Razvoj web aplikacije za rent a boat servis

Stašić, Eugen

Sveučilište Jurja Dobrile u Puli
Tehnički fakultet u Puli

EUGEN STAŠIĆ

RAZVOJ WEB APLIKACIJE ZA RENT ZA BOAT SERVIS

Završni rad

Pula, Veljača, 2024 godine

Sveučilište Jurja Dobrile u Puli
Tehnički fakultet u Puli

EUGEN STAŠIĆ

RAZVOJ WEB APLIKACIJE ZA RENT A BOAT SERVIS

Završni rad

JMB: 0081147725, redoviti student

Studijski smjer: Računarstvo

Predmet: Inženjerska grafika i konstruiranje

Znanstveno područje: Tehničke znanosti
Znanstveno polje: Računarstvo
Mentor: doc. dr. sc. tech. Marko Kršulja

Pula, Veljača, 2024 godine

Sažetak

Ovaj rad fokusira se na razvoj web aplikacije namijenjene za servis *rent-a-boat*, što je segment tržišta koji bilježi rastuću potražnju u Hrvatskoj. U suvremenom digitalnom svijetu, web aplikacije postaju neizostavan dio poslovanja, posebno u turističkom sektoru gdje je online prisutnost ključna za uspjeh. Cilj rada je razviti web aplikaciju koja će se istaknuti na tržištu svojom funkcionalnošću i fokusom na korisničko iskustvo, te na taj način biti konkurentna postojećim rješenjima.

Razvoj aplikacije obuhvaća korištenje modernih tehnologija i pristupa. Za izradu *frontend* dijela aplikacije odabrana je tehnologija *React.js*, jedan od najpopularnijih JavaScript alata za izradu korisničkih sučelja. React.js ističe se svojom fleksibilnošću, pogotovo kada se koristi u kombinaciji sa *Redux-om*, koji omogućava efikasno upravljanje stanjem aplikacije. Takav pristup omogućava brzu i dinamičnu interakciju s korisnicima te pruža izuzetno korisničko iskustvo.

Backend aplikacije izrađen je korištenjem Node.js, *event-driven* platforme koja omogućava izgradnju skalabilnih i efikasnih web aplikacija. U kombinaciji s Express.js, minimalističkim i fleksibilnim web aplikacijskim okvirom, stvara se robusna struktura koja podržava zahtjeve i funkcionalnosti potrebne za *rent-a-boat* servis. *MongoDB*, kao odabrana baza podataka, nudi visoke performanse, laku skalabilnost i fleksibilnost, što je idealno za upravljanje složenim podacima za ovakve vrste aplikacija.

Kroz rad, detaljno se objašnjavaju sve korištene tehnologije i metodologije razvoja, uz poseban naglasak na integraciju *frontend* i *backend* dijelova.

Sadržaj

1. Uvod

1.1. Hipoteza

U ovom završnom radu, fokus stavljen je na razvoj web aplikacije za *rent-a-boat* servisa, segment tržišta koji u Hrvatskoj doživljava rastuću potražnju. Cilj je stvoriti aplikaciju koja će se na tržištu istaknuti svojom funkcionalnošću i izvanrednim korisničkim iskustvom, čime će postati konkurentna alternativa postojećim digitalnim rješenjima. Aplikacija koristi napredne tehnologije, uključujući React.js za izradu interaktivnog korisničkog sučelja, podržan *Redux-om* za efikasno upravljanje stanjem aplikacije. Ovaj pristup doprinosi dinamičkoj interakciji s korisnicima. *Backend* je razvijen koristeći Node.js platformu, u kombinaciji s Express.js, što rezultira snažnom i fleksibilnom strukturom koja podržava sve funkcionalnosti potrebne za *rent-a-boat* servis. Kao baza podataka, odabran je *MongoDB*, zbog svoje visoke performanse i fleksibilnosti, idealan za upravljanje složenim podacima u ovakvoj vrsti aplikacije. Rad detaljno objašnjava korištene tehnologije i metodologije razvoja, s naglaskom na integraciju *frontend* i *backend* dijelova aplikacije.

1.2. Predmet istraživanja

Predmet istraživanja temelji se na dubinskoj analizi kako skup odabranih tehnologija može pridonijeti razvoju efikasnog i skalabilnog web sustava za najam plovila. Korištenje tehnologija kao što su *React.js*, *Node.js*, *Express.js* i *MongoDB* ključno je za postizanje ciljeva projekta. Fokus istraživanja usmjeren je na razumijevanje kako svaka od ovih tehnologija doprinosi kreiranju koherentnog korisničkog putovanja, počevši od trenutka kada korisnik prvi put posjeti platformu, preko procesa rezervacije, pa sve do konačne realizacije usluge.

1.3. Problem istraživanja

Problem istraživanja u ovom radu usmjerava se na usklađivanje niza softverskih rješenja kako bi se stvorilo sučelje za korisnike koje odgovara specifičnostima najma plovila. Izazov leži u integraciji modernih tehnoloških rješenja, kao što su React.js za *frontend* i *Node.js*, *Express.js* te *MongoDB* za *backend*, kako bi se osiguralo intuitivno i ergonomski prihvatljivo korisničko iskustvo. U radu se objašnjava kako primjena ovih tehnologija može doprinijeti ne samo poboljšanju korisničkog iskustva, već i operativnoj učinkovitosti web aplikacije, ključnoj za uspjeh u segmentu *rent-a-boat* usluga. Ovaj pristup zahtijeva pažljivo balansiranje između tehničke izvedbe i korisničkih potreba, te stavlja naglasak na dizajniranje sučelja koje je jednostavno za korištenje, ali i dovoljno robustno da podrži različite funkcionalnosti potrebne za gospodarsku aktivnost najma plovila.

1.4. Ciljevi istraživanja:

- Projektiranje web aplikacije
- Primjena *backend* tehnologija
- Primjena *frontend* tehnologija
- Optimizacija web aplikacije

1.5. Metodologija istraživanja

Metodologija istraživanja u ovom završnom radu temelji se na agilnom razvojnom modelu koji pruža esencijalnu fleksibilnost i prilagodljivost kroz cijeli proces razvoja. UML dijagrami igraju ključnu ulogu u preciznom modeliranju i dokumentiranju arhitekture sustava, olakšavajući razumijevanje strukture i funkcionalnosti aplikacije te osiguravajući jasnu komunikaciju unutar razvojnog tima. *REST* arhitektura je primijenjena za dizajniranje *API-ja*, omogućavajući efikasan razvoj, testiranje i održavanje aplikacije. *RESTful* pristup pruža standardiziran i intuitivan način za izgradnju i interakciju s *API-jem*, poboljšavajući integraciju i skalabilnost. Dodatno, korištenje *Reduxa* kao rješenja za upravljanje stanjem unutar *React* okruženja omogućava centralizirano i predvidljivo upravljanje stanjem aplikacije. Ova metodologija olakšava razdvajanje logike i sučelja, što pojednostavljuje razvoj, testiranje i održavanje aplikacije.

