Izrada web aplikacije u .NET tehnologiji

Ogrinec, Tomislav

SVEUČILIŠTE U ZAGREBU

FAKULTET ORGANIZACIJE I INFORMATIKE

V A R A Ž D I N

Tomislav Ogrinec

IZRADA WEB APLIKACIJE U .NET TEHNOLOGIJI

DIPLOMSKI RAD

Varaždin, 2023.

SVEUČILIŠTE U ZAGREBU

FAKULTET ORGANIZACIJE I INFORMATIKE

V A R A Ž D I N

Tomislav Ogrinec

JMBAG: 0016135746

Studij: Informacijsko i programsko inženjerstvo

IZRADA WEB APLIKACIJE U .NET TEHNOLOGIJI

DIPLOMSKI RAD

Mentor:

Prof. dr. sc. Dragutin Kermek

Varaždin, kolovoz 2023.

Tomislav Ogrinec

Izjava o izvornosti

Izjavljujem da je moj diplomski rad izvorni rezultat mojeg rada te da se u izradi istoga nisam koristio drugim izvorima osim onima koji su u njemu navedeni. Za izradu rada su korištene etički prikladne i prihvatljive metode i tehnike rada.

Autor potvrdio prihvaćanjem odredbi u sustavu FOI-radovi

Sažetak

Diplomski rad pokriva teorijsko obrazloženje pojmova jednostraničnih aplikacija (*eng. Single Page Application*), čiste arhitekture (*eng. Clean Architecture*), domene vođene dizajnom (*eng. Doman Driven Design*) i druge pojmove vezane uz razvoj modernih web aplikacija. Prvo je definirano što aplikaciju čini web aplikacijom te koja je razlika između web stranice i web aplikacije. Nakon toga su objašnjeni pojmovi vezani uz programiranje i koncipiranje rada aplikacije putem web preglednika te sama arhitektura sustava koja definira aplikaciju. Na kraju teorijskog dijela rada, ukratko je uspoređena odabrana tehnologija sa sličnim tehnologijama.

Aplikacija koja je napravljena kao praktičan dio rada ima primarnu svrhu demonstracije koncepata i principa obrađenih u teorijskom dijelu rada. Ona ima ukomponirane razne poglede krajnjih korisnika na ideji simulacije vođenja željezničkog kolodvora te same kupovine prijevoznih karata.

Cilj rada je razjasniti arhitekturne koncepte, kao i one manje važne poput upravljanja komponentizacijom dijelova programskog koda i međusobnom interakcijom. Kao što je opisano u odlomku iznad, web aplikacija služi kao primjer primjene teorije na stvarnom slučaju razvoja web aplikacije za vođenje željezničkog kolodvora i prodaje karata.

Ključne riječi: ASP.NET, Blazor, jednostranična aplikacija (*eng. Single Page Application*), Web Assembly, čista arhitektura (*eng. Clean Architecture*), domenom vođen dizajn (*eng. Doman Driven Design*)

Sadržaj

1. Uvod

Web aplikacija danas je jedna od najčešćih oblika aplikacije nakon računalnih videoigara, te je to također primjer najčešće razvijanih aplikacija. Ta učestalost, kao i sam razvoj tehnologije, prouzročila je prijelaz stolnih aplikacija u oblik klasičnih web aplikacija ili web aplikacija u računalnom oblaku. Iz tog razloga danas imamo tehnologije poput .NET i Java pomoću kojih možemo vrlo brzo razviti web stranicu bez prevelikog financijskog ili vremenskog troška.

Sigurnosni model web preglednika određuje da na njemu nema memorijskog prostora za pohranjivanje velikog sadržaja aplikacije, pa je to prva dimenzija na koju se mora obratiti pažnju prilikom osmišljavanja arhitekture. Također, ne želi se da se svaka stranica učitava po nekoliko sekundi što znači da aplikacija mora biti brza i procesno lagana kako bi se izbjegla velika čekanja. Druga dimenzija na koju se nailazi jest sigurnost podataka. Internet nije osmišljen kao sigurno mjesto pa iz tog razloga aplikacija mora biti podložna raznim sigurnosnim pravilima. Treća dimenzija koja je u ovom radu obrađena jest i samo korisničko iskustvo prilikom korištenja web aplikacije. Kod interneta normalna je pojava gubljenja signala, brzine ili pak gubljenje paketa putem mreže. Na sve te probleme se mora paziti prilikom izrade web aplikacije.

Kako bi se pripremilo programsko rješenje koje će se samo moći prilagođavati svim spomenutim zahtjevima, mora se osmisliti adekvatna arhitektura sustava. Na početku se koristila monolitna arhitektura sustava. Iz tog principa počeli su se razdvajati slojevi s dodatnim apstrakcijama kako taj programski kod ne bi bio strogo povezan, već labavo. Takav pristup omogućuje odvajanje implementacije od same apstrakcije te kasnije promjene na/u sustavu ne zahtijevaju promjene nad cijelim projektom, već samo dijelom (slojem). Taj koncept zove se slojevita arhitektura (*eng. N-Layered architecture*). Kako bi se bolje odvojila domenska logika od aplikacijske razine, te modernizirao razvoj web aplikacije, u radu se prebacuje iz arhitekture u čistu arhitekturu (*eng. Clean Architecture*). Razloge kako, što i zašto je to napravljeno objašnjeno je konceptom razvoja domene vođene dizajnom (*eng. Doman Driven Design*).

Nakon definiranih osnovnih elemenata za razvoj web aplikacije, nastavlja se razvoj koji se sastoji od puno funkcionalnosti, dizajnerskih koncepata, načina pisanja programskog koda te drugih. Neki od tih koncepata, poput pametnih/glupih komponenti (*eng. Smart/Dumb Components*) i pozadinskog generiranja korisničkog sučelja (*eng. Backend For Frontend*) objašnjeni su i primijenjeni u ovom radu.

Tehnologije koje se koriste u ovom radu su iz porodice C++ jezika, konkretno C#. Osnovni programski okvir koji je korišten za razvoj moderne web aplikacije je ASP.NET. On omogućuje brzu i jednostavnu izgradnju web aplikacija, kao i web servisa. Druga važna tehnologija je EF Core (*eng. Entity Framework Core*), koja se koristi kako bi se ubrzalo dohvaćanje podataka kao i samo pisanje upita na bazu podataka. Kako web aplikacija ne bi bila samo sirov HTML, CSS i JavaScript, uvedene su Blazor komponente koje omogućuju uređivanje HTML komponenata s C# programskim kodom koji se prilikom pokretanja prevodi u JavaScript. Također, uveden je i MudBlazor, to jest proširenje, kako bi se iskoristile već gotove komponente te tako izbjegla potrošnja vremena na razvoj stilizacije.

Web aplikacija koja je izrađena za potrebe ovog rada koncipirana je na svim spomenutim i obrađenim metodama i tehnikama rada. Glavna domena aplikacije je administracija vezana uz vođenje jednog željezničkog kolodvora te sama prodaja karata online kupovinom. Glavne funkcionalnosti aplikacije su: evidencija vlakova, vođenje ljudskih potencijala, evidencija voznih redova, zapisivanje popravaka perona/zgrade/tračnica, blagajna (online kupovina).

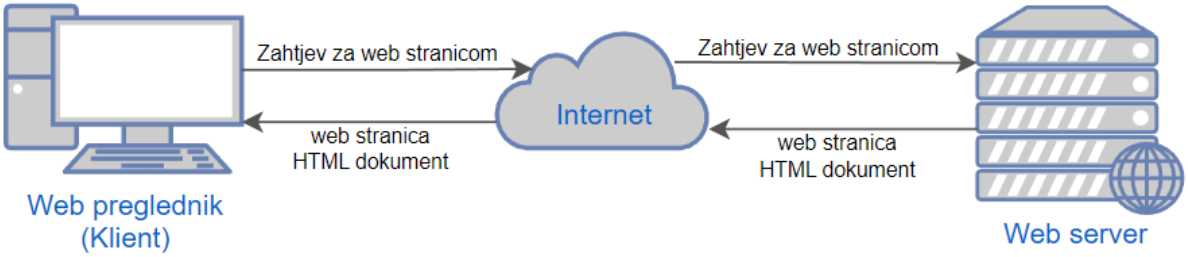
2. Web aplikacija

U početcima interneta nisu postojale web aplikacije već su to bile samo web stranice čija je osnovna svrha bila prikaz informacija. Izuzetak su bile stranice koje su sadržavale funkcionalnosti pretraživanja svoje arhive koja je bila statičan popis podataka. Prva značajna takva stranica bila je od Tim Berners-Lee-a koja je pretraživala telefonski imenik. Upravo ovakav primjer web stranice bio je okidač za razvoj dinamičkih web stranica. [1, str. 19]

Pitanja koja se vežu uz pojam web aplikacije su: „Što je zapravo web aplikacija?“, „Što je web stranica?“, te „Koja je razlika između web stranice i web aplikacije?“. U daljnjem kontekstu pokušalo se odgovoriti na postavljena pitanja te pronaći definicija koja u potpunosti definira oba pojma te razliku između njih.

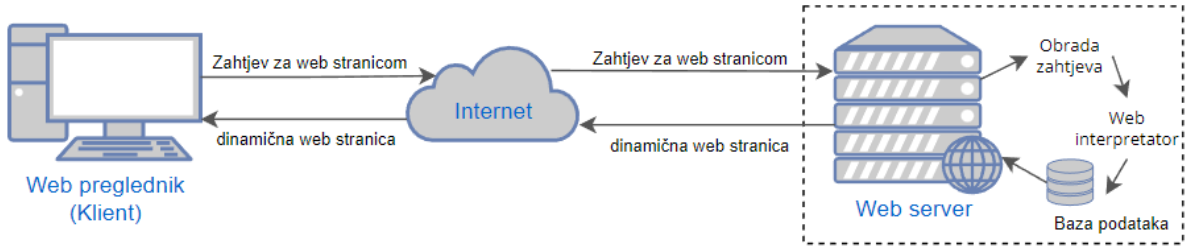
Prema WiX-u [2], web stranica je dokument sastavljen od skupa HTML (*eng. HyperText Markup Language*) elemenata tako da ga web preglednik može učitati i prikazati. Technopedia [3] tumači web stranicu kao dokument za WWW (*eng. World Wide Web*) identificiran s jedinstvenom URL (*eng. Uniform Resource Locator*) poveznicom, te govori kako se sastoji od skupine HTML elemenata koji čine sadržaj stranice. Također, navodi se svojstvo promjene web stranice putem navigacijskih poveznica poznatijih kao „hypertext links“.

Tutorialspoint [4] web stranice dijeli na statične i dinamične. Statične web stranice su stranice koje su pohranjene na serveru te se njihov sadržaj ne mijenja s obzirom na akcije korisnika. One se sastoje isključivo od HTML elemenata i CSS (*eng. Cascading Style Sheets*) stilizacija [4].



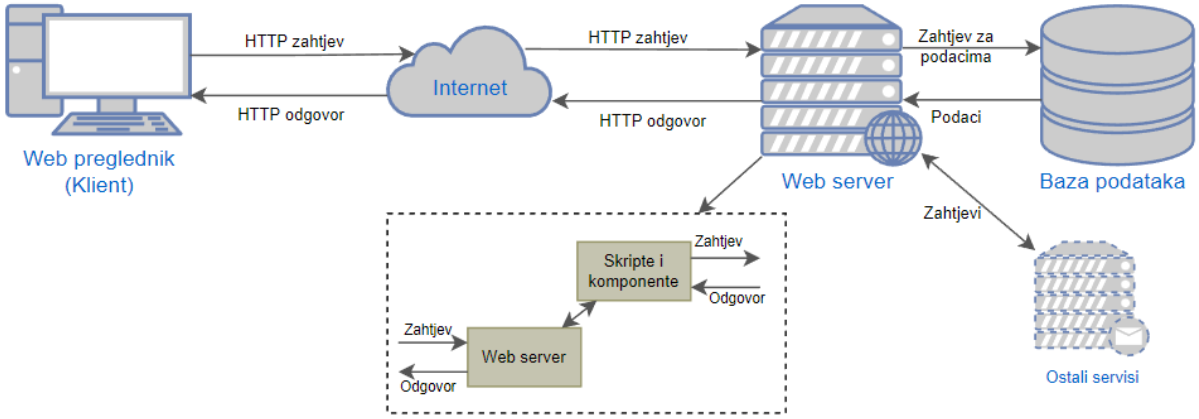
Slika 1. Komunikacija kod statičnih web stranica

Dinamične web stranice su stranice koje putem raznih web servisa dohvaćaju svoje podatke sa servera. Dinamične web stranice su svojom popularnošću omogućile razvoj objektnih modela dokumenata (*eng. DOM – Document Object Model*). [4]



Slika 2. Komunikacija kod dinamičkih web stranica

Upravo je dinamičan način rada web stranica omogućio rad sa skriptnim jezicima i okvirima te tako otvorio vrata za razvoj tehnologija poput ASP.NET i Java JSP.

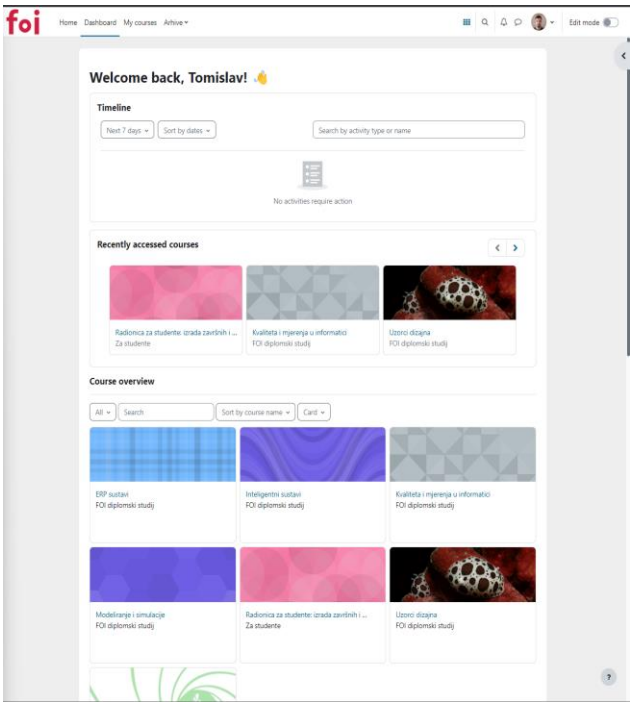


Slika 3. Detaljizirana moderna komunikacija na dinamičnim web stranicama i/ili aplikacijama

Web aplikacija je aplikacija koja je spremljena na server i pristupa joj se putem web preglednika [5]. Prema definiciji TechTargeta, ova definicija suštinski je vrlo slična definicijama web stranice pa se iz tog razloga postavlja pitanje: „Što je to web aplikacija i čemu služi?“. Web aplikaciju je najlakše opisati na temelju razlika između nje same i web stranice. Web aplikacija je vrlo slična samoj web stranici, no ona se ne mora sastojati od samo jedne stranice, već može sadržavati njih više, izuzev koncepta jednostraničnih web aplikacija (*eng. SPA - Single Page Application*). Glavna razlika između web stranice i web aplikacije jest prilagođavanje sadržaja korisniku, dodatne usluge i vanjski servisi [5].

Kako bi se razjasnila razlika između web stranice i web aplikacije uspoređena je web stranica jedne gimnazije i Moodle sustav (web aplikacija) Fakulteta organizacije i informatike.

Slika 4. Pregled web stranice gimnazije "Fran Galović" Koprivnica [6]



Slika 5. Pregled naslovne stranice Moodle sustava (Pregled vlastitog računa)

Većina razlika se može vidjeti vizualno, prvenstveno iz razloga što su web aplikacije novije te tako i modernijeg izgleda. Slijedi tablična usporedba tih dvaju pojmova.

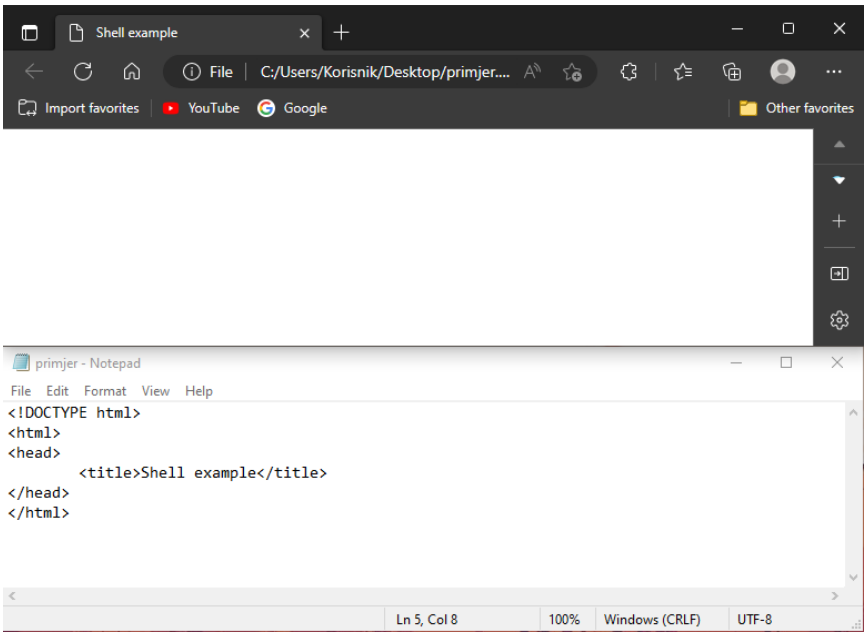
Parametar	Web aplikacija	Web stranica
Zbog čega je napravljena?	Aplikacija koja služi za interakciju sa krajnjim korisnikom	Prikaz informacija; u većini slučajeva dostupna svima
Korisničko iskustvo	Korisnik može uređivati i brisati podatke	Korisnik može samo pregledavati sadržaj
Autentifikacija	Potrebna iz razloga što se aplikacija prilagođava korisniku	Najčešće se koristi samo u svrhe skrivanja sadržaja
Kompleksnost izrade	Puno kompleksnije i tehnički teže za razvoj	Čisti prikaz podataka u određenom obliku
Tip softvera	Ponekad web aplikacija nije samostalna, već je samo dio neke web stranice	Samostalni proizvod kojeg učitava web preglednik
Kompilacija	Mora se kompilirati prije isporuke	Učitava se čisti kod
Isporuka	Cijeli projekt je kompilira i postavlja na određenu domenu	Učitava se HTML kod i ne zahtjeva održavanje

Tablica 1. Razlike između web aplikacije i web stranice [7]

2.1. Jednostranične web aplikacije

Prema I. Wigmore [8] jednostranična web aplikacija je web aplikacija koja je prezentirana kao jedna HTML stranica kako bi bila responzivnija te što bliža razvoju desktop aplikacija. U ovom obliku aplikacije sva prezentacijska logika te pogledi separirani su u zasebnu domenu te podalje od servera [9].

