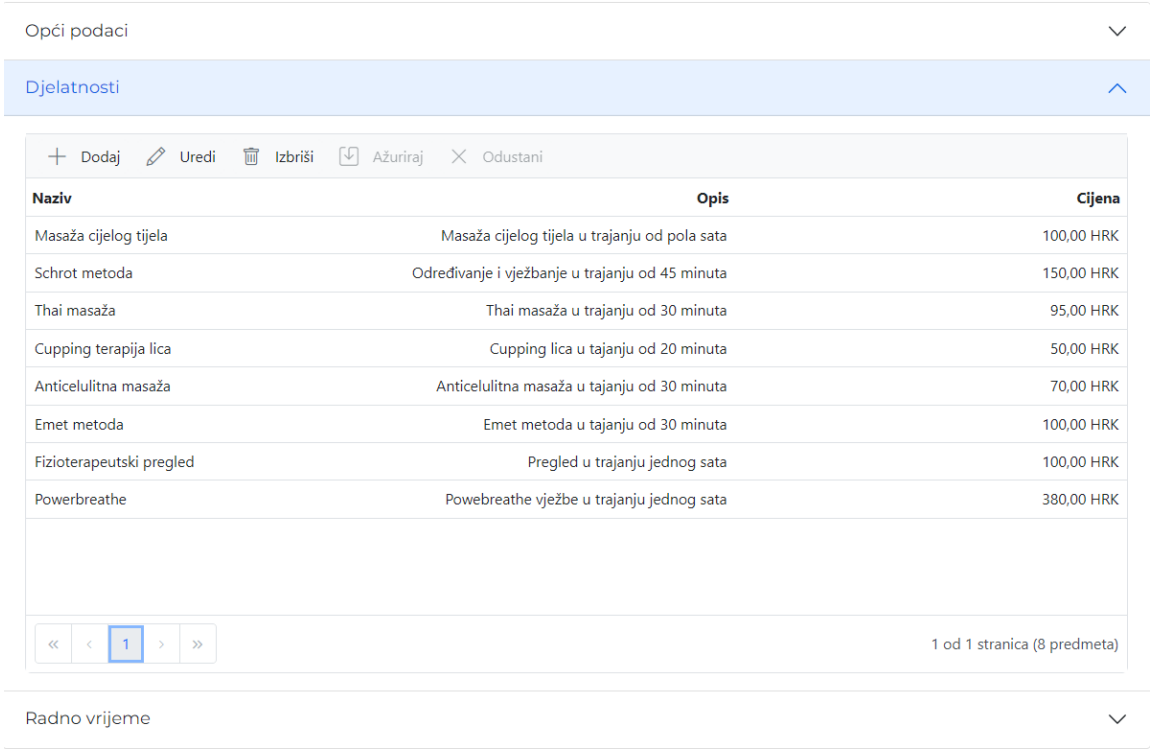



*Slika 17. Stranica Kontrolna ploča*

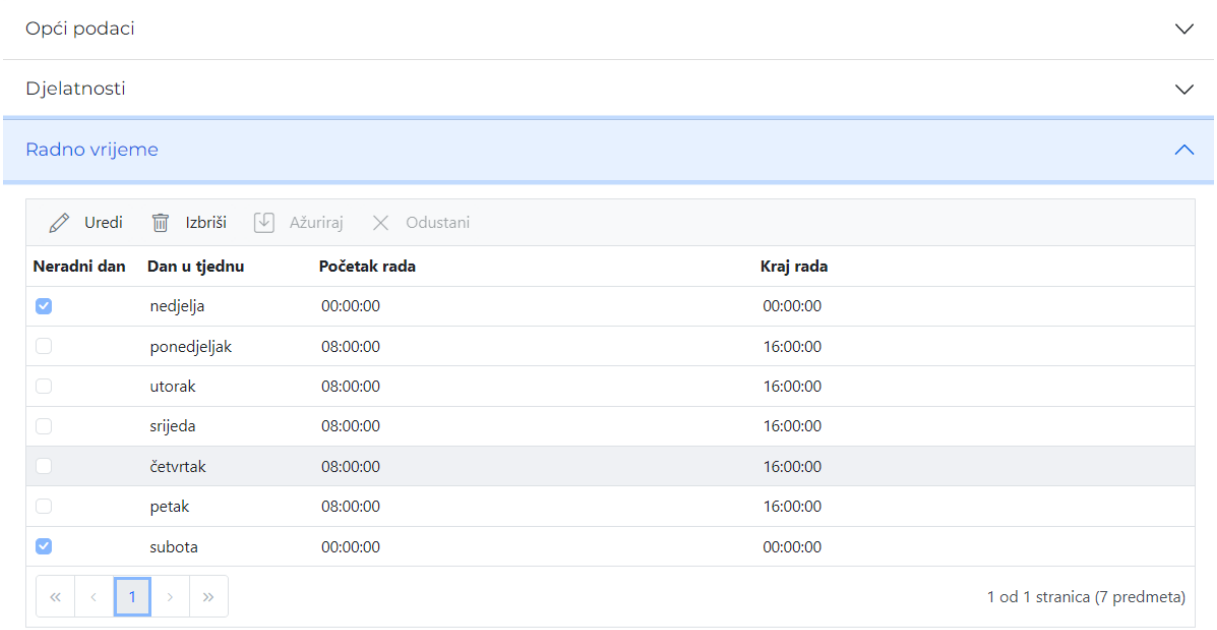
Na slikama 18., 19. i 20. prikazana je stranica *Moj profil* koja služi liječnicima da za uređivanje osobnih informacije. Opći podaci uključuju ime, prezime, titulu, specijalizaciju, adresu i sliku profila. Djelatnosti su popis svih djelatnosti koje liječnik nudi uključujući opis i cijenu, a u dijelu radno vrijeme liječnik dodaje svoje raspoložive termine u obliku dana i termina unutar tog dana.




