Razvoj web aplikacije korištenjem MEVN paketa tehnologija

Milažar, Dario

SVEUČILIŠTE U ZAGREBU
FAKULTET ORGANIZACIJE I INFORMATIKE
V A R A Ž D I N

Dario Milažar

RAZVOJ WEB APLIKACIJE KORIŠTENJEM MEVN PAKETA TEHNOLOGIJA

ZAVRŠNI RAD

Varaždin, 2023.

SVEUČILIŠTE U ZAGREBU

FAKULTET ORGANIZACIJE I INFORMATIKE

V A R A Ž D I N

Dario Milažar

Matični broj: 0016124116

Studij: Primjena informacijske tehnologije u poslovanju

RAZVOJ WEB APLIKACIJE KORIŠTENJEM MEVN PAKETA TEHNOLOGIJA

ZAVRŠNI RAD

Mentor:

doc. dr. sc. Matija Novak

Varaždin, rujan 2023.

Dario Milažar

Izjava o izvornosti

Izjavljujem da je moj završni/dimplomski 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

Korištenje web aplikacija postalo je ključno za suvremeno poslovanje i komunikaciju s rastom interneta. Web aplikacije omogućuju tvrtkama dostavu usluga putem interneta, proširujući njihovu publiku i poslovanje. Pri izboru tehnologije za izgradnju web aplikacija važno je prilagoditi ih potrebama tvrtke. Postoji mnogo tehnologija za web razvoj, uključujući HTML, CSS i JavaScript, koji pružaju strukturu, stiliziranje i interaktivnost. Također, poslovna logika na strani poslužitelja je ključna za brzo i učinkovito upravljanje podacima i zahtjevima korisnika. Najpopularniji programski jezik za web razvoj je JavaScript, koji omogućuje dinamičke web aplikacije. Node.js proširuje njegovu primjenu, omogućujući izvršavanje JavaScript koda na strani poslužitelja. To pojednostavljuje razvoj jer se koristi isti jezik i pruža kohezivno iskustvo.

Ključne riječi: HTML, CSS, JavaScript, web aplikacije, Node.js

Sadržaj

1. Uvod

Korištenje aplikacija putem interneta posljednjih godina postalo je ključni aspekt modernog poslovanja i komunikacije. Web aplikacije nude fleksibilan i ekonomičan način isporuke informacija i usluga korisnicima putem Interneta, omogućujući tvrtkama da dosegnu širu publiku i usmjere svoje poslovanje. S eksponencijalnim rastom interneta, porasla je i potražnja za web aplikacijama, kao i razvoj tehnologija koje se koriste za izgradnju istih.

Kod odabira tehnologije koja će se koristiti za izgradnju aplikacije treba uzeti u obzir da aplikacija bude dizajnirana tako da odražava i oponaša poslovnu logiku tvrtke kojoj služi, što znači da bi aplikacija trebala biti prilagođena kako bi zadovoljila jedinstvene potrebe i tijek rada tvrtke, jer korištenje generičkog, tj. istog pristupa prilikom izgradnje aplikacija, ne odgovara svakoj organizaciji i njihovim zahtjevima.

U današnje vrijeme postoji mnogo tehnologija i okvira koje se koriste za izgradnju web aplikacija. HTML (HyperText Markup Language), CSS (Cascading Style Sheets) i JavaScript su osnovni gradivni blokovi web razvoja. HTML osigurava strukturu web stranica, CSS se koristi za njihovo stiliziranje, JavaScript za interaktivne funkcionalnosti, a u novije vrijeme i za izgradnju logike na strani poslužitelja i povezivanje s bazom podataka. Poslovna logika na strani poslužitelja treba biti u stanju nositi se s visokim razinama prometa i korisničkih zahtjeva, te biti u mogućnosti pohraniti i dohvatiti podatke brzo i učinkovito. Popularne tehnologije odnosno programski jezici koji se mogu koristiti za ispunjavanja navedenih zahtjeva su JavaScript, PHP, Python, Ruby, C#, itd.

Najkorišteniji programski jezik koji se koristi za razvoj web aplikacija je JavaScript. JavaScript je skriptni jezik na strani klijenta koji se izvršava u web pregledniku korisnika, omogućujući interaktivne i dinamičke funkcionalnosti web aplikacije. Pojava tehnologije koja pokreće i omogućava izvršavanje JavaScript koda izvan web preglednika i na strani poslužitelja zove se Node.js. Node.js je uvelike proširio raspon aplikacija koje se mogu izraditi pomoću JavaScripta. Programerima je omogućena izgradnja web aplikacija korištenjem jednog programskog jezika što pruža kohezivnije i jednostavnije razvojno iskustvo.

2. Razvoj web aplikacija

Razvoj web aplikacija doživio je prekretnicu posljednjih godina, pri čemu tehnologija igra ključnu ulogu u oblikovanju digitalnog krajolika. Danas programeri imaju velik izbor tehnologija na raspolaganju koje im omogućuju stvaranje dinamičkih, skalabilnih i značajkama bogatih web aplikacija koje zadovoljavaju raznolike korisničke potrebe. U sljedećim poglavljima detaljnije će biti opisane tehnologije i najkorišteniji okviri u web aplikacijama.

2.1. Tehnologije u web aplikacijama

Razvoj web aplikacija oslanja se na korištenje raznolikih tehnologija koje omogućuju programerima izgradnju robustnih i učinkovitih aplikacija, obuhvaćaju programske jezike, okvire, knjižnice i alate koji zajedno stvaraju dinamička i interaktivna web iskustva. Neke od ključnih tehnologija koje se koriste u razvoju web aplikacija uključuju:

1. HTML (Hypertext Markup Language): Osnova web razvoja, HTML pruža strukturu i sadržaj web stranica. Koristi se za označavanje elemenata na web stranici kako bi se definirao njihov sadržaj i organizacija, omogućavajući pregledniku da pravilno prikaže informacije. Najnovija verzija, HTML5, donosi brojne nove elemente i atribute koji podržavaju modernu web tehnologiju. Uz ugrađenu podršku za video, audio, grafičke elemente i bolje semantičke oznake.

2. CSS (Cascading Style Sheets): CSS kontrolira prikaz i izgled web stranica, omogućavajući programerima prilagodbu izgleda elemenata. Pomoću CSS-a možete definirati boje, fontove, pozicije i druge stilističke osobine elemenata kako bi se postigao željeni vizualni izgled stranice. CSS3, posljednja verzija CSS-a, donosi napredne stilističke mogućnosti kao što su gradijenti, sjene, animacije i prijelazi. Osim toga, postoji i praksa korištenja preprocesora poput SASS i LESS, koji dodatno proširuju mogućnosti stilizacije.

3. JavaScript: Univerzalni programski jezik koji omogućuje interaktivne i responzivne značajke, poput korisničkih interakcija, manipulacije podacima i ažuriranja u stvarnom

vremenu. JavaScript se izvršava u pregledniku korisnika i omogućava dinamično mijenjanje sadržaja i ponašanja web stranica. JavaScript je izuzetno popularan jezik zbog svoje svestranosti i široke primjene. Osim toga, popularni frontend okviri i biblioteke poput Reacta, Angulara i Vue.js dodano pridonose rastu i popularnosti JavaScripta.

4. Tehnologije poslužiteljske strane: Za funkcionalnost na poslužiteljskoj strani, web programeri koriste programske jezike poput Pythona, Rubyja, Java-e ili Node.js, zajedno s okvirima poput Django-a, Ruby on Rails-a, Spring-a ili Express.js-a. Ove tehnologije omogućavaju izradu dinamičnih web aplikacija koje mogu obrađivati zahtjeve, pristupati bazama podataka i upravljati poslovnom logikom.

5. Okviri za korisničko sučelje: Popularni frontend okviri poput: Reacta, Angulara i Vue.js olakšavaju razvoj složenih korisničkih sučelja, nudeći ponovno upotrebljive komponente i učinkovito upravljanje stanjem. Oni omogućavaju brz i efikasan razvoj dinamičnih aplikacija koje reagiraju na korisničke akcije.

6. Tehnologije baza podataka: Web aplikacije često zahtijevaju pohranu podataka, a najpopularnije tehnologije baza podataka uključuju MySQL, PostgreSQL, MongoDB i Firebase. MySQL i PostgreSQL su relacijske baze podataka koje pružaju strukturiranu pohranu podataka, dok MongoDB nudi fleksibilnu i skalabilnu NoSQL bazu podataka. Firebase je platforma bazirana na oblaku (*eng. cloud*) koja omogućava brzu implementaciju baze podataka i autentifikaciju.

7. API (*eng. Application Programming Interface*): API-ji omogućuju komunikaciju između različitih softverskih sustava. REST API (Representational State Transfer) i GraphQL su dva popularna pristupa za izgradnju API-ja. REST API koristi HTTP za komunikaciju između klijenta i poslužitelja te se oslanja na različite HTTP metode za upravljanje resursima. GraphQL omogućava klijentima da precizno specificiraju podatke koje žele dohvatiti, smanjujući nepotrebni prijenos podataka i poboljšavajući učinkovitost komunikacije.

8. Tehnologije u oblaku: Cloud platforme poput AWS-a, Azure-a i Google Cloud-a pružaju skalabilnu infrastrukturu, pohranu i mogućnosti implementacije, osiguravajući

visoku dostupnost i performanse web aplikacija. Ove tehnologije omogućavaju razvoj, implementaciju i upravljanje aplikacijama putem udaljenih servera, smanjujući potrebu za lokalnom infrastrukturom.

2.2. Paketi korišteni za razvoj web aplikacija

Odabir odgovarajućeg paketa ili okvira za projekt koji se razvija ključan je za izgradnju modernih, učinkovitih i robustnih web aplikacija. Kroz vrijeme, pojavilo se nekoliko popularnih kombinacija okvira koje su postale omiljen izbor programera, pružajući sveobuhvatan skup alata i tehnologija potrebnih za razvoj. Dvije takve kombinacije su MERN (MongoDB, Express.js, React, Node.js) i MEVN (MongoDB, Express.js, Vue.js, Node.js), koje pokazuju snagu JavaScripta i nude besprijekornu integraciju između poslužiteljskih i klijentskih komponenti, omogućavajući programerima izradu potpunih JavaScript aplikacija. Nadalje, LAMP paket (Linux, Apache, MySQL, PHP/Python/Perl) je dugo vremena bio najpopularniji u web razvoju, pružajući stabilnu platformu s Linuxom kao operacijskim sustavom, Apacheom kao web poslužiteljem, MySQL-om kao bazom podataka i PHP/Python/Perlom kao jezikom za skriptiranje na poslužiteljskoj strani.

Naposljetku, Django, okvir temeljen na Pythonu, nudi sveobuhvatan alatni skup za razvoj web aplikacija, uključujući ORM (*Object-Relational Mapping*), administrativno sučelje i snažne sigurnosne značajke. [1]

Nabrojani su samo neki od danas najkorištenijih paketa i okvira, no treba istaknuti da svaki okvir ima svoj skup značajki i filozofiju prilagođenu različitim potrebama i preferencijama programera. Korištenje okvira može značajno ubrzati razvoj, poboljšati održivost koda i potaknuti skalabilnost sustava koji se razvija. Programeri mogu iskoristiti ekosustav okvira, dokumentaciju i podršku zajednice kako bi riješili uobičajene izazove i izgradili visokokvalitetne aplikacije. Međutim, okviri mogu također nametnuti ograničenja ili zahtijevati puno učenja, budući da programeri moraju slijediti strukturu i konvencije okvira kojeg koriste.

2.2.1. Usporedba najkorištenijih paketa

"*Citius, altius, fortius*". Kao i latinski moto Olimpijskih igara, i kod odabira odgovarajućeg paketa tehnologija, razvojni timovi također teže odabrati kombinaciju

koja će omogućiti brži, efikasniji i napredniji razvoj aplikacija. Budući da danas postoji puno izbora, svaki sa svojim prednostima i nedostacima, u nastavku je prikazana kratka usporedba najkorištenijih paketa tehnologija.

Svijet web razvoja se konstantno mijenja velikom brzinom, a kao rezultat toga, nove i nastajuće tehnologije poput MERN-a kontinuirano preuzimaju vodeću ulogu. U toj utrci, LAMP je već daleko ostao iza, a MERN je postao konačan izbor mnogih web programera diljem svijeta. [2]

LAMP, kao akronim, označava četiri bitne komponente. Linux, operativni sustav, čini bazu, Apache služi kao softver web poslužitelja, MySQL, sustav upravljanja relacijskom bazom podataka, odgovoran je za pohranu i upravljanje podacima, dok PHP djeluje kao skriptni jezik na strani poslužitelja. U LAMP paketu, razvoj korisničkog sučelja se obično oslanja na HTML, CSS i koristeći JavaScript bez ikakvih dodatnih biblioteka i okvira. Iako se PHP može koristiti za izradu predložaka, prvenstveno se fokusira na logiku na strani poslužitelja. LAMP paket se pokazao pouzdan izbor kroz povijest, a njegove komponente kao robusne i stabilne. Također, LAMP paket tehnologija, ima ogromnu zajednicu i obilje dostupnih resursa zbog svoje dugogodišnje prisutnosti u industriji. Rješavanje problema i pronalaženje rješenja općenito je lakše s tako zrelim paketom tehnologija što je i dalje bitna stavka kod odabiru paketa tehnologija za korištenje.

MERN paket izgrađen je s četiri različite tehnologije. MongoDB služi kao NoSQL baza podataka, koja nudi fleksibilnost i skalabilnost. Express.js je minimalistički okvir web aplikacije za Node.js, odgovoran za rukovanje logikom na strani poslužitelja. Node.js, djeluje kao okruženje za izvršavanje JavaScript koda na strani poslužitelja, a React.js je moćna front-end biblioteka za izgradnju korisničkih sučelja.

MERN paket, s jakim naglaskom na JavaScriptu, nudi kohezivnije razvojno iskustvo. Korištenje jednog programskog jezika za izgradnju klijentske i poslužiteljske strane aplikacije, omogućuje programerima da upotrijebe svoje vještine u cijelokupnom procesu razvoja.

Korištenje NoSQL baza podataka kao što je MongoDB, omogućuje lakše horizontalno skaliranje i distribuciju podataka na više poslužitelja. Dodatno, neblokirajuća priroda Node.js-a omogućuje učinkovito rukovanje velikim brojem istodobnih veza, poboljšavajući ukupnu izvedbu. Krivulja učenja MERN paketa može biti strmija za programere koji su novi u JavaScript programskom jeziku. Međutim, za

one koji su već dobro upoznati s JavaScriptom, prijelaz je lakši i oni mogu iskoristiti svoje postojeće znanje u cjelokupnom procesu razvoja.