1.6. Struktura rada

U ovom radu detaljno se opisuju tehnologije i metodologije korištene tijekom razvoja web aplikacije za najam plovila. Struktura rada osmišljena je tako da jasno predstavi sve faze razvoja aplikacije, od konceptualizacije do implementacije.
U prvom dijelu rada fokus je na konceptualizaciji *Rent-a-Boat* web aplikacije. Ovdje se razmatraju hipoteza i ciljevi projekta, predmet istraživanja, te metodologija koja se koristi. Također, predstavljen je i prototip web aplikacije, uključujući njegovu strukturu i UML dijagram.

Dalje, rad se bavi tehnologijama i razvojnim okruženjima koji su ključni za realizaciju projekta. *Frontend* tehnologije poput *React.js* i *Reduxa* koriste se za razvoj dinamičkog i interaktivnog korisničkog sučelja, dok se za *backend* koriste *Node.js*, *Express.js* i *MongoDB*, koji zajedno čine snažnu platformu za upravljanje podacima i poslovnom logikom aplikacije.

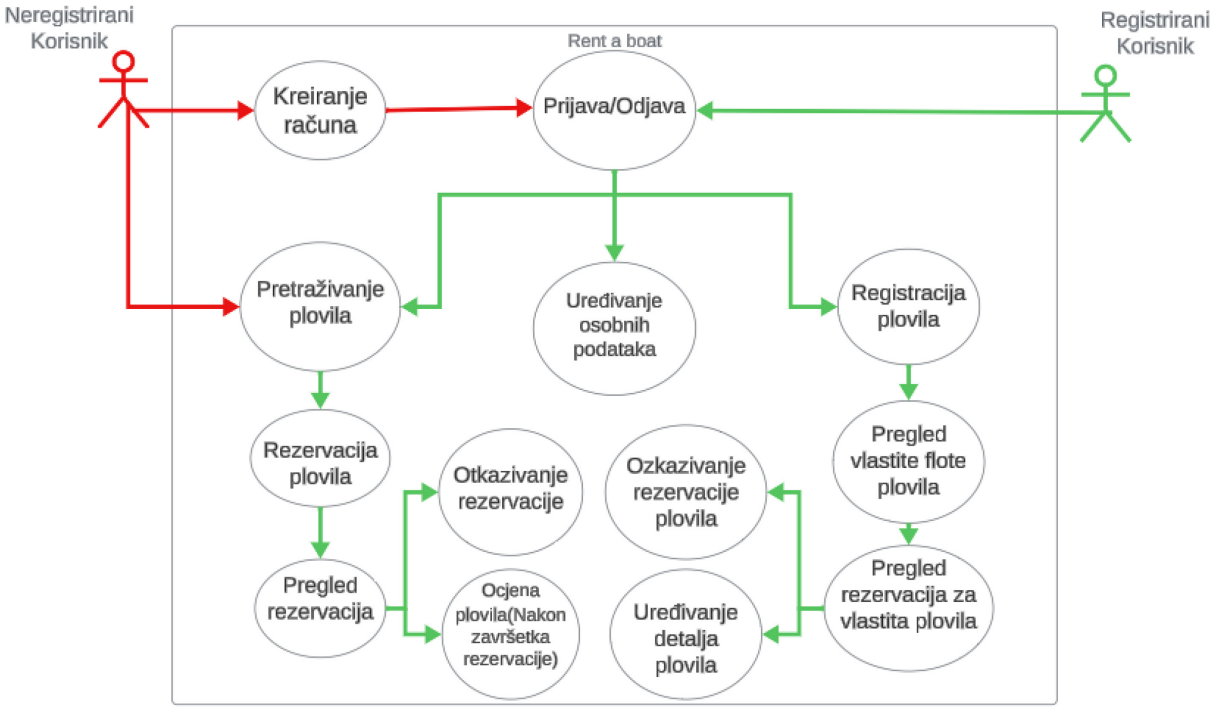
Poseban naglasak stavljen je na detaljan opis *backend* i *frontend* dijelova aplikacije. U poglavlju o *backendu* objašnjavaju se ključni aspekti kao što su upravljanje paketima, konfiguracija, server.js datoteka, *middleware*, modeli, i rute. U dijelu o *frontendu* opisani su *Redux, actions, reducers, services*, te komponente i stranice koje čine korisničko sučelje aplikacije.

2. Konceptualizacija Rent-A-Boat Web Aplikacije

2.1. UML Dijagram i Struktura Aplikacije

UML dijagram prikazuje arhitektonsku strukturu aplikacije, uključujući ključne komponente i njihove interakcije. Detaljno objašnjenje dijagrama pruža uvid u logičku organizaciju sustava, odnos entiteta i procesa koji omogućuju *rent-a-boat* transakcije. Dijagram je podijeljen u tri glavna segmenta koji odražavaju različite korisničke uloge i njihove interakcije s aplikacijom: Neregistrirani Korisnik, Rent a *boat* (aplikacija) i Registrirani Korisnik.