*Slika 18. Moj profil liječnika - opći podaci*

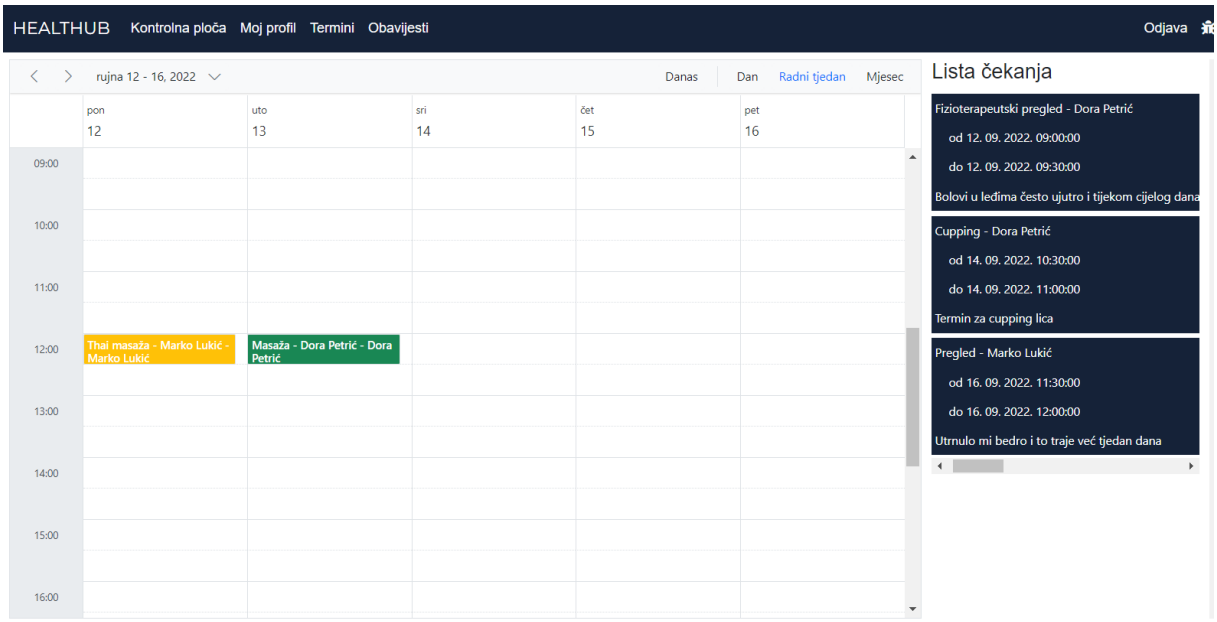



*Slika 19. Moj profil liječnika - djelatnosti*



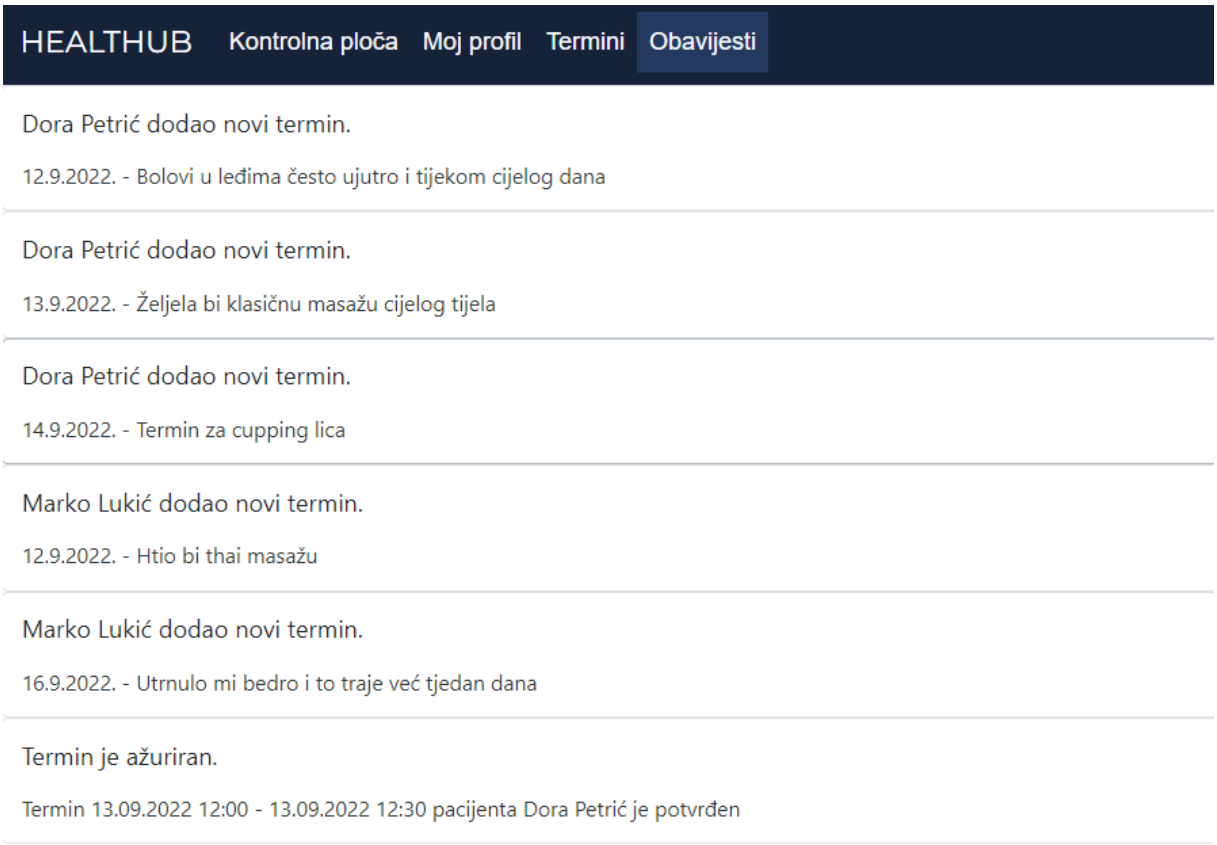

*Slika 20. Moj profil liječnika - radno vrijeme*

Stranica *Termini* na slici 21 sastoji se od dva dijela. Prvi dio je kalendar s terminima gdje liječnik može vidjeti sve zakazane termine. Dvostrukim klikom na željeni termin otvara se pregled