I LAMP i MERN paket tehnologija imaju svoje prednosti i prikladni su za različite vrste projekata i razvojnih timova. Stabilnost LAMP-a i široka podrška zajednice čine ga pouzdanim izborom za tradicionalniji tip web aplikacije, posebno one koje postoje već neko vrijeme. S druge strane, fokus MERN paketa je na JavaScript-u i njegova fleksibilnost čini ga idealnim za moderne, dinamične web aplikacije, posebno one koje zahtijevaju ažuriranje i skalabilnost u stvarnom vremenu. MERN paket je relativno noviji, brzo je prihvaćen, a njegovo korištenje kontinuirano raste. Iako za razvoj novijih projekata MERN, MEVN ili MEAN paketi tehnologija postaju popularniji izbor od LAMP-a, LAMP se i dalje intezivno koristi i ostaje popularan izbor kada se gleda stabilnost i pouzdanost. [3]

2.3. JavaScriptu i izgradnja web aplikacija

Kao programski jezik koji se izdvojio kao dominantna snaga koja oblikuje način interakcije s web stranicama i online aplikacijama, JavaScript ima bogatu i zanimljivu povijest koja seže do ranih dana interneta.

JavaScript je kreiran u samo deset dana, 1995. godine, od strane inžinjera Brendana Eicha, tadašnjeg zaposlenika tvrtke Netscape s ciljem dodavanja interaktivnosti kod korištenja web preglednika.

Glavne primjene bile su provjera unosa podataka u obrascima, iskočni prozori (*eng.pop-up*), prikazivanje više slika u ograničenom prostoru na stranici (eng. slide show), prikaz padajućih izbornika, kreiranje dinamičkih efekta i jednostavnijih animacija.

Početkom 2000-ih, sposobnosti JavaScripta značajno su napredovale s uvođenjem Asinkronog JavaScripta i XML-a (AJAX). Ova revolucionarna tehnika omogućila je ažuriranje određenog sadržaja web stranica bez potrebe za potpunim ponovnim učitavanjem stranice što je značajno poboljšalo korisničko iskustvo. AJAX je također odigrao ključnu ulogu u popularizaciji aplikacija s jednom stranicom tzv. SPA (*eng. Single Page Application*), gdje cijela aplikacija radi unutar jedne web stranice, a sadržaj se dinamički ažurira ovisno interakciji korisnika s aplikacijom.

Tradicionalno, JavaScript je bio ograničen samo na razvoj sučelja klijentske strane, međutim, 2009. godine Ryan Dahl predstavio je Node.js.

Node.js je okruženje za pokretanje JavaScripta na poslužiteljskoj strani, izvan web preglednika, sjedinjujući programski jezik i omogućujući razvoj cjelokupnog softvera u JavaScriptu.

Od početaka JavaScripta kao brzog rješenja za dodavanje interaktivnosti na web stranice do pozicije svestranog jezika koji pokreće suvremeni web razvoj, JavaScript je prošao dug put. Njegov dinamičan razvoj, uz pojavu biblioteka i okvira, omogućio je programerima stvaranje sofisticiranih, interaktivnih web aplikacija bogatih značajkama. Kako ulazimo u budućnost, važnost JavaScripta u web razvoju čini se neosporiva. [4]

2.3.1. TypeScript

TypeScript je nadskup JavaScripta, što znači da radi sve što i JavaScript, ali s nekim dodanim značajkama. TypeScript je razvio Microsoft 2012. godine kao odgovor na izazove s kojima se susreću programeri koji rade na velikim JavaScript projektima. Primarna motivacija razvoja TypeScripta bila je uvođenje statičkog tipkanja i poboljšanog alata u JavaScript ekosustav. Dinamička priroda JavaScripta dopuštala je fleksibilan kod, ali mu je nedostajala mogućnost prepoznavanja određenih vrsta pogrešaka tijekom razvoja. TypeScript je imao za cilj riješiti te probleme dodavanjem sustava tipova koji pomažu u hvatanju pogrešaka u vrijeme prevođenja, a ne u vrijeme izvođenja. Kao rezultat, korištenje TypeScripta učinilo je napisani kod više samodokumentirajući, što je olakšalo razumijevanje i održavanje napisanog koda.

Glavni razlog korištenja TypeScripta je dodavanje statičkog tipkanja u JavaScript. Statički tip znači da se tip varijable ne može promijeniti ni u jednom trenutku u programu što može spriječiti mnogo grešaka. JavaScript je jezik s dinamičkim tipovima, što znači da varijable mogu mijenjati vrstu kao što je vidljivo na primjeru:

```
// JavaScript
let ime = "Marko";
name = 99;
```

U primjeru iznad vidimo varijablu pod nazivom „ime“ koja ima vrijednost „Marko“, tipa string. Liniju ispod, varijabli “ime” dodana je nova vrijednost „99“, čiji je tip number. JavaScript ne vidi u tome nikakav problem i nastavlja s izvršavanjem koda bez greške.

Dopustivost JavaScripta da “prisilno” mijenja tipove varijabli može se činiti zgodnim na površini i ne predstavlja preveliki problem u projektima manje veličine, no u složenijim aplikacijama, odnosno većim projektima može dovesti do neočekivanih problema i grešaka kod izvođenja.

Zbog toga mnogi programeri preferiraju jezike kao što je TypeScript, koji pružaju strožu provjeru tipa i pomažu u otkrivanju ovakvih problema tijekom razvoja umjesto da im dopuštaju da se manifestiraju tijekom izvođenja. Primjer korištenja TypeScripta u istoj situaciji:

```
// TypeScript
let ime = "Marko";
name = 99;  // ERROR - ime cannot change from string to number
```

U primjeru iznad vidi se greška na koju je upozorio TypeScript i onemogućio daljnje izvršavanje koda. Greška u prijevodu znači da varijabla “ime” ne može promijeniti svoj tip iz string u number. U nastavku je prikazno još nekoliko primjera oznovnih značajki TypeScripta.

U TypeScriptu moguće je definirati koju vrstu podataka niz može sadržavati, na način da se poslije naziva varijable doda znak dvotočke i zatim naziv tipa kao što je prikazno u primjeru ispod:

```
let brojevi: number[] = [1,2,3,4,5];
```

Također se može dodati i unija tipova kojom se može dozvoliti više tipova vrijednosti:

```
let ime: string | number = "Marko";
ime = 99;


let brojevi: (string, number, boolean)[] = [„Marko“, 54, false];
brojevi[0] = 23;
brojevi[1] = {zanimanje: „Programer“}; // Error - brojevi array can't
contain objects
```

Ili je moguće omogućiti dodavanje bilo kojeg tipa vrijednosti s tipom „any“ kao što je prikazano ispod:

```
let ime: any = "Marko";
ime = true;
```

Iako je u primjerima iznad specificiranje tipova prikazano samo na varijabli i nizu, treba napomenuti da se statičko tipkanje u TypeScriptu može također dodati objektima, funkcijama, klasama i bilo kojim drugim konstrukcijama unutar baze koda.

Vrlo bitna značajka TypeScripta su sučelja (*eng. Interfaces*). Sučelja je alat koji omogućuje definiranje strukture ili oblika objekata te pružaj način za opisivanje očekivanih svojstava, metoda i tipova koje objekt treba imati. Sučelje je posebno korisno za definiranje ugovora između različitih dijelova koda, osiguravajući da se određeni objekti pridržavaju određene strukture. U nastavku je prikazan primjer u kojem je definirano sučelje pod nazivom “osoba” koje sadrži tri svojstva: “ime”, “dob”, “zaposlen” i svakom svojstvu je definiran tip vrijednosti koji joj može biti pridružen.

```
interface Osoba {
   ime: string;
   dob: number;
   zaposlen: boolean;
}
```

Nakon što je sučelje definirano, može se koristiti za definiranje objekata koji odgovaraju njegovoj strukturi. Stvaranje objeka koji se pridržava sučelja osobe je prikazan u nastavku:

```
const osoba: Osoba = {
   ime: Marko,
   dob: 54,
   zaposlen: true
};
```

3. MEVN paketi

Kombinirajući MongoDB, Express.js, Vue.js i Node.js, MEVN paket je skup otvorenih tehnologija koji nudi sveobuhvatan i funkcionalno bogat okvir za razvoj web aplikacija koristeći JavaScript. MEVN paket je stekao značajnu popularnost u zajednici za web razvoj i stoji kao konkurentan izbor uz druge popularne JavaScript pakete, poput MERN i MEAN (MongoDB, Express.js, Angular.js i Node.js,). Svaki navedeni paket tehnologija ima svoje prednosti i slučajeve kada ih je najbolje upotrijebiti, a njihova popularnost može varirati ovisno o čimbenicima kao što su: preferencije programera, projektni zahtjevi i trendovi u industriji. Očita razlika kod odabira korištenja paketa između MEVN, MERN i MEAN paketa tehnologija je odabir okvira za razvoj klijentske strane aplikacije, odnosno korisničkog sučelja.

Angular.js, React.js i Vue.js su tri najpopularnija JavaScript okvira koji se koriste za izgradnju korisničkih sučelja i razvoj web aplikacija. Iako dijele više sličnosti nego razlika, svaki okvir ima svoje različite značajke i filozofiju dizajna. U nastavku je rečeno ponešto o svakome od njih.

Angular.js je razvio Google 2010. godine. Dok se mnogi JavaScript okviri fokusiraju na proširenje mogućnosti samog JavaScripta, AngularJS umjesto toga pruža metode za poboljšanje HTML-a. AngularJS proširuje mogućnosti HTML-a izvan jednostavnog jezika za označavanje.

React.js je front-end biblioteka, razvijena i održavana od strane Facebook-a (Meta), koja je postupno postala najpopularniji okvir za moderni web razvoj unutar JavaScript zajednice. U Reactu, aplikacija se razvija na način da se stvaraju komponente za višekratnu upotrebu koje se mogu zamisliti kao neovisne Lego kocke. Te komponente su pojedinačni dijelovi konačnog sučelja, koji, kada se sastave, tvore cjelokupno korisničko sučelje aplikacije.

Vue.js, kreiran 2014. godine od strane bivšeg zaposlenika Google-a, Evana You. You je kod kreiranja Vue.js okvira uzeo, po njegovom mišljenju, najbolje značajke Angular i React okvira. Danas je Vue.js jedan od najkorištenijih JavaScript okvira i trenutno je na svojoj trećoj verziji. U nastvku rada detaljnije će se opisati njegove značajke i funkcionalnosti. [5]

3.1. MongoDB

MongoDB je NoSQL baza podataka orijentirana na dokumente koja nudi fleksibilne i skalabilne značajke. MongoDB pohranjuje podatke u zbirkama dokumenata, slično JSON (*JavaScript Object Notation*) objektima, za razliku od tipičnih relacijskih baza podataka koje drže podatke u tablicama s unaprijed određenim redovima i stupcima. U SQL tablici svaki podatak ima istu shemu, što znači da nakon što je tablica inicijalizirana, podaci koji se umeću moraju imati sva polja prikazana u tablici, čak i ako vrijednosti za neka polja nisu prisutne, tada se ta polja popunjavaju s podacima 'NULL' što znači da su polja statična. U slučaju NoSQL-a, podaci ne moraju bitit umetnuti u sva polja, odnosno polja su dinamička. Ovo je jedan od najvažnijih čimbenika koji razlikuju NoSQL od SQL baze podataka. [6]

Primjer dokumenta u MongoDB bazi podataka:

```
{
  "_id": ObjectId("5ce45d7606444f199ac"),
  "ime": "Joe",
  "prezime": "Doe",
  "email": "email@example.com",
  "dob": 27,
  "adresa": {
              "ulica": "Pavlinksa",
              "broj": "2",
              "mjesto": "Varaždin",
             },
  "datumRođenja": "1980-11-11",
  "uloga": "manager",
}
```

Jedan dokument, kao dokument prikazan na prethodnom primjeru, približno odgovara jednom unosu u određenoj tablici unutar SQL baze podataka. Dokumenti se pohranjuju u tzv. kolekciju, što je ekvivalent tablici ili relaciji u SQL bazama podataka.

Dok se relacijskim bazama podataka postavljaju upiti korištenjem Structured Query Language (SQL), u MongoDB upiti se postavljaju korištenjem MongoDB Query Language (MQL). MQL upit je zahtjev za dohvaćanje podataka iz MongoDB baze podataka koji uključuje pretraživanje, filtriranje, kreiranje, dohvaćanje i brisanje dokumenata iz zbirke na temelju određenih kriterija. Sljedeći primjer prikazuje jednostavan upit za dohvaćanje korisnika po kriteriju imena i prezimena iz MongoDB baze podataka:

```
db.korisnici.find({ "ime": "Joe" }, { "prezime": "Doe"})
```

MongoDB je popularan izbor NoSQL baze podataka kod korištenja u kombinaciji s Node.js aplikacijama i općenito u aplikacijama napisanim u JavaScript programskim jezikom.

Već je spomenuto da MongoDB pohranjuje podatke u dokumente slične JSON objektima, format koji je također sličan JavaScript objektima. Usklađivanje s JavaScriptom čini MongoDB praktičnim za slanje i interpretiranje podataka, bez potrebe za opsežnim transformacijama.

3.1.1. Mongoose

Sustavi objektno-relacijskog preslikavanja (*eng. Object-Relational Mapping*), nadalje ORM, pojavili su se kao ključni alati koji se koriste za komunikaciju s bazom podataka u projektima objektno organizirane strukture i organizacije zbog jednostavnije komunikacije s SQL bazom podataka. Međutim, kada se radi o svijetu baze podataka NoSQL, specijalizirani alat koji se zove modeliranje podataka o objektu (*eng. Object Data Modeling*), nadalje ODM, koristi se radi slične uloge. Najpoznatiji ODM kod korištenja s MongoDB bazom podataka je Mongoose.

Mongoose je robusna ODM biblioteka koja pojednostavljuje interakcije s MongoDB unutar Node.js aplikacija. Omogućuje pristup temeljen na shemi za definiranje strukture, pravila provjere valjanosti i ponašanja dokumenata unutar MongoDB kolekcija. U svojoj srži, Mongoose omogućuje razvojnim programerima modeliranje podataka aplikacije pomoću JavaScript klasa, što olakšava rad s MongoDB strukturom usmjerenom na dokumente.

Kako bi Mongoose koristio u projektu, prvo ga je potrebno instalirati naredbom “`npm install moongoose`” i povezati s aplikacijom na jedan od načina koji je prikazan u primjeru ispod.

```
import mongoose from ‘mongoose’;

mongoose.connect('mongodb://localhost/mojaBaza');

const db = mongoose.connection;
```

3.1.2. MongoDB shema

Shema u MongoDB-u, koja se naziva "MongoDB shema" (eng. MongoDB Schema), definira strukturu dokumenata unutar zbirke. MongoDB kao NoSQL baza

podataka, poznata po svojoj fleksibilnosti, korištenjem sheme pruža organizaciju i provjeru valjanosti podataka, posebno u složenijim ili relacijskim podacima.

MongoDB shema definira polja i njihove vrste podataka koje svaki dokument u zbirci treba sadržavati. Slično je shemi tablice u tradicionalnim relacijskim bazama podataka. Polja mogu biti različitih tipova podataka, kao što su niz, broj, Booleov, datum, ugrađeni dokumenti i još mnogo toga. Također, MongoDB sheme mogu sadržavati pravila provjere podataka. Time se osigurava da podaci dodani u bazu podataka zadovoljavaju određene kriterije. Pravila provjere valjanosti mogu uključivati zahtjeve kao što su obvezna polja, format podataka, maksimalne duljine ili prilagođene funkcije provjere valjanosti i unaprijed zadane vrijednosti polja.