Jednostranična web aplikacija gradi se iz ljuske što znači da postoji samo inicijalni HTML dokument koji se popunjava komponentama [9]. To znači da aplikacija ima samo jednom postavljen predložak za HTML te se sadržaj unutar njega mijenja ovisno o navigacijskim elementima i funkcionalnostima. Na slici ispod može se vidjeti kako izgleda inicijalna web stranica.



Slika 6. Prikaz inicijalne HTML stranice

Jedna od glavnih prednosti jednostraničnih aplikacija jest brzina i jednostavnost navigacije [10]. Kada korisnik želi pristupiti nekoj funkcionalnosti, ne mora se prebacivati na drugu stranicu, već ima sve što mu je potrebno na trenutnoj stranici. Na taj se način smanjuje vrijeme koje je potrebno za navigaciju te samo korištenje funkcionalnosti.

Druga prednost jednostraničnih web aplikacija jest njihova stabilnost i sigurnost. Kada korisnik radi s višestraničnom web aplikacijom (*eng. MPA - Multi Page Application*), svaka promjena stranice zahtijeva ponovno učitavanje stranice što može dovesti do grešaka i problema sa sigurnošću [11]. Jednostranične web aplikacije, s druge strane, ne zahtijevaju ponovno učitavanje stranice pri svakoj promjeni, što smanjuje vjerojatnost grešaka i povećava sigurnost.

Međutim, iako postoje brojne prednosti jednostraničnih aplikacija, postoje i nedostaci koje je važno uzeti u obzir. Prvi nedostatak je veća zahtjevnost za računalo i potreba za brzim internetom što može ograničavati upotrebljivost za neke korisnike [12]. Drugi nedostatak je lošija optimizacija za pretraživanje što može otežati korištenje same web stranice za neke korisnike [12].

2.2. Progresivne web aplikacije

Progresivne web aplikacije su novi trend u razvoju web tehnologija [13]. To su aplikacije koje kombiniraju prednosti web i nativnih aplikacija stvarajući jedinstveno korisničko iskustvo [14]. Korištenjem tehnologija poput Service Workera i Web App Manifesta, progresivne web aplikacije omogućuju da se aplikacije preuzmu i koriste bez potrebe za instalacijom putem trgovine aplikacijama što je velika prednost za korisnike [15]. Osim toga, progresivne web aplikacije su sposobne funkcionirati čak i u lošim uvjetima mreže što omogućuje kontinuirano korištenje bez prekida [16].

Međutim, korištenje progresivnih web aplikacija također ima neke nedostatke. Na primjer, progresivne web aplikacije možda neće imati istu funkcionalnost kao nativne aplikacije, osobito ako se radi o korištenju specifičnih uređaja ili funkcija [17]. Osim toga, progresivne web aplikacije ovisne su o funkcioniranju mreže što znači da će rad aplikacije biti usporen ako internetska mreža nije dovoljno brza i/ili stabilna [18].

2.3. Višestranične web aplikacije

Višestranične web aplikacije su tradicionalni način razvoja web aplikacija gdje se svaka stranica, koja se prikazuje korisniku, učitava zasebno. Ovaj dizajn se često koristi za velike i složene web stranice gdje se želi osigurati jednostavna navigacija i organizacija sadržaja. Prema T. Brucksu [19], višestranične web aplikacije su često izbor za razvoj korporativnih web-stranica i e-trgovina jer omogućuju različite dizajne stranica i funkcionalnosti na pojedinačnoj razini.

Međutim, višestranične web aplikacije također imaju nedostatke. Na primjer, svaki put kada korisnik želi promijeniti stranicu potreban je novi zahtjev prema serveru što može rezultirati sporijim učitavanjem stranica i manjom interaktivnošću. Stoga kod velikih i kompleksnih web-stranica, višestranične web aplikacije mogu dovesti do lošijeg korisničkog iskustva. Prema R. P. Nguyenu [10], višestranične web aplikacije su u prošlosti često bile izbor za razvoj web stranica, ali s rastućom popularnošću jednostraničnih web aplikacija, sve više poduzeća prelazi na razvoj jednostraničnih web aplikacija.

2.4. Usporedba i odabir strukture web aplikacije

Skoro pa je nemoguće sa sigurnošću reći koji tip web aplikacije je bolji jer svaki od njih ima svoje prednosti i nedostatke. Ukoliko je cilj da aplikacija nudi potpunu neovisnost te da se svaka stranica uređuje na svoje načine i ima svoje pogodnosti, onda najbolje odgovara višestranična web aplikacija. Kada se razvija web aplikacija za jednostavne i male do srednje aplikacije koje nemaju potrebe za izvan mrežnim načinom rada, onda je logičan izbor jednostranična web aplikacija, ali u slučaju dodavanja vanjskih servisa, ovisnosti o dostupnosti mreže, velikog korištenja priručne memorije (*eng. Cache memory*) i slično, bolji odabir je progresivna web aplikacija.

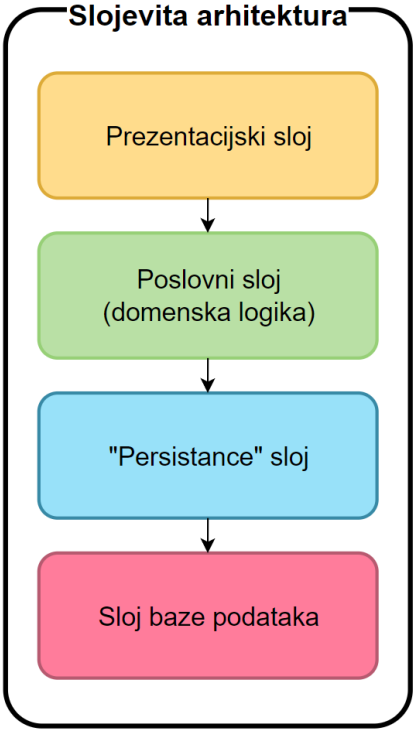
Kako je web aplikacija u radu napravljena samo u svrhe prikazivanja implementacije modernih principa izrade web aplikacije, odluka oko prezentacije i mogućnosti ove aplikacije je jednostavna. Web aplikacija razvijana je u obliku jednostranične web aplikacije jer ne zahtjeva izvan mrežni način rada, no također nema potrebe za razvojem velike aplikacije koja će ovisiti o vanjskim servisima.

3. Slojevita arhitektura

Kako aplikacije postaju sve složenije, jedan od načina da se spriječi ta kompleksnost jest podjela aplikacije prema odgovornostima ili brigama. Na taj se način slijedi princip odvajanja briga, što olakšava organiziranje koda i pronalaženje određenih funkcionalnosti koje se implementiraju. Međutim, osim organizacije koda, slojevita arhitektura ima i druge prednosti. Organizacijom koda u slojeve, zajednička funkcionalnost niske razine može se ponovno koristiti kroz aplikaciju. Ova ponovna uporaba je korisna jer znači manje programskog koda te može omogućiti standardizaciju na jedinstvenoj implementaciji slijedeći princip „ne ponavljaj se“ (*eng. DRY – Don't Repeat Yourself*). [20]

Slojevita arhitektura je struktura softverskog sustava koja se sastoji od više različitih slojeva, a svaki od njih bavi se specifičnim zadatkom. Ova arhitektura omogućuje razdvajanje funkcionalnosti sustava na jasne, međusobno odvojene slojeve što povećava modularnost, skalabilnost i učinkovitost sustava. [21]

Sastavnice u obliku slojevite arhitekture organizirane su u vodoravne slojeve, a svaki sloj obavlja specifičnu ulogu unutar aplikacije (npr. prezentacijska ili poslovna logika). Iako oblik slojevite arhitekture ne specificira broj i vrste slojeva koji moraju postojati u obrascu, većina slojevitih arhitektura sastoji se od četiri standardna sloja: prezentacije, poslovnog sloja, „persistence“ i baze podataka. [22]



Slika 7. Prikaz osnovne raspodijele slojeva

Korištenjem slojevite arhitekture, aplikacija može kontrolirati koji dijelovi sustava mogu komunicirati međusobno. To pomaže u održavanju svakog sloja odvojenim i samodostatnim što olakšava promjenu ili zamjenu jednog sloja bez utjecaja na ostale. Ograničavanjem ovisnosti između slojeva omogućuje se da utjecaj bilo kakvih promjena može biti minimiziran te kako bi se ažuriranja mogla napraviti što učinkovitije i s manjim rizikom od uzrokovanja novih problema u cijeloj aplikaciji. [20]

Prema Alexandri A. [23] slojevita arhitektura dijeli se na tri osnovna dijela:

- prezentacija,
- domenska logika,
- i baza podataka (podatkovni sloj).

Prezentacijski sloj najviši je sloj u web aplikaciji i predstavlja korisničko sučelje. Glavna zadaća mu je prevođenje naredbi i rezultata na način da ih korisnik razumije. Poslovi sloj ima zadaću da koordinira cijelu aplikaciju, procesira korisničke naredbe, radi logične odluke te provodi razne izračune. Također, on je središnji sloj koji spaja preostala dva sloja te surove podatke pretvara u korisniku razumljive informacije. Podatkovni sloj ima zadaću da sve podatke sprema u bazu podataka te kreira zaštitnički sloj koji će onesposobiti spontane upite i pokušaje upada. [23]

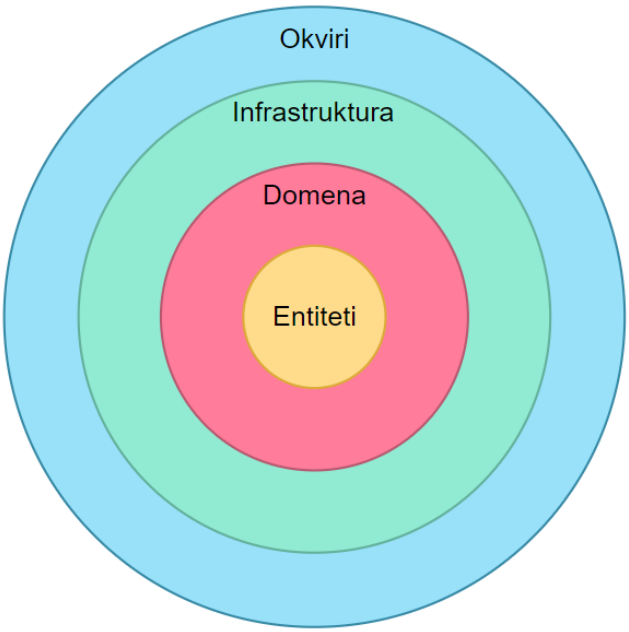
Slojevita arhitektura najčešće se implementira kao infrastruktura kao usluga (*eng. IaaS – Infrastructure as a Service*), pri čemu svaki sloj radi na zasebnom skupu virtualnih strojeva (*eng. VM – Virtual Machine*). Međutim, slojevita aplikacija ne mora biti isključivo bazirana na IaaS-u. Često je korisno koristiti upravljane usluge (*eng. Managed Services*) za neke dijelove arhitekture, naročito za memoriranje, razmjenu poruka i pohranu podataka. [24]

Slojevita arhitektura nudi nekoliko prednosti, poput prenosivosti između računalnog oblaka i lokalnih okruženja, te mogućnosti rada na heterogenim okruženjima kao što su Windows i Linux. Osim toga, slojevita arhitektura ima manji zahtjev za učenjem i predstavlja prirodni razvoj za razliku od tradicionalnog modela web aplikacija. Međutim, postoji nekoliko izazova povezanih sa slojevitom arhitekturom. Jedan potencijalni problem jest stvaranje srednjeg sloja koji samo obavlja CRUD (*eng. Create Read Update Delete*) operacije nad bazom podataka dodajući nepotrebnu latenciju bez bilo kakvog doprinosa korisnog rada. Osim toga, upravljanje IaaS-a općenito zahtijeva više posla nego upravljanje aplikacijom koja koristi samo upravljane usluge, a jasno je da može biti teško upravljati sigurnošću mreže u velikom sustavu. [24]

4. Čista arhitektura

Prema Ashish P. [25], čista arhitektura jest stil pisanja programskog koda na način da se postigne slaba povezanost implementacije upotrebe slučaja. Upravo ti slučajevi središnja su organizacijska struktura koja je odvojena od okvira i tehnologije [25]. Drugi izvor [20, str. 31], tvrdi da čista arhitektura stavlja poslovnu logiku i model aplikacije u samo središte, umjesto da poslovna logika ovisi o pristupu podacima i/ili o drugim infrastrukturnim pitanjima.

Čista arhitektura nam pomaže da držimo cijeli programski kod aplikacije pod kontrolom. Glavni cilj čiste arhitekture je održavanje i razvoj programskog koda/logike koji se vjerojatno neće mijenjati. Način na koji se postiže čista arhitektura jest pisanje programskog koda/logike tako da nema izravnih ovisnosti te da su vanjske ovisnosti u potpunosti zamjenjive. [26]



Slika 8. Prikaz koncepta čiste arhitekture

Kako bi se objasnila čista arhitektura, prvo se treba razumjeti ideja ovisnosti, a ona glasi: „Ovisnosti programskog koda smiju samo biti usmjerene prema unutrašnjem sloju, prema pravilima više razine.“ [27, str. 203].

Ništa od sloja ispod ne smije znati ni o čemu iz vanjskog sloja, a to uključuje: funkcije, klase, varijable, metode ili bilo koju drugu softversku jedinicu [27]. Na primjer, servis u domenskom sloju ne smije znati za kontrolera iz vanjskog sloja sučelja za programiranje web aplikacija (*eng. Web API – Application Programming Interface*). Također, želi se izbjeći da vanjski okvir ima ikakav utjecaj na unutarnje krugove [27].

Entiteti predstavljaju kritična poslovna pravila širom poslovne domene. Entitet može biti objekt s metodama ili skup struktura podataka i funkcija. Bitno je samo da se oni mogu koristiti u različitim aplikacijama. [27]

Domenski sloj nalazi se u samom središtu arhitekture te sadrži entitete i njihove specifikacije [26]. Prema P. Ashish [25], domena je osnovni i centralni dio projekta o kojem svi ostali slojevi ovise. Također, tvrdi kako se domenski sloj sastoji od:

- entiteta,
- agregata,
- vrijednosnih objekata,
- domenskih događaja,
- enumeracija i
- konstanta [25].

Aplikacijski sloj predstavlja implementaciju slučajeva upotrebe aplikacije na temelju domene. Slučaj upotrebe može se shvatiti kao interakcija korisnika s korisničkim sučeljem (*eng. UI – User Interface*). Ovaj sloj utjelovljuje svu aplikacijsku logiku, ali je ovisan o sloju domene te predstavlja zadnji sloj pozadinske strane razvoja aplikacije prema van [25]. U ovom sloju mogu se pronaći sljedeći elementi:

- apstrakcije (sučelja),
- aplikacijski servisi,
- komande i upiti,
- iznimke,
- modeli – objekti prijenosa podataka (*eng. DTO – Data Transfer Object*),
- validatori,
- ponašanja i
- specifikacije [25].

Infrastruktura predstavlja skup objekata modela koji će se održavati, sve migracije i sve objekte konteksta baze podataka. U slučaju korištenja uzorka dizajna o repozitorijima u ovom sloju imali bismo i definiciju metoda svih repozitorija [20], [26]. Neki od osnovnih elemenata su:

- infrastrukturni servisi,
- migracije i
- implementacije pristupa podatcima i mnoge druge [20].

Prezentacijski sloj predstavlja rezultat komunikacije između krajnjeg korisnika i web aplikacije. Ovaj sloj sastoji se od sljedećih elemenata:

- kontroleri,
- filteri,
- posredni programi,
- pogledi i
- konfiguracije [20].

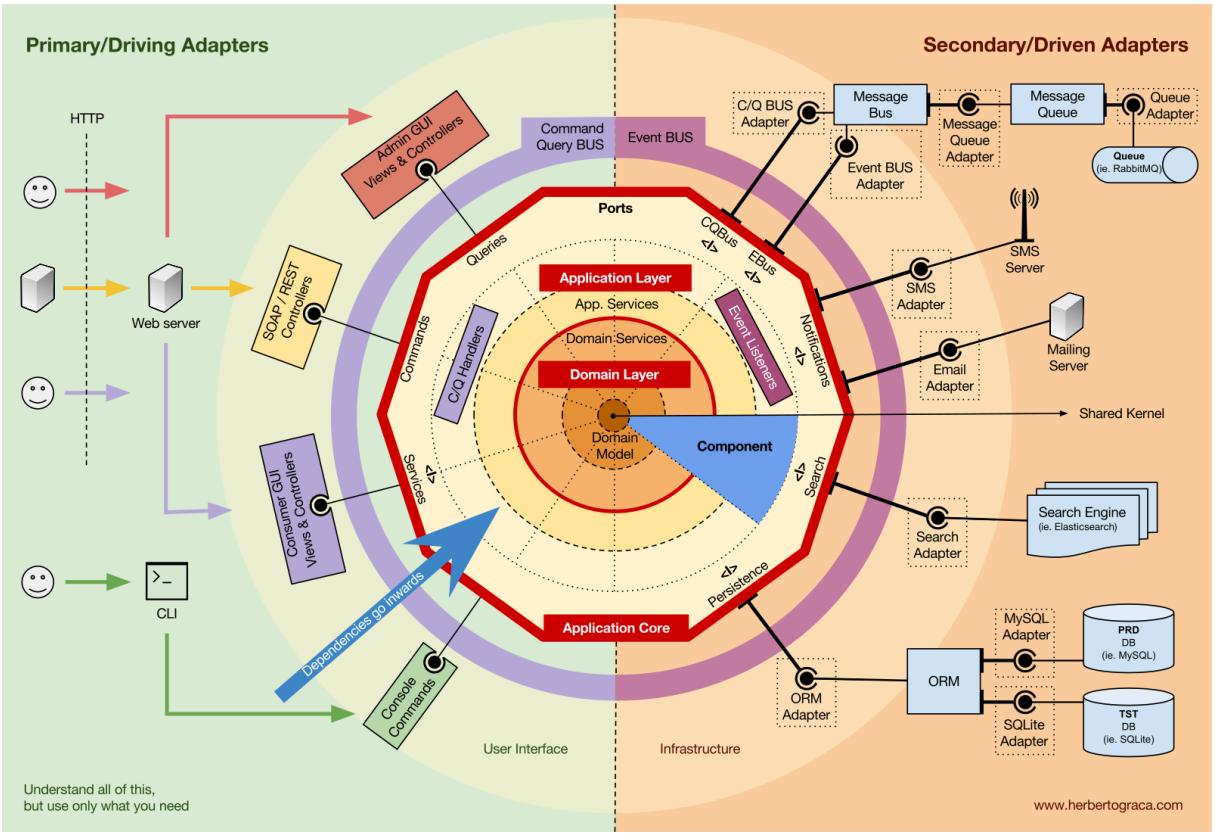
Čista arhitektura preporučuje se za projekte koji imaju kompleksna poslovna pravila ili zahtijevaju visoku fleksibilnost u pogledu promjena zahtjeva. Također, povoljna je za projekte koji imaju očekivan dugi vijek trajanja ili zahtijevaju česte nadogradnje. [27]

Jedan od glavnih razloga za upotrebu čiste arhitekture jest poboljšanje održivosti sustava te omogućavanje testiranja cijelog sustava. Razdvajanjem poslovne logike od detalja implementacije omogućava se lakše pisanje jediničnih testova te ujedno i ažuriranje dijelova sustava bez utjecaja na ostatak. Kombinacija tih koristi može rezultirati značajnijim uštedama u troškovima i prilagođavanjem sustav za kasnije promjene. [28], [29]

5. Domenom vođen dizajn

U području razvoja softvera, domenom vođen dizajn (*eng. DDD - Domain-Driven Design*) je popularni pristup koji ima za cilj uskladiti softverske sustave s poslovnim područjem. DDD je način razmišljanja koji naglašava važnost razumijevanja poslovnog područja i njegovih složenosti kako bi se stvarali učinkoviti softverski sustavi. [30]

Prema D. Laribee [31], domena vođena dizajnom je skup principa i uzoraka koji pomažu programerima da stvore elegantne objektne sustave. Ukoliko je pravilno primijenjen, može dovesti do softverskih apstrakcija nazvanih domenski modeli. Upravo ti modeli inkapsuliraju složenu poslovnu logiku smanjujući jaz između poslovne stvarnosti i programskog koda.