Slika 1 UML Dijagram



Izvor: Obrada autora

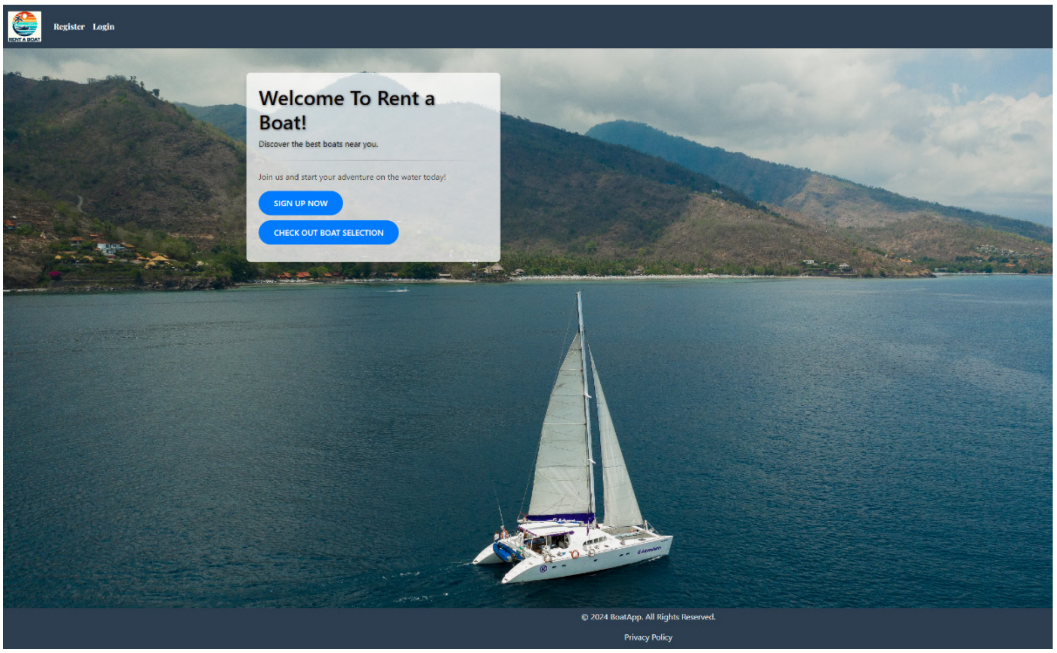
- Neregistrirani Korisnik: Ova kategorija korisnika odnosi se na one koji još nisu stvorili račun unutar aplikacije. Njihov put započinje opcijom "Kreiranje računa", koja je osnovni korak prema pristupu dodatnim funkcionalnostima. Neregistrirani korisnik također može pregledavati ponudu plovila, ali nije u mogućnosti koristiti ostale značajke web aplikacije.
- Registrirani Korisnik: Registrirani korisnici imaju pristup širem spektru funkcionalnosti. Osim sposobnosti pregledavanja i upravljanja vlastitim rezervacijama i plovilima, oni također imaju pristup funkcionalnostima kao što su "Pregled vlastite flote plovila" i "Pregled rezervacija za vlastita plovila"

2.2. Prototip Web Aplikacije

Sljedeći odjeljak ilustrira prototipiranu verziju "*Rent a Boat*" aplikacije, s fokusom na intuitivno korisničko sučelje koje reflektira funkcionalnost i estetiku.

2.2.1. Početna Stranica

Slika 2 Početna stranica

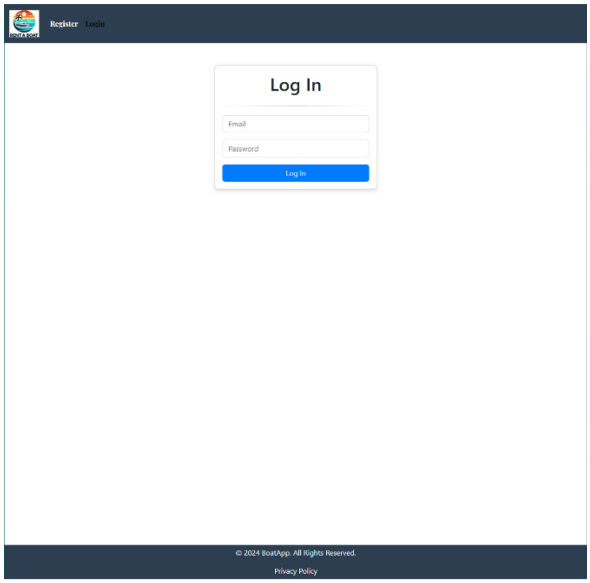


Izvor: Obrada autora

Početna stranica web aplikacije "*Rent a Boat*" prva je točka interakcije između korisnika i platforme. . Na slici se nalazi brod, more i planine u zaleđu što je i cilj teme tj. izrada web aplikacije za najam brodova. Dizajnirana je da odmah privuče pažnju i prenese osnovnu vrijednost usluge a to je najam brodova, ova stranica kombinira vizualnu privlačnost(more, plava boja, zaleđe, brdo, zelena boja) s jasnoćom poruke(najam plovila) kako bi ohrabrila korisnike na daljnje istraživanje i korištenje aplikacije. U gornjem lijevom kutu nalaze se gumbi za registraciju i prijavu, koji su istaknuti kako bi se novim korisnicima omogućio brz pristup procesu stvaranja računa ili prijave na već postojeći.

2.2.2. Prijava/Registracija

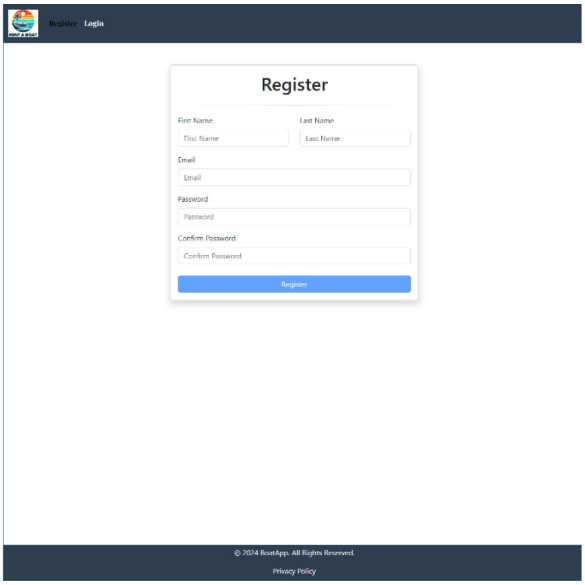
Slika 3 Stranica za prijavu



Izvor: Obrada autora

Na stranici za prijavu, korisnici su dočekani s čistim i nenametljivim sučeljem koje naglašava esencijalne elemente: polja za unos email adrese i lozinke te gumb za prijavu.

Slika 4 Stranica za registraciju

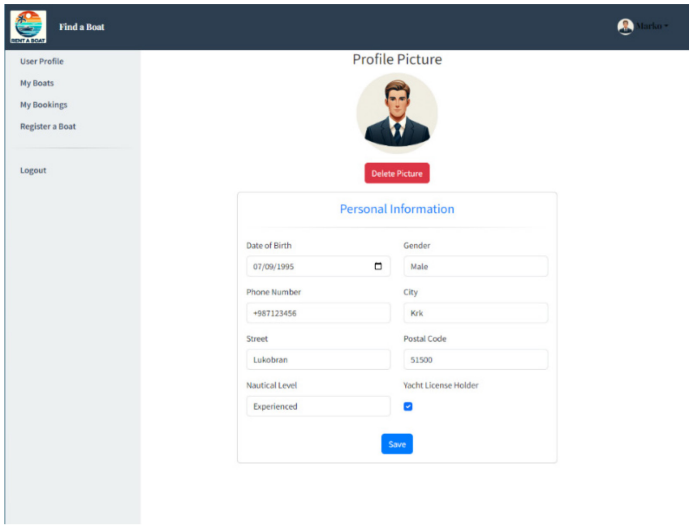


Izvor: Obrada autora

Stranica za registraciju nastavlja ovu temu jednostavnosti, nudeći jasno strukturiran obrazac s poljima za ime, prezime, email, lozinku i potvrdu lozinke. Svaka akcija na ovoj stranici je intuitivna, vodeći nove korisnike kroz proces stvaranja računa korak po korak, bez nepotrebnih komplikacija.