U nastavku je prikazano jednostavno definiranje sheme i njenih polja za dokument pod nazivom “Knjiga” korištenjem Mongoosa.

```
import moongoose from 'mongoose';

const knjigaSchema = new mongoose.Schema({
   naslov: {
     type: String,
     required: true,
  },
   autor: {
      type: String,
      required: true,
   },
    godina: {
      type: Number,
      required: true,
   },
});

const Knjiga = mongoose.model('Knjiga', knjigaSchema);
```

3.1.3. MongoDB i Node.js

Dizajn MongoDB baze usklađen je s asinkronim modelom koji koristi Node.js. Prilikom interakcije s bazom podataka, kao što je postavljanje upita ili umetanje podataka, te se operacije mogu izvoditi asinkrono pomoću povratnih poziva (eng. Callback), obećanja (eng. Promises) ili sintakse async/await. Ovaj pristup sprječava blokiranje aplikacije dok čeka da se operacije baze podataka završe. Na primjer, kada Node.js aplikacija pošalje upit MongoDB-u, ne zaustavlja njegovo izvršenje sve dok upit ne vrati rezultate. Umjesto toga, aplikacija nastavlja obrađivati druge zadatke ili rukovati dolaznim zahtjevima. Nakon što MongoDB dovrši upit, pokreće pridruženi

povratni poziv ili rješava obećanje, dopuštajući aplikaciji da rukuje podacima. Izbjegavanjem nepotrebnog vremena čekanja, aplikacije mogu brže obrađivati zadatke, što dovodi do poboljšane ukupne izvedbe.

U nastavku je prikazana jednostavna Node.ja aplikacija koja se spaja na mongoDB bazu podataka, definira shemu korisnika i komunicira s MongoDB bazom podataka koristeći async/await sintaksu s ciljem dohvata korisničkih podataka.

```
import express from 'express';
import moongoose from 'mongoose';

const app = express();
mongoose.connect(' mongodb://localhost/mydatabase');

const userSchema = new mongoose.Schema({
   username: String,
   email: String,
});

const User = mongoose.model('User', userSchema);

app.get('/users', async (req, res) => {
   try {
     const users = await User.find();

     res.json(users);
   } catch (err) {
      res.status(500).json({ error: err.message });
   }
});

app.listen(3000, () => {
  console.log('Server started on port 3000');
});
```

Riječ „await“ koristi se za pauziranje izvršavanja rukovatelja rutom, dok MongoDB operacija “User.find()” ne završi dokvat korisnika iz baze. Ovo osigurava da aplikacija i dalje reagira i da može obraditi druge zahtjeve dok čeka da operacija baze podataka završi. Nakon što se upit u bazu izvrši i rezultat bude uspješan “res.json(users)” vraća dohvaćene podatke iz baze u JSON formatu, a ako rezultat bude neuspješan aplikacija vraća grašku statusa 500 i poruku koja govori u čemu je greška.

3.2. Node.js

Rastom popularnosti JavaScript-a kao jezika za razvoj web aplikacija na strani klijenta koji je mogao biti izveden samo u web preglednicima, tvrtke su pokušale koristiti JavaScript i za pokretanje web poslužitelja (skriptiranje na strani poslužitelja). Pokušaji su uključivali Netscapeov Livewire i Microsoftove Active Server Pages, alli to nikada nije postao način razvoja web poslužitelja koji koristi JavaScript.

Godine 2008. Google je najavio novi, optimizirani web preglednik koji brzo izvršava JavaScript, pod nazivom Chrome. Kada je Crome preglednik objavljen, napravio je revoluciju u svijetu pregledavanja interneta zbog velikog poboljšanja korisničkog iskustvo na webu.

Razlog zašto je Google Chrome mogao tako brzo izvršavati JavaScript kod bio je taj što je JavaScript mehanizam, pod nazivom V8, korišten unutar Chromea.

V8 mehanizam bio je odgovoran za prihvaćanje JavaScript koda, optimiziranje koda te njegovo izvršavanje na računalu.

Google Chrome je postao vodeći web preglednik, a korištenje njegovog V8 mehanizma odgovarajuće rješenje za izvršavanje JavaScript-a na strani klijenta.

Kritizirajući neučinkovito podnošenje velikog broja korisničkih veza u stvarnom vremenu (10 000 +) tradicionalnog poslužitelja za izgradnju web servisa Apache HTTP poslužitelja, inžinjer Ryan Dahl 2009. godine kreirao je Node.js koji koristi Googleov V8 mehanizam za razumijevanje i izvođenje JavaScript koda izvan preglednika. Node.js se pokazao kao izvrsna alternativa tradicionalnom Apache HTTP poslužitelju i polako je postao prihvaćen među razvojnom zajednicom. [7]

U srcu Node.js-a leži njegov I/O (input/output) model koji nije blokiran za razliku od tradicionalnih web poslužitelja koji obrađuju svaki zahtjev klijenta sekvencijalno, Node.js radi na petlji događaja s jednom dretvom. Node.js može rukovati s višestrukim istodobnim vezama bez stvaranja zasebnih dretva za svaku, što ga čini vrlo učinkovitim u rukovanju aplikacijama koje intenzivno koriste podatke u stvarnom vremenu i omogućuje Node.js-u izvršavanje drugih zadataka dok čeka da se dovrše I/O operacije. Kao rezultat toga, poslužitelj se ne blokira, što mu omogućuje istovremeno rukovanje brojnim vezama. Node.js nadmašuje tradicionalne tehnologije na strani poslužitelja kao što su PHP i Ruby, što ga čini izvrsnim izborom za aplikacije koje zahtijevaju nisku latenciju i visoku propusnost. [8]

3.2.1. Node.js biblioteke

U današnjem svijetu izgranje softwera, potrebno se stalno ažurirati najnovijim i najboljim bibliotekama jer današnji razvoj softvera ovisi o njima, ako poznajete dobre biblioteke, to potencionalno može uštedjeti vrijeme, naporan rad i također pomoći u izgradnji kvalitetnog softvera. U nastavku poglavlja biti će spomenute neke najkorištenije Node.js biblioteke koje su također korištene kod izgradnje aplikacije u praktičnom dijelu rada.

3.2.1.1. CORS

CORS (Cross-Origin Resource Sharing) štiti korisnike od web stranica koje bi mogle pokušati učiniti ono što ne bi smjele. Ako web-aplikacija s jednog mjesta želi nešto s drugog mjesta, preglednik provjerava slaže li se s tim. Ako ne, aplikacija ne može dobiti ono što želi, osim ako drugo mjesto ne kaže da je u redu pomoću posebnih CORS uputa. Korištenjem biblioteke, CORS osigurava da aplikacija šalje potrebna CORS zaglavlja kako bi se omogućili zahtjevi iz pouzdanih izvora, što je važno kod izrade API-ja ili web-aplikacija koje trebaju komunicirati s različitim domenama uz održavanje snažnog sigurnosnog položaja. Za korištenje CORS biblioteke u Node.js aplikaciji, prvo je potrebno instalirati CORS naredbom `npm install cors`, zatim ga inicijalizirati u projekt kao što je prikazano u primjeru ispod.

```
import express from 'express';
import cors from 'cors';

const app = express();
app.use(cors({ origin: 'https://example.com'}));
```

3.2.1.2. Body Parser

Body Parser je međuprogramski modul za rukovanje dolaznim tijelima HTTP zahtjeva u Node.js aplikacijama. Kada klijent pošalje podatke poslužitelju, poput slanja obrasca ili slanja JSON podataka, podaci su uključeni u tijelo zahtjeva. Body Parser pomaže analizirati i izdvojiti te podatke iz zahtjeva kako bi ih server mogao obraditi i koristiti. Za korištenje body-parser biblioteke u Node.js aplikaciji, bodiy-parser je prvo potrebno instalirati naredbom `npm install body-parser`, zatim inicijalizirati u projekt kao što je prikazano u primjeru ispod.

```
import express from 'express';
import bodyParser from 'body-parser';
```

```
const app = express();
app.use(bodyParser.json());
app.use(bodyParser.urlencoded({ extended: true }));
```

3.2.1.3. Joi

Joi je popularna knjižnica za provjeru valjanosti za JavaScript koja se često koristi u Node.js aplikacijama za provjeru ispranosti podataka, posebno unosa korisnika i korisnih opterećenja API zahtjeva. Omogućuje jednostavan i deklarativan način za definiranje pravila provjere valjanosti i ograničenja za različite vrste podataka, osiguravajući da podaci zadovoljavaju očekivani format i kriterije prije daljnje obrade. Za korištenje Joi biblioteke u Node.js aplikaciji, prvo ga je potrebno instalirati naredbom `npm install joi`, zatim inicijalizirati u projekt kao što je prikazano u primjeru ispod.

```
import express from 'express';
import Joi from 'joi';

const app = express();

app.post('/user', (req, res) => {
  const schema = Joi.object({
    ime: Joi.string().required(),
    email: Joi.string().email().required(),
    dob: Joi.number().min(18).required()
  });

  const { error, value } = schema.validate(req.body);
```

3.2.1.4. Multer

Multer je međuprogram za Node.js, koji se prvenstveno koristi za učitavanje datoteka i pojednostavljanja proces rukovanja datotekama. Može se jednostavno dodati u Express.js rute za obradu prijenosa datoteka do određene krajnje točke koja se definira u konfiguraciji. Multer je lako konfigurabilan i omogućuje postavljanje raznih opcija za kontrolu ponašanja kod učitavanja datoteka. Neke od značajnijih konfiguracija su:

1. Možete definirati kako će učitane datoteke biti pohranjene na poslužitelju. Multer podržava različite mehanizme za pohranu, kao što je pohrana na disku (lokalni sustav datoteka), pohrana u memoriji, pa čak i rješenja za pohranu u oblaku kao što je Amazon S3.
2. Multer može obraditi više datoteka učitanih u jednom zahtjevu. Može se konfigurirati da prihvati više datoteka i odrediti kako pohraniti ili rukovati svakom od njih.

Primjer konfiguracije i korištenja Multer biblioteke bi će prikazan u praktičnom dijelu rada.

3.2.1.5. Axios

Axios je svestrana JavaScript biblioteka koja omogućuje postavljanje HTTP zahtjeva sa strane poslužitelja (Node.js) i sa strane klijenta (web preglednik). Jedna od ključnih prednosti Axiosa je njegov korisnički prilagođen API. Programeri mogu brzo shvatiti njegovu sintaksu i početi postavljati HTTP zahtjeve. Axios pruža jednostavno sučelje za različite HTTP metode kao što su GET, POST, PUT i DELETE. Jednostavnost omogućuje programerima da se usredotoče na izgradnju funkcionalnosti umjesto da se bore sa složenim kodom zahtjeva. U primjeru ispod je prikazan jednostavan primjer HTTP GET zahtjeva, koristeći Axios.

```
import axios from 'axios';

axios.get('https://api.example.com/data')
   .then(response => {
      console.log(response.data);
   })
   .catch(error => {
      console.log(error);
   });
```

3.2.2. NPM

Node.js velik dio svog uspjeha duguje Node Package Manageru (NPM), moćnom i opsežnom upravitelju paketa za JavaScript. NPM pruža pristup ogromnom repozitoriju biblioteka i modula koji se mogu ponovno koristiti, omogućujući programerima da značajno ubrzaju svoj razvojni proces. Bilo da se radi o izgradnji web poslužitelja, rukovanju vezama baze podataka ili integraciji s API-jima trećih strana, NPM nudi mnoštvo unaprijed izgrađenih rješenja, štedeći programerima dragocjeno vrijeme i trud. U nastavku je prikazan primjer komande koja pomoću NPM-a instalira Express.js, najpopularniji web okvir za Node.js.

```
npm install -D express
```

3.3. Express.js

Dok je Node.js okruženje koje omogućuje izvršavanje JavaScript koda na strani poslužitelja, Express.js je najpopularniji web okvir koji pojednostavljuje proces izrade

web aplikacija u Node.js okruženju. Pruža skup robusnih značajki i uslužnih programa za stvaranje aplikacija i API-ja na strani poslužitelja. Express je izgrađen na temelju Node.js i koristi njegovu neblokirajuću arhitekturu vođenu događajima za učinkovito rukovanje brojnim istodobnim vezama. Najznačajnije značajke Express.js okvira opisane su u nastavku.

Express.js je okvir bez unaprijed definiranih smjernica kod razvoja, što znači da programerima omogućuje slobodu strukturiranja koda po želji, umjesto da se nameće određena struktura koda. Također, Express.js je minimalistički okvir, što znači da inicijalno pruža samo osnovne alate i značajke potrebne za izradu web-aplikacija i API-ja. Za razliku od nekih drugih web okvira koji dolaze u paketu s mnogo unaprijed izgrađenih komponenti, Express.js svoju jezgru drži malom i prepušta programerima da po potrebi dodaju dodatne funkcije.

Express.js je uveo koncept međuprograma (*eng. middleware*). Međuprogrami su funkcije koje se mogu izvršiti prije konačnog obrađivača zahtjeva. Međuprogramski sustav omogućuje dodavanje specifične funkcionalnosti svojim aplikacijama bez pretrpavanja koda glavne aplikacije. Međuprogram se može primijeniti i na razini aplikacije i na razini rute, kao i biti povezan u lance. Može se dodati gotovo bilo koji kompatibilni međuprogram u lanac obrade zahtjeva, gotovo bilo kojim redoslijedom. Ova fleksibilnost pomaže da se jezgra Express.js-a održi jednostavnom, a istovremeno omogućuje programerima da prošire njegovu funkcionalnost kada je to potrebno. Ispod je prikazan primjer jednostavnog međuprograma koji prikazuje funkciju pod nazivom helloWorld koja će se izvršiti prije prijenosa kontrole sljedećem međuprogramu ili rukovatelju rute.

```
import express from 'express';

const app = express();

const helloWorld = (req, res, next) => {
  console.log('Hello World!');
  next();
}

app.use(helloWorld)

app.get(/hello, (req, res) =>{
  res.status(201);
});
```

3.3.1. REST API i Express.js

U modernom web razvoju, koncept RESTful API-ja postao je neizostavan dio besprijekorne komunikacije između klijenata i poslužitelja. REST (*eng. Representational State Transfer*) je arhitektonski stil koji omogućuje interakciju s RESTful web servisima. API (*eng. Application programming interface*) je skup definicija i protokola za izgradnju i integraciju aplikacijskog softvera. Jednostavno rečeno, ako želimo komunicirati s računalom ili sustavom kako bi dohvatili informacije ili izvršili funkciju, API nam pomaže priopćiti ono što želimo reći tom sustavu, kako bi taj sustav mogao interpretirati, odnosno razumjeti i ispuniti zahtjev.

RESTful API-ji dizajnirani su oko koncepta resursa, koji predstavljaju entitete kojima se može manipulirati pomoću HTTP metoda. Express.js, sa svojim mogućnostima usmjeravanja i međuprograma (*eng. middleware*), popularan je okvir za izgradnju RESTful API-ja.