Slika 9. Detaljni prikaz mogućnosti domene vođene dizajnom [32]

U domeni vođenom dizajnom, ograničeni konteksti (*eng. Bounded Context*) koriste se za definiranje jasnih granica oko jezika i modela korištenih u određenim područjima sustava [30]. Upravo ovaj oblik granica pomaže u izoliranju kompleksnosti i pruža zajedničko razumijevanje sustava za sve uključene u njegov razvoj. Pokazalo se da su ograničeni konteksti posebno korisni u velikim i složenim sustavima [33].

U domeni vođenom dizajnom, sveprisutni jezik (*eng. Ubiquitous Language*) je zajednički jezik koji koriste svi članovi razvojnog tima kako bi komunicirali o domeni sustava [30]. Ovaj jezik pomaže osigurati da svi uključeni u proces razvoja imaju zajedničko razumijevanje domene i njezine terminologije. Sveprisutni jezik je ključni dio stvaranja domenskog modela koji točno odražava poslovnu domenu [33].

Agregat je skupina povezanih objekata koji se tretiraju kao jedna cjelina u svrhu promjene podataka i održavanja dosljednosti [30]. Agregati su odgovorni za provođenje varijanti i granica dosljednosti u sustavu te su ključni element u stvaranju domenskog modela koji odražava poslovnu domenu. Također, agregati se često koriste u kombinaciji s ograničenim kontekstima kako bi se osigurale jasne granice oko područja sustava te pomoglo u izoliranju složenosti [33].

U domeni vođenom dizajnom, entitet je objekt koji ima identitet koji ostaje konstantan tijekom njegovog životnog ciklusa [30]. Entiteti se koriste za prikazivanje pojmova iz poslovne domene i često su fokus poslovnih pravila i procesa. Entiteti se obično modeliraju kao objekti u sustavu i odgovorni su za održavanje vlastitog stanja i upravljanje vlastitim ponašanjem [33]. Prema S. Miteva [34], entiteti su kombinacija podataka i ponašanja poput korisnika ili proizvoda, oni imaju identitet, ali predstavljaju točke podataka s ponašanjem.

Grmljavina događaja (*eng. Event storming*) je način strategijskog dizajna koji uključuje suradnju domenskih stručnjaka i programera kako bi se učinkovito dizajnirao sustav. „Event storming“ uključuje sve potrebne sudionike usluge da se zajedno okupe i sudjeluju u procesu. Fokus je na događajima i poslovnim procesima, a ne na razvojnim elementima. Cijeli tim može povećati razumijevanje usluge, dok domenski stručnjaci mogu proširiti svoje razumijevanje i dobiti nova saznanja. [32]

Uzorak repozitorija (*eng. Repository pattern*) je uzorak dizajna u domeni vođenom dizajnom koji ima za cilj zadržati brigu o perzistenciji izvan modela domene sustava. Jedna ili više apstrakcija perzistencije - sučelja – definirane su unutar modela domene, a ove apstrakcije imaju implementacije u obliku adaptera specifičnih za perzistenciju koji su definirani drugdje u aplikaciji. [35]

Implementacije repozitorija su klase koje inkapsuliraju logiku potrebnu za pristup izvorima podataka. One centraliziraju zajedničku funkcionalnost pristupa podacima pružajući bolju održivost i razdvajajući infrastrukturu ili tehnologiju korištenu za pristup bazi podataka od samog modela domene. [35]

Prema M. Flower [36], repozitorij obavlja poslove posrednika između slojeva modela domene i preslikavanja podataka djelujući slično kao skup domenskih objekata u memoriji.

Repozitoriji također podržavaju svrhu jasnog i jednosmjernog razdvajanja ovisnosti između radnog područja i dodjele ili preslikavanja podataka.

6. Principi softverskog dizajna

6.1. Pametne/glupe komponente

6.1.1. Pametne komponente

Pametne komponente su softverske komponente koje posjeduju visok stupanj inteligencije i sposobne su obavljati složene zadatke. Obično se grade pomoću sofisticiranih algoritama i tehnika strojnog učenja. Pametne komponente su dizajnirane da autonomno donose odluke na temelju primljenih podataka te su sposobne prilagoditi se promjenjivim uvjetima dok uče iz svojih iskustava. [37]

Korištenjem uzorka dizajna kontejnera (*eng. Container Design Pattern*), kontejnerske komponente odvojene su od prezentacijskih komponenti i svaka od njih obrađuje svoje zadatke. Kontejnerske komponente obavljaju teške zadatke i prenose podatke prema prezentacijskim komponentama kao svojstva. Ove komponente često sadrže povratne funkcije (*eng. Callback functions*) koje se koriste za ažuriranje stanja i prijenos podataka prema njihovim podkomponentama kao svojstva. [38]

Pametne komponente su vrlo korisne u razvoju softvera jer omogućuju razvojnim programerima da uštede vrijeme i napore. One također poboljšavaju performanse aplikacije jer mogu optimizirati vlastito ponašanje. [37]

Primjer pametne komponente jest najčešće komponenta koja ima registriranu rutu u navigaciji te pritom služi kao oslonac za sve podkomponente koje se pokreću od strane nje. Osnovni razlog tome jest što glupe komponente nisu vezane uz neko postojanje ili fizičku lokaciju unutar aplikacije, već su primarno za apstrakciju i ponovno korištenje. Drugi način kako se može prepoznati pametna komponenta jest komunikacija komponente sa vanjskim resursima, kao što su „Web API„ ili „Application Gateway“.

6.1.2. Glupe komponente

Glupe komponente, također poznate kao pasivne komponente, su vrsta softverskih komponenti koje imaju nižu razinu inteligencije i nisu sposobne izvršavati složene zadatke. Ove komponente su obično dizajnirane da obavljaju određenu funkciju i nisu u stanju samostalno donositi odluke. [39]

Prema J. Arnold [38], jedina odgovornost koju glupe komponente imaju jest prikazivanje informacija u objektnim modelima dokumenata. Kada se taj zadatak obavi, komponenta nije odgovorna za daljnje praćenje ili održavanje podataka. Informacije se jednostavno prikazuju na stranici, a komponenta nastavlja s drugim zadacima.

Glupe komponente su jednostavnije i lakše za razvoj i održavanje u odnosu na pametne komponente. To je zato što ne zahtijevaju upotrebu složenih algoritama ili tehnika strojnog učenja, pa su lakše za razumijevanje i izmjenu. [39]

Jedna od mana glupih komponenti jest što nisu sposobne prilagoditi se promjenjivim uvjetima i ne mogu samostalno donositi odluke [40].

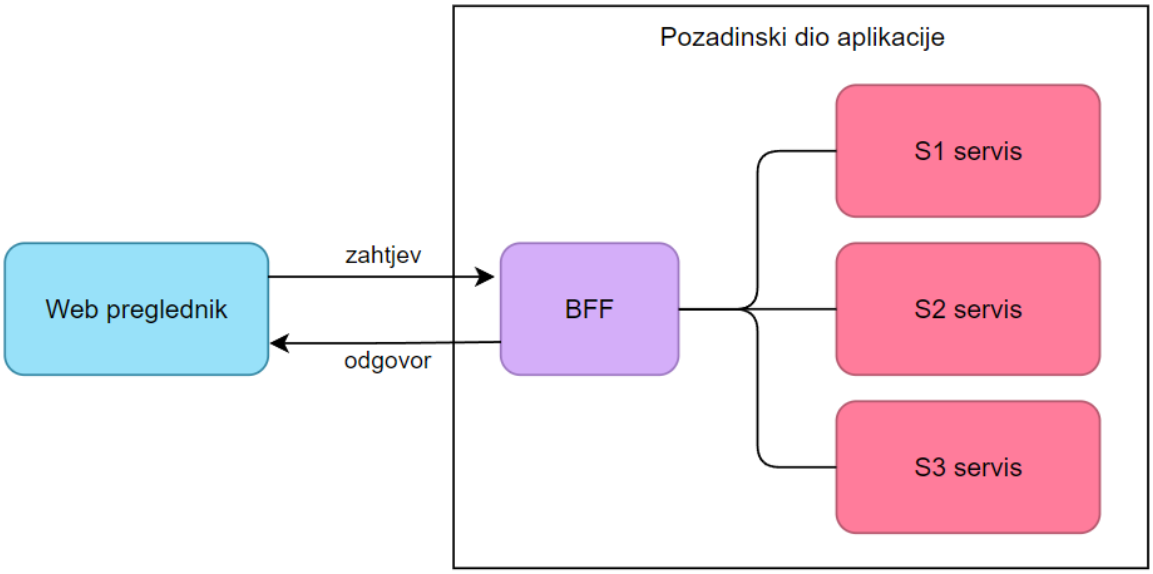
Primjer glupe komponente jest apstrakcija tekstualnog unosa. Unutar nje se definira izgled i ponašanje tekstualnog unosa koji će se koristiti u svim formama aplikacije. Najčešće se vizualne komponente razdvajaju u glupe komponente kako bi se lakše snalazili u programskom kodu neke komponente. Time bi logiku dohvaćanja i obrade podatka razdvojili od logike koja služi za implementaciju korisničkog iskustva.

6.2. Pozadinsko generiranje korisničkog sučelja

Prednji i pozadinski dio (*eng. Frontend and backend*) su dvije važne komponente razvoja softvera. Prednji dio je korisnički vidljivi dio aplikacije, dok je pozadinski dio tehnologija koja pokreće aplikaciju. Pozadinski dio se bavi svim teškim zadacima poput pohrane i dohvaćanja podataka, programiranja na strani poslužitelja i razvoja API-ja, dok se prednji dio usredotočuje na stvaranje besprijekornog i intuitivnog korisničkog iskustva. [41]

Princip pozadinskog generiranja korisničkog sučelja sugerira da bi programeri trebali izgraditi poseban pozadinski dio specifičan za prednji dio aplikacije. Ovaj posvećeni pozadinski dio, poznat kao BFF (*eng. Backend for Frontend*), pruža učinkovitiji način za rukovanje jedinstvenim potrebama i zahtjevima prednjeg dijela. Izgradnjom posvećenog pozadinskog dijela, programeri mogu pojednostaviti proces razvoja i osigurati da prednji i pozadinski dio bez problema surađuju. [42]

Prema [43], izgradnjom pozadinskog generiranja korisničkog sučelja mogu se stvoriti modularnije i skalabilnije aplikacije što može dovesti do bržih procesa razvoja, jednostavnijeg održavanja i manje pogrešaka. Jedan BFF je fokusiran samo na jedno korisničko sučelje. To pomaže da se održava jednostavnost prednjeg dijela aplikacije i ujedinjava prikaz podataka kroz njegov pozadinski dio [44].



Slika 10. Konceptualni prikaz pozadinskog generiranja korisničkog sučelja

7. Tehnologija

7.1. ASP.NET

ASP.NET je poslužiteljski okvir za razvoj web aplikacija koji je razvio Microsoft i omogućuje programerima stvaranje dinamičkih web aplikacija pomoću .NET jezika kao što su C# i Visual Basic [45]. Od svog predstavljanja 2002. godine, ASP.NET je evoluirao u snažan i svestran okvir koji podržava različite programske modele uključujući Web Forms, MVC i Web API [46].

ASP.NET je dizajniran s obzirom na proširivost i performanse. Uključuje značajke poput izlaznog predmemoriranja koje smanjuje opterećenje poslužitelja pohranjivanjem renderiranog HTML-a stranice i njegovog posluživanja zahtjevima [47]. Osim toga, ASP.NET koristi točno na vrijeme (*eng. JIT – Just In Time*) kompilaciju za optimizaciju izvršavanja programskog koda što dovodi do poboljšanih performansi [48].

ASP.NET dolazi s opsežnom .NET Framework klasnom bibliotekom (*eng. Class Library*), koja programerima pruža pristup širokom spektru unaprijed izgrađenih funkcionalnosti kao što su pristup podacima, kriptografija i obrada XML-a [49]. To omogućuje programerima da učinkovitije grade složene aplikacije koristeći prethodno postojeće dijelove (*eng. Modules*) programskog koda.

Uz uvođenje .NET Core, ASP.NET sada podržava razvoj koji se može koristiti na različitim platformama omogućujući programerima stvaranje web aplikacija koje se mogu pokrenuti na sustavima Windows, Linux i macOS [50]. Ova kompatibilnost s različitim platformama proširuje doseg ASP.NET aplikacija i čini okvir svestranijim.

Django je okvir za razvoj web aplikacija temeljen na Pythonu koji slijedi arhitektonski uzorak Model-Prikaz-Predložak (*eng. MVT – Model-View-Template*) [51]. Iako i ASP.NET i Django nude snažne biblioteke i modularan pristup razvoju, Django ima prednosti zbog jednostavnosti korištenja i čitljivosti Pythona. Međutim, značajke ASP.NET-a vezane uz performanse i skalabilnost, kao što su izlazno predmemoriranje i JIT kompilacija, pružaju mu prednost u pogledu optimizacije performansi [48].

Ruby on Rails, često nazivan Rails, je još jedan široko korišteni okvir za razvoj web aplikacija temeljen na programskom jeziku Ruby. Rails usvaja arhitekturu Model-Prikaz-Kontroler (*eng. MVC – Model-View-Controller*) i promiče konvencije nasuprot konfiguraciji, što pojednostavljuje razvojni proces [52]. Iako Rails dijeli neke sličnosti s ASP.NET-om, kao što

su modularni pristup i snažna biblioteka, prvenstveno je dizajniran za brzo prototipiranje te možda nije jednako prikladan za visoko učinkovite aplikacije kao što je ASP.NET [27].

7.2. Blazor

Blazor je suvremeni web-aplikacijski okvir koji omogućuje programerima da iskoriste svoje trenutno znanje C# i .NET-a za stvaranje dinamičnih i otpornih web aplikacija. Blazor se temelji na .NET okviru i koristi WebAssembly (Wasm) standard, koji je binarni format uputa koji olakšava učinkovito izvršavanje koda u današnjim web preglednicima. Okvir se sastoji od dva poslužiteljska modela: Blazor Server i Blazor WebAssembly. [53], [54]

Blazor nudi nekoliko značajki koje ga čine atraktivnim izborom za izgradnju modernih web aplikacija:

- arhitektura zasnovana na komponentama (*eng. Component-based architecture*),
- C# i Razor sintaksa,
- interoperabilnost s JavaScriptom i
- injekcija zavisnosti (*eng. DI – Dependency Injection*) [54].

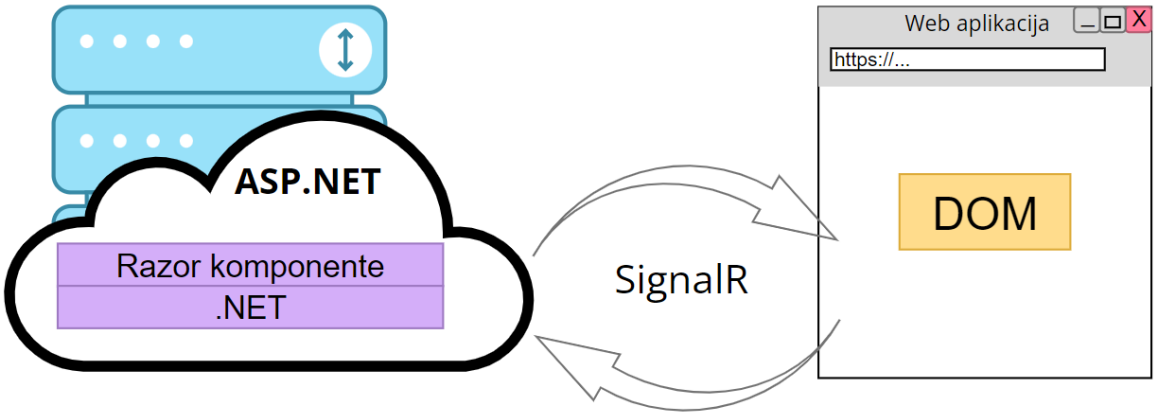
Blazor koristi strukturu zasnovanu na komponentama što omogućuje programerima dizajniranje ponovno upotrebljivih i modularnih elemenata korisničkog sučelja (*eng. UI elements – User Interface elements*) koji se mogu lako integrirati u izgradnju složenih korisničkih sučelja. Iskorištavajući mogućnosti C# i Razor sintakse, Blazor omogućuje programerima pisanje programskog koda za klijentsku i poslužiteljsku stranu koristeći ujedinjeni jezik i okvir. Okvir osigurava glatku interoperabilnost s JavaScriptom olakšavajući programerima pozivanje JavaScript funkcija iz C# programskog koda i obrnuto. Osim toga, ugrađena injekcija zavisnosti u Blazoru pojednostavljuje upravljanje zavisnostima među različitim komponentama i uslugama za programere. [53]

7.2.1. Blazor Server

Blazor Server upravlja komponentama aplikacije na strani poslužitelja i koristi prednosti SignalR-a za trenutna ažuriranja korisničkog sučelja. Ovaj poslužiteljski model omogućuje brzo vrijeme učitavanja i kompaktno početno preuzimanje (*eng. download*) budući da klijent mora preuzeti samo elemente korisničkog sučelja aplikacije. [55]

Ovaj pristup oslanja se na trajnu vezu između klijenta i poslužitelja što potencijalno rezultira izazovima s latencijom i skalabilnošću. Štoviše, zbog izvršavanja komponenti

aplikacije na poslužitelju, može rezultirati povećanom upotrebom resursa na strani poslužitelja i ograničiti mogućnost rada prilikom odsutnosti interneta. [54]

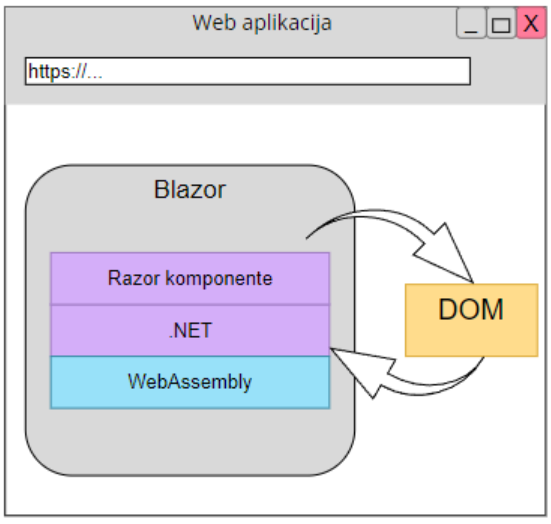


Slika 11. Prikaz rada Blazor Servera

7.2.2. Blazor WebAssembly

Blazor WebAssembly pokreće aplikaciju izravno u korisnikovom pregledniku koristeći .NET runtime baziran na WebAssemblyu za obradu komponenti. Ovaj poslužiteljski model pruža poboljšane performanse i omogućuje aplikaciji da funkcionira bez prisutnosti interneta jer su svi potrebni resursi preuzeti na klijentskoj strani. [54]

Međutim, početna veličina preuzimanja je znatnija jer klijent mora preuzeti kompletnu web aplikaciju, uključujući *„runtime“* i zavisnosti, što može dovesti do produženog početnog vremena učitavanja [55]. Nadalje, zbog izvršavanja web aplikacije na klijentskoj strani, mogla bi koristiti više klijentskih resursa, što potencijalno utječe na performanse korisnikovog uređaja [54].