2.2.3. Korisnička Upravljačka Ploča i Pregled Rezervacija

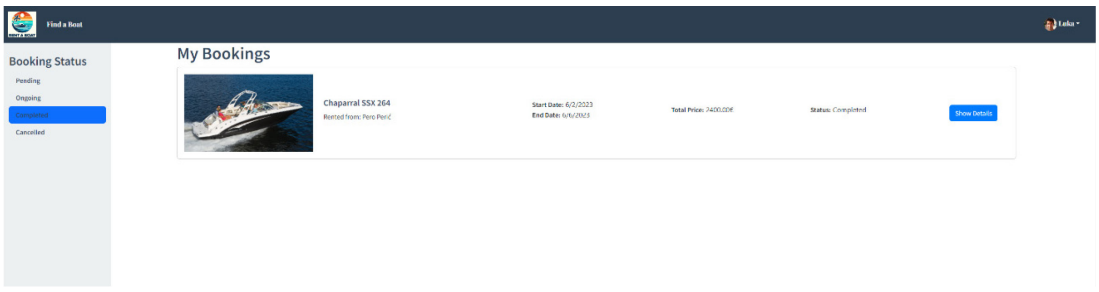
Slika 5 Upravljačka ploča



Izvor: Obrada autora

Na ovoj stranici, korisnici imaju mogućnost pregleda i uređivanja svojih informacija, što pridonosi osjećaju kontrole i sigurnosti unutar aplikacije. Lijevi bočni izbornik nudi brz pristup različitim sekcijama korisničke ploče, uključujući profil, plovila, rezervacije i opciju za registraciju plovila. Također, tu je i gumb za odjavu, što korisnicima omogućuje brzo i jednostavno napuštanje svojeg računa. Centralni dio stranice je posvećen uređivanju profila, gdje se korisnicima prikazuje forma za ažuriranje osobnih informacija kao što su datum rođenja, kontakt broj, adresa i nautička razina iskustva. Osim toga, korisnici mogu upravljati svojom profilnom slikom, s mogućnošću dodavanja ili brisanja slike.

Slika 6 Stranica MyBookings

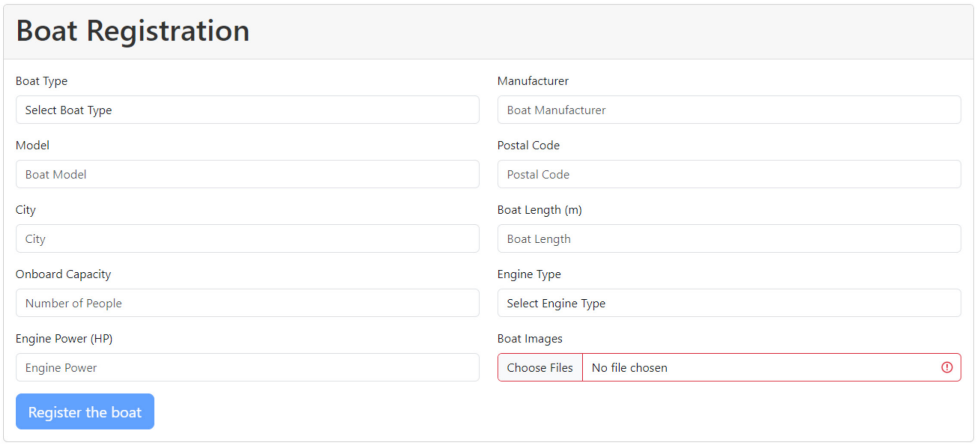


Izvor: Obrada autora

Stranica "My Bookings" u aplikaciji "Rent a Boat" omogućava korisnicima pregled i upravljanje svojim rezervacijama. Ova stranica je dizajnirana kako bi korisnicima pružila brz i organiziran pregled njihovih aktivnih, nadolazećih i prošlih najmova plovila.

2.2.4. Registracija i Pregled Plovila

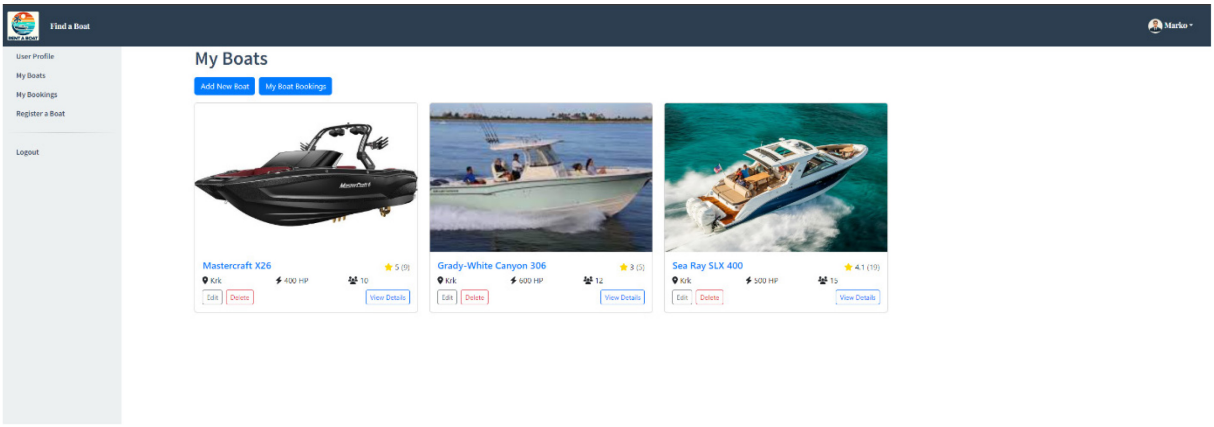
Slika 7 Forma za registraciju plovila



Izvor: Obrada autora

Forma za registraciju uključuje sve relevantne podatke koji su potrebni za registraciju plovila, uključujući tip plovila, proizvođača, model, grad u kojem se plovilo nalazi, poštanski broj, duljinu, tip motora, snagu motora, kapacitet osoba na brodu, kao i mogućnost dodavanja slika plovila. Ovakav pristup osigurava da se potencijalni najmoprimci mogu informirati o svim aspektima plovila prije nego što odluče izvršiti rezervaciju.