Osnova RESTful API-ja leži u mapiranju HTTP metoda (GET, POST, PUT, DELETE) u CRUD (*eng. Create, Read, Update, Delete*) operacije na resursima. Express.js sustav usmjeravanja čini ovo mapiranje jednostavnim.

U nastavku su prikazani jednostavni primjeri kreiranja RESTful API-ja s HTTP metodama u express.js-u.

Kreiranje knjige (POST):

```
app.post('/knjiga', (req, res) => { });
```

Dohvačanje knjige (GET):

```
// dohvačanje svih knjiga
app.get('/knjiga', (req, res) => { });

// dohvačanje specifične knjige po "id" GET parametru
app.get('/knjiga/:id', (req, res) => { });
```

Ažuriranje knjige (PUT/PATCH)

```
app.put('/knjiga/:id', (req, res) => { });
```

Brisnje knjige (DELETE)

```
app.delete('/knjiga/:id', (req, res) => { });
```

3.4. Vue.js

Vue.js je okvir za izgradnju korisničkih sučelja u JavaScriptu koji se lako nadovezuje na standardne web tehnologije poput HTML-a, CSS-a i JavaScripta,

omogućujući deklarativno programiranje temeljeno na komponentama. Deklarativna paradigma omogućuje učinkovit razvoj kako jednostavnih, tako i složenih korisničkih sučelja. Jedna od glavnih prednosti Vue.js-a je njegova svestranost i prilagodljivost. Može zadovoljiti različite zahtjeve za razvoj korisničkog sučelja, prilagođavajući se različitim scenarijima upotrebe. Bez obzira na vrstu web aplikacije koja se gradi, Vue.js nudi fleksibilnost i postupnu prilagodljivost koja omogućuje razvijanje korisničkih sučelja na način koji najbolje odgovara traženim zahtjevima.

Jedan od načina na koji se Vue može iskoristiti je poboljšanje statičkog HTML-a. Programeri mogu jednostavno ugraditi mogućnosti Vue-a u postojeći HTML kod, omogućujući im stvaranje interaktivnih i dinamičkih korisničkih sučelja bez napora kao što je vidljivo u primjeru ispod koji prikazuje `v-if` , koji kao uvijet prikaza vrijednosti varijable “ime” provjerava da li varijabla pod nazivom „ime“ sadrži neku vrijednost, ako nema oznaka prikazati će se vrijednost linije ispod gdje se nalazi `v-else`.

```
<template>
   <div>
       <p v-if=”ime.length > 0”>{{ ime }}</p>
       <p v-else>Varijabla Ime nema vrijednost.</p>
   </div>
</tamplate>
```

Drugi pristup je ugradnja Vue-a kao web komponente na bilo koju stranicu. Koristeći Vue na ovaj način, programeri mogu kapsulirati komponente s njihovom funkcionalnošću i stilovima. U primjeru ispod prikazana je komponenta s funkcionalnošću jednostavnog brojača koji na klik gumba povećava vrijednost varijable “broj” za 1, također je prikazana CSS stilizacija za HTML objekte gumb i paragraf.

```
<script setup>
   import { ref } from "vue";

   const broj = ref(0);
</script>

<template>
   <div>
       <button @click=“broj++“>Klikni ovdje!</button>
       <p>{{ broj }}</p>
   </div>
</tamplate>
<style>
   p{ color: black; font-weight: bold; }
   button{ color: white; background: black; }
 </style>
```

Nakon što je kreirana komponenta, ponovno je upotrebljiva i lako integrirana u ostale komponente kroz aplikaciju. Komponentu koju integriramo unutar neke druge komponente nazivamo dječjom komponentom (eng. child component), a komponentu unutar koje je dječja kopmonenta intergirana, roditelj komponenta (eng. parent component). U primjeru ispod prikazano je ponovno korištenje komponente iz prethodnog primjera u nekoj drugoj komponenti.

```
import Brojac from „./brojac.vue“;

<template>
   <Brojac />
</tamplate>
```

Za složenije aplikacije, Vue je izvrstan izbor za izradu SPA. Sa svojim reaktivnim povezivanjem podataka i arhitekturom temeljenom na komponentama, Vue omogućuje razvojnim programerima stvaranje besprijekornog i responzivnog korisničkog iskustva s glatkim prijelazima između stranica. [10]

3.4.1. Props i Emits

U prethodnom poglavlju spomenuti su i prikazani primjeri dječjih (*eng. child*) i roditeljskih (*eng. parent*) komponenti u Vue.js-u. Komponente se mogu smatrati građevnim blokovima, s roditeljskim komponentama koje enkapsuliraju dječje komponente. Hijerarhijska struktura omogućuje rastavljanje složenih korisničkih sučelja na manje dijelove kojima se može upravljati. Svaka komponenta može imati vlastitu logiku, predložak i stil, poticajući mogućnost ponovne upotrebe i održavanja koda.

Komunikacija između dječjih i roditeljskih komponenti je temelj za izgradnju višekratno upotrebljivih i održivih aplikacija temeljenih na komponentama. U ovom poglavlju biti će objašnjeno na koji način one međusobno komuniciraju odnoso na koji način je moguće slati podatke između njih.

Props skraćeno od (*eng. properties*) omogućuje prijenos podataka od roditeljske komponente do dječje komponente. Ovaj mehanizam osigurava da dječje komponente primaju potrebne informacije za ispravno funkcioniranje i prikazivanje informacija. U kodu roditeljske komponente, svojstva podataka vezana su za dječje komponente pomoću direktive `v-bind` ili njezine skraćenice `:`. Na primjer, `:propIme="ime"` šalje

vrijednost varijable „ime“ u dječju komponentu. Da bi vrijednost varijable „ime“ bila inicijalizirana u dječjoj komponenti, prvo je potrebno definirati sve prop vrijednosti koje komponenta očekuje; `const props = defineProps(["propIme"]);` Nakon definiranja vrijednost propIme se dohvaća na način; `props.propIme;` U seljedećem primjeru prikazana je dječja komponenta pod nazivom “Kolegij” kojoj je prosljeđen prop koji sadrži popis studenata i primjer kako je vrijednost dohvaćena i prikazana.

```
// Roditelj komponenta
<script setup>
 import Kolegij from „@/components/kolegij.vue“;
 const studenti = { ime: Dario, prezime: Milažar,
                    ime: Pero, prezime: Perić,
                    ime: Hrvoje, prezime: Horvat }
</script>
<template>
   </ Kolegij :studenti>
</template>

// Komponenta djete (Kolegij)
<script setup>
   Const props = defineProps([„studenti“]);
</script>
<template>
   <h1>Popis studenata:</h1>
   <div v-for=“student in props.studenti“>
      Ime: {{ student.ime }} Prezime: {{ student.prezime }}
   </div>
</template>
```

Dok props olakšavaju jednosmjerni protok podataka od roditelja do djeteta, Vue.js koristi emite kako bi dječje komponente mogle komunicirati sa svojim roditelj komponentama. Dječja komponenta definira emite na način; `const emit = defineEmits(["uspjeh"]),` dok ih roditelj komponenta dohvaća s prefiksom `@` prije naziva emita kojeg očekuje. U sljedećem primjeru prikazana je komunikacija između dječje i roditelj komponente pomoću emita.

```
// Komponenta djete
<script setup>
   const emit = defineEmits([„uspjeh“]);
   const emitUspjeh = () => {
     emit(„uspjeh“);
   }
</script>
<template>
   <button @click=“emitUspjeh“>
     Klikni ako je radnja uspješna!
   </button>
</template>

// Komponenta roditelj
<template>
   </ Kolegij @uspjeh=““
</template>
```

3.4.2. Pinia

Kako aplikacija raste, tako se povećava i složenost upravljanja podacima. U velikoj aplikaciji u kojoj imamo velik broj komponenti, upravljanje njihovim stanjem postaje zahtjevno. Upravljanje stanjem aplikacije može se predočiti na način da kada korisnik stupa u interakciju s aplikacijom, ona radi neke promjene na kontrolama korisničkog sučelja, dohvaća podatke s poslužitelja, pokreće animaciju, mijenja rute itd. Dakle, svaki put kada korisnik stupa u interakciju ili radi neke promjene, mijenja stanje aplikacije. Mijenjanje stanja zahtijeva određeno upravljanje, gdje dolazi do upravljanja stanjem koje pomaže upravljati različitim stanjem koje se mijenja tijekom vremena i pomaže proširiti aplikaciju s boljom kontrolom nad njom.

Vue.js za upravljanje stanjem koristi Piniu, rješenje za upravljanje stanjem posebno dizajnirano za Vue 3, koje pruža centraliziranu pohranu za stanje aplikacije. Pinia usvaja pristup temeljen na trgovini (*eng. store*), gdje je stanje izolirano u trgovinama i pristupa mu se putem dobro definiranog API-ja. Odvajanje osigurava modularnost, mogućnost testiranja i lakše održavanje. U primjeru ispod prikazano je definiranje jednostavne Pinia trgovine koja ima za zadatak spremanje stanja pod

nazivom “broj” i manipuliranje stanja zavisno o akciji koja se pozove. Akcija “povecaj” povećava vrijednost stanja za 1, dok akcija “smanji” smanjuje vrijednost za 1.

```
import { defineStore } from 'pinia';

export const useBrojacStore = defineStore('brojac', {
  state: () => ({
    broj: 100,
  }),
  actions: {
    povecaj() {
      this.broj++;
    },
    smanji() {
      this.broj--;
    },
  },
});

export default useBrojacStore;
```

Kao što je već ranije rečeno jedna trgovina se može koristiti u više komponenti na način da se uvede (*eng. import*) u komponentu gdje se želi koristiti, kao što je prikazano u sljedećem primjeru.

```
<script setup>
import { useBrojacStore } from “@/stores/brojac.js”;

const brojacTrgovina = useBrojacStore();
</script>

<template>
  <div>
    {{ brojacTrgovina.$state.broj }}

    <button @click=“brojacTrgovina.povecaj“>Povečaj za 1!</button>
    <button @click=“brojacTrgovina.smanji“>Smanji za 1!</button>
   </div>
</template>
```

3.4.3. Nuxt.js

Nuxt.js je okvir temeljen na Vue.js-u koji nudi snažan ekosustav za jednostavno izgradnju web aplikacija. Adresira neke nedostatke korištenja samog Vue.js-a i pruža nekoliko prednosti, posebno u pogledu SEO-a i performansi aplikacije.

Jedan od glavnih nedostataka korištenja samog Vue.js-a je nedostatak ugrađene podrške za SSR (eng. server-side rendering) i statičku generaciju web stranica (*eng. Static site generator*), nadalje SSG. Kada se web aplikacije grade samo s Vue.js-om, optimizacija za tražilice (SEO) može biti izazovna jer tražilice mogu imati

problema s pretraživanjem i indeksiranjem sadržaja jednostraničnih aplikacija (SPA). Međutim, Nuxt.js dolazi s podrškom za SSR i SSG, omogućavajući aplikacijama da se renderiraju na strani poslužitelja prije nego što se pošalju klijentu ili da generiraju statičke HTML datoteke tijekom izgradnje. To značajno poboljšava SEO omogućujući tražilicama pristup potpuno renderiranom sadržaju, što dovodi do boljih rangiranja na tražilicama i poboljšane vidljivosti za web stranice s puno sadržaja.

Dodatno, Nuxt.js pojednostavljuje postupak konfiguracije pružajući razumne zadane postavke i slijedi princip konvencije nad konfiguracijom. To smanjuje količinu potrebnog standardnog koda i postavljanja, što omogućuje programerima da brzo krenu u rad bez žrtvovanja performansi ili fleksibilnosti. Nasuprot tome, kada se koristi samo Vue.js, programeri moraju postaviti i konfigurirati postupak izgradnje i SSR-a ručno, što može biti vremenski zahtjevno i zahtijevati dublje razumijevanje alata za izgradnju i koncepta server-side renderinga. [11]

4. Izrada vlastite web aplikacije

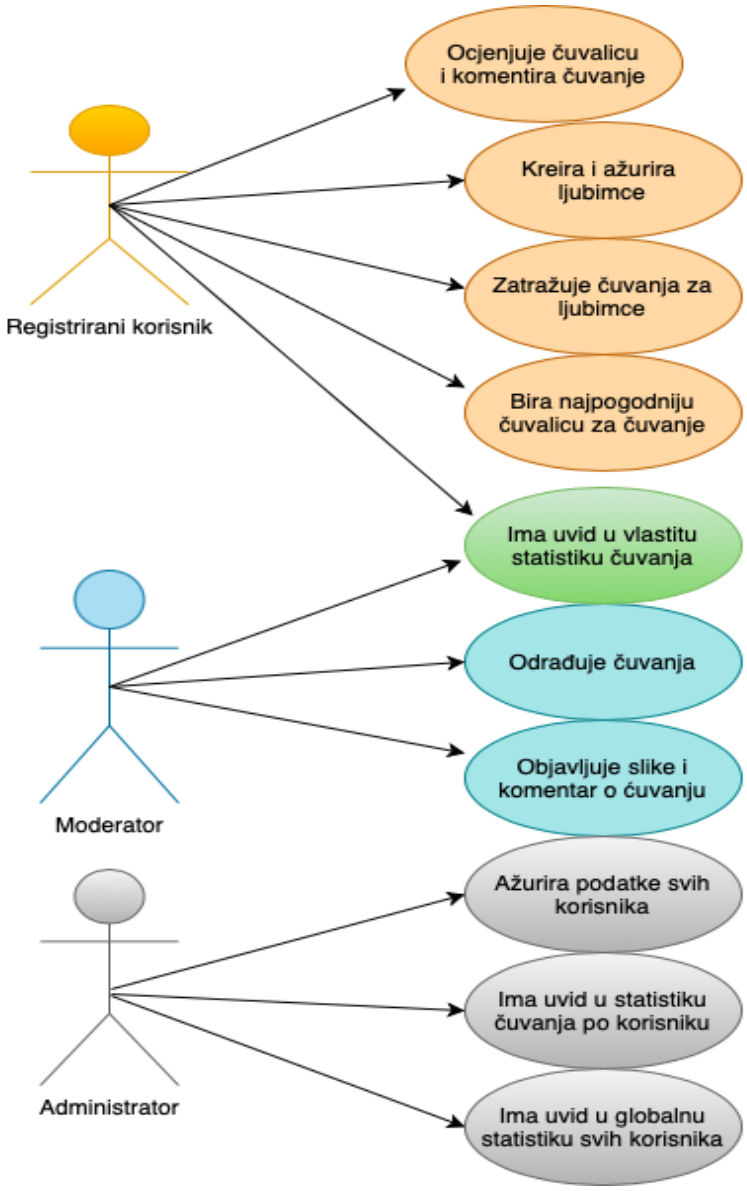
Web aplikacija je zamišljena kao online platforma koja služi za pronalazak čuvalica kućnih ljubimaca. Usluga rješava jedan od najvažnijih problema s kojima se susreću vlasnici kućnih ljubimaca, a to je poteškoća u pronalaženju pouzdane skrbi za svoje ljubimce kada oni nisu u mogućnosti biti prisutni. Kućni ljubimci imaju različite potrebe, a pronalazak skrbnika koji može udovoljiti specifičnim zahtjevima može biti izazov. Aplikacija za čuvanje kućnih ljubimaca nudi platformu na kojoj vlasnici mogu opisati potrebe svojih ljubimaca kao što su raspored hranjenja, medicinski zahtjevi, a pri tom omogućavaju dadiljama pružanje prilagođenih usluga. Personalizirani pristup osigurava da kućni ljubimci dobiju brigu i pažnju koju zaslužuju, što dovodi do sretnijih i zdravijih životinja. Također, uvidom u profil potencijalnih čuvalica, vlasnici mogu na temelju prosjećne ocjene, iskustva i osobnih informacija čuvalica, birati osobu za koju smatraju da bi najviše odgovarala za čuvanje.