detalja o tom terminu gdje ujedno ima mogućnost promjene statusa termina ili brisanja termina. Osim toga, liječnik ima mogućnost dodati termin u kalendar koji poprima status ostalo što mu omogućava rezervaciju termina za druge obaveze, koje možda nisu usko vezane uz poslove koje nudi na aplikaciji. Drugi dio stranice je lista čekanja u kojemu se nalaze svi zahtjevi za termine u obliku kartica s predmetom, imenom i prezimenom pacijenta, željenim terminom i opisom problema. Prevlačenjem kartice na kalendar u određeno vrijeme otvara se prozor gdje liječnik ima opciju postavljanja statusa u potvrđeno ili otkazano i prilagođavanja vremena termina.




*Slika 21. Stranica Termini*

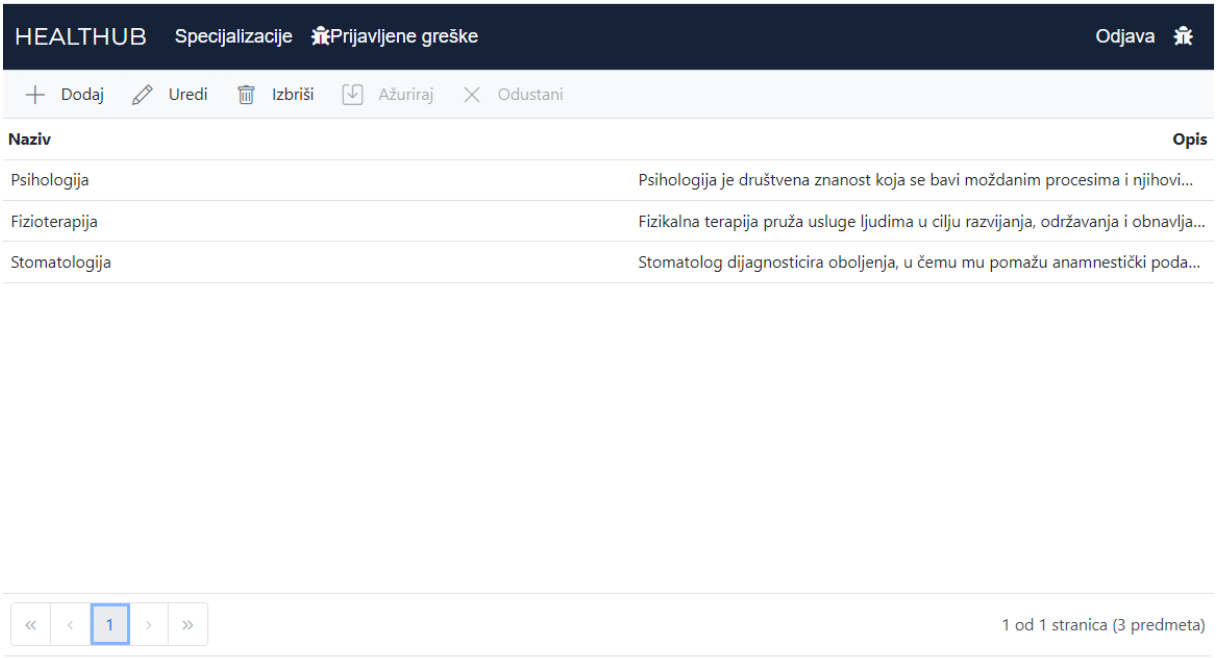
Također, liječnik ima stranicu s obavijestima o zaprimljenim zahtjevima za termine te potvrđenim i otkazanim terminima - slika 22.