Slika 8 Stranica "Moja Plovila"

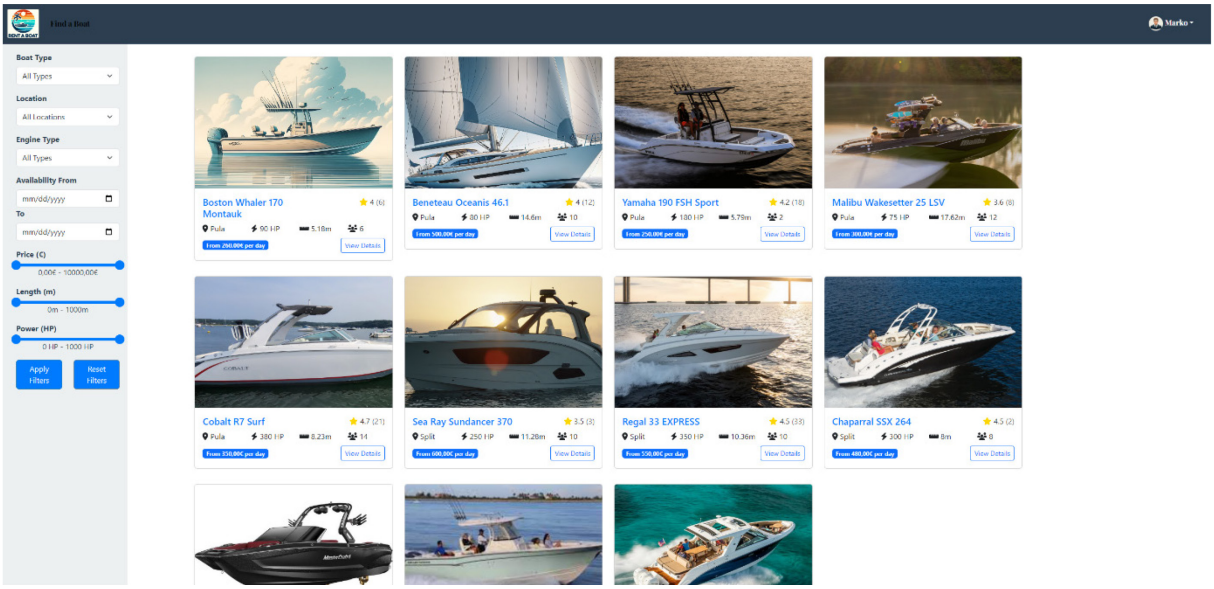


Izvor: Obrada autora

Na slici stranice "*My Boats*", prikazane su kartice plovila s osnovnim informacijama kao što su ime plovila, lokacija, snaga motora, broj raspoloživih mjesta, kao i korisničke ocjene. Ovo omogućava korisnicima brz uvid u karakteristike svakog plovila, kao i trenutnu reputaciju istih na platformi. Svako plovilo ima pridružene akcije poput "*Edit*" i "*Delete*" koje vlasniku plovila omogućavaju da izvrši potrebne izmjene ili ukloni plovilo s liste.

2.2.5. Pretraga Ponude Plovila

Slika 9 Stranica za pretraživanje plovila

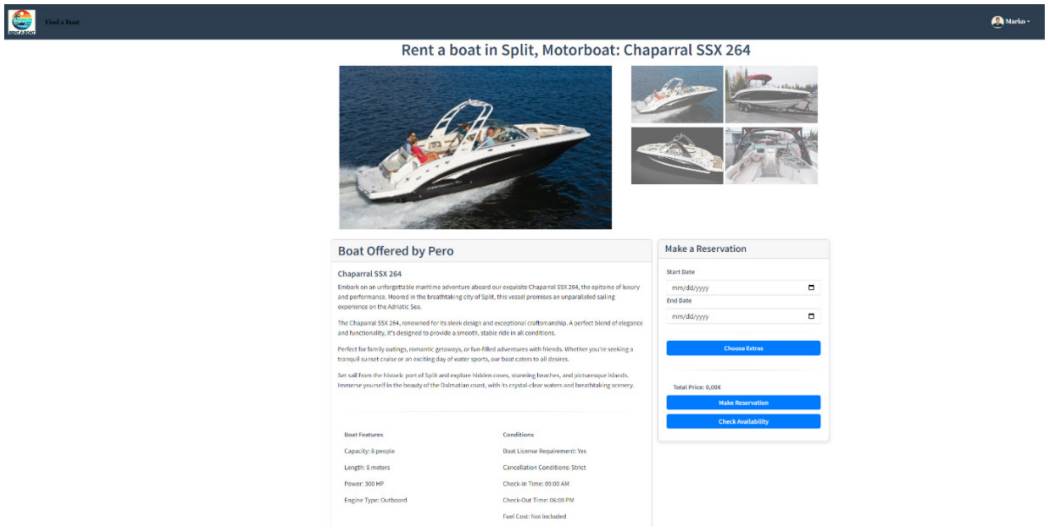


Izvor: Obrada autora

Stranica za pretragu u "*Rent a Boat*" aplikaciji ključni je resurs za korisnike koji traže idealno plovilo za najam. Sa složenim filterima i korisničkim sučeljem koje je jednostavno za korištenje, ova stranica omogućuje korisnicima da pronađu plovila koja najbolje odgovaraju njihovim potrebama. Korisnici imaju na raspolaganju set filtera koji omogućavaju usmjeravanje pretrage prema tipu plovila, lokaciji, tipu motora, raspoloživosti, cijeni, dužini i snazi motora.

2.2.6. Oglas Plovila

Slika 10 Oglas plovila



Izvor: Obrada autora

Na prikazanoj slici, stranica je pažljivo organizirana kako bi predstavila sve važne informacije o plovilu, uključujući naslov s lokacijom i tipom plovila, kao i galeriju slika koja pruža vizualni prikaz plovila. Središnji dio stranice sadrži opis koji vlasnik plovila pruža, dajući potencijalnim najmoprimcima uvid u jedinstvene karakteristike i mogućnosti koje plovilo nudi. S desne strane, korisnici mogu pronaći interaktivnu formu za rezervaciju koja uključuje kalendare za odabir datuma početka i završetka najma, kao i gumb za provjeru dostupnosti. Također, korisnici mogu vidjeti ukupnu cijenu najma i, ako je potrebno, odabrati dodatne usluge.

Slika 11 Recenzije

Izvor: Obrada autora

Sekcija za recenzije na stranici pojedinog plovila u "*Rent a Boat*" aplikaciji pruža korisnicima mogućnost da pročitaju iskustva prethodnih najmoprimaca. Kako je prikazano na slici, recenzije su lako vidljive na dnu stranice i uključuju ocjenu u zvjezdicama, ime recenzenta, datum recenzije i tekstualni opis iskustva.

3. Korištene Tehnologije i Razvojna Okruženja

U ovom poglavlju fokusiramo se na tehnologije i razvojna okruženja koja su korištena u izradi web aplikacije za *rent-a-boat* servis. Odabir tehnologija temelji se na njihovoj efikasnosti, fleksibilnosti, i sposobnosti da podrže razvoj visokokvalitetnih web rješenja. Predstavljene tehnologije uključuju *Visual Studio Code (VS Code)* kao razvojno okruženje, *React.js* i *Redux* za *frontend*, te *Node.js, Express.js,* i *MongoDB* za *backend* aplikacije.