Aplikacija sadrži tri vrste korisnika: registrirani korisnik, moderator i administrator. Prva vrsta korisnika su registrirani korisnici koji imaju mogućnost napraviti profil s osobnim informacijama kao i informacijama svog kućnog ljubimca ili više njih. Svoj profil korisnik može ažurirati i deaktivirati, također i profil kućnih ljubimaca. Registrirani korisnik može zatražiti čuvanje na način da ispuni određene informacije vezane za čuvanje koje zatražuje, poput točno vrijeme početka i točno vrijeme završetka čuvanja, grad u kojem se nalazi, opis čuvanja i dodane napomene. Također, može zatražiti čuvanje više ljubimaca u jednom čuvanju. Registrirani korisnik nakon kreiranja određenog čuvanja može birati između više čuvalica koje su se prijavile za čuvanje, te odabrati onu čuvalicu za koju smatra da bi bila najbolja. Kao asistenciju u odabiru ima mogućnost pregleda svih profila čuvalica koje su se prijavile za posao te ima mogućnost pregleda njihovih osobnih informacija, ocjene i količine čuvanja koju je čuvalica odradila...itd. Promjene planova u zadnji tren nikome nisu strane, tako i registrirani korisnik ima mogućnost brisanja zatraženih čuvanja. Nakon što čuvanje završi, registrirani korisnik ima mogućnost ocijeniti čuvalicu i napisati komentar.

Čuvalica, odnosno moderator, ima mogućnost kreiranja i ažuriranja vlastitiog profila s osobnim infomacijama kao i informacijama koliko naplaćuje čuvanje po satu i danu. Ima pregled svih čuvanja koja su zatražena u istom gradu u kojem se i on nalazi. Kod pregleda potencijalnih prijava za čuvanje, moderator može pogledati profil s

detaljnim informacijama o ljubimcima za određeno čuvanje. Moderator šalje zahtjev za čuvanje za ponuđene poslove i čeka odobrenje registriranog korisnika. Nakon završtek čuvanja moderator ima mogućnost objaviti nekoliko slika čuvanja kao i komentar.

Administrator ima mogućnost pretraživanja svih korisnika, ažuriranje i deaktiviranje njihovih profila, pregled statistike vezane za svakog moderatora.

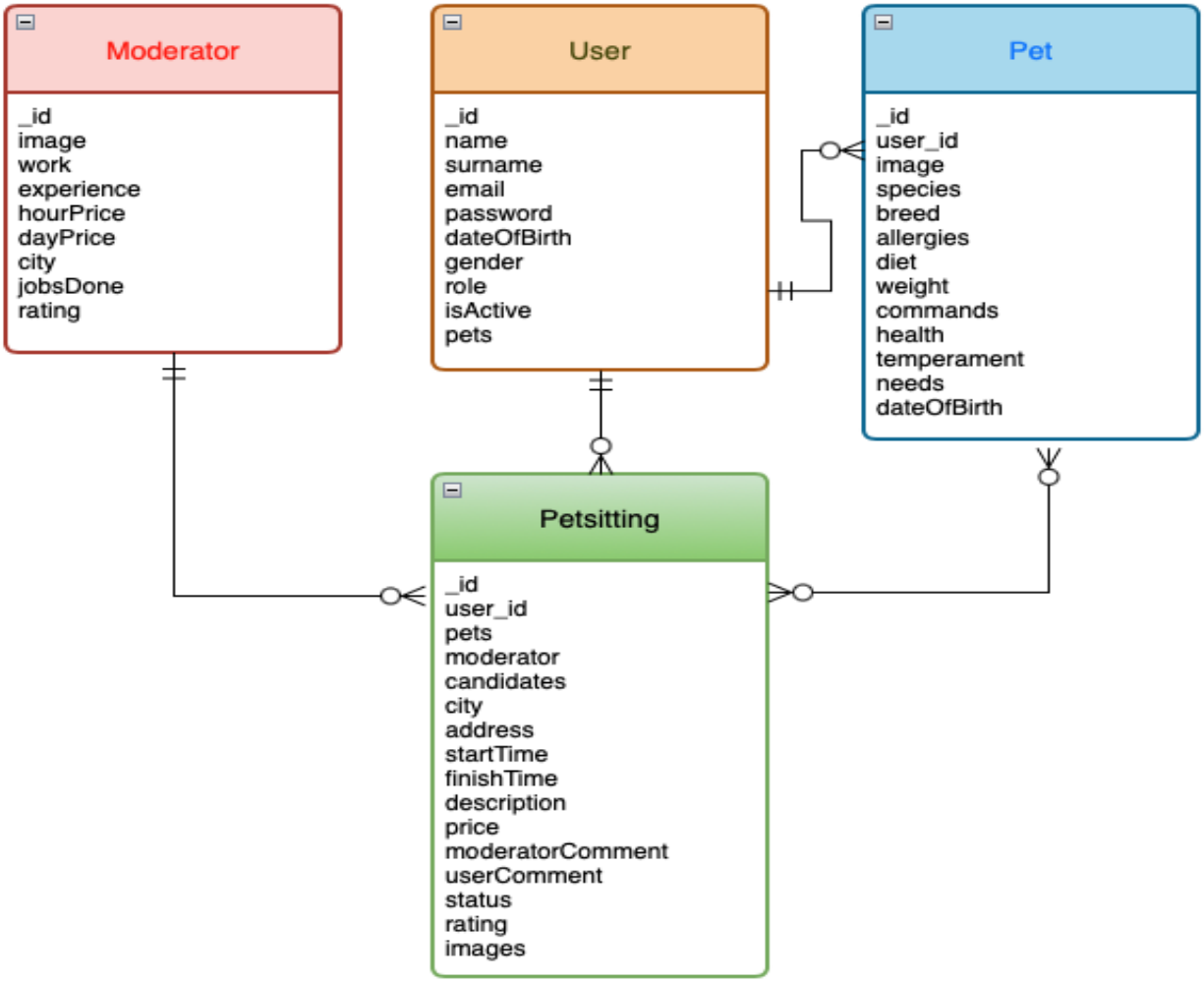


Slika 1 Dijagram slučaja upotrebe [autorski rad]

4.1. ERA Model

ERA model (eng. Entity Relationship Attribute model), koji je konceptualna tehnika modeliranja podataka koja se koristi za dizajn baze podataka. U ERA modelu entiteti predstavljaju objekte iz stvarnog svijeta, odnosi opisuju kako su entiteti

međusobno povezani, a atributi definiraju svojstva ili karakteristike entiteta. Veze, odnosno relacije, prikazuju odnose između objekata. Ograničenja, koje se nazivaju još i kardionalnosti, govore o tome koliko entiteta jednog tipa može biti u vezi s entitetom drugog tipa. Sljedeća slika prikazuje ERA model aplikacije napravljene za potrebe ovog rada, te su opisane veze i kardionalnosti između njih.



Slika 2 ERA dijagram [autorski rad]

Moderator odrađuje Petsitting: 1:n (1 prema više), svaki moderator može održavati više čuvanja.

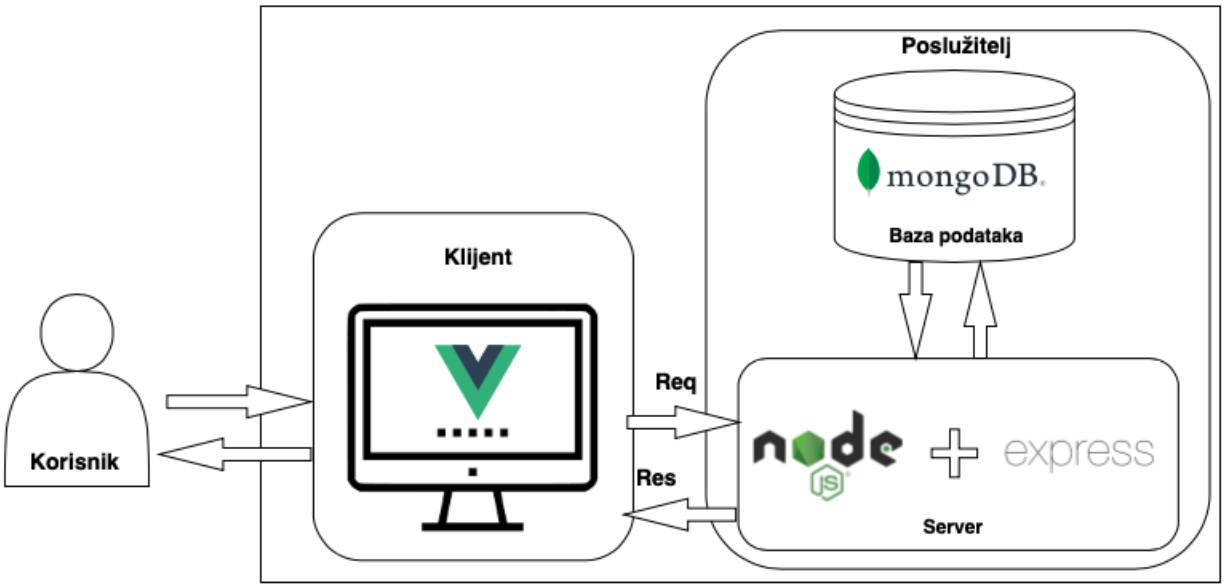
User zatražuje čuvanja i vlasnik je ljubimca. Odnos prema objektu Pet: 1:n (1 prema više), svaki korisnik može imati više kućnih ljubimaca.
Odnos prema objektu Petsitting: 1:n (1 prema više), svaki korsnik može zatražiti više čuvanja.

Objekt Pet prema objektu User, n:1 (više prema jedan), više različitih ljubimaca mogu imati jednog valsnika. Objekt Pet prema objektu Petsitting, n:m (više prema više), više ljubimaca može sudjelovati u više čuvanja.

Objekt Petsitting prema objektu Moderator, n:1 (više prema jedan), više čuvanja mogu imati istog moderatora. Objekt Petsitting prema objektu User, n:1 (više prema jedan), više čuvanje mogu biti zatražena od jednog korisnika. Objekt Petsitting prema objektu Pet n:m (više prema više), više čuvanje mogu imati više ljubimaca.

4.2. Dijagram arhitekture sustava

Dijagram arhitekture sustava, vizualni je prikaz strukture komponenti softverskog sustava ili aplikacije. Dijagrami pružaju grafički pregled načina na koji različiti dijelovi sustava međusobno djeluju i rade zajedno. U nastavku je prikazana slika dijagrama arhitekture sustava koji opisuje aplikaciju napravljenu za potrebe ovog rada i objašnjenje svih djelova.



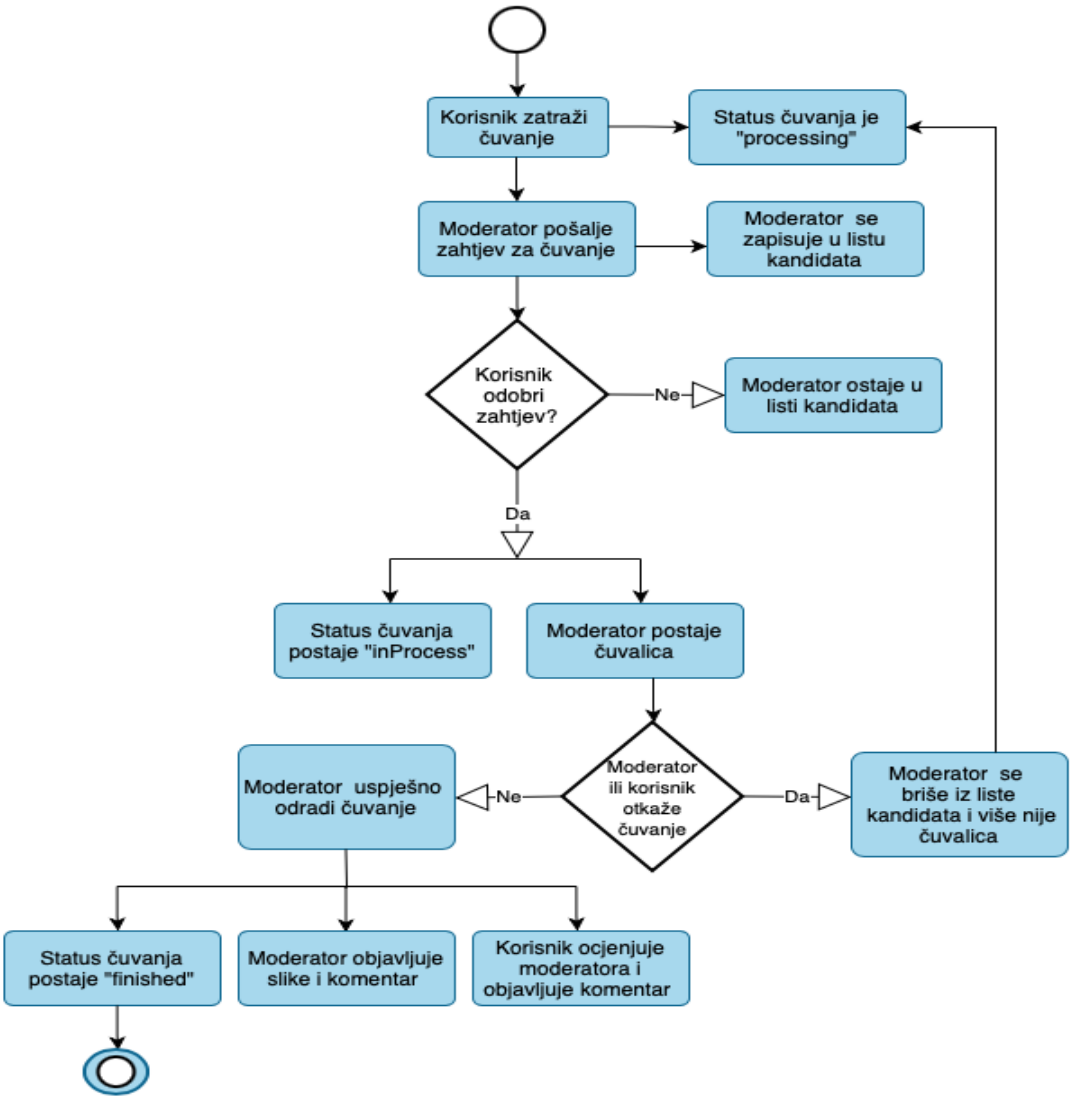
Slika 3 Dijagram arhitekture [autorski rad]

Kada korisnik stupi u interakciju sa sučeljem na strani klijenta, pokreće radnje poput postavljanja zahtjeva, slanja obrazaca ili navigacije kroz aplikaciju. Navedene radnje pokreću HTTP zahtjeve pomoću krajnjih točaka RESTful API-ja poslane s klijenta na poslužitelj. Poslužitelj Express.js, koji radi na Node.js okruženju, prima te zahtjeve i usmjerava ih na odgovarajuće API krajnje točke. Poslužitelj je u interakciji s MongoDB-om kako bi dohvatio ili ažurirao podatke prema potrebi za ispunjavanje

zahtjeva korisnika. Nakon obrade zahtjeva, poslužitelj šalje HTTP odgovor nazad klijentu, u JSON formatu, koji sadrži tražene podatke ili potvrdu poduzete radnje. Aplikacija Vue.js na strani klijenta ažurira korisničko sučelje kako bi odražavalo odgovoru poslužitelja.