Slika 12. Prikaz rada Blazor WebAssemblya

7.2.3. Usporedba poslužiteljskih modela

Pri odabiru između Blazor Servera i Blazor WebAssembly-a programeri bi trebali uzeti u obzir aspekte kao što su zahtjevi aplikacije, ciljana publika i ograničenja infrastrukture. Blazor Server može biti prikladniji izbor za aplikacije koje zahtijevaju ažuriranja u stvarnom vremenu i daju prednost brzom vremenu učitavanja, dok bi Blazor WebAssembly mogao biti prikladniji za aplikacije koje zahtijevaju funkcionalnost izvan mreže i performanse na klijentskoj strani. [54]

Blazor Server i Blazor WebAssembly nude različite prednosti i kompromise. Razumijevanjem prednosti i nedostataka svakog modela, programeri su bolje opremljeni za donošenje informiranih odluka i odabir najprikladnijeg rješenja za svoje web aplikacije.

7.2.4. Usporedba s drugim tehnologijama

Angular je sveobuhvatan okvir za web aplikacije razvijen od strane Google-a koji koristi TypeScript, statički tipizirani nadskup JavaScripta, te nudi bogat skup ugrađenih funkcionalnosti. Iako je Angular moćan, ima strmiju krivulju učenja u usporedbi s Blazorom. Nadalje, C# i .NET integracija u Blazoru može biti privlačnija za developere s iskustvom u Microsoft tehnologijama. [56], [57]

React je popularna JavaScript biblioteka razvijena od strane Facebooka za izgradnju korisničkih sučelja. S velikom zajednicom i širokim ekosustavom biblioteka i alata, React zahtijeva od developera učenje i korištenje JavaScripta, dok Blazor omogućuje razvoj punog stoga u C#. Osim toga, React ne nudi ugrađeni sustav injektiranja zavisnosti, koji je dostupan u Blazoru. [58]

Vue je lagan i fleksibilan okvir za web aplikacije koji je jednostavan za učenje i integraciju u postojeće projekte. Kao i React, Vue se oslanja na JavaScript za razvoj. Blazor ima prednost nad Vueom u pogledu ugrađenog sustava injektiranja zavisnosti i mogućnosti korištenja C# i .NET vještina. Međutim, Vue-ova jednostavnost i jednostavnost upotrebe čine ga privlačnom opcijom za manje projekte i developere koji traže brzi početak. [59]

8. Aplikacija

Koncept ove aplikacije temelji se na stvaranju integrirane platforme koja omogućava sveobuhvatno upravljanje željezničkim prometom. Kombinirajući različite funkcionalnosti, aplikacija ne samo da omogućava evidentiranje vlakova i vozni red, već također pruža mogućnost prodaje karata.

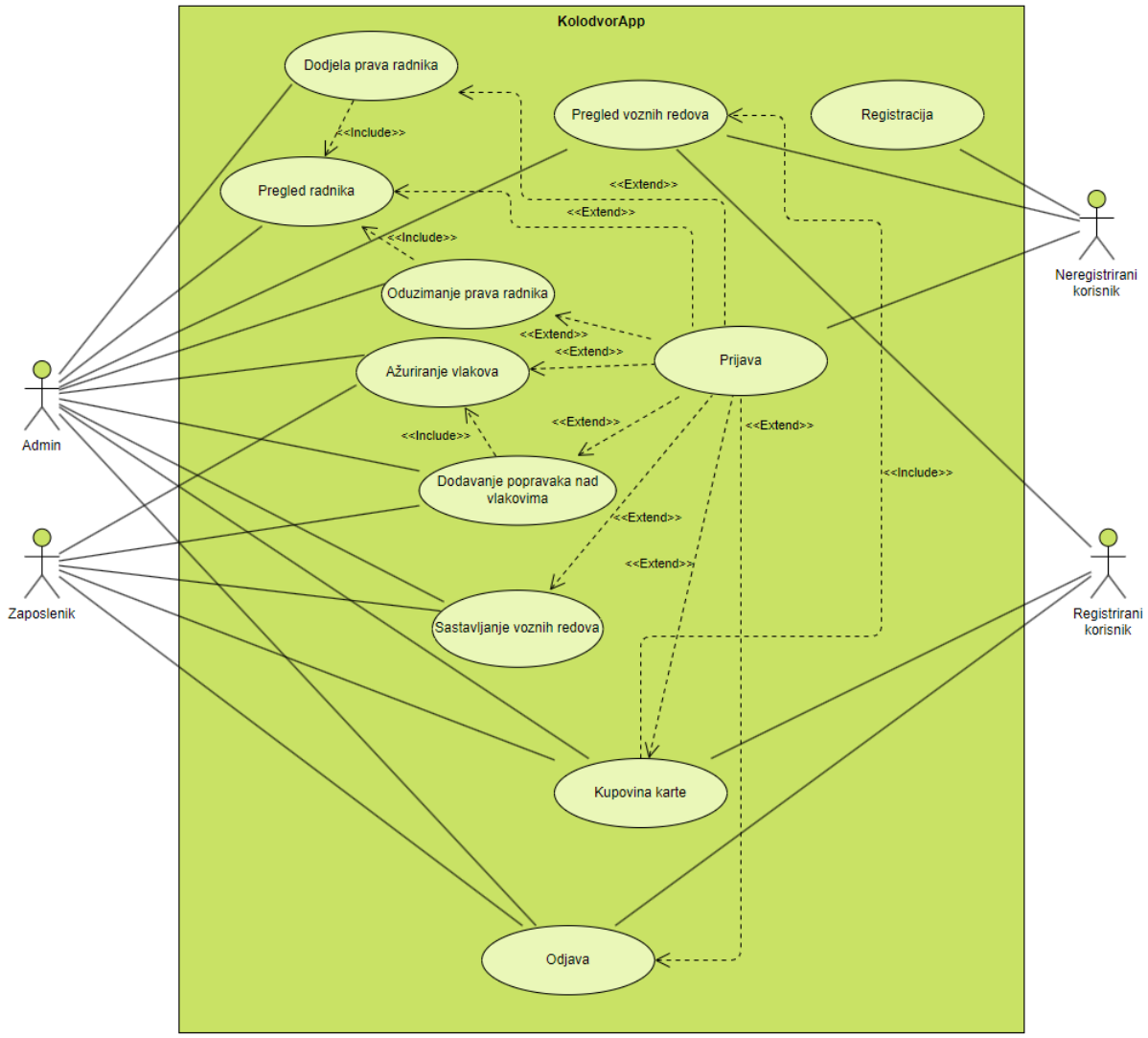
U kontekstu evidencije vlakova, aplikacija ne pruža samo suhoparnu listu vlakova, već nudi detaljne informacije o svakom vlaku, uključujući njegovu trenutnu dostupnost i kapacitet. Primjerice, korisnici će za putovanje od Koprivnice do Varaždina moći vidjeti je li dostupan vlak s tri ili četiri vagona ili je možda na raspolaganju samo stariji vlak s dva vagona.

S obzirom na upravljanje kadrovima, svaki član osoblja koristit će osobne akreditacije za prijavu u aplikaciju. Ove akreditacije određivat će korisničke ovlasti unutar aplikacije, ograničavajući pregled informacija ovisno o funkcionalnoj ulozi svakog korisnika.

Raspored voznih redova bit će predstavljen kao zaseban segment aplikacije. Zaduženo osoblje bit će odgovorno za sastavljanje i ažuriranje ovog rasporeda koji će služiti kao oglasna ploča za sve vožnje dostupna svim korisnicima za pregled.

Aplikacija će sadržavati i evidenciju svih popravaka na vlakovima omogućujući transparentno praćenje troškova povezanih s održavanjem vlakova. Na taj način će se moći pratiti koliko je novca utrošeno na pojedini popravak.

Konačno, aplikacija će imati segment primarno usmjeren na korisnike nudeći im jednostavan i intuitivan proces kupnje karata za raspoložive vozne redove. Ova korisnički orijentirana komponenta doprinosi sveobuhvatnosti i funkcionalnosti aplikacije nudeći potpunu uslugu za korisnike i osoblje.



Slika 13. Prikaz UML dijagrama slučajeva korištenja

Pregled radnika je proširenje funkcionalnosti prijave kako bi se osiguralo da samo administrator ima pristup. Ova funkcionalnost djeluje na principu da samo korisnici s ovlastima administratora imaju pravo pristupa i interakcije s funkcionalnošću za upravljanje korisnicima.

Ova proširena funkcionalnost „Pregled radnika“ dodatno uključuje mogućnosti unapređenja i degradiranja korisnika. Sama funkcionalnost pregledavanja korisnika, iako je u svojoj osnovi samo-inkluzivna, omogućuje administratorima da pregledaju korisničke trenutne ovlasti te da vrše promjene statusa korisnika, tj. unapređenja ili degradiranja. Ovaj korak je ključan u postavljanju različitih razina pristupa i kontrola unutar aplikacije.

Naime, vezom "uključi" između funkcionalnosti pregledavanja korisnika i unapređivanja/degradiranja korisnika, prikazuje se kako su ove dvije funkcionalnosti usko povezane. Promjena statusa korisnika (unapređivanje/degradiranje) ne može se izvesti bez prethodnog pregledavanja korisnika. Dakle, pregled korisnika je osnovna funkcionalnost koja omogućuje unapređivanje ili degradiranje korisnika.

Funkcionalnost prijave ima ključni utjecaj na upotrebljivost aplikacije jer definira koji korisnici mogu pristupiti kojim dijelovima aplikacije. Ona je esencijalna u održavanju sigurnosti i integriteta informacija unutar aplikacije budući da ograničava pristup i manipulaciju osjetljivim podacima na isključivo ovlaštene korisnike.

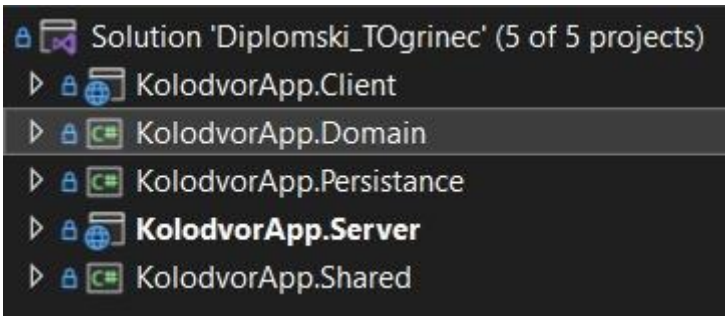
Neregistrirani korisnici, odnosno korisnici koji nisu izvršili prijavu, imaju ograničen pristup funkcionalnostima aplikacije. U ovom slučaju mogu se koristiti samo s funkcionalnošću "Pregled voznih redova" što znači da mogu pregledavati dostupne rute bez potrebe za pristupom osjetljivim funkcionalnostima. Ovo je bitno za privlačenje i informiranje potencijalnih korisnika omogućujući im da vide ključne informacije o rutama prije nego što se odluče za registraciju i prijavu.

Funkcionalnost "Pregled voznih redova" osmišljena je tako da je jednostavna za korištenje i dostupna svima, što poboljšava upotrebljivost aplikacije i privlačnost za nove korisnike. Istovremeno, ograničavanje pristupa drugim funkcionalnostima osigurava da se osjetljive informacije i kontrolni elementi aplikacije održavaju sigurnima i dostupnima samo ovlaštenim korisnicima.

Funkcionalnosti prijave i registracije predstavljaju vrata do ostalih funkcionalnosti aplikacije. Kroz proces registracije korisnici se mogu uključiti u sustav, dok proces prijave osigurava autentifikaciju korisnika prije pristupa daljnjim funkcionalnostima. Ova dva koraka su neophodna u postavljanju granica i kontrole pristupa unutar aplikacije te stvaraju temelj za sigurno i učinkovito korištenje aplikacije.

8.1. Arhitektura aplikacije

Odabrana arhitektura za web aplikaciju jest Clean arhitektura koja je koncipirana kao višeslojna arhitektura u skladu s arhitektonskim obrascem. Ova struktura je dalje obogaćena principima domenom vođenog dizajna (*eng. DDD – Domain-Driven Design*) kako bi se osigurala još veća strukturalna organiziranost i jasnoća pozadinskog rada aplikacije. Implementacija DDD-a doprinosi boljem modeliranju poslovne domene unutar arhitekture, omogućavajući preciznije usklađivanje stvarnog svijeta na softverski model. Ovakav pristup rezultira visokom razinom apstrakcije, modularnosti i nezavisnosti među komponentama sustava, olakšavajući održavanje i testiranje. Svaki sloj u arhitekturi ima definiranu ulogu i odgovornost čime se smanjuje složenost i povećava čitljivost koda.



Slika 14. Raspored projekata unutar rješenja

Implementacija DDD-a izvedena je kroz identifikaciju i modeliranje ključnih agregatnih korijena (*eng. Aggregate Roots*) koji predstavljaju grupu povezanih entiteta kojima se pristupa kao jedinstvenoj jedinici za čitanje i pisanje. Također, uveden je uzorak repozitorija što pruža apstrakciju od specifičnih detalja upravljanja podacima, omogućavajući time univerzalne operacije za dohvat i manipulaciju objektima domene bez ometanja poslovne logike.

Dodatno, poslovne funkcionalnosti su razdijeljene na domenske i aplikacijske servise. Domenski servisi upravljaju složenim operacijama unutar poslovne domene, dok aplikacijski servisi koordiniraju domenske servise kako bi izveli specifične operacije na visokoj razini često uključujući upravljanje transakcijama i koordinaciju između različitih domenskih objekata. Ova podjela odgovornosti omogućuje jasniju strukturu i čitljivost koda te bolje održavanje i testiranje.

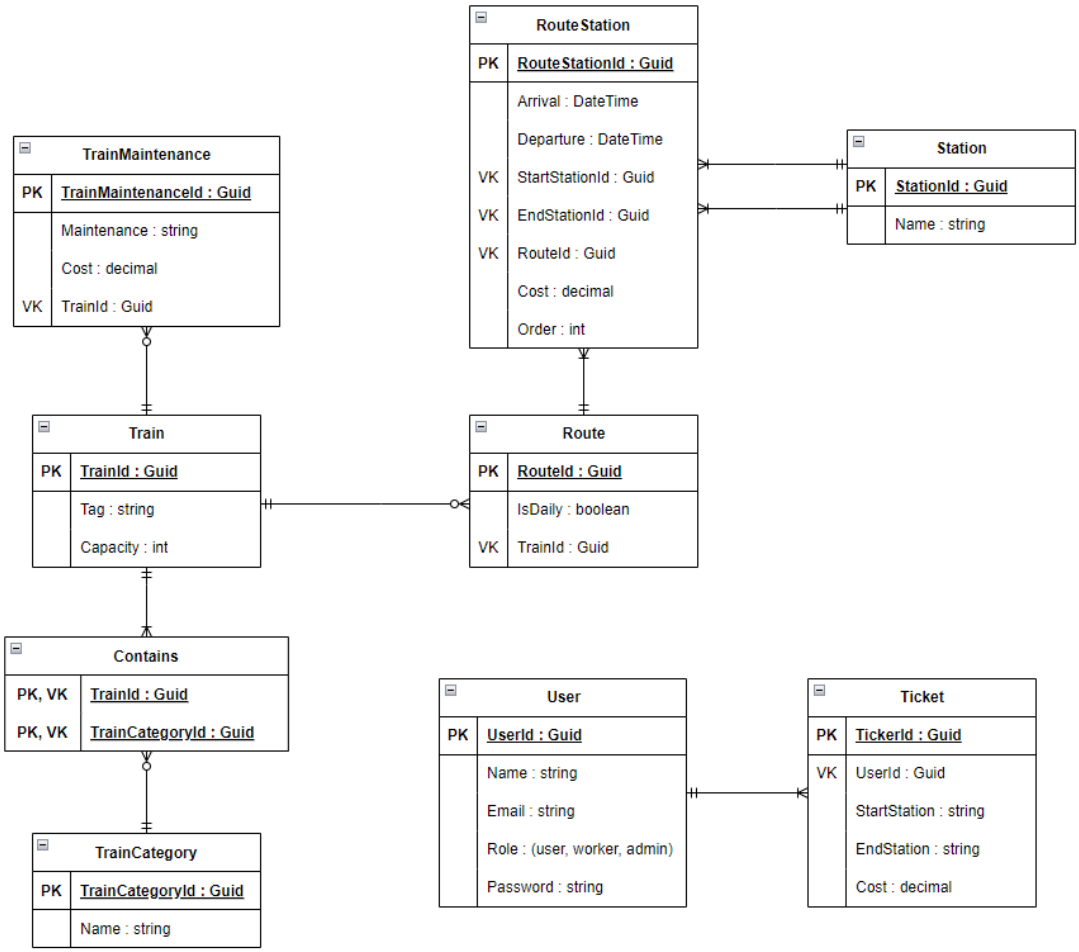
8.2. Baza podataka

Podatkovni sloj aplikacije dizajniran je koristeći se SQL bazom podataka koja je migrirana iz Entity Framework (EF) Core ORM-a. Ova baza podataka funkcionira kao temeljna struktura za pohranu, preuzimanje i manipulaciju podacima unutar aplikacije.

U kontekstu Entity Framework Core ORM-a, to je objektno-relacijski maper (*eng. ORM – object-related mapper*) koji preslikava objekte u relacijske modele. Efikasnost EF Core ORM-a leži u njegovoj sposobnosti da automatski generira SQL naredbe koje se izvršavaju na bazi podataka. Na taj način, EF Core olakšava posao programerima pružajući apstraktan način za manipulaciju podacima bez potrebe za ručnim pisanjem SQL naredbi. [60]

Baza podataka korištena je u projektu Perzistencije unutar .NET rješenja. Projekti Perzistencije, u kontekstu razvoja softvera, odnose se na komponente ili module u aplikaciji koji su odgovorni za interakciju s bazom podataka ili drugim mehanizmima za pohranu podataka. U ovom slučaju, Perzistencija projekta upravlja svim operacijama povezanim s bazom podataka uključujući čitanje, pisanje, ažuriranje i brisanje podataka, dok je sam kod implementacije servisa u Domenskom projektu koji koristi sučelje repozitorija iz projekta Perzistencije.