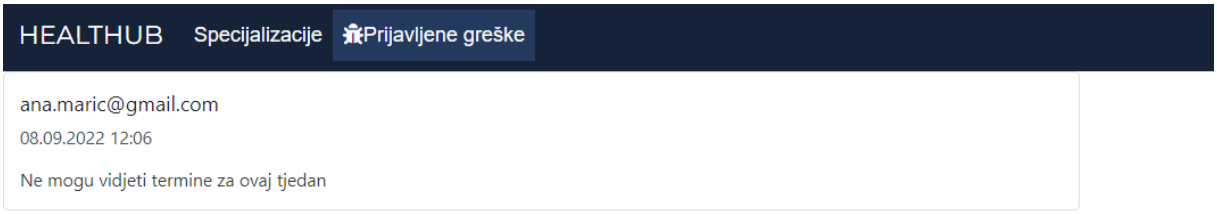

*Slika 22. Obavijesti liječnika*

### 6.6.4. Održavanje aplikacije

Administrator je zaslužan za održavanje aplikacije. *Specijalizacije i Prijavljene greške* su stranice na koje samo administrator ima pristup. Slika 23. prikazuje stranicu *Specijalizacije* na kojoj administrator može dodavati nove i ažurirati već postojeće specijalizacije. Na stranici *Prijavljene greške* (slika 24.) administrator zaprima prijavljene poteškoće s kojima su se korisnici susreli pri radu aplikacije ili prijedloge za nove funkcionalnosti aplikacije.




*Slika 23. Administracija specijalizacija*




*Slika 24. Stranica Prijavljene greške*

# 7. Zaključak

Dugi niz godina JavaScript je prevladavao kao glavni jezik za implementaciju klijentske strane web aplikacija u kombinaciji s nekim od jezika za implementaciju poslužitelja. Veliku promjenu u razvoju web aplikacija uveo je Microsoft razvojem okvira Blazor koji pruža mogućnost implementacije klijentske i poslužiteljske strane aplikacije unutar samo jednog programskog jezika C#. Mogućnost izvršavanja web aplikacije na klijenskoj strani, jednostavnost Blazor komponenti, velika zajednica i sigurna podrška .NET-a pridonose sve većoj popularnosti ovog okvira. Uz Blazor, Docker također uvodi velike promjene pri razvoju web aplikacija. Mogućnost kontejnerizacije web aplikacija pridonijelo je olakšanom razvoju aplikacija, boljim performansama i lakšem skaliranju aplikacija.

U ovom radu objašnjeni su najvažniji teorijski koncepti korišteni pri razvoju aplikacije, a zatim je opisana aplikacija HealthHub mikroservisne arhitekture, kojoj je svrha omogućiti rezervaciju termina kod specijalista različitih grana medicine. Aplikacija je napravljena za korisnike koji mogu biti u ulozi pacijent, liječnik ili administrator. Ovisno o toj ulozi korisnik ima omogućene specifične funkcionalnosti. Pacijent može pretražiti liječnike, pregledati detalje lječnika, slati zahtjeve za određeni termin liječniku i pregledati vlastite zakazane termine. Liječnikove funkcionalnosti su pregled statistike termina, pregled zahtjeva za termine u listi čekanja i kalendara s trenutno potvrđenim terminima, dodavanje termina i premještanje termina iz liste čekanja u kalendar, potvrđivanje i odbacivanje termina, uređivanje vlastitog profila. Uloga administratora je održavanje aplikacije te on može dodavati nove specijalizacije i uređivati stare specijalizacije. Osim toga, prima obavijesti od korisnika koji su prijavili poteškoće pri radu aplikacije. Tehnologije korištene za izradu ove aplikacije su Blazor WebAssembly, ASP.NET 6.0, Docker, Apache Kafka i ApiGateway.

Mogu reći da sam kroz zadatak završnog rada i predmet Projektiranje informacijskih sustava na koji se veže zadatak završnog rada unaprijedila svoje znanje pri razvoju web aplikacija. Korištenjem okvira Blazora, *runtime* okruženja Dockera, alata Kafke, Ocelot API pristupnika i .NET 6.0 stekla sam nova znanja i iskustva koja će mi pomoći u daljnjem razvoju kao programskog inženjera.