4.3. Dijagram aktivnosti

Dijagram aktivnosti vizualni je prikaz koji se koristi u području softverskog inženjerstva za modeliranje dinamičkih aspekata sustava ili procesa. U aplikaciji napravljenoj za potrebe ovog rada, jezgra poslovne logike je zatraživanje i odrađivanje poslova čuvanja kućnih ljubimaca, gdje su glavni čimbenici registrirani korisnik koji zatražuje čuvanja i moderator kojih ih odrađuje. U nastavku je prikazan dijagram aktivnosti za navedene radnje i detaljnije objašnjenje.



Slika 4 Dijagram aktivnosti [autorski rad]

Kada korisnik incijalno zatraži čuvanje polja koja u MongoDB dokumentu popunjava su početak (startTime) i kraj (finishTime) čuvanja, nekon odabira početka i kraja, aplikacija automatski računa koliko je trajanje (duration) gdje se zapisuje dan i sat. Sljedeći atributi koji su ispunjeni kod inicijalnog zatraživanja čuvanja su grad (city), adresa (address), ljubimci (pets), opis (description) i napomena (note). Ispod je prikazano sučelje za Petsitting.

```
interface PetSitting {
 user: ObjectId;
 duration: { day: number; hour: number };
 pets: Pet[];
 moderator: User;
 candidates: User[];
 city: string;
 address: string;
 startTime: string;
 finishTime: string;
 description: string;
 note: string;
 price: number;
 moderatorComment: string;
 userComment: string;
 status: string;
 rating: number;
 images: string[]; }
```

Nakon što je čuvanje zatraženo i podaci su poslani prema poslužitelju, automatski se dodaje status “processing” i u tom trenutku svi moderatori koji se nalaze u istom gradu u kojem je zatraženo čuvanje imaju mogućnost poslati zahtjev za čuvanje. Kada moderator pošalje zahtjev za čuvanje, objekt tog moderatora se sprema u polje “candidates”. Više moderatora može zatražiti zahtjev za isto čuvanje što polje “candidates” čini nizom u koji se spremaju objekti tipa User.

Korisnik koji je zatražio čuvanje ima uvid u sve moderatore koji su poslali zahtjev za čuvanje i ima mogućnost odabira moderatora za kojeg samtra da bi najbolje odgovarao za čuvanje. Kod odabira moderatora njegov objekt se sprema u polje moderator i status posla se mijenja iz “processing” u “inProcess”. Moderator ili korisnik imaju mogućnost prije početka čuvanja otkazati čuvanje, pri čemu se iz baze briše moderatorov objekt iz polja “candidates” i “moderator”, također status posla se vraća na “processing”. Budući da atribut u dokumentu “candidates” sadrži sve moderatore koji su se prijavili za čuvanje, registrirani korisnik može odabrati nekog drugog moderatora.

Ukoliko moderator ne otkaže zahtjev i odradi čuvanje, status posla se mijenja u “finished” i moderator ima mogućnost dodati svoj komentar i polje “moderatorComment” i slike u polje “images”. Kada je čuvanje odrađeno registrirani korisnik također ima mogućnost ocjeniti čuvalicu odnosno moderatora koje se zapisuje u polje “rating” i ostaviti svoj komentar koji se zapisuje u polje “userComment”.

4.4. Baza podataka

Tehnologija korištena za implementaciju baze podataka je MongoDB. Kao što je već rečeno, MongoDB je NoSQL baza podataka koja za razliku od tradicionalnih SQL baza podataka koristi kolekcije umjesto tablice. Feksibilna struktura omogućava lakše skaliranje, upravljanje podacima, brži i fleksibilniji razvoj aplikacije. Budući da je za razvoj aplikacije korišten i Node.js, za modeliranje MongoDB objekata koristit će se paket mongoose.

Imajuću na umu poslovnu logiku i sve sudionike u procesu čuvanja kućnih ljubimaca, napravljeni su svi potrebni modeli za aplikaciju. Prvo su napravljeni modeli sudionika u procesu čuvanja, a to su registrirani korisnik, moderator i administrator.

Uzevši u obzir veličinu modela sa svim poljima i korištenje TypeScripta za tipiziranje shema, u nastavku će biti prikazani svi tipovi podataka koji se pohranjuju u bazu, u obliku sučelja. Svako sučelje odgovara pojedinačnom dokumentu u MongoDB kolekciji.

Registrirani korisnik i moderator dijele isti model koji se sastoji od sljedećih atributa: ime, prezime, email, lozinka, spol, datum rođenja, uloga, je li korisnik aktivan, kućni ljubimci i moderator. Način na koji se registrirani korisnik razlikuju je vrijednost atributa uloga (*eng. role*). Registrirani korisnik ima ulogu “user”, a moderator “moderator”. Osim vrijednosti atributa uloga, registrirani korisnik može dodavati kućne ljubimce u atribut “pets” koji predstavlja niz tipa “Pet” što znači da registrirani korisnik može dodati više od jednog ljubimca. S druge strane, moderator ne može dodavati ljubimce, ali može napraviti moderator profil na način da ispuni dodane podatke koji se mogu vidjeti u moderator sučelju. Kako bi odmah bilo jasno koju ulogu korisnik ima, u aplikaciji je napravljena logika da nakon registracije, kod prijave, korisnik mora odabrati svoju ulogu i, ovisno o odabranoj ulozi, ispunjava informacije o svom ljubimcu ili informacije vezane za profil moderatora. Nakon odabira uloge i ispunjavanja potrebnih

informacija, prilikom spremanja u bazu automatski se zapisuje uloga korisnika u atribut “role” i tek tada je omogućeno daljnje korištenje aplikacije. U primjeru ispod prikazana su sučelje korisnik (*eng. User*), ljubimac (*eng. Pet*), moderator.

```
interface User {
  _id: string;
  name: string;
  surname: string;
  email: string;
  password: string;
  dateOfBirth: Date;
  gender: string;
  role: string;
  isActive: boolean;
  moderator: Moderator;
  pets: Pet[];
}
```

```
interface Pet {
  user: ObjectId;
  image: string;
  name: string;
  species: string;
  breed: string;
  allergies: string;
  diet: string;
  weight: string;
  commands: string;
  health: string;
  temperament: string;
  needs: string;
  other: string;
  dateOfBirth: Date;
  gender: string;
}
```

```
interface Moderator {
  userId: ObjectId;
  image: string;
  work: string;
  experience: string;
  hourPrice: number;
  dayPrice: number;
  city: string;
  jobsDone: number;
  rating: number;
}
```

4.4.1. Mongoose međuprogram

Korištenje međuprograma (*eng. middlewarea*) za izvođenje radnji prije spremanja podataka u bazu podataka značajka je koju pruža Mongoose. To je praktičan način implementacije radnji koje bi se trebale poduzeti prije pohranjivanja podataka. U primjeru ispod prikazan je međuprograma koji prije spremanja podataka provjerava je li lozinka promijenjena. Ako je lozinka uistinu promjenjena, koristi se bcrypt algoritam za kriptiranje lozinke i sprema ju u bazu.

```
UserSchema.pre<User>("save", async function (next) {
  if (!this.isModified("password")) {
    return next();
  }

  const hash = await bcrypt.hash(this.password, 10);
  this.password = hash;
  next();
});
```

4.5. Implementacija poslužitelja

Kao što je već ranije rečeno, tehnologije za implementaciju poslužitelja su Express.js i okruženje koje mu omogućuje rad, odnosno Node.js. Arhitektura je bazirana na REST servisu, što znači da se koristi HTTP protokol za komunikaciju između klijenata i poslužitelja. Također je već spomenuta parktičnost i benefiti korištenja međuprograma u Express.js aplikaciji. U sljedećim poglavljima biti će

prikazani značajniji primjeri upotrebe međuprograma u aplikaciji napravljenoj za potrebe ovog rada.

4.5.1. Autentifikacija korisnika

Jedna od najčešćih implementacija međuprograma je implementacija u svrhe autentifikacije korisnika prilikom prijave u aplikaciju ili prilikom pristupanja nekom API-ju odnosno resursu, prilikom čega je bitno provjeriti ovlasti korisnika. U kontekstu autentifikacije, često se primjenjuje JWT (JSON Web Token) kao popularan mehanizam za sigurnu provjeru identiteta i autorizaciju korisnika. JWT omogućava aplikacijama da generiraju token koji sadrži korisničke podatke i digitalni potpis, što omogućava sigurno prenošenje i verifikaciju korisnika putem HTTP zahtjeva.

Kako bi koristili JWT u Express.js aplikaciji prvo ga je potrebno instalirati naredbom `npm install jsonwebtoken`, zatim kreirati funkcije za kreiranje i verifikaciju JWT-a prikazane u nastavku.

```
export const createToken = (user: User): string => {
  return jwt.sign({ id: user._id }, process.env.JWT_SECRET as jwt.Secret,
{
    expiresIn: "1h",
  });
      };


export const verifyToken = async (
  token: string
): Promise<jwt.VerifyErrors | Token> => {
  return new Promise((resolve, reject) => {
    jwt.verify(token, process.env.JWT_SECRET as jwt.Secret, (err,
payload) => {
      if (err) return reject(err);

      resolve(payload as Token);
    });
  });
};

export default { createToken, verifyToken };
```

Funkcija za kreiranje tokena koristi se kod prijave korisnika, a funkcija za verifikaciju kod međuprograma za autentifikaciju korisnika kada pristupaju nekom API-ju. Također, u funkciji za kreiranje tokena specificirano je da valjanost tokena ističe nakon jednog sata. Nakon što su funkcije za kreiranje i provjeru ispravnosti JWT tokena kreirane, može se napraviti međuprogram za autentifikaciju.

```
async function authenticatedMiddleware(
  req: Request,
  res: Response,
  next: NextFunction
): Promise<Response | void> {
  const bearer = req.headers.authorization;
  if (!bearer || !bearer.startsWith("Bearer ")) {
    return next(new HttpException(401, "Unauthorised"));
  }

  const accessToken = bearer.split("Bearer ")[1].trim();

  try {
    const payload: Token | jwt.JsonWebTokenError = await
      verifyToken( accessToken );

    if (payload instanceof jwt.JsonWebTokenError) {
      return next(new HttpException(401, "Unauthorised"));
    }

    const user = await
    UserModel.findById(payload.id).select("password").exec();

    if (!user) {
      return next(new HttpException(401, "Unauthorised"));
    }

    req.user = user;

    return next();
  } catch (error) {
    return next(new HttpException(401, "Unauthorised"));
  }
}

export default authenticatedMiddleware;
```

Funkcije međuprograma su funkcije koje mogu presresti i obraditi zahtjeve prije nego što dođu do konačnog rukovatelja rutom. U sljedećem primjeru međuprogram `authenticatedMiddleware` je implementiran na način da se izvršava prije glavnog rukovatelja zahtjevom odnosno funkcije `this.getPetsitting`, koja ima za zadatak dohvatiti čuvanja. Međuprogram, `authenticatedMiddleware` je odgovoran za provjeru dolazi li zahtjev od autentificiranog korisnika.

```
this.router.get(
      `api/petsitting/get/:role/:status/:id`,
      authenticatedMiddleware,
      this.getPetsitting
);
```

4.5.2. Spremanje dokumenata

Učinkovito upravljanje i pohrana dokumenata, posebno slika, čine temelj modernih aplikacija. Bez obzira izrađuje li se društvena mreža, platformu za e-trgovinu koja prikazuje proizvode putem slika ili bilo koja aplikacija koja se pokreće sadržajem koji generiraju korisnici, metode za pohranjivanje dokumenata su različite.

U aplikaciji napravljenoj za potrebe ovog rada korištena je motoda pohranjivanja reference slike odnosno, naziva slike u MongoDB dok su stvarne slike spremljene na poslužitelju u posebnom dokumentu napravljenom određeno za tu svrhu. U navedenom procesu korišten je Multer međuprograma, koji je opisan ranije u poglavlju 3.2.1.Node.js biblioteke. Ovaj hibridni pristup spremanja dokumenata, gdje se u bazu sprema jedinstveni naziv dok je stvarni dokument spremljen na poslužitelj, uravnotežuje snagu baze podataka kao što je MongoDB za upravljanje metapodacima s prednostima izvedbe i skalabilnosti pohrane slika na strani poslužitelja.