Kombinacijom ovih tehnologija i pristupa, podatkovni sloj aplikacije optimiziran je za efikasno upravljanje podacima, omogućavajući aplikaciji da ostvari svoje funkcionalnosti na način koji je skalabilan i održiv [60].



Slika 15. Prikaz ER dijagrama

Podatkovni model sastoji se od različitih entiteta koji predstavljaju ključne koncepte u domeni željezničke usluge. Ovi entiteti uključuju "User", "Ticket", "Train", "TrainMaintenance", "TrainCategory", "Contains", "Station", "RouteStation" i "Route". Navedeni entiteti su međusobno povezani preko različitih veza koje omogućuju modeliranje kompleksnih relacija u domeni problema.

Entitet "User" predstavlja korisnika sustava i posjeduje jedinstveni identifikator e-pošte. Svaki korisnik može kupiti nekoliko karata, što je modelirano kroz jedan-naprema-više vezom prema entitetu "Ticket".

Entitet "Ticket" predstavlja kartu koju korisnik kupuje. Ovaj entitet sadrži strani ključ "UserId" koji povezuje kartu s korisnikom koji je kupio tu kartu.

Entitet "Train" predstavlja vlak koji ima sposobnost pripadanja više kategorijama ("TrainCategory") koje su modelirane preko veznog entiteta "Contains". Svaki vlak može imati niz održavanja ("TrainMaintenance") i može biti dodijeljen različitim rutama ("Route").

Entitet "TrainMaintenance" predstavlja održavanje vlaka i povezan je s vlakom preko stranog ključa "TrainId".

Entitet "Station" predstavlja stanicu koja može biti ili početna ili završna stanica za različite rute. Ova veza je modelirana preko entiteta "RouteStation" koji također sadrži informacije o vremenima dolaska i odlaska na svaku stanicu.

Entitet "Route" predstavlja rutu koja je povezana s vlakom i sadrži niz stanica modeliranih preko entiteta "RouteStation".

Primjerice, konfiguracija entiteta "RouteStation" zahtijeva definiranje jedinstvenog ključa sastavljenog od "StartStationId", "EndStationId", "RouteId" i "Order". EF Core će to interpretirati i stvoriti odgovarajući složeni indeks unutar relacijske tablice za entitet "RouteStation", čime se osigurava jedinstvenost kombinacije ovih atributa.

Nadalje, entitet "RouteStation" sadrži relacijske veze prema entitetima "Station" i "Route". Ove veze omogućuju entitetu "RouteStation" da posjeduje referentne točke do povezanih entiteta koje će EF Core uspostaviti kroz strane ključeve unutar odgovarajuće tablice.

Konačno, neki atributi entiteta, poput "ArrivalTime" i "DepartureTime", imaju posebne zahtjeve za upravljanje vremenom. EF Core osigurava zanemarivanje tih polja prilikom mapiranja što omogućuje programerima da održavaju visoki stupanj apstrakcije pri rukovanju s datumima i vremenima, dok istodobno osiguravaju ispravno pohranjivanje podataka u bazu podataka.

```
protected override void OnModelCreating(
      ModelBuilder modelBuilder){
            ...
            modelBuilder.Entity<RouteStation>(b =>
            {
             b.ToTable("RouteStations");
             b.HasIndex(x => new { x.StartStationId, x.EndStationId,
                 x.RouteId, x.Order })
                 .IsUnique();
             b.Property(x => x.Order)
                 .IsRequired();
             b.Property(x => x.Cost)
                 .HasPrecision(18, 2)
                 .IsRequired();
```

```
        b.Property(x => x.Arrival)
            .IsRequired();
        b.Property(x => x.Departure)
            .IsRequired();
        b.Ignore(x => x.ArrivalTime);
        b.Ignore(x => x.DepartureTime);
        b.Navigation(x => x.StartStation)
            .AutoInclude();
        b.Navigation(x => x.EndStation)
            .AutoInclude();
        b.HasOne(x => x.Route)
            .WithMany(x => x.RouteStations)
            .HasForeignKey(x => x.RouteId)
            .OnDelete(DeleteBehavior.Cascade)
            .IsRequired();
      });
      ...
}
```

8.3. Serverski sloj aplikacije

Serverski sloj aplikacije razvijen je koristeći Blazor WebAssembly tehnologiju kao dio .NET ekosustava. Aplikacija koristi domenom vođen dizajn (DDD) kao pristup u svom arhitektonskom dizajnu kako bi se postigla bolja organizacija koda, jasnoća poslovnih zahtjeva i veća održivost sustava.

U okviru ovog pristupa, poslovna logika i entiteti aplikacije modelirani su kroz domenske entitete i vrijednosne objekte koji su pažljivo osmišljeni kako bi odražavali stvarne poslovne procese i zahtjeve. Ovi domenski objekti formiraju temelj aplikacije i služe kao centralna točka za definiranje i implementaciju poslovne logike.

Aplikacija također koristi uzorak generičkog repozitorija (*eng. Generic Repository pattern*) kao dio svoje arhitekture pristupa podacima. Ovaj obrazac omogućuje apstrakciju pristupa podacima putem generičkih repozitorija koji se mogu primijeniti na različite domenske entitete. Kroz ovaj pristup, logika pristupa podacima je centralizirana što rezultira manje redundantnim kodom i olakšava testiranje i održavanje aplikacije.

```
public class Repository<TEntity> : IRepository<TEntity>, IDisposable
    where TEntity : class, IEntity
{
    internal readonly KolodvorAppContext _dbContext;

    public Repository(KolodvorAppContext context)
    {
        _dbContext = context;
    }

    public async Task<TEntity> GetAsync(Guid id)
    {
        return await FindAsync(entity => entity.Id == id) ?? throw
           new KeyNotFoundException();
    }

    public async Task<TEntity?> FindAsync(
      [NotNull]Expression<Func<TEntity, bool>> predicate)
    {
        return await GetDbSet()
           .Where(predicate).FirstOrDefaultAsync();
    }

    public virtual IQueryable<TEntity> GetAll()
      => GetDbSet().AsQueryable();

    public IQueryable<TEntity> WithDetails(
      params Expression<Func<TEntity, object>>[] propertySelectors)
    {
        return Repository<TEntity>.IncludeDetails(
            GetDbSet().AsQueryable(), propertySelectors);
```

```
    }

    public async Task<TEntity> InsertAsync([NotNull] TEntity entity)
    {
        var savedEntity = (await
           GetDbSet().AddAsync(entity)).Entity;

        await _dbContext.SaveChangesAsync();

        return savedEntity;
    }

    public async Task<TEntity> UpdateAsync([NotNull] TEntity entity)
    {
        var updatedEntity = _dbContext.Update(entity).Entity;

        await _dbContext.SaveChangesAsync();

        return updatedEntity;
    }

    public virtual async Task DeleteAsync(Guid key)
    {
        var entity = await FindAsync(entity => entity.Id == key) ??
           throw new KeyNotFoundException();

        GetDbSet().Remove(entity);

        await _dbContext.SaveChangesAsync();
    }

    private DbSet<TEntity> GetDbSet()
    {
        return _dbContext.Set<TEntity>();
    }

    private static IQueryable<TEntity> IncludeDetails(
        IQueryable<TEntity> query,
        Expression<Func<TEntity, object>>[] propertySelectors)
    {
        if (propertySelectors.Length > 0)
        {
            foreach (var propertySelector in propertySelectors)
            {
                query = query.Include(propertySelector);
            }
        }
        return query;
    }

    public void Dispose()
    {
        _dbContext.Dispose();
        GC.SuppressFinalize(this);
    }
}
```

Ovaj programski kod predstavlja generički repozitorij koji enkapsulira CRUD operacije za određenu vrstu entiteta u aplikaciji. Ova klasa radi s entitetima koji implementiraju sučelje „IEntity“, koje pretpostavljamo da sadrži svojstvo „Id“ tipa „Guid“. Ovu implementaciju koristi Entity Framework Core za rukovanje operacijama s podacima.

Korištenje „async“/“await“ u metodama omogućuje da operacije budu neblokirajuće, što je korisno za performanse aplikacije. Ovdje repozitorijski obrazac pomaže apstrahirati sloj pristupa podacima i promiče čistu odvojenost odgovornosti u aplikaciji.

U aplikaciji "KolodvorApp", domenski sloj igra ključnu ulogu u modeliranju poslovne logike i strukturalnih komponenata sustava. To je sloj koji sadrži sve entitete, domenske servise i kalkulacijske servise koji predstavljaju temeljne koncepte i operacije unutar domene problema koju aplikacija rješava.

- **Entiteti:** U ovom kontekstu, entiteti su klase koje predstavljaju različite poslovne objekte i koncepte koji su relevantni za problem koji aplikacija rješava. Oni predstavljaju ključne sastavnice domenskog modela i sadrže atribute i svojstva koja reflektiraju karakteristike i ponašanja tih objekata u stvarnom svijetu [30]. Na primjer, u aplikaciji "KolodvorApp" entiteti poput "Train", "Station", "Route", i "Ticket" predstavljaju različite poslovne koncepte koji su ključni za rad sustava.
- **Domenski servisi:** Domenski servisi su klase koje enkapsuliraju poslovnu logiku i operacije koje nisu prirodno dio samih entiteta. Oni obavljaju operacije koje uključuju više entiteta i koordiniraju interakcije između njih. Domenski servisi igraju ključnu ulogu u orkestraciji kompleksnih poslovnih procesa i pružaju sredstva za obavljanje akcija koje prelaze granice pojedinih entiteta. [61]
- **Kalkulacijski servisi:** Kalkulacijski servisi su specifična vrsta domenskih servisa koji se fokusiraju na izvršavanje složenih matematičkih i financijskih operacija unutar domenskog modela [36]. Ovi servisi mogu biti odgovorni za proračun cijena karata, obračun troškova održavanja vlakova ili izvođenje drugih analitičkih i izračunskih operacija koje su ključne za rad sustava.

```
public interface IEntity
{
    public Guid Id { get; set; }
}

public abstract class BaseEntity : IEntity
{
    public Guid Id { get; set; }
}
```

```
public class RouteStation : BaseEntity
{
    public DateTime Arrival { get; set; }

    public TimeOnly ArrivalTime
    {
        get => TimeOnly.FromDateTime(Arrival);
        set => Arrival = new DateOnly(2020, 1, 1).ToDateTime(value);
    }

    public DateTime Departure { get; set; }

    public TimeOnly DepartureTime
    {
        get => TimeOnly.FromDateTime(Departure);
        set => Departure = new DateOnly(2020, 1, 1)
                .ToDateTime(value);
    }

    public Guid StartStationId { get; set; }

    public virtual Station StartStation { get; set; } = null!;

    public Guid EndStationId { get; set; }

    public virtual Station EndStation { get; set; } = null!;

    public Guid RouteId { get; set; }

    public Route Route { get; set; } = null!;

    public decimal Cost { get; set; }

    public int Order { get; set; }
}
```

Svaka klasa u ovom primjeru ima svoju svrhu i funkcionalnost koja je u skladu s principima DDD-a i objektno orijentiranog programiranja.

- **Sučelje IEntity:** Ovo je sučelje koje definira osnovno svojstvo koje svaki entitet treba imati - identifikator (ID). Identifikator je ključna komponenta entiteta jer omogućuje jedinstvenu identifikaciju entiteta u sustavu. U ovom slučaju, identifikator je tipa Guid.
- **Klasa BaseEntity:** Ova klasa implementira sučelje „IEntity“ i predstavlja osnovnu implementaciju entiteta. Sadrži svojstvo „Id“ koje je definirano sučeljem. Ova klasa služi kao apstraktna osnova za sve ostale entitete u sustavu i osigurava da svaki entitet ima svoj jedinstveni identifikator.
- **Klasa RouteStation:** Ova klasa nasljeđuje „BaseEntity“ i predstavlja konkretan domenski entitet. Sadrži različita svojstva koja reflektiraju svojstva stvarnog entiteta u poslovnu domenu. U ovom slučaju, RouteStation predstavlja postaju unutar određene rute u željezničkom sustavu. Svojstva ove klase uključuju informacije o vremenu

dolaska i odlaska, povezanim stanicama i cijeni. Također, sadrži navigacijska svojstva (StartStation, EndStation, Route) koja omogućuju povezivanje s drugim entitetima u sustavu.

```
public class RouteService : IRouteService
{
    private readonly IMapper _mapper;
    private readonly IRepository<Route> _repository;
    private readonly ITrainService _trainService;

    public RouteService(IMapper mapper,
     IRepository<Route> repository, ITrainService trainService)
    {
        _mapper = mapper;
        _repository = repository;
        _trainService = trainService;
    }

    public List<RouteDto> GetAll()
    {
        var routeList = _repository.GetAll();
        return _mapper.Map<List<RouteDto>>(routeList);
    }

    public async Task<RouteDto> CreateOrUpdateAsync(
     RouteDto routeDto)
    {
        var route = _mapper.Map<Route>(routeDto);

        if (routeDto.Id is null)
        {
            route = await _repository.InsertAsync(route);
        }
        else
        {
            try
            {
                var entity = await _repository.GetAsync(route.Id);
                _mapper.Map(route, entity);
                entity.Train = await
                      _trainService.GetSpecificAsync(route.TrainId);
                route = await _repository.UpdateAsync(entity);
            }
            catch (KeyNotFoundException)
            {
                throw new InvalidOperationException("Tried to update
                     an non-existing entity.");
            }
        }
        route = await _repository.GetAsync(route.Id);
        return _mapper.Map<RouteDto>(route);
    }
```

```
    public async Task DeleteAsync(Guid id)
    {
        await _repository.DeleteAsync(id);
    }
}
```

Ovaj primjer ilustrira kako domenski servis orkestrira različite operacije na domenskim entitetima i kako koristi različite komponente, poput repozitorija i drugih servisa, kako bi obavljao poslovne operacije. Implementacija sučelja „IRouteService“ osigurava da se ovaj servis može koristiti kao dio većeg sustava u skladu s principima DDD-a i objektno orijentiranog programiranja.

Klasa „RouteService“ sadrži različite komponente koje su ključne za izvršavanje poslovnih operacija na entitetu „Route“. Sadrži sljedeće komponente:

- **Imapper _mapper:** Objekt koji se koristi za preslikavanje modela između različitih tipova objekata. U ovom slučaju, koristi se za preslikavanje između entiteta „Route“ i DTO objekta „RouteDto“.
- **IRepository<Route> _repository:** Repozitorij koji pruža pristup entitetima tipa Route u bazi podataka. Omogućuje izvršavanje CRUD operacija na ovim entitetima.
- **ITrainService _trainService:** Servis koji pruža pristup funkcionalnostima vezanim za entitet „Train“.

```
public class RouteCalculatorService : IRouteCalculatorService
{
    private readonly IRouteService _routeService;

    public RouteCalculatorService(IRouteService routeService)
    {
        _routeService = routeService;
    }

    public List<MergedRoutesDto> FindTravelPaths(
      RouteSearchDto routeSearch)…

    private static List<RouteDto> IsRouteAvailable(
      List<RouteDto> routes, DateTime dateOfTravel)…

    private static List<RouteSegmentDto> PrepareRouteSegments(
      List<RouteDto> filteredRoutes)…

    private static List<RouteSegmentDto> GetNextSegments(
      List<RouteSegmentDto> routeSegments,
      RouteStationDto endStation)…
}
```

Za razliku od uobičajenih domenskih servisa, kalkulacijski servisi obično nemaju direktan pristup bazi podataka. Umjesto toga, oni su usmjereni na obradu podataka koji su im prethodno predani putem drugih servisa unutar domenskog sloja. Na primjer, kalkulacijski

servis može biti odgovoran za proračun ukupnih troškova putovanja na temelju cijena karata za pojedinačne rute ili za izračun popusta na temelju određenih kriterija.

Takav pristup ima nekoliko prednosti. Prvo, omogućuje veću razinu modularnosti i odvajanja odgovornosti unutar aplikacije. Kalkulacijski servisi mogu biti neovisni o infrastrukturnim detaljima i mogu se razvijati i održavati neovisno o ostatku sustava. Drugo, olakšava testiranje kalkulacijskih servisa, budući da se testovi mogu fokusirati isključivo na proračunske i analitičke operacije, bez potrebe za simuliranjem pristupa bazi podataka. Treće, poboljšava se pouzdanost i performanse aplikacije jer kalkulacijski servisi mogu biti optimizirani za brzo i učinkovito izvršavanje proračuna bez opterećenja povezanog s pristupom i upravljanjem podacima u bazi podataka.