3.1. Razvojno Okruženje

Visual Studio Code (VS Code) predstavlja vrhunac razvojnih okruženja, nudeći sveobuhvatne alate za razvoj i *debugiranje*, što ga čini idealnim izborom za složene projekte kao što je ovaj. Njegova popularnost među programerima proizlazi iz jednostavnosti korištenja, podrške za mnoge programerske jezike i okvire, te bogate biblioteke dodataka koji značajno poboljšavaju produktivnost.

Jedna od ključnih prednosti *VS Code-a* u ovom projektu je njegova integracija s Gitom. Ova funkcionalnost omogućava lako praćenje promjena i upravljanje verzijama koda, što je posebno korisno u projektima gdje više suradnika radi na istom kodu. Integracija s Gitom omogućava vizualno označavanje promjena i pregled povijesti izmjena.

Dodatno*, VS Code* pruža podršku za JavaScript, što je neophodno za razvoj aplikacije koristeći React.js i Node.js. Uz to, postoji niz specifičnih alata i dodataka za te tehnologije koji su se koristili tijekom razvoja. Jedan od takvih dodataka je '*Prettier*', popularni alat za formatiranje koda. *Prettier* automatski formatira kod kako bi bio čitljiviji i u skladu s općeprihvaćenim stilskim pravilima, što olakšava održavanje koda.

Korištenje *VS Code-a* u ovom projektu nije samo povećalo efikasnost i produktivnost u razvoju, već je također osiguralo da je kod visokokvalitetan, dosljedan i lako održiv. Ukratko, *VS Code* je bio nezamjenjiv alat u razvoju ove web aplikacije, pružajući sve potrebne resurse i alate za učinkovit i uspješan razvojni proces.

3.2. Frontend tehnologije

Kombinacija *React.js-a* i *Reduxa* omogućila je stvaranje robusne, efikasne i lako proširive frontend arhitekture za web aplikaciju *rent-a-boat* servisa. React.js je korišten za izgradnju dinamičkih komponenata poput prikaza dostupnih plovila i formi za rezervaciju, dok je *Redux* upravljao stanjima ovih komponenata, omogućavajući korisnicima glatko pregledavanje i interakciju s aplikacijom.

3.2.1. React.js

React.js, razvijen od strane Facebooka, je deklarativna, efikasna i fleksibilna JavaScript biblioteka za izgradnju korisničkih sučelja. Njegova primjena posebno dolazi do izražaja u velikim aplikacijama gdje se podaci često mijenjaju. Jedan od ključnih aspekata React.js-a koji je bio vrlo važan za ovaj projekt je JSX (JavaScript XML). JSX omogućava pisanje UI komponenata koje izgledaju gotovo identično kao HTML, ali s punom snagom JavaScripta. Ovo je značajno poboljšalo čitljivost koda i olakšalo izradu kompleksnih korisničkih sučelja. *Reactova* komponentna arhitektura bila je temeljna u ovom projektu, omogućavajući modularan pristup razvoju i ponovnu upotrebu koda. Svaka komponenta je neovisna i može se lako integrirati u druge dijelove aplikacije, što doprinosi efikasnosti i skalabilnosti.

3.2.2. Redux

Redux, kao predvidljivi kontejner stanja za JavaScript aplikacije, igra ključnu ulogu u upravljanju kompleksnim stanjem i akcijama unutar *React* aplikacija. Svojom jednostavnom filozofijom "jedan izvor istine", *Redux* pruža središnje mjesto za stanje cijele aplikacije, što omogućava lakšu kontrolu, praćenje i *debugiranje*. Ova centralizacija stanja posebno je korisna u složenim aplikacijama, gdje se isti podaci mogu koristiti i mijenjati na različitim mjestima u aplikaciji.

U *Reduxu*, promjene stanja se iniciraju akcijama - objektima koji opisuju što se događa. *Reduceri* su čiste funkcije koje uzimaju prethodno stanje i akciju te vraćaju novo stanje. Ovaj pristup omogućava jasno definiranje kako se stanje aplikacije mijenja u odgovoru na akcije, što dovodi do predvidljivijeg i lakše održivog koda. U ovom projektu, korištenje *Reduxa* omogućilo je efikasnije upravljanje stanjem, lakše *debugiranje* i bolje testiranje aplikacije.

3.3. Backend Tehnologije

Kombinacija *Node.js-a, Express.js-a* i *MongoDB-a* omogućila je izgradnju efikasne i sigurne *backend* arhitekture, koja podržava ključne funkcionalnosti i zahtjeve aplikacije, uz istovremeno održavanje visokih standarda performansi i sigurnosti. *Node.js* je upotrjebljen za upravljanje asinkronim zahtjevima prilikom rezervacija plovila, dok je *Express.js* olakšao razvoj *API-ja* za obradu podataka o plovilima i korisnicima.

3.3.1. Node.js

Node.js, temeljen na *Chrome's V8 JavaScript engineu*, ključan je za razvoj visoko-*performansnih* i skalabilnih mrežnih aplikacija. U ovom projektu, *Node.js* se istaknuo svojom sposobnošću u efikasnom upravljanju asinkronim događajima, što je bilo od vitalnog značaja za obradu višestrukih zahtjeva bez blokiranja I/O operacija. Korištenje *NPM-a (Node Package Manager)* omogućilo je jednostavno upravljanje zavisnostima, s pristupom širokoj paleti modula i biblioteka koje podržava njegova snažna zajednica. U kontekstu ovog projekta, Node.js je poslužio kao temelj za *backend*, podržavajući razvoj funkcionalnosti kao što su autentifikacija korisnika i upravljanje podacima. Posebno, implementirana je *JWT (JSON Web Token)* autentifikacija, omogućujući sigurnu provjeru identiteta korisnika.

3.3.2. Express.js

Express.js, kao minimalistički i fleksibilni web aplikacijski okvir za Node.js, odigrao je centralnu ulogu u strukturiranju i razvoju *backend* dijela aplikacije. Njegova jednostavnost i modularnost omogućile su brz razvoj i lako održavanje *RESTful API-a*. U ovom projektu, *Express.js* je korišten ne samo za osnovne funkcionalnosti web aplikacije već i za specifične potrebe kao što su validacija ulaznih podataka i upravljanje rutama.

Za provjeru ispravnosti i sigurnosti podataka prilikom zahtjeva, implementiran je *express-validator*. Ovaj alat je omogućio detaljnu validaciju podataka prije njihove obrade, što je bilo ključno za održavanje integriteta i sigurnosti aplikacije. Također, zaštita korisničkih lozinki ostvarena je kroz *hashiranje* prije spremanja u bazu, što je dodatno poboljšalo sigurnost aplikacije.