# 8. Popis literature


[1] Alok Ranjan, Abhilasha Sinha, Ranjit Battewad, *JavaScript for Modern Web Development: Building a Web Application Using HTML, CSS, and JavaScript*: BPB Publications © 2020

[2] Dušan Petković, *Microsoft SQL Server 2019: Beginner's Guide, Seventh Edition*: Oracle Press © 2020

[3] Google LLC . (2020). G Suite. Dostupno na: https://gsuite.google.com

[4] Merkel, D. (2014). Docker: lightweight linux containers for consistent development and deployment: Linux Journal, 2014(239), 2.

[5] Michele Aponte, *Building Single Page Applications in .NET Core 3: Jumpstart Coding Using Blazor and C#*: Apress © 2020

[6] Microsoft Corporation, Microsoft tehnička dokumentacija, SAD, Pristupljeno: 10.srpnja 2022 [Online]. Dostupno na: https://docs.microsoft.com/en-us/docs/

[7] Neha Narkhede, Gwen Shapira, Todd Palino, *Kafka: The Definitive Guide*: O'Reilly Media, Inc. 2017

[8] Nigel Poulton, *Docker Deep Dive: Zero to Docker in a Single Book!*: JJNP Consulting Limited © 2020

[9] Peter Himschoot, *Microsoft Blazor: Building Web Applications in .NET 6 and Beyond, 3rd Edition*: Apress © 2021

[10] Rami Vemula, *Real-Time Web Application Development: With ASP.NET Core, SignalR, Docker, and Azure*: Apress © 2017

# 9. Popis slika

# Obrazac 5: Izjava o autorstvu

Veleučilište u Virovitici

**OBRAZAC 5**

**IZJAVA O AUTORSTVU**

Ja, FABIJANIĆ DORA

izjavljujem da sam autor/ica završnog/diplomskog rada pod nazivom

Web aplikacija za rezervaciju termina kod privatnih specijalista različitih grana medicine

Svojim vlastoručnim potpisom jamčim sljedeće:

- da je predani završni/diplomski rad isključivo rezultat mog vlastitog rada koji se temelji na mojim istraživanjima i oslanja se na objavljenu literaturu, a što pokazuju korištene bilješke i bibliografija,
- da su radovi i mišljenja drugih autora/ica, koje sam u svom radu koristio/la, jasno navedeni i označeni u tekstu te u popisu literature,
- da sam u radu poštivao/la pravila znanstvenog i akademskog rada.

**Potpis studenta/ice**

Fabijanić Dora

# Obrazac 6: Odobrenje za pohranu i objavu završnog/diplomskog rada

Veleučilište u Virovitici

**OBRAZAC 6**

**ODOBRENJE ZA POHRANU I OBJAVU ZAVRŠNOG/DIPLOMSKOG RADA**

Ja FABIJANIĆ DORA

dajem odobrenje za objavljivanje mog autorskog završnog/diplomskog rada u javno dostupnom digitalnom repozitoriju Veleučilišta u Virovitici te u javnoj internetskoj bazi završnih radova Nacionalne i sveučilišne knjižnice bez vremenskog ograničenja i novčane nadoknade, a u skladu s odredbama članka 83. stavka 11. Zakona o znanstvenoj djelatnosti i visokom obrazovanju (NN 123/03, 198/03, 105/04, 174/04, 02/07, 46/07, 45/09, 63/11, 94/13, 139/13, 101/14, 60/15, 131/17).

Potvrđujem da je za pohranu dostavljena završna verzija obranjenog i dovršenog završnog/diplomskog rada. Ovom izjavom, kao autor navedenog rada dajem odobrenje i da se moj rad, bez naknade, trajno javno objavi i besplatno učini dostupnim:

a) široj javnosti
b) studentima i djelatnicima ustanove
c) široj javnosti, ali nakon proteka 6 / 12 / 24 mjeseci (zaokružite odgovarajući broj mjeseci).

Potpis studenta/ice

Fabijanić Dora

U Virovitici, 3.9.2022

**U slučaju potrebe dodatnog ograničavanja pristupa Vašem završnom/diplomskom radu, podnosi se pisani obrazloženi zahtjev.*