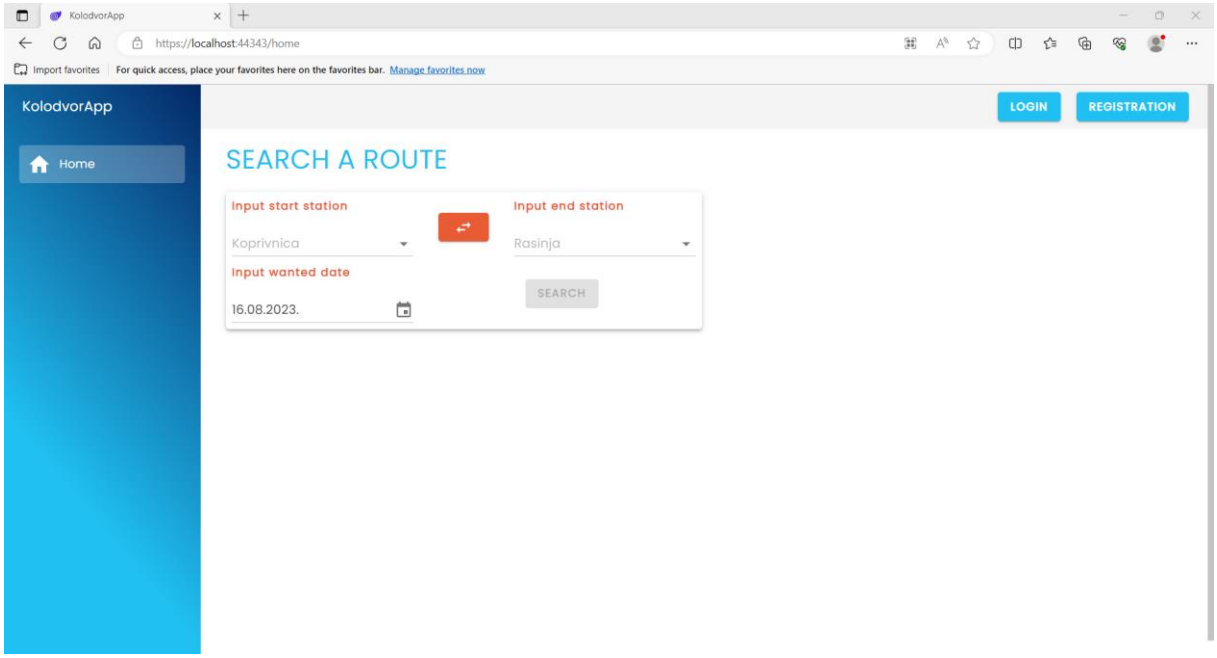
```
[ApiController]
[Route("[controller]")]
public class RoutesController : ControllerBase
{
    private readonly IRouteService _service;
    private readonly IRouteCalculatorService
     _routeCalculatorService;

    public RoutesController(IRouteService service,
     IRouteCalculatorService routeCalculatorService)
    {
        _service = service;
        _routeCalculatorService = routeCalculatorService;
    }

    [HttpGet]
    public ActionResult<List<RouteDto>> GetAll()
    {
        var result = _service.GetAll();

        return Ok(result);
    }

    [HttpPost]
    public async Task<ActionResult<RouteDto>>
     CreateOrUpdateAsync([FromBody] RouteDto routeDto)
    {
        try
        {
            var result = await
                _service.CreateOrUpdateAsync(routeDto);
            return Ok(result);
        }
        catch (InvalidOperationException ex)
        {
            return BadRequest(ex.Message);
        }
        catch (Exception)
        {
```

```
            return Conflict("There's an error on the server.");
        }
    }

    [HttpPost("search")]
    public ActionResult<List<MergedRoutesDto>>
     SearchRoutes([FromBody] RouteSearchDto searchInfo)
    {
        var result = _routeCalculatorService
           .FindTravelPaths(searchInfo);
        return Ok(result);
    }

    [HttpDelete("{id}")]
    public async Task<IActionResult> DeleteAsync(Guid id)
    {
        try
        {
            await _service.DeleteAsync(id);
            return NoContent();
        }
        catch (KeyNotFoundException)
        {
            return BadRequest($"Route {id} does not exist.");
        }
        catch (Exception)
        {
            return Conflict("There's an error on the server.");
        }
    }
}
```

Kontroler je implementiran u programskom okruženju .NET Core, koristeći atribut "ApiController" i specifikaciju rute za krajnje točke (*eng. Endpoint*) API-ja. Kontroler koristi dva servisa: "IRouteService" i "IRouteCalculatorService", koji su proslijeđeni u konstruktor kontrolera putem injektiranja ovisnosti. Ovo omogućuje kontroleru da komunicira s domenskim i kalkulacijskim servisima.

8.4. Korisnički sloj aplikacije

U okviru klijentske strane navedene web aplikacije implementirano je korisničko sučelje koje se odlikuje visokim stupnjem sofisticiranosti i intuitivnosti. Sučelje je konstruirano s ciljem pružanja transparentnosti i jasnoće korisnicima, uz istovremenu primjenu suvremenih estetskih kriterija u dizajnu. Strukturirana i efikasna navigacija osigurava korisnicima optimizirano korisničko iskustvo. Temeljna grafička rješenja i vizualna prezentacija korisničkog sučelja realizirana su primjenom tehnologija HTML5, CSS3 i JavaScript. Za implementaciju dinamičkih i interaktivnih komponenata na klijentskoj strani, korišten je okvir Blazor WebAssembly. Ovaj okvir, koji omogućuje razvoj visoko učinkovitih aplikacija pomoću programskog jezika C# koji se izvršava u pregledniku putem WebAssembly tehnologije, pruža značajnu fleksibilnost i proširivost u razvoju suvremenih web aplikacija.

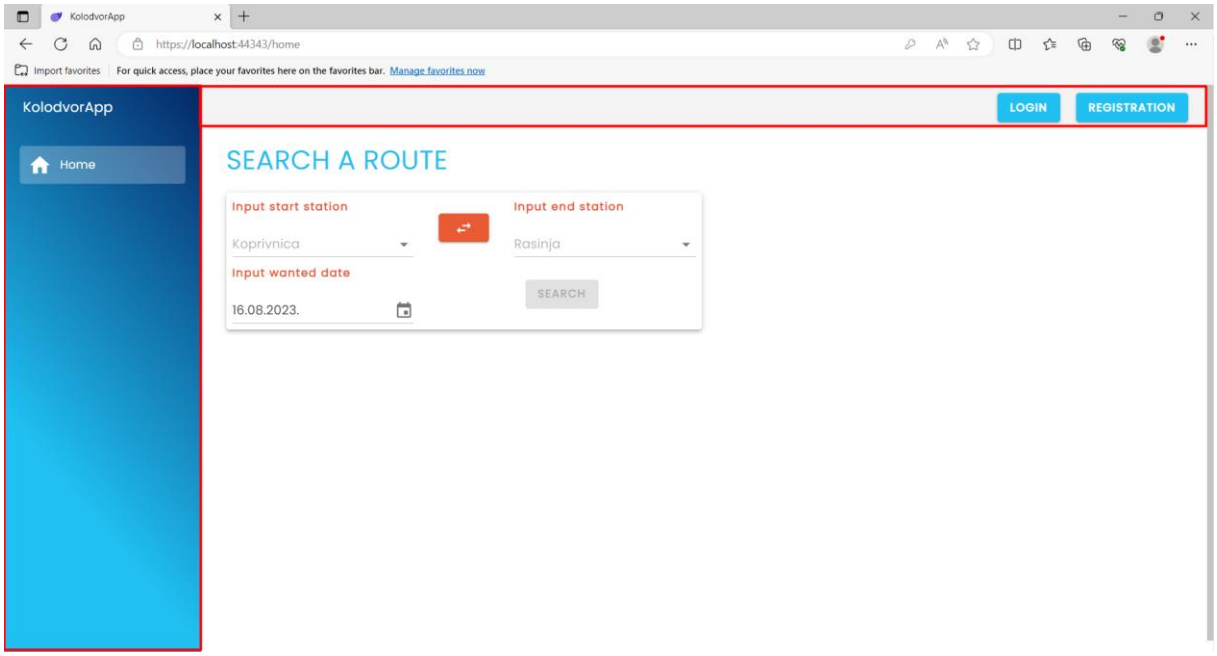


Slika 16. Naslovna stranica aplikacije

U okviru aplikacije, korisnicima se pruža minimalističko korisničko sučelje koje je vizualno osmišljeno s naglaskom na jednostavnost i funkcionalnost. Na početnoj stranici aplikacije, u izborniku se nalazi samo jedna opcija. Dodatno, korisnicima su dostupne opcije za prijavu i registraciju unutar sustava. Također, korisnicima je omogućeno pretraživanje dostupnih ruta za specifični datum putem odgovarajućeg pretraživačkog alata.

Na Slika 17. koja slijedi može se uočiti trodijelna struktura korisničkog sučelja web aplikacije. Prva sekcija predstavlja izborničku traku koja se proteže duž cijele visine stranice i smještena je na lijevom rubu. Sadržaj izborničke trake varira ovisno o razini prava pristupa korisnika. Stoga, za neregistrirane korisnike, izbornička traka prikazuje isključivo opciju za

pristup naslovnoj stranici. Druga sekcija obuhvaća naslovnu traku (u daljnjem tekstu zaglavlje) koja je pozicionirana na vrhu aplikacije i zauzima preostali prostor u širini koji nije zauzet izborničkom trakom. Unutar naslovne trake smješteni su gumbi za prijavu, registraciju i odjavu, kao i poruka dobrodošlice koja je vidljiva prijavljenim korisnicima. Treća sekcija predstavlja središnji dio stranice na kojem se prikazuje sadržaj pojedinih stranica u skladu s odabranom opcijom iz izbornika.



Slika 17. Prikaz strukture aplikacije

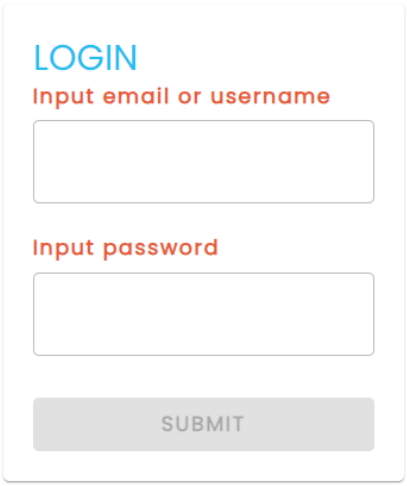
```
@inherits LayoutComponentBase

<div class="page">
    <CascadingAppState>
        <div class="sidebar">
            <NavMenu />
        </div>

        <main>
            <div class="top-row px-4">
                <Header />
            </div>

            <article class="content px-4">
                @Body
            </article>
        </main>
    </CascadingAppState>
</div>

<MudThemeProvider Theme=@currentTheme />
<MudDialogProvider />
<MudSnackbarProvider />
```

Prva linija, **`@inherits LayoutComponentBase`**, ukazuje da je ovaj kod dijelom hijerarhije komponenata u Blazor aplikaciji, specifično, da je ovo osnovni raspored koji će se koristiti u aplikaciji. Nakon toga, kod koristi **`<div class="page">`** kako bi definirao osnovnu strukturu stranice koja se sastoji od traka s izbornikom, zaglavlja i sadržaja.

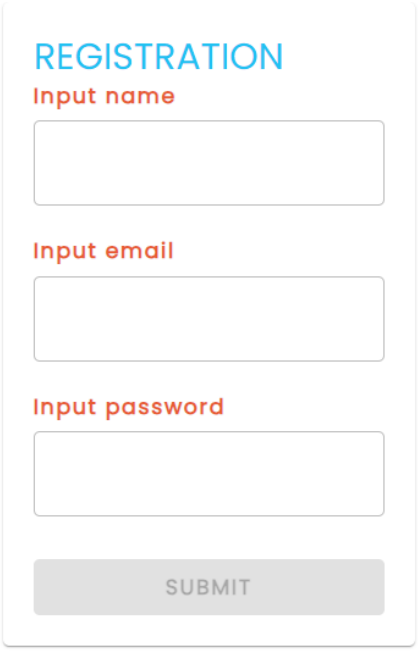
U **`<CascadingAppState>`** kontekstu, postavljena je temeljna struktura koja omogućuje komunikaciju i dijeljenje podataka između komponenata aplikacije. Unutar tog konteksta, **`<div class="sidebar">`** i **`<NavMenu />`** koriste se za definiranje i prikaz izbornika navigacije, dok **`<main>`** element sadrži zaglavlje i sadržaj stranice.

Unutar **`<main>`** elementa, **`<div class="top-row px-4">`** i **`<Header />`** koriste se za definiranje i prikaz zaglavlja stranice. **`<article class="content px-4">`** i **`@Body`** koriste se za prikaz sadržaja stranice. Ovo je mjesto gdje će sadržaj svake stranice unutar aplikacije biti učitan i prikazan.

Na kraju, **`<MudThemeProvider Theme=@currentTheme />`**, **`<MudDialogProvider />`**, i **`<MudSnackbarProvider />`** su dodani kako bi omogućili tematizaciju, dijaloške prozore i obavijesti unutar aplikacije koristeći MudBlazor biblioteku.



Slika 18. Prikaz obrasca za prijavu



Slika 19. Prikaz obrasca za registraciju

Procesi prijave i registracije u aplikaciji karakterizirani su sličnostima u pogledu vizualnog prikaza i interakcije s korisnicima. Očigledno je da su oba postupka dizajnirana na jednostavan i intuitivan način.

```
<CascadingAuthenticationState>
    <Router AppAssembly="@typeof(App).Assembly">
        <Found Context="routeData">
            <AuthorizeRouteView RouteData="@routeData"
                DefaultLayout="@typeof(MainLayout)">
                <NotAuthorized>
                    <PageTitle>You aren't allowed here</PageTitle>
                    <LayoutView>
                        <p role="alert">Sorry, you don't have access
                           rights for this address.</p>
                    </LayoutView>
                </NotAuthorized>
            </AuthorizeRouteView>
            <FocusOnNavigate RouteData="@routeData" Selector="h1" />
        </Found>
        <NotFound>
            <PageTitle>Not found</PageTitle>
            <LayoutView Layout="@typeof(MainLayout)">
                <p role="alert">Sorry, there's nothing at this
                     address.</p>
            </LayoutView>
        </NotFound>
    </Router>
</CascadingAuthenticationState>
```

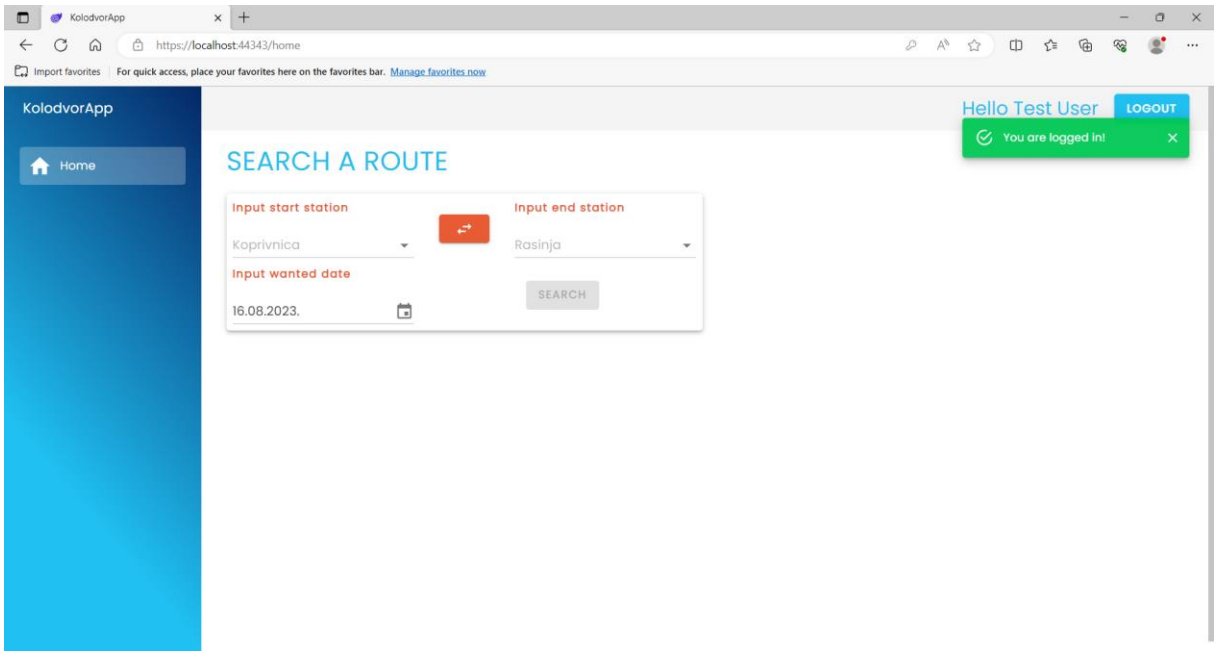
Koristeći **<CascadingAuthenticationState>** omotač, omogućeno je praćenje autentikacijskog stanja kroz cijelu aplikaciju što omogućuje komponentama unutar tog omotača da znaju je li korisnik autentificiran ili nije.

Koristeći **<Router>** komponentu postavljena je osnova za usmjeravanje unutar aplikacije prema različitim komponentama koje predstavljaju stranice aplikacije.

Kroz **<AuthorizeRouteView>** komponentu omogućeno je kontroliranje pristupa određenim putanjama (rutama) na temelju autentikacijskog stanja korisnika. Ako korisnik nije ovlašten za pristup određenoj ruti, ispisuje se poruka "Sorry, you don't have access rights for this address.".

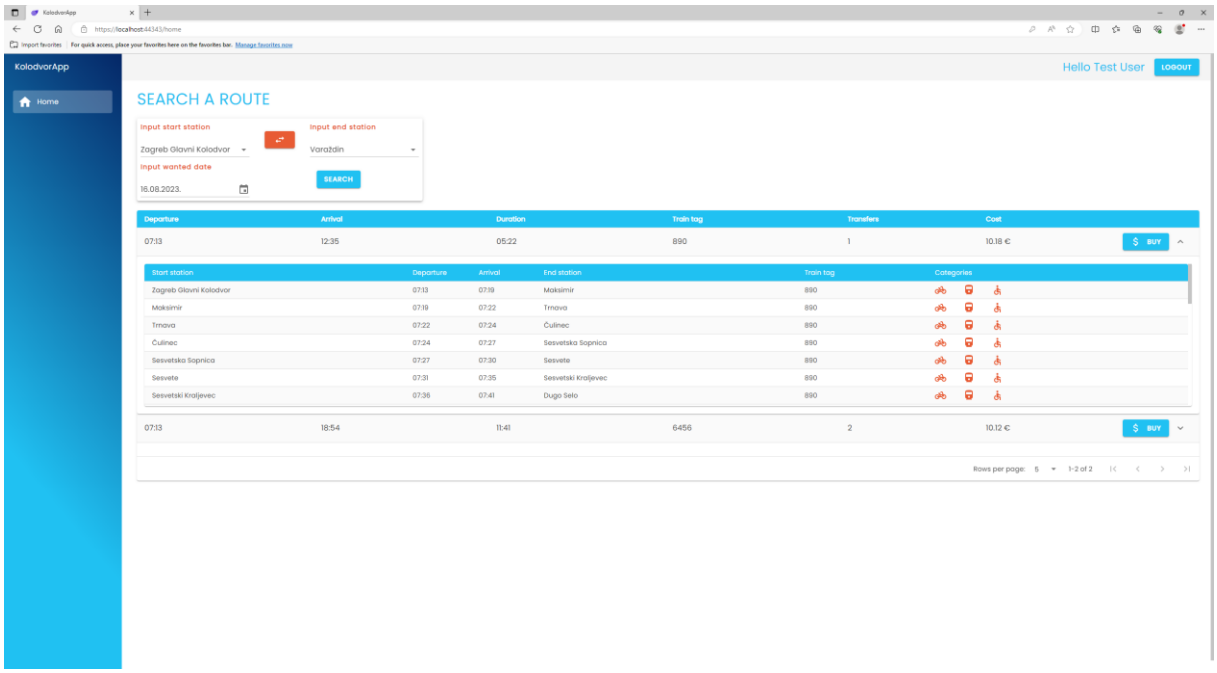
Korištenjem **<FocusOnNavigate>** komponente postignuto je automatsko fokusiranje na naslovne elemente (h1) nakon navigacije na novu stranicu, što pomaže u poboljšanju pristupačnosti aplikacije.

U slučaju nevažećeg usmjeravanja (kad korisnik unese nepostojeću rutu u adresnu traku), koristeći **<NotFound>** komponentu, korisniku se prikazuje poruka "Sorry, there's nothing at this address.", obavještavajući ga da tražena stranica ne postoji.