Za korištenje Multer međuprograma u Node.js/Express.js aplikaciji prvo ga je potrebno instalirati komandom `npm install multer`, zatim je potrebno napraviti konfiguracijski dokument sa željenim opcijama poput; specificiranje putanje datoteke gdje će se spremati dokumenti i postavljanje konvencije kod imenovanja datoteka. U aplikaciji napravljenoj za potrebe ovog rada napravljena je datoteka pod nazivom “images” gdje će se spremati slike, a za imenovanje slika odabrana je popularna metoda kojom se, prije spremanja slike, stavlja trenutni datum i vrijeme kako bi se osiguralo da svaka slika ima jedinstveni naziv. Na sljedećem primjeru prikazana je konfiguracija Multer međuprograma.

```
import multer from "multer";

const fileStorageEngine = multer.diskStorage({
  destination: (req, file, cb) => {
    cb(null, "./images");
  },
  filename: (req, file, cb) => {
    cb(null, Date.now() + "--" + file.originalname);
  },
});

const upload = multer({ storage: fileStorageEngine });

export default upload;
```

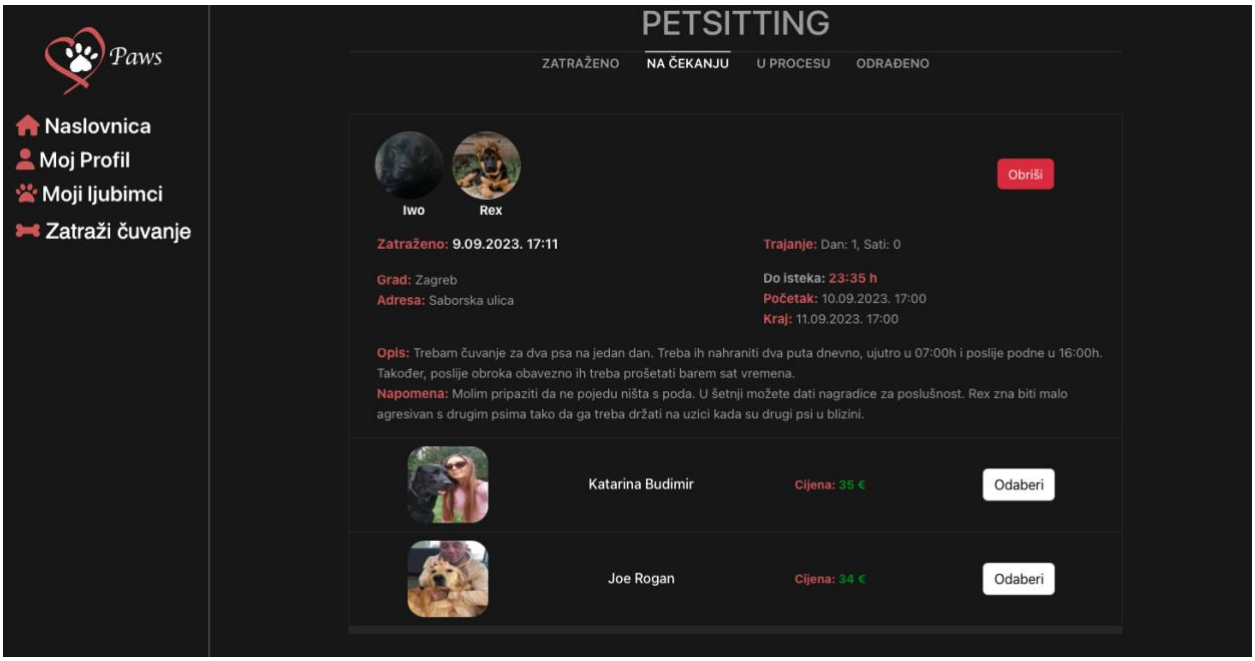
U nastavku je prikazano korištenje Multer međuprograma na API-ju za ažuriranje čuvanja, gdje je na istanci Multer međuprograma `upload`, specificirano

funkcijom `.array("images", 4)`. Prvi argument navodi naziv polja u HTML obrascu za koje se očekuje da će sadržavati učitane datoteke. U ovom slučaju traži polje za unos s nazivom "`images`" u obrascu. Drugi argument specificira koliko Multer međuprogram može maksimalno prihvatiti dokumenata, u ovom slučaju je to 4 dokumenta.

```
import upload from "@/middleware/multer.middleware.js";
this.router.put(
      `api/petsitting/update/:role/:status/:id`,
       authenticatedMiddleware,
       upload.array("images", 4),
       this.update
    );
```

4.6. Implementacija klijenta

Klijentska strana ove aplikacije razvijena je korištenjem Vue.js okvira treće verzije. Uz Vue.js, Bootstrap je korišten za poboljšanje korisničkog sučelja i dizajna aplikacije. Stil i vizualni elementi prilagođeni su pomoću CSS/SCSS, osiguravajući vizualno privlačno i jednostavno iskustvo za korisnike aplikacije. U nastavku ovog poglavlja biti će prikazane najbitniji djelovi razvoja klijentske strane aplikacije pomoću Vue.js.



Slika 5 Aplikacija - Kandidati koji su se prijavili za čuvanje [autorski rad]

4.6.1. Navigation Guards

Vue.js nudi moćnu značajku poznatu kao "Navigation Guards" (*hrv. Zaštitari Navigacije*). Navigation Guards su funkcije koje programerima omogućuju kontrolu toka navigacije unutar aplikacije. Nude detaljan pristup upravljanju prijelazima ruta, omogućujući programerima izvršavanje prilagođene logike prije, tijekom ili nakon promjena rute. Također, omogućuju programerima da provedu autentifikaciju, izvrše dohvaćanje podataka i obrađuju uvjete specifične za rutu, poboljšavajući cjelokupno korisničko iskustvo i sigurnost Vue.js aplikacije.

U izradi aplikacije, funkcije koje pruža Navigation Guard su korištene za autentifikaciju i autorizaciju korisnika, kontrolu kojoj ruti korisnik može pristupiti i dohvaćanje podataka korisnika s poslužitelja kako bi sve navedene akcije i provjere mogle biti moguće. U primjeru ispod prikazana je Navigation Guard `.beforeEach` funkcija koja se izvršava prije učitavanja svih ruta osim ruta pod nazivom „login“ i „register“ jer tamo nije potrebno. Funkcija služi kao provjera odnosno autentifikacija korisnika na način da provjerava je li JWT token spremljen u lokalnom spremištu i, ako je, je li istekao, budući da je na poslužitelju u funkciji koja kreira JWT specificirano da traje jedan sat. Ukoliko token nije dostupan ili je istekao, Navigation Guard automatski šalje korisnika na stranicu prijave.

```
router.beforeEach(async (to, from, next) => {
  const token = localStorage.getItem("token");
  const tokenExpiration = localStorage.getItem("tokenExpiration");
  const present = Date.now().toString();

  if (
    to.name !== "login" &&
    to.name !== "register" &&
    token &&
    tokenExpiration &&
    tokenExpiration < present
  ) {
    next({ name: "login" });
  } else {
    next();
  }
});
```

U sljedećem primjeru biti će prikazana autorizacija korisnika na klijentskoj strani koristeći meta podatke rute i Navigation Guards `.beforeEach` funkciju.

Budući da aplikacija ima više različitih korisnika, svaka vrsta korisnika ima svoju ulogu, što znači da je potrebno na temelju uloge korisnika definirati kojim resursima korisnik ima pristup. U nastavku je prikazana ruta „pets“ koja ima definirane meta

podatke „premissions“ u kojima je specificirano da samo korisnik s ulogom „user“ može pristupiti toj ruti.

```
const router = createRouter({
  history: createWebHistory(import.meta.env.BASE_URL),
  routes: [
    {
      path: "/pets",
      name: "pets",
      component: () => import("../views/Pets.vue"),
      meta: {
        permissions: ["user"],
      }
    }]
   });
```

Nakon toga napravljena je `.beforeEach` funkcija koja dohvaća dozvole rute kojoj se pristupa. Nakon toga dohvaćaju se podaci korisnika gdje se nalazi atribut „role“, odnosno uloga. U aplikaciji je napravljena logika da korisnik nakon registracije ima ulogu „undefined“, a nakon prijave u aplikaciju mora odabrati ulogu „user“ ili „moderator“. Sve dok korisnik ne odabere ulogu i ispuni potrebne podatke za tu ulogu, nije mu dozvoljen pristup niti jednoj drugoj ruti.

Ukoliko je korisnik odabrao ulogu „user“ ili „moderator“, provjerava se nalazi li se njegova uloga u dopuštenjima za rutu kojoj želi pristupiti, ukoliko da, Navigation Guard dopušta učitavanje rute, ukoliko ne, korisnik se prosljeđuje na rutu „NotFound“.

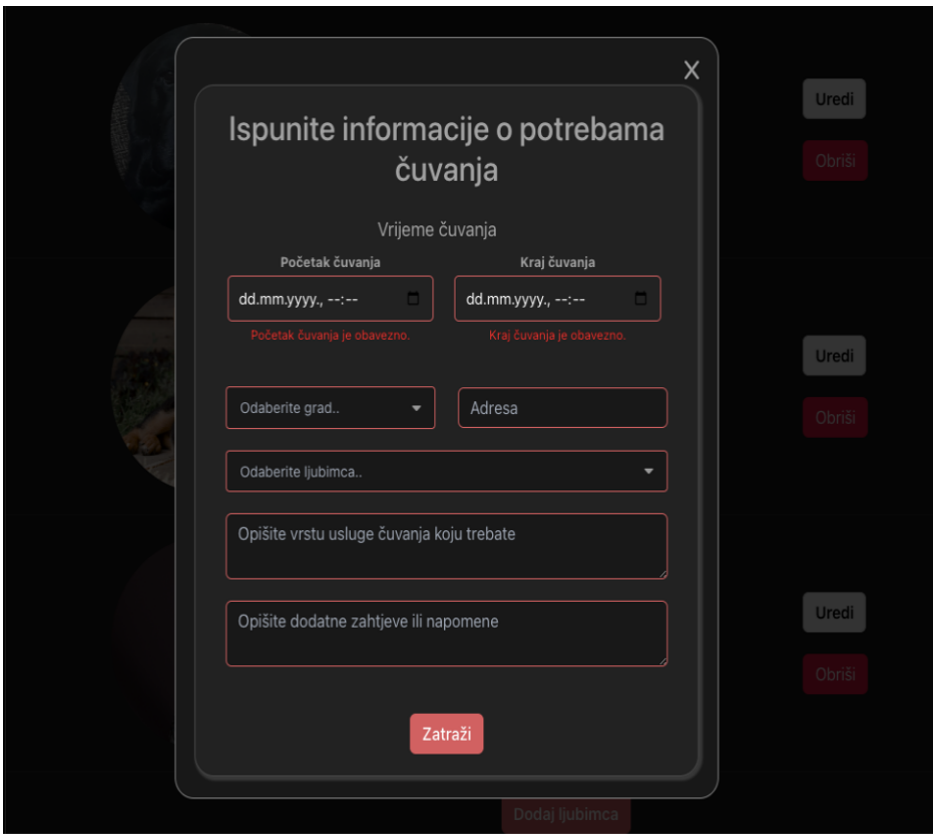
```
router.beforeEach(async (to, from, next) => {
  const permissions = to.meta.permissions as string[];

  if (permissions) {
    const userStore = useUserStore();
    await userStore.getUser();

    const role = userStore.userData?.role;

    if (to.name === "setup" && role === "undefined") {
      next();
    } else if (role === undefined || role === "undefined") {
      next({ name: "setup" });
    } else if (permissions.includes(role)) {
      next();
    } else {
      next({ name: "NotFound" });
    }
  } else {
    next();
  }
});
```

4.6.2. Modal komponenta

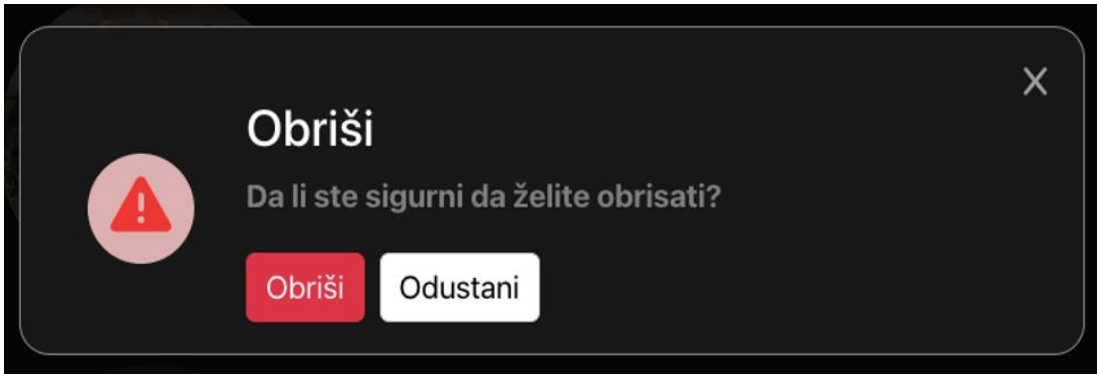
Jedno od ključnih načela koje pokreće učinkovitost i mogućnost održavanja u razvoju Vue.js aplikacija je praksa ponovne upotrebe komponenti. Neke od prednosti koje nudi su:

1. Učinkovitost koda: Komponente za višekratnu upotrebu smanjuju redundantnost, što dovodi do sažetijeg koda koji se može održavati. Programeri mogu izraditi i testirati komponente jednom, a zatim ih koristiti u različitim dijelovima aplikacije.
2. Dosljednost: Komponente osiguravaju dosljedan dizajn i korisničko iskustvo u cijeloj aplikaciji. Dosljednost poboljšava ukupnu upotrebljivost i estetiku.
3. Ušteda vremena: Iskorištavanjem postojećih komponenti, programeri mogu značajno smanjiti vrijeme razvoja što omogućuje brži završetak projekta.

Jedna od takvih komponenata je "Modal". Modal je definiran unutar Pinia dućana i njegova svrha je da omogući integraciju različitih komponenata unutar skočnih prozora. Modal komponenta se koristi za situacije gdje su potrebne interakcije korisnika, kao što su potvrda brisanja, potvrda slanje podataka iz ispunjenih obrazaca i slično.



Slika 6 Korištenje forme unutar Modal komponente [autorski rad]



Slika 7 Korištenje komponente za povrdu brisanja unutar Modal komponente [autorski rad]

Modal dućan sadrži funkciju „open“ koja kao prvi argument prima komponentu koju želimo prikazati i kao drugi opcionalni parametar podatke koje želimo prikazati unutar komponente.

```
import { markRaw } from "vue";
import { defineStore } from "pinia";
import type User from "@/interfaces/UserInterface";
import type Pet from "@/interfaces/PetInterface";

export const useModal = defineStore("modal", {
  state: (): Modal => ({
    isOpen: false,
    isConfirmed: false,
    view: {},
    data: null,
  }),
  actions: {
    async open(view: object, data?: User | Pet): Promise<Modal> {
      return new Promise((resolve) => {
        if (data) {
          this.data = data;
        }
        this.isOpen = true;
        this.isConfirmed = false;
        this.view = markRaw(view);

        const unsubscribe = this.$subscribe(() => {
          if (this.isConfirmed === true) {
            resolve(this.$state);
            unsubscribe();
          }
        });
      });
    },
    close() {
      this.isOpen = false;
      this.view = {};
    },
    confirm() {
      this.isConfirmed = true;
      this.isOpen = false;
    },
  },
```

```
});

export default useModal;
```

4.7. Statistika

Aplikacija za čuvanje kućnih ljubimaca nije platforma koja odgovara svima, zato je potrebno prikupljati podatke o korisnicima kako bi na temelju tih podataka poboljšali korisničko iskustvo aplikacije i pružili sveobuhvatno bolji proizvod i uslugu.

Statistika omogućuje ovakvom tipu aplikacije prikupljanje vrijednih podataka o ponašanju i preferencijama korisnika. Analizom ovih podataka, programeri mogu prilagoditi značajke, preporuke i usluge aplikacije tako da odgovaraju individualnim potrebama korisnika. U nastavku će biti prikazana statistika koja je napravljena u aplikaciju u sklopu ovog rada.