3.3.3. Baza Podataka: MongoDB

MongoDB, kao *NoSQL* baza podataka orijentirana na dokumente, odabran je zbog svoje visoke performanse, lakoće skalabilnosti i velike fleksibilnosti. Fleksibilnost *MongoDB-a* pokazala se izuzetno korisnom u ovom projektu, posebno u kontekstu promjenjivih zahtjeva i rasta aplikacije. Mogućnost lako prilagodljivih shema pokazala se vrlo korisnom za dinamičko upravljanje različitim entitetima aplikacije, kao što su korisnički podaci, informacije o čamcima i rezervacijama.

Korištenje *MongoDB-a* omogućilo je efikasno upravljanje složenim upitima i odnosima među podacima, što je bilo vitalno za brzu i učinkovitu obradu podataka. Ova sposobnost da se prilagodi promjenjivim potrebama i strukturi podataka u aplikaciji omogućila je kontinuiran rast i evoluciju aplikacije bez potrebe za kompleksnim restrukturiranjem baze podataka.

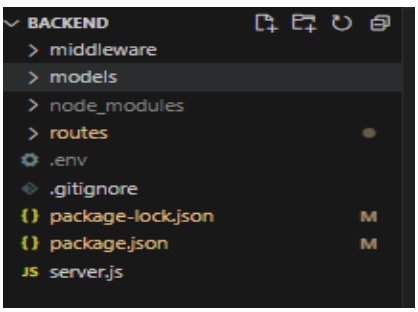
4. BACKEND

Backend dio web aplikacije "*Rent a boat*" izveden je koristeći Node.js okruženje u kombinaciji s *Express.js* okvirom. Ovaj pristup omogućava brzo i učinkovito upravljanje zahtjevima klijenta, kao i interakciju s bazom podataka. *Node.js* platforma odabrana je zbog svoje izvrsne performanse u obradi asinkronih zahtjeva te izuzetne podrške za rad s mrežnim operacijama. *Express.js*, kao modularni web okvir, koristi se za definiranje *API* ruta, što omogućava jasno strukturiranje poslovnih operacija i lakše održavanje koda.

Projekt je strukturiran u više direktorija koji olakšavaju organizaciju i razvoj aplikacije:

- *models*: Sadrži definicije *Mongoose* shema koje strukturiraju *i validiraju* podatke prije njihovog spremanja u *MongoDB* bazu podataka.
- *routes*: Definira *endpointe* koji se koriste za obradu HTTP zahtjeva, povezujući frontend s odgovarajućim logikama poslovnih operacija *backend-a*.
- *middleware*: Ovaj direktorij uključuje funkcije *middleware-a* koje se izvršavaju u lančanim pozivima, pružajući slojeve za autentifikaciju, obradu pogrešaka, i druge prethodne provjere.
- *node_modules*: Direktorij generiran od strane NPM-a koji sadrži sve pakete treće strane potrebne za projekt.
- *.env*: Skrivena datoteka koja čuva konfiguracijske varijable i osigurava da osjetljive informacije ostanu privatne.
- *.gitignore*: Omogućava isključenje određenih datoteka i direktorija iz *Git* repozitorija kako bi se spriječilo neželjeno dijeljenje osjetljivih podataka.
- *package.json* i *package-lock.json*: Centralne datoteke projekta koje sadrže *metapodatke* i održavaju evidenciju o svim zavisnostima aplikacije.
- server.js: Primarna izvršna datoteka koja inicijalizira i pokreće Express server te povezuje sve *middleware* i rute.

Slika 12 Struktura direktorija Backend-a



Izvor: Obrada autora

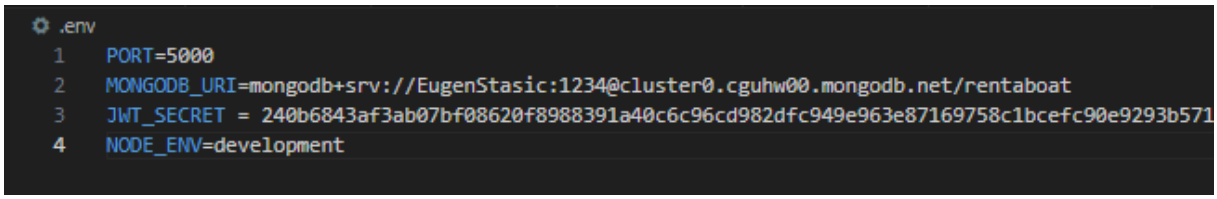
4.1. Upravljanje paketima i konfiguracija

Ključna komponenta svakog Node.js projekta je direktorij *node_modules*, koji služi kao spremište za sve pakete treće strane koje projekt koristi. Ovi paketi, instalirani i upravljani putem *Node Package Managera (NPM),* esencijalni su za razvoj, testiranje i produkciju aplikacije, pružajući bogat set funkcionalnosti koje se mogu jednostavno integrirati u projekt. Konfiguracija projekta i upravljanje zavisnostima centralizirani su unutar *package.json* datoteke, koja sadrži metapodatke projekta, skripte za izvršavanje zadataka te popis ovisnosti. Ova datoteka omogućava brzu inicijalizaciju i uspostavu radnog okruženja, bilo da se radi o lokalnom razvoju ili produkciji na poslužitelju.

package-lock.json datoteka osigurava integritet instaliranih paketa, zaključavajući verzije i osiguravajući konzistentnost okruženja među različitim razvojnim i produkcijskim sredinama. Ovaj mehanizam značajno smanjuje mogućnost neslaganja i konflikata koji mogu nastati uslijed razlika u verzijama paketa.

Da bi se zaštitili osjetljivi podaci i konfiguracijski parametri, koristi se *.env* datoteka koja čuva okruženje varijable potrebne za različite aspekte aplikacije, uključujući konfiguracije baze podataka, tajne ključeve i *API endpointe*.

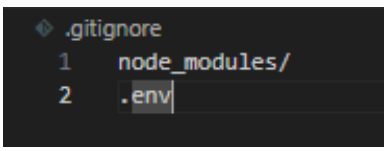
Slika 13 Sadržaj .env datoteke



Izvor: Obrada autora

Za potrebe verzioniranja koda koristi se *Git*, a *.gitignore* datoteka je neophodna kako bi se iz izvornog koda isključili direktoriji poput *node_modules* i datoteke poput *.env*, koje nisu namijenjene dijeljenju ili praćenju unutar *Git* repozitorija. Ovim pristupom se osigurava da osjetljivi i nepotrebni elementi ostanu privatni i izvan dometa verzioniranja, što je standardna praksa za održavanje sigurnosti i čistoće koda.