Slika 20. Prikaz uspješne prijave u aplikaciju

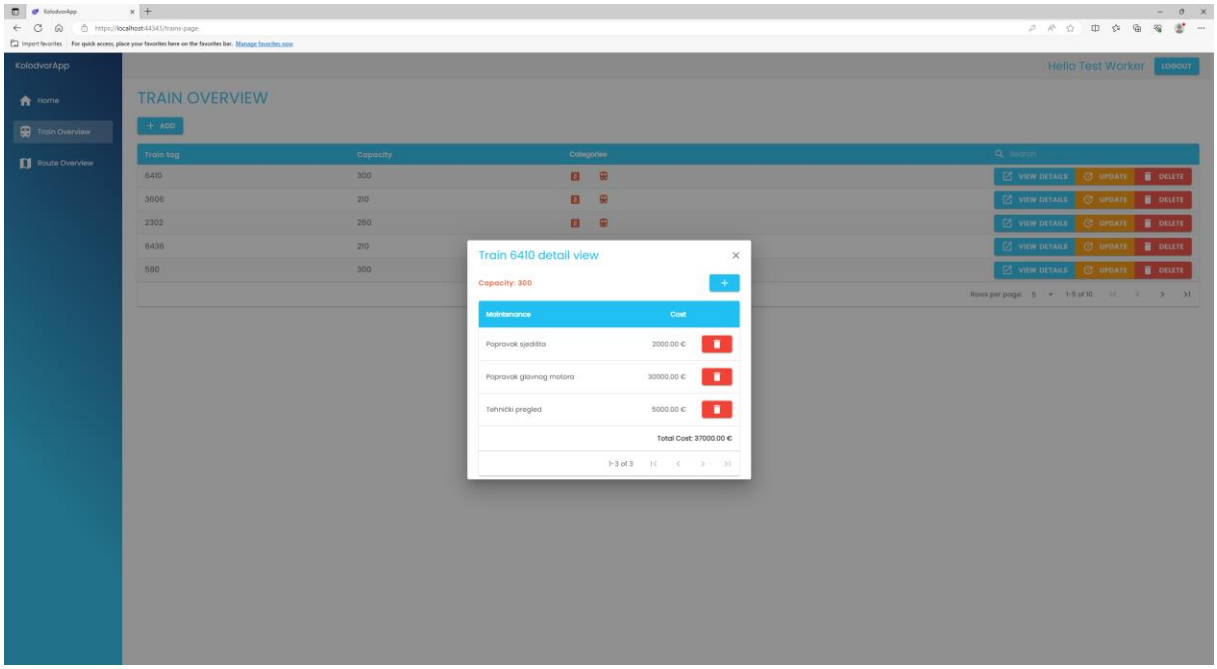
Na priloženoj Slika 20. može se uočiti promjena u prikazu naslovne trake u odnosu na identitet prijavljenog korisnika. Nadalje, sadržaj izbornika također je prilagodljiv u skladu s karakteristikama korisnika, iako u ovom konkretnom slučaju korisnik ne posjeduje posebne privilegije, stoga izbornik ostaje nepromijenjen.



Slika 21. Prikaz pregleda mogućih ruta na traženi datum, početnu i završnu stanicu

Na priloženoj Slika 21. ilustrirana je interakcija korisnika s aplikacijom tijekom pretraživanja dostupnih ruta te detaljnog pregleda jedne od ponuđenih opcija. Uz to, može se

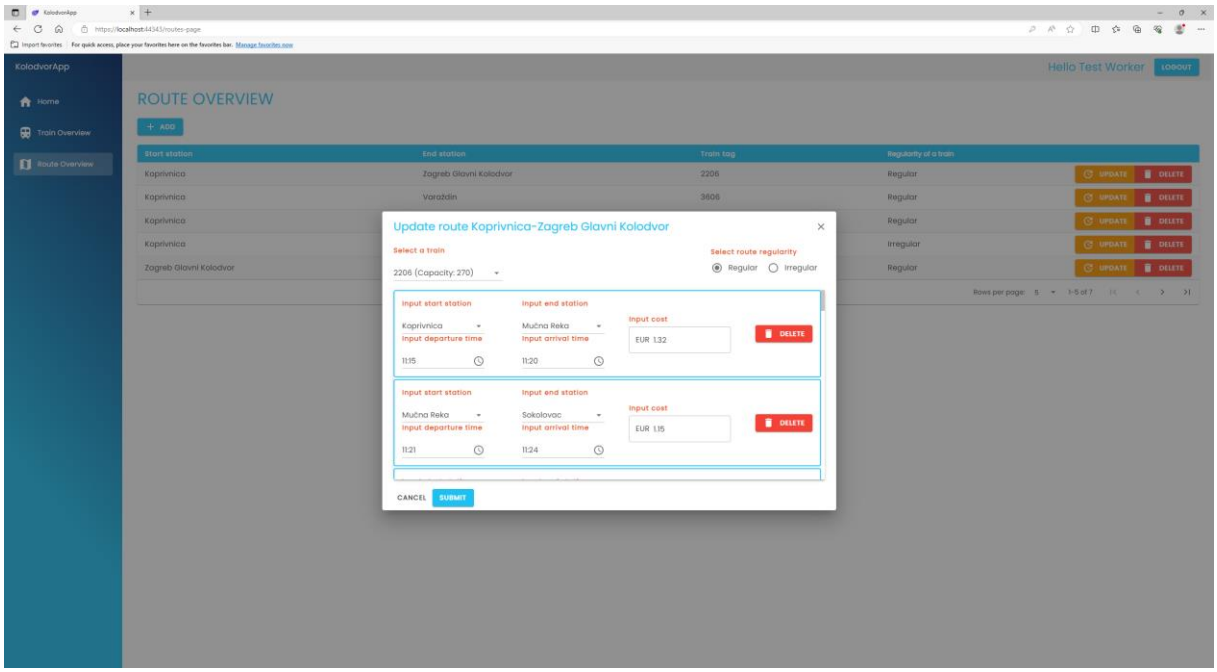
uočiti da korisnik, koji je prijavljen na platformu, ima ovlaštenje za iniciranje procesa kupnje karte za odabranu rutu što je vidljivo u desnom segmentu prikazane rute.



Slika 22. Prikaz stranice za pregled vlakova sa otvorenim dijalogom

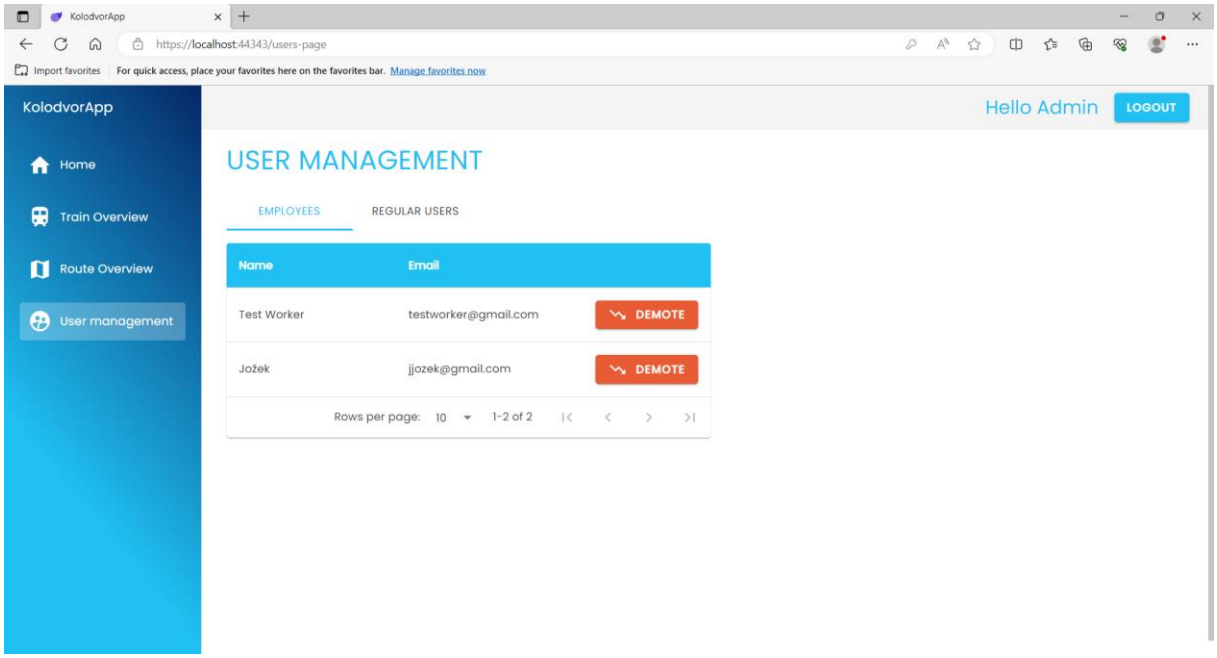
Na priloženoj Slika 22. možemo uočiti korisničko sučelje aplikacije na kojem su popisani svi vlakovi registrirani u sustavu. Istaknuti dijalog u središnjem dijelu slike otvara se kada korisnik odabere opciju za detaljan pregled određenog vlaka. U sklopu tog dijaloga korisnik ima mogućnost pregledavanja svih troškova i popravaka koje je bilo potrebno izvesti kako bi se vlak održavao u ispravnom i operativnom stanju za prijevoz putnika. Također, unutar dijaloga postoji funkcionalnost za dodavanje i brisanje zapisa o popravcima te je moguće vidjeti ukupne troškove vezane uz odabrani vlak.

Na stranici za pregled vlakova korisniku je omogućeno dodavanje novih vlakova ili ažuriranje podataka o postojećim vlakovima. To se postiže definiranjem šifre vlaka, njegovog kapaciteta i kategorija u koje spada dotični vlak. Naposljetku, korisnik ima opciju brisanja određenog vlaka iz sustava pod uvjetom da za taj vlak nisu evidentirani nikakvi troškovi kako bi se izbjegao gubitak važnih podataka.



Slika 23. Prikaz stranice za pregled ruta sa otvorenim dijalogom

Na priloženoj Slika 23. može se uočiti korisničko sučelje sustava koji prikazuje sve rute koje su unesene u sustav. Istaknuti dijalog u središnjem dijelu slike otvara se kada korisnik odluči ažurirati postojeću rutu ili kreirati novu. Unutar ovog dijaloga korisnik ima priliku pregledati sve stanice koje su uključene u odabranu rutu, kao i sve podatke povezane s vremenima dolaska i odlaska, kao i cijenama pojedinačnih putovanja od stanice do stanice. Nadalje, korisnik može unutar ovog dijaloga odabrati koji će vlak voziti na odabranoj ruti te odabrati hoće li ruta biti svakodnevna ili ne. Konačno, korisnik može obrisati odabranu rutu.



Slika 24. Prikaz administrativne stranice za upravljanje ulogama registriranih korisnika

Na posljednjoj Slika 24. prikazan je administrativni pogled na registrirane korisnike u kojem administrator ima ovlasti dodjeljivati ili oduzimati ulogu radnika. Radnici s ovom ulogom imaju ovlaštenje za upravljanje vlakovima i rutama.

Prva komponenta nazvana je "routes-page" i ona predstavlja "pametnu" komponentu. Ova komponenta sadrži veću količinu poslovne logike i ima direktan pristup servisima koji komuniciraju s poslužiteljem. Ona preuzima podatke s poslužitelja, obrađuje ih i prosljeđuje "glupoj" komponenti za prikaz. Ova komponenta također upravlja korisničkim interakcijama i reagira na različite događaje kao što su dodavanje nove rute, ažuriranje postojeće rute ili brisanje rute.

```
@page "/routes-page"

@attribute [Authorize]

@using KolodvorApp.Client.Features.Routes.Components
@using KolodvorApp.Client.HttpServices
@using KolodvorApp.Shared.DTOs

@inject IRouteService _routeService
@inject IStationService _stationService
@inject ITrainService _trainService
@inject IDialogService _dialogService
@inject ISnackbar _snackbar

<MudText Typo="Typo.h4" Style="color: var(--primary-
color)">@localizer["PAGE_TITLE_ROUTE_OVERVIEW"]</MudText>
<br />

<MudTooltip Text=@localizer["ADD_ROUTE"]>
    <MudButton Variant="Variant.Filled" Class="btn-primary"
     OnClick="AddNewRoute">
        <MudIcon Style="color: var(--white)"
           Icon="@Icons.Material.Filled.Add" />
        <span class="ml-2"
           style="white-space: nowrap">@localizer["ADD"]</span>
    </MudButton>
</MudTooltip>
<br />
<br />

<RoutesTable RouteList="_routeList" UpdateRoute="UpdateRoute"
     DeleteRoute="DeleteRoute" />

@code {

    #region General Summary

    private List<RouteDto> _routeList = new();

    private List<StationDto> _stationsList = new();
```

```
private List<TrainSelectorDto> _trainsList = new();

#endregion

protected override async Task OnInitializedAsync()
{
    await base.OnInitializedAsync();

    _routeList = await _routeService.GetAll();
    _stationsList = await _stationService.GetAll();
    _trainsList = await _trainService.GetAllTrainsForSelect();
}

private async Task AddNewRoute()
{
    var parameters = new DialogParameters();
    parameters.Add("TrainsList", _trainsList);
    parameters.Add("StationsList", _stationsList);

    var result = await RouteAddOrUpdateDialog(parameters);
    if (result is not null)
    {
        _routeList.Add(result);
        StateHasChanged();
    }
}

private async Task<RouteDto> RouteAddOrUpdateDialog(
 DialogParameters parameters)
{
    var options = new DialogOptions()
        {
            CloseButton = true,
            MaxWidth = MaxWidth.Medium,
            FullWidth = true
        };

    var dialog = _dialogService.Show<RouteCreateDialog>("",
       parameters, options);
    return (RouteDto)(await dialog.Result).Data;
}

private async Task UpdateRoute(RouteDto route)
{
    var parameters = new DialogParameters();
    parameters.Add("TrainsList", _trainsList);
    parameters.Add("StationsList", _stationsList);
    parameters.Add("Model", route);

    var result = await RouteAddOrUpdateDialog(parameters);
    if (result is not null)
    {
        _routeList.Remove(route);
        _routeList.Add(result);
        StateHasChanged();
    }
```

```
    }

    private async Task DeleteRoute(RouteDto route)
    {
        var parameters = new DialogParameters { ["Message"] =
           localizer["ARE_YOU_SURE"] + $""" "{route.StartStation}-
           {route.EndStation}"? """ };

        var options = new DialogOptions()
            {
                CloseButton = true,
                MaxWidth = MaxWidth.ExtraSmall,
                FullWidth = true
            };

        var dialog = _dialogService.Show<AreYouSureDialog>("",
           parameters, options);
        var result = (await dialog.Result).Data;

        if (result is null) return;

        var response = await
           _routeService.DeleteRoute(route.Id!.Value);

        if (response.IsSuccessStatusCode)
        {
            _snackbar.Add(localizer["ROUTE_SUCCESSFUL_DELETED"],
                Severity.Success);
            _routeList.Remove(route);
            StateHasChanged();
        }
        else _snackbar.Add(localizer["SERVER_PROBLEMS"],
                Severity.Warning);
    }
}
```

S druge strane, druga komponenta, koja je nazvana "RoutesTable", predstavlja "glupu" komponentu. Ova komponenta prikazuje tablicu s podacima o rutama koje su proslijeđene kao parametar. Ova komponenta ne obavlja nikakve poslove vezane uz preuzimanje podataka s poslužitelja, obradu podataka ili interakciju s korisnikom, osim što prikazuje podatke koje je primila i reagira na korisničke događaje kao što su klikovi na gumb. Njena glavna svrha je prikaz podataka u strukturiranom formatu.

```
@using KolodvorApp.Shared.DTOs

<MudTable Items="RouteList" FixedHeader="true" Striped="true"
     Hover="true" Dense="true" Virtualize="true"
     Breakpoint="Breakpoint.Sm">
    <ColGroup>
        <col style="width: 30%;" />
        <col style="width: 30%;" />
        <col style="width: 20%;" />
        <col style="width: 20%;" />
```

```
</ColGroup>

<HeaderContent>
    <MudTh Style="background-color: var(--primary-color);
       color: var(--white);">
        <MudTableSortLabel SortBy="new Func<RouteDto, object>(
            x =>
       x.StartStation!)">@localizer["TABLE_START_STATION"]</MudT
       ableSortLabel>
    </MudTh>
    <MudTh Style="background-color: var(--primary-color);
       color: var(--white);">
        <MudTableSortLabel SortBy="new Func<RouteDto, object>(
            x =>
       x.EndStation!)">@localizer["TABLE_END_STATION"]</MudTable
       SortLabel>
    </MudTh>
    <MudTh Style="background-color: var(--primary-color);
       color: var(--white);">
        <MudTableSortLabel SortBy="new Func<RouteDto, object>(
            x =>
       x.TrainTag!)">@localizer["TABLE_TRAIN_TAG"]</MudTableSort
       Label>
    </MudTh>
    <MudTh Style="background-color: var(--primary-color);
       color: var(--white); font-weight: 600;">
       @localizer["TABLE_DAILY_ROUTE"]</MudTh>
    <MudTh Style="background-color: var(--primary-color)" />
</HeaderContent>

<RowTemplate>
    <MudTd Style="font-size: 16px">@context.StartStation</MudTd>
    <MudTd Style="font-size: 16px">@context.EndStation</MudTd>
    <MudTd Style="font-size: 16px">@context.TrainTag</MudTd>
    <MudTd Style="font-size: 16px">@(context.IsDaily ?
       localizer["REGULAR"] : localizer["IRREGULAR"])</MudTd>
        <MudTd Class="mr-3" Style="text-align: right">
            <MudButtonGroup Variant="Variant.Filled"
                 OverrideStyles="false">
                <MudTooltip
                       Text=@localizer["OPEN_ROUTE_DETAILS"]>
                    <MudButton Variant="Variant.Filled"
                       Class="btn-warning" OnClick="@((e) =>
                       OnUpdateRoute(context))">
                        <MudIcon Style="color: var(--white)"
                            Icon="@Icons.Material.Filled.Update"
                            />
                        <span class="ml-3" style="white-space:
                            nowrap">@localizer["UPDATE"]</span>
                    </MudButton>
                </MudTooltip>
                <MudTooltip Text=@localizer["DELETE_BUTTON"]>
                    <MudButton Variant="Variant.Filled"
                       Class="btn-error" OnClick="@((e) =>
                       OnDeleteRoute(context))">
                        <MudIcon Style="color: var(--white)"
```

```
                                    Icon="@Icons.Material.Filled.Delete"
                                    />
                                <span class="ml-3" style="white-space:
                                    nowrap">@localizer["DELETE"]</span>
                            </MudButton>
                        </MudTooltip>
                    </MudButtonGroup>
                </MudTd>
            </RowTemplate>
            <PagerContent>
                <MudTablePager PageSizeOptions="new int[] { 5, 10 }" />
            </PagerContent>
        </MudTable>

        @code {
        [Parameter, EditorRequired]
        public List<RouteDto> RouteList { get; init; } = new();

        [Parameter, EditorRequired]
        public EventCallback<RouteDto> UpdateRoute { get; init; }

        [Parameter, EditorRequired]
        public EventCallback<RouteDto> DeleteRoute { get; init; }

        private async Task OnUpdateRoute(RouteDto route)
            => await UpdateRoute.InvokeAsync(route);

        private async Task OnDeleteRoute(RouteDto route)
            => await DeleteRoute.InvokeAsync(route);
}
```

Ključna razlika između ove dvije vrste komponenata leži u njihovim odgovornostima i složenosti. "Glupe" komponente su jednostavne i imaju ograničene odgovornosti, uglavnom se fokusirajući na prezentaciju podataka. S druge strane, "pametne" komponente obavljaju složenije zadatke i upravljaju interakcijama s korisnicima, kao i komunikacijom s poslužiteljem.