Proces započinjemo na poslužitelju, prikupljajući sve potrebne podatke i prilagođavanje formata tih podataka kako bi ih bilo što lakše prikazati na klijentskoj strani aplikacije. Kao primjer biti će prikazana statistika za moderatora.

```
public async getModeratorStats(userId: string): Promise<object | void> {
    try {
      const petsittings = await this.petsitting
        .find({
          status: "finished",
          moderator: userId,
        })
        .sort({ startTime: 1 })
        .populate({ path: "pets" });

      let totalJobs = 0;
      let totalMoney = 0;
      let speciesCount = [];
      const monthlyMoney = [];
      const monthlyJobs = [];

      petsittings.forEach((ps) => {
        totalMoney += parseInt(ps.price.toString());
        totalJobs++;

        // Broj tipa čuvanih ljubimaca
        ps.pets.forEach((pet) => {
          const key = pet.species;
          const existingSpecies = speciesCount.find((item) => item.key
=== key);
          if (existingSpecies) {
            existingSpecies.value++;
          } else {
            speciesCount.push({ key, value: 1 });
          }
        });
```

```
        const startTime = new Date(ps.startTime);
        const year = startTime.getFullYear();
        const monthName = startTime.toLocaleString("hr", { month: "long"
}).toUpperCase();
        const key = `${monthName}, ${year}`;

        // Mjesečna zarada
        const existingMonth = monthlyMoney.find((item) => item.key ===
key);
        if (existingMonth) {
          existingMonth.value += parseInt(ps.price.toString());
        } else {
          monthlyMoney.push({
            key,
            value: parseInt(ps.price.toString()),
          });
        }

        // Broj odrađenih poslova po mjesecu
        const existingMonths = monthlyJobs.find((item) => item.key ===
key);
        if (existingMonths) {
          existingMonths.value++;
        } else {
          monthlyJobs.push({
            key,
            value: 1,
          });
        }
      });
```

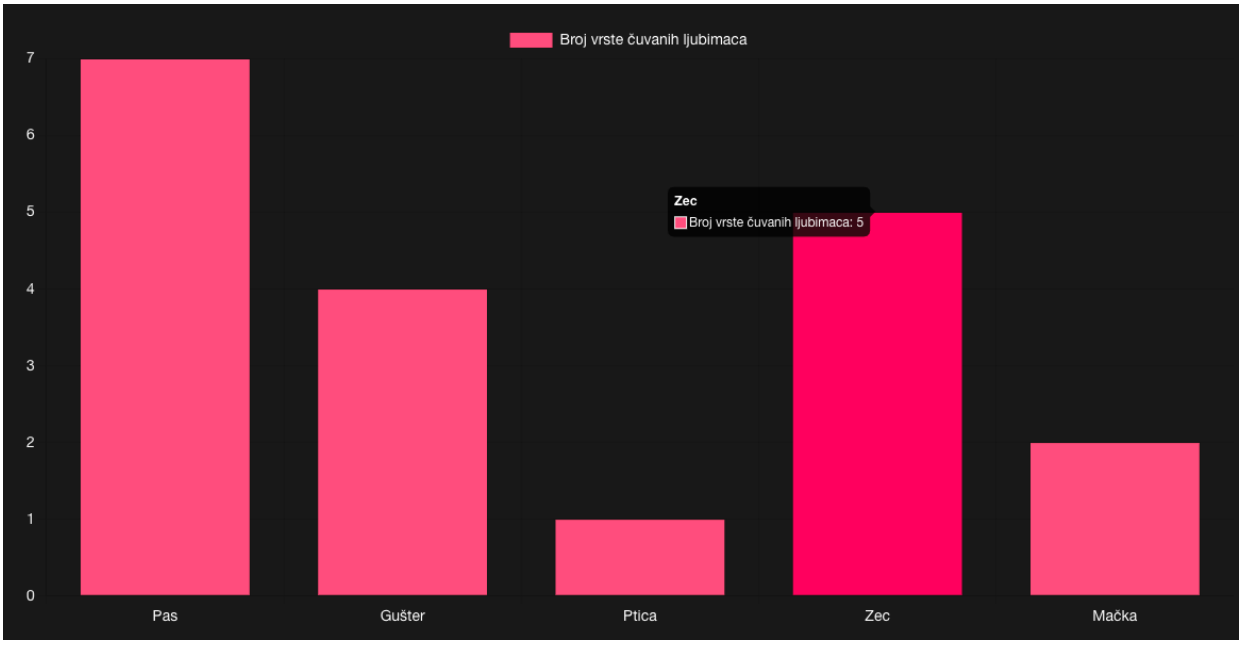
Nakon što su podaci prikupljeni, klijent dohvaća te podatke na način da pristupa API-ju na kojem se nalazi statistika. Dohvaćeni podaci se spremaju u Pinia dućan.

```
export const useModeratorStore = defineStore("ModeratorStore", {
  state: (): {
    stats: Stats | null;
  } => ({
    stats: null
  }),
  actions: {
    async getStats(id: string) {
      const token = localStorage.getItem("token");
      const headers = {
        Authorization: `Bearer ${token}`,
      };

      try {
        const response: AxiosResponse = await axios.get(
          `http://localhost:3000/api/moderator/stats/${id}`,
          { headers }
        );

        this.stats = response.data["data"];
      } catch (error) {
        const axiosError = error as AxiosError;
        console.log(axiosError.message);
```

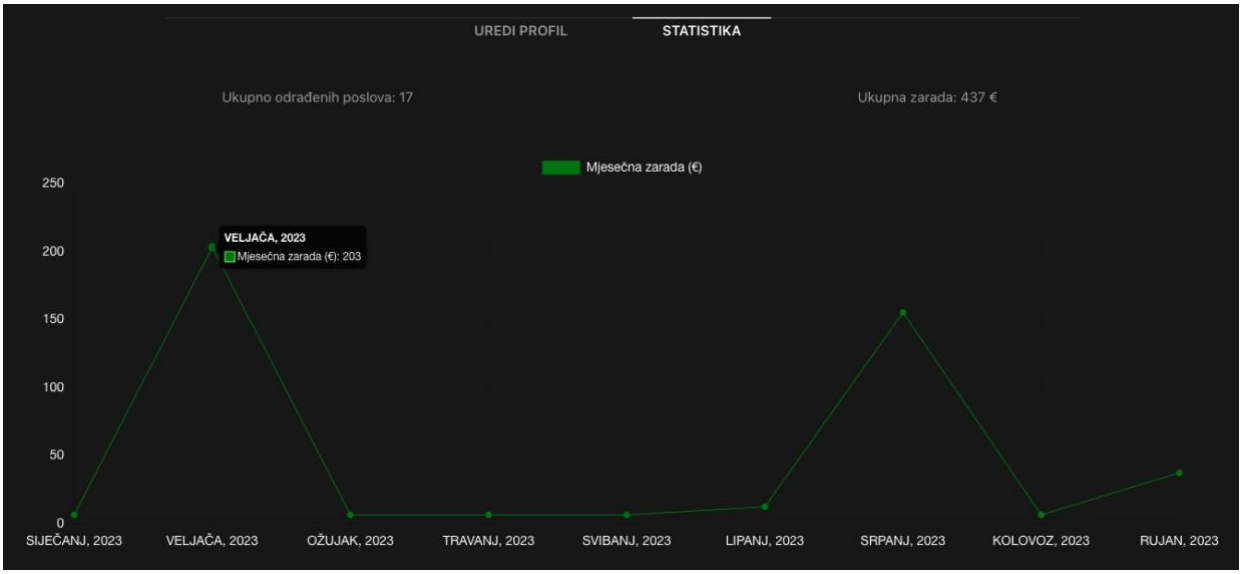
```
      }
    },
  },
});
```

Dohvaćeni podaci su spremni za prikaz. Radi boljeg vizualnog korisničkog iskustva i lakšeg pregleda statističkih podataka, podaci će biti prikazani u grafovima, koristeći JavaScript biblioteku Chart.js.

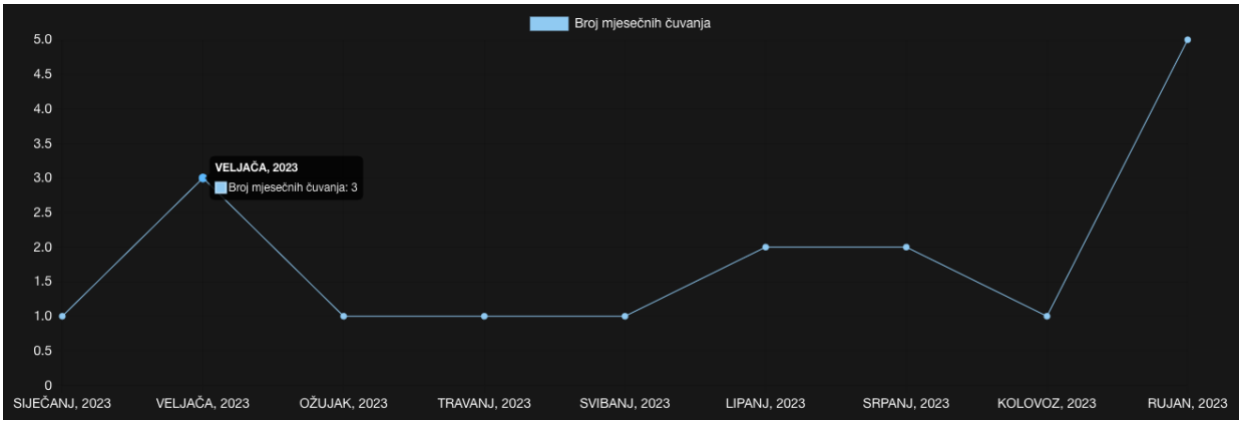
```
<script setup lang="ts">
import { ref, onMounted } from "vue";
import BarChart from "../charts/BarChart.vue";
import LineChart from "../charts/LineChart.vue";
import { useModeratorStore } from "@/stores/ModeratorStore";
import { useUserStore } from "@/stores/UserStore";

const modStore = useModeratorStore();
const userStore = useUserStore();
const barStats = ref([]);
const lineStats = ref([]);

onMounted(async () => {
  if (userStore.userData.role === "moderator") {
    await modStore.getStats(userStore.userData._id!);

    barStats.value.push(modStore.stats?.speciesCount);

    lineStats.value.push(
      modStore.stats?.monthlyMoney,
      modStore.stats?.monthlyJobs
    );
  }
});
</script>
<template>
  <div class="container">
    <div class="row mb-5">
      <div class="col-6">
        Ukupno odrađenih poslova: {{ modStore.stats?.totalJobs }}
      </div>
      <div class="col-6">Ukupna zarada: {{ modStore.stats?.totalMoney }}
€</div>
    </div>
    <div class="row charts-wrap">
      <div class="col-12" v-if="lineStats" v-for="(value, index) in
lineStats">
        <LineChart :stats="value" :index="index" />
      </div>
      <div class="col-12" v-if="barStats" v-for="(value, index) in
barStats">
        <BarChart :stats="value" :index="index" />
      </div>
    </div>
  </div>
</template>
```



Slika 8 Prikaz statistike - broj vrste čuvanih ljubimaca [autorski rad]



Slika 9 Prikaz statistike - mjesečna zarada [autorski rad]



Slika 10 Prikaz statistike - broj odrađenih čuvanja po mjesecu [autorski rad]

5. Zaključak

Istraživanje razvoja aplikacija korištenjem MEVN (MongoDB, Express.js, Vue.js, Node.js) paketa tehnologija prikazan je ogroman potencijal i svestranost ovog tehnološkog paketa tehnologija. Kroz ovaj dokument, dubinski su istraženi različiti aspekti razvoja MEVN aplikacija, od postavljanja razvojnog okruženja do stvaranja robusnih i skalabilnih aplikacija. Jedna od istaknutih prednosti MEVN skupa je njegova sposobnost pružanja besprijekornog i dosljednog razvojnog iskustva tijekom cijelog razvoja. MongoDB nudi fleksibilnu NoSQL bazu podataka, Express.js pojednostavljuje logiku na strani poslužitelja, Vue.js pruža snažan i intuitivan front-end okvir, a Node.js omogućuje skriptiranje na strani poslužitelja i mogućnosti u stvarnom vremenu. Zajedno, ove tehnologije stvaraju dobro zaokružen JavaScript ekosustav za izgradnju modernih web aplikacija. Dodatno, MEVN paketi promiču korištenje JavaScripta u cijelosti, olakšavajući razvojnim programerima rad i na klijentskoj i na poslužiteljskoj strani aplikacije, što pojednostavljuje proces razvoja.

Nadalje, istražili smo važnost RESTful API-ja, provjere autentifikacije i upravljanja stanjem u razvoju MEVN aplikacija. Ovi su koncepti temeljni u stvaranju sigurnih, korisniku prilagođenih i responzivnih web aplikacija. Savladavanjem ovih elemenata, programeri mogu kreirati web aplikacije koje zadovoljavaju potrebe i očekivanja današnjih korisnika. MEVN paketi također obuhvaćaju koncept full-stack razvoja, omogućujući programerima sveobuhvatno razumijevanje cjelokupne arhitekture aplikacija. MEVN paketi su moćan i relevantan izbor za razvoj modernih web aplikacija. Njihova fleksibilnost, izvedba i prilagođenost programerima čine ga idealnim izborom za širok raspon projekata. Kako se tehnologija nastavlja razvijati, MEVN paket tehnologija je dobro pozicioniran da se prilagodi i ostane vrijedan alat u rukama programera.

Literatura

[1] Melanie Harris (2019). Comparing Popular Web Development Stacks: MERN vs. MEVN vs. LAMP vs. Django. Pristupljeno 13.07.2023 s https://medium.com/@Melanie_Harris/lets-talk-stacks-and-i-don-t-mean-money-web-development-stacks-explained-9bea22603ed0

[2] Ankita Kapoor (2021). MERN vs LAMP. Preuzeto 13.07.2023. s https://enlear.academy/mern-vs-lamp-f0653b0dc96a

[3] Emma Jhonson (2022) LAMP Stack vs. MEAN Stack: Which Web Development Stack is Right for You? Preuzeto 13.07.2023 s https://medium.com/nerd-for-tech/lamp-stack-vs-mean-stack-which-web-development-stack-is-right-for-you-ce8cfc22e282

[4] David Flanagan (2011) E-book. JavaScript: The Definitive Guide: https://pepa.holla.cz/wp-content/uploads/2016/08/JavaScript-The-Definitive-Guide-6th-Edition.pdf

[5] Eric Goebelbecker (2022). 9 popular JavaScript frameworks (and how to choose one for your project). Preuzeto 14.07.2023. s https://raygun.com/blog/popular-javascript-frameworks/

[6] Ashiq KS (2009). Working with MongoDB. Preuzeto 20.07.2023. https://medium.com/@ashiqgiga07/working-with-mongodb-afee14557a92

[7] Ryan Dahl' (2009). Node.js Introduction and Motivation JSConf EU prezentacija. Dostupno 07.08.2023 https://www.youtube.com/watch?v=ztspvPYybIY

[8] Node.js Event Loop (službena Node.js dokumentacija) (bez. Dat.). Preuzeto 02.08.2023. s https://nodejs.org/en/docs/guides/event-loop-timers-and-nexttick

[9] Getting Started with Express.js (2023). Preuzeto 04.08.2023. s https://developer.mozilla.org/en-US/docs/Learn/Server-side/Express_Nodejs

[10] Vue.js službena dokumentacija. Preuzeto 07.08.2023 s https://vuejs.org/guide/introduction.html

[11] D. Svjetličić (18.10.2019.) Nuxt.js over Vue.js: when should you use it and why. Preuzeto 07.08.2023 s https://www.bornfight.com/blog/nuxt-js-over-vue-js-when-should-you-use-it-and-why/

Popis slika

6. Prilozi

Uz ovaj završni rad, prilažem i ZIP datoteku s kodom aplikacije.