9. Zaključak

U području modernog web razvoja, izrada web aplikacija postala je sve složeniji zadatak karakteriziran složenim detaljima i opsežnim mogućnostima za prilagodbu. U svjetlu toga, ovaj rad je u početku pokušao razjasniti razlike između web stranica i web aplikacija. Nakon toga, ovaj rad je nastojao objasniti osnovnu arhitekturu koja se koristi u suvremenim malim i jednostavnim aplikacijama koje zbog svoje nekomplicirane poslovne domene ne zahtijevaju upotrebu složenijih i robusnijih arhitektura.

U ovom radu naglašava se utjecaj odluka o arhitekturi na dizajn sustava, ističući kako dobro promišljene odluke mogu značajno pridonijeti razvoju sustava. Često je najučinkovitije započeti razvoj s višeslojnom arhitekturom u kontekstu čiste arhitekture jer omogućuje jednostavno razdvajanje i ne ograničava sustav na nedjeljivu cjelinu, omogućavajući napredak bez potpune realizacije arhitekture. Ova fleksibilnost omogućuje sustavu prilagodbu novim ili izvanrednim zahtjevima i pruža vremena za konačne odluke, definiranje smjera kretanja arhitekture i završetak razvoja sustava. Također, važno je napomenuti da ovaj pristup otvara vrata naprednim arhitekturama kao što su popratno vozilo, mikroservisi i ostalo.

Usvajanje višeslojne arhitekture, u kontekstu čiste arhitekture, može donijeti nekoliko ključnih prednosti razvoju sustava. Jedna od glavnih prednosti je mogućnost neovisnog razvoja komponenata. Svaki sloj se može izgraditi, testirati i usavršiti bez potrebe za istodobnim izmjenama u drugim slojevima. Ovaj modularni pristup ne samo da povećava agilnost i brzinu razvojnog procesa već i povećava robusnost sustava osiguravajući da promjene u jednom sloju ne utječu negativno na druge slojeve.

Osim toga, ova arhitektura omogućuje programerima da uspostave jasno razdvajanje odgovornosti što olakšava dodjelu specifičnih odgovornosti različitim slojevima. Ovo razdvajanje pojednostavljuje proces održavanja i otklanjanja grešaka jer programeri mogu brzo identificirati izvor bilo kojih problema i riješiti ih bez utjecaja na nevezane dijelove sustava. Također omogućuje fokusiraniju i učinkovitiju suradnju tima, jer svaki tim može raditi na određenom sloju bez ometanja rada drugih timova.

Osim toga, fleksibilnost koju nudi ova arhitektura olakšava prilagodbu promjenama u poslovnim zahtjevima ili uključivanje novih tehnologija. Ukoliko je potrebna nova značajka ili funkcionalnost, može se dodati u odgovarajući sloj bez potrebe za preuređenjem cijelog sustava. Slično tome, ako određena tehnologija postane zastarjela ili se identificira učinkovitije rješenje, može se zamijeniti u relevantnom sloju bez utjecaja na ostatak sustava.

Višeslojna arhitektura pruža čvrstu osnovu za skaliranje sustava, bilo horizontalno dodavanjem više resursa ili vertikalno poboljšavanjem sposobnosti postojećih resursa. Ova skalabilnost osigurava da sustav može podnijeti povećanu potražnju i nastaviti optimalno raditi.

Pri dizajniranju web aplikacije, primjena principa „pametnih“/“glupih“ komponenata i pozadinskog generiranja korisničkog sučelja (BFF) može značajno poboljšati fleksibilnost, održivost i performanse aplikacije. „Pametne“ komponente, poznate i kao spremnik komponenata, odgovorne su za rukovanje logikom podataka i komunikaciju sa servisima, dok se „glupe“ komponente ili prezentacijske komponente fokusiraju isključivo na vizualnu prezentaciju korisničkog sučelja. Ovo razdvajanje odgovornosti omogućuje programerima da stvaraju modularniji i ponovno upotrebljiv kod, jer se „glupe“ komponente mogu jednostavno ponovno koristiti u različitim dijelovima aplikacije bez vezanja za specifične podatke ili logiku. S druge strane, BFF služi kao posrednički sloj između prednjeg i pozadinskog dijela sustava, prilagođen specifičnim potrebama prednjeg dijela sustava. Centralizacijom logike dohvaćanja podataka u BFF-u, prednji dio sustava može učinkovitije preuzimati i koristiti podatke, što dovodi do glatkijeg korisničkog iskustva. Osim toga, BFF može apstrahirati složenosti pozadinskog dijela sustava, omogućujući programerima prednjeg dijela sustava da se usredotoče na stvaranje responzivnijeg i intuitivnijeg korisničkog sučelja. Integracijom ovih principa u arhitekturu web aplikacije, programeri mogu postići organiziraniju, skalabilniju i učinkovitiju aplikaciju koja udovoljava evoluirajućim potrebama korisnika.

Kako bi se potvrdile navedene arhitektonske odluke i principi, razvijena je i implementirana sveobuhvatna web aplikacija. Ova aplikacija služi kao praktična inkarnacija opisanih koncepta, uključujući „pametne“/“glupe“ komponente i pristup pozadinskog generiranja korisničkog sučelja (BFF). Posebna pažnja posvećena je implementaciji arhitektonskih slojeva i komponenata, osiguravajući da su odabrane dizajnerske odluke vjerno primijenjene. Važno je napomenuti da je programski kod ove aplikacije strukturiran s naglaskom na čitljivost, modularnost i pridržavanje najboljih praksi. To, ne samo da olakšava razumijevanje implementiranih koncepta, već i poboljšava održivost koda. Pregledom izvornog koda čitatelji mogu steći opipljivo razumijevanje kako se primjenjuju raspravljane arhitektonske odluke i principi u stvarnom kontekstu te kako doprinose organiziranijoj i koherentnijoj softverskoj arhitekturi.

Literatura

[1] L. Shklar i R. Rosen, „Web Application Architecture: Principles, Protocols and Practices“, sv. 2, izd. 1, str. 620, 2009.

[2] „Web Page vs Website A Definition“, *WiX*. https://www.wix.com/encyclopedia/definition/webpage?utm_source=google&utm_medium=cpc&utm_campaign=13708482660^124757113432&experiment_id=^^530755701272^^_DSA&gclid=CjwKCAiA0JKfBhBIEiwAPhZXD8tTFUSThn8cqT-mHhcVcLFtcdnjtZrlkLPzq1yML0f7S9Q1owNBBBoCWi4QAvD_BwE (pristupljeno 09. veljača 2023.).

[3] „What is a Web Page? - Definition from Techopedia“, *Technopedia*, 12. lipanj 2012. https://www.techopedia.com/definition/4774/web-page-page (pristupljeno 09. veljača 2023.).

[4] „Web Pages“, *tutorialspoint.com*. https://www.tutorialspoint.com/internet_technologies/web_pages.htm (pristupljeno 09. veljača 2023.).

[5] TechTarger Contributor, „What is Web Application (Web Apps) and its Benefits“, *TechTarget*, siječanj 2023. https://www.techtarget.com/searchsoftwarequality/definition/Web-application-Web-app (pristupljeno 09. veljača 2023.).

[6] „Gimnazija ‚Fran Galović‘ - Naslovnica“, *Gimnazija „Fran Galović“ Koprivnica*. http://www.gimnazija-fgalovic-koprivnica.skole.hr/?loginfailed=1 (pristupljeno 09. veljača 2023.).

[7] M. Matthew, „Difference between Website and Web Application (Web App)“, *Guru99*, 28. siječanj 2023. https://www.guru99.com/difference-web-application-website.html (pristupljeno 09. veljača 2023.).

[8] I. Wigmore, „Single-page application (SPA)“, *TechTarget*, prosinac 2016. https://www.techtarget.com/whatis/definition/single-page-application-SPA (pristupljeno 10. veljača 2023.).

[9] E. A. Scott, F. O. By, i B. Holland, *SPA Design and Architecture*. Shelter Island: Manning Publications Co., 2016.

[10] R. P. Nguyen, „Single-Page Web Applications: Navigation and User Experience“, *User Experience*, sv. 11, str. 45–55, 2012.

[11] Y. F. Wang, „Single-Page Web Applications: Security and Performance“, *Security and Performance Journal*, sv. 7, str. 33–42, 2013.

[12] A. K. Gupta, „Single-Page Web Applications: Limitations and Future Directions“, *Future Directions in Web Development*, sv. 4, str. 11–21, 2014.

[13] M. B. Patel, „Progressive Web Applications: An Overview“, *Journal of Web Development*, sv. 5, str. 12–24, 2015.

[14] R. K. Singh, „Progressive Web Applications: Features and Benefits“, *Web Development Journal*, sv. 7, str. 45–54, 2016.

[15] A. K. Sharma, „Progressive Web Applications: The Future of Web Development“, *Future Directions in Web Development*, sv. 9, str. 33–41, 2019.

[16] R. Ćosić, „The principles of building Single-Page Applications (SPAs)“, *LinkedIn*, stu. 2022, Pristupljeno: 10. veljača 2023. [Na internetu]. Dostupno na: https://www.linkedin.com/pulse/principles-building-single-page-applications-spas-ratko-%25C4%2587osi%25C4%2587/?trackingId=Pzzx1QBkRqeW62ZT09Tpmw%3D%3D

[17] N. T. Nguyen, „Progressive Web Applications: Limitations and Challenges“, *Limitations and Challenges Journal*, sv. 14, str. 45–55, 2022.
[18] Y. F. Chen, „Progressive Web Applications: Performance and Security“, *Performance and Security Journal*, sv. 13, str. 11–21, 2021.
[19] T. Brucks, „Multi-Page Web Applications vs. Single-Page Web Applications: A Comparison“, *Journal of Web Engineering*, sv. 7, str. 201–212, 2008.
[20] S. Smith, *Architect Modern Web Applications with ASP.NET Core and Azure*, 7. izd. Microsoft Developer Division, 2022. [Na internetu]. Dostupno na: https://www.microsoft.com
[21] J. Doe, „An Overview of N-Layered Architecture“, *Journal of Software Engineering*, sv. 5, izd. 2, str. 89–101, 2020.
[22] R. Overview, *Software Architecture Patterns*, sv. 32, izd. 5 Suppl. 2014. [Na internetu]. Dostupno na: http://www.ncbi.nlm.nih.gov/pubmed/24798474
[23] Altvater Alexandra, „What is N-Tier Architecture? Examples, Tutorials & More“, *Stackify*, 19. svibanj 2017. https://stackify.com/n-tier-architecture/ (pristupljeno 09. ožujak 2023.).
[24] „N-tier architecture style - Azure Architecture Center | Microsoft Learn“, *Microsoft*. https://learn.microsoft.com/en-us/azure/architecture/guide/architecture-styles/n-tier (pristupljeno 09. ožujak 2023.).
[25] A. Patel, „Clean Architecture with .NET and .NET Core — Overview | by Ashish Patel | .NET Hub | Medium“, 02. rujan 2021. https://medium.com/dotnet-hub/clean-architecture-with-dotnet-and-dotnet-core-aspnetcore-overview-introduction-getting-started-ec922e53bb97 (pristupljeno 12. ožujak 2023.).
[26] S. Mudassar Ali Khan, „Clean Architecture In ASP.NET Core Web API“, *C# Corner*, 23. srpanj 2022. https://www.c-sharpcorner.com/article/clean-architecture-in-asp-net-core-web-api/ (pristupljeno 12. ožujak 2023.).
[27] Robert C. Martin, *Clean Architecture*, sv. 1. Pearson Education, Inc., 2018. [Na internetu]. Dostupno na: www.EBooksWorld.ir
[28] M. Noback, *Clean Architecture: Patterns, Practices, and Principles*, 3. izd., sv. 35. IEEE Software, 2018.
[29] R. K. Gupta, M. Venkatachalapathy, i F. K. Jeberla, „Challenges in Adopting Continuous Delivery and DevOps in a Globally Distributed Product Team: A Case Study of a Healthcare Organization“, u *2019 ACM/IEEE 14th International Conference on Global Software Engineering (ICGSE)*, IEEE, svi. 2019, str. 30–34. doi: 10.1109/ICGSE.2019.00020.
[30] E. Evans, *Domain-Driven Design: Tackling Complexity in the Heart of Software*. Boston: Addison-Wesley Professional, 2003.
[31] D. Laribee, „Best Practice - An Introduction To Domain-Driven Design | Microsoft Learn“, *Microsoft Docs*, veljača 2009. https://learn.microsoft.com/en-us/archive/msdn-magazine/2009/february/best-practice-an-introduction-to-domain-driven-design (pristupljeno 22. ožujak 2023.).
[32] „DDD(Domain Driven Design) - Incheol’s TECH BLOG“. https://incheol-jung.gitbook.io/docs/q-and-a/architecture/ddd (pristupljeno 22. ožujak 2023.).
[33] V. Vernon, „Effective Aggregate Design“, u *Domain-Driven Design Distilled*, Addison-Wesley Professional, 2016, str. 95–115.
[34] S. Miteva, „The Concept of Domain-Driven Design Explained“, *Microtica*, 03. srpanj 2020. https://medium.com/microtica/the-concept-of-domain-driven-design-explained-3184c0fd7c3f (pristupljeno 22. ožujak 2023.).

[35] „Designing the infrastructure persistence layer“, *Microsoft Learn*, 21. veljača 2023. https://learn.microsoft.com/en-us/dotnet/architecture/microservices/microservice-ddd-cqrs-patterns/infrastructure-persistence-layer-design (pristupljeno 24. ožujak 2023.).
[36] M. Fowler, *Patterns of Enterprise Application Architecture*, 1st izd. Addison-Wesley Professional, 2002.
[37] S. Chandra i P. Misra, „Components: A Comparative Analysis“, *Int J Comput Appl*, sv. 174, izd. 15, str. 1–4, 2021.
[38] J. Arnold, „Dumb Components and Smart Components“, *Medium*, 25. veljača 2017. https://medium.com/@thejasonfile/dumb-components-and-smart-components-e7b33a698d43 (pristupljeno 25. ožujak 2023.).
[39] S. Sengupta, S. Roy, i S. Saha, „A Comprehensive Study on the Characteristics and Utilization of Smart and Dumb Components“, *International Journal of Advanced Computer Science and Applications*, sv. 10, izd. 2, str. 110–115, 2019.
[40] J. Doe, „Smart vs. Dumb Components: A Comparison“, *Journal of Software Engineering*, sv. 10, izd. 2, str. 50–55, 2022.
[41] „Introduction to the DOM - Web APIs“, *MDN*, 22. veljača 2023. https://developer.mozilla.org/en-US/docs/Web/API/Document_Object_Model/Introduction (pristupljeno 02. travanj 2023.).
[42] N. Salas, „Backend for Frontend as a Microservices Architecture Pattern“, u *Proceedings of the 4th International Conference on Internet of Things, Big Data and Security*, 2019, str. 1–6.
[43] N. Ding, H. Ma, C. Zhao, Y. Ma, i H. Ge, „Driver’s Emotional State-Based Data Anomaly Detection for Vehicular Ad Hoc Networks“, u *2019 IEEE International Conference on Smart Internet of Things (SmartIoT)*, IEEE, kol. 2019, str. 121–126. doi: 10.1109/SmartIoT.2019.00027.
[44] V. Wickramarachchi, „The BFF Pattern (Backend for Frontend): An Introduction“, *Bits and Pieces*, 23. veljača 2021. https://blog.bitsrc.io/bff-pattern-backend-for-frontend-an-introduction-e4fa965128bf (pristupljeno 30. ožujak 2023.).
[45] M. Esposito i D. Rettig, „Introducing ASP.NET“, u *Programming Microsoft ASP.NET Core*, Redmond: Microsoft Press, 2018, str. 1–30.
[46] A. Freeman, „The Evolution of ASP.NET“, u *Pro ASP.NET Core MVC 2*, New York: Apress, 2017, str. 1–20.
[47] S. Walther, K. Scott, i D. Sussman, „Caching“, u *Professional ASP.NET 4.5 in C# and VB*, Hoboken: John Wiley & Sons, 2013, str. 523–568.
[48] D. Esposito, „ASP.NET Core Runtime“, u *Programming Microsoft ASP.NET Core*, Remond: Microsoft Press, 2018, str. 31–52.
[49] G. Bradski i A. Kaehler, „Using Libraries“, u *Learning OpenCV*, Sebastopol: O’Reilly Media, 2008, str. 7–24.
[50] J. Galloway, D. R. Kuhn, i J. C. Novak, „ASP.NET Core: Cross-Platform Web Development“, u *Mastering ASP.NET Core 2.0*, Birmingham: Packt Publishing, 2017, str. 1–30.
[51] A. Holovaty i J. Kaplan-Moss, „Getting Started with Django“, u *The Django Book*, Berkeley: Apress, 2009, str. 1–10.
[52] D. H. Hansson, „Ruby on Rails: Up and Running“, u *Rails: Up and Running*, Sebastopol: O’Reilly Media, 2006, str. 1–12.

[53] S. Roth, „Blazor: A New .NET Web Framework for Single-Page Applications“, *MSDN Magazine*, str. 54–61, listopad 2018.
[54] D. Roth, „Get started building .NET web apps with Blazor“, u *Microsoft Developer Blogs 2019*, 2019.
[55] D. Fowler i S. Sanderson, „ASP.NET Core SignalR: Build Real-Time Web Apps with Blazor WebAssembly and SignalR“, *MSDN Magazine*, str. 22–28, 2019.
[56] Angular Team, „Angular: One framework. Mobile & desktop“, *Angular.io*. https://angular.io/ (pristupljeno 06. travanj 2023.).
[57] „TypeScript: JavaScript that scales“, *TypeScriptLang.org*. https://www.typescriptlang.org/ (pristupljeno 06. travanj 2023.).
[58] Inc. Facebook, „React: A JavaScript library for building user interfaces“, *ReactJs*. https://react.dev/ (pristupljeno 06. travanj 2023.).
[59] E. You, „Vue.js: The Progressive JavaScript Framework“, *Vue.js.org*. https://vuejs.org/ (pristupljeno 06. travanj 2023.).
[60] „Overview of Entity Framework Core - EF Core | Microsoft Learn“, *Microsoft Learn*, 25. svibanj 2021. https://learn.microsoft.com/en-us/ef/core/ (pristupljeno 29. srpanj 2023.).
[61] V. Vernon, “Implementing Domain-Driven Design”, 1. izd. Boston, MA, SAD: Addison-Wesley, 2013.

Popis slika

Popis tablica

Prilozi

Link za GitHub repozitorij: https://github.com/Beertija/Diplomski_TOgrinec