Slika 14 Sadržaj .gitignore datoteke



Izvor: Obrada autora

4.2. Server.js Datoteka

Datoteka *server.js* u *Node.js* aplikacijama služi kao ulazna točka ili pokretački centar aplikacije. Njezina osnovna funkcija je postavljanje poslužitelja, konfiguriranje *middleware-a,* povezivanje ruta i upravljanje bazom podataka.

Slika 15 Server.js datoteka

```
JS server.js > ...
1   require('dotenv').config();
2   const express = require('express');
3   const cors = require('cors');
4   const mongoose = require('mongoose');
5   const cookieParser = require('cookie-parser');
6
7   const authRoutes = require('./routes/auth');
8   const userRoutes = require('./routes/user');
9   const boatRoutes = require('./routes/boat');
10
11  const app = express();
12
13  app.use(cors({
14      origin: 'http://localhost:3000',
15      credentials: true
16  }));
17  app.use(express.json());
18  app.use(cookieParser());
19  app.use('/auth', authRoutes);
20  app.use('/user', userRoutes);
21  app.use('/boat', boatRoutes);
22
23  mongoose.connect(process.env.MONGODB_URI,{
24      useNewUrlParser: true,
25      useUnifiedTopology: true
26  })
27  .then (() => console.log('Connected to Database'))
28  .catch(err => console.error('Could not connect to Database'));
29
30  const PORT = process.env.PORT;
31
32  app.listen(PORT, () => {
33      console.log(`Server running on ${PORT}`);
34  });
```

Izvor: Obrada autora

Importovi:

- *dotenv*: Modul *dotenv* se koristi za učitavanje okruženje varijabli iz .env datoteke. Ovo je prva linija koda, što osigurava da se sve konfiguracijske varijable učitaju prije bilo koje druge logike aplikacije.
- *express*: Osnovni web *framework* za Node.js koji se koristi za stvaranje web poslužitelja i upravljanje HTTP zahtjevima i odgovorima.
- *cors*: *Middleware* koji omogućuje ili ograničava resurse na web stranici da budu zatraženi s drugih domena izvan domene iz koje je prva resurs poslan, što je poznato kao *cross-origin requests.*

- *mongoose*: *ODM (Object Data Modeling)* biblioteka za *MongoDB* i *Node.js* koja omogućuje definiranje objekata u skladu sa strukturom dokumenta u *MongoDB,* kao i upravljanje povezivanjem, transakcijama i drugim operacijama nad bazom podataka.
- *authRoutes, userRoutes, boatRoutes*: Ovi importi učitavaju definicije ruta iz odgovarajućih datoteka unutar direktorija *routes*. Ove rute određuju kako aplikacija odgovara na klijentske zahtjeve na određenim *endpointima*, s pripadajućim kontrolerima i logikom

Konfiguracija *Middleware-a* i Ruta:

- *app*: Stvaranje instance Express aplikacije.
- *app.use(cors())*: Primjena CORS *middleware-a* s konfiguracijom koja dozvoljava zahtjeve s određene domene (http://localhost:3000) i omogućava slanje *kredencijala* kao što su kolačići i autorizacijski *headeri*.
- *app.use(express.json()): Middleware* za *parsiranje JSON* tijela zahtjeva, što omogućava lak pristup poslanim podacima u JSON formatu.
- *app.use(cookieParser()):* Omogućava aplikaciji da čita podatke spremljene u kolačićima klijenta.
- *app.use('/auth', authRoutes):* Specifikacija ruta za autentifikaciju. Sve zahtjeve koji započinju s */auth* će se obraditi kroz *authRoutes.*

Povezivanje sa Bazom podataka i pokretanje *backend* servisa:

- *mongoose.connect():* Pokreće povezivanje s *MongoDB* bazom podataka koristeći URL iz varijabli okruženja. Opcije kao što su *useNewUrlParser* i *useUnifiedTopology* osiguravaju pravilno povezivanje u skladu s novijim *MongoDB* driver standardima.
- *app.listen(PORT, callback):* Zapovijeda* *Express* poslužitelju da počne slušati dolazne zahtjeve na definiranom portu, što se također dohvaća iz varijable okruženja. *Callback* funkcija se izvršava nakon što je poslužitelj uspješno pokrenut.

4.3. Middleware

Datoteka *authenticate.js* unutar direktorija *middleware* predstavlja sigurnosnu komponentu u Node.js aplikacijama koja služi za autentifikaciju korisnika. Njena primarna svrha je provjeriti je li zahtjev koji je poslan poslužitelju valjan i da li dolazi od ovlaštenog korisnika. Ovo se postiže kroz verifikaciju *JWT (JSON Web Tokena)*, koji je standard za sigurno prenošenje informacija između klijenta i poslužitelja u obliku *JSON* objekta.

Slika 16 Sadržaj authenticate.js datoteke

middleware > JS authenticate.js > ...

```
1   const jwt = require('jsonwebtoken');
2
3   const authenticate = (req, res, next) => {
4       const token = req.headers.authorization?.split(' ')[1];
5       if (!token) {
6           return res.status(401).json({ message: 'No token provided.' });
7       }
8
9       try {
10          const decoded = jwt.verify(token, process.env.JWT_SECRET);
11          req.user = decoded;
12          next();
13      } catch (ex) {
14          res.status(400).json({ message: 'Invalid token.' });
15      }
16  };
17
18  module.exports = authenticate;
```

Izvor: Obrada autora

U praksi, kada se izvrši *HTTP* zahtjev prema serveru, *authenticate* funkcija se aktivira kao *middleware* prije nego što se zahtjev proslijedi dalje u lanac obrade. Prvo, funkcija izvlači token iz '*Authorization*' zaglavlja zahtjeva. Ako token nije prisutan, *middleware* prekida obradu zahtjeva i vraća odgovor s *HTTP* statusom 401, što ukazuje na to da zahtjev nije autoriziran zbog nedostatka tokena. Ako je token prisutan, *authenticate* funkcija pokušava verificirati token korištenjem *jwt.verify* metode. Ova metoda dekodira token koristeći tajni ključ definiran u varijablama okruženja (*.env* datoteka), osiguravajući da je token izdan od strane servera i da nije istekao ili na neki drugi način kompromitiran. Uspješna verifikacija postavlja dekodirani JWT objekt u *req.user*, čime se omogućava pristup ovim podacima u narednim fazama obrade zahtjeva. U slučaju da verifikacija tokena ne uspije, bilo zbog nevažećeg tokena ili problema s verifikacijom, funkcija vraća odgovor s HTTP statusom 400, označavajući neispravnost tokena.

4.4. Models

Model u *Mongooseu* je visoko strukturirani okvir koji predstavlja kolekciju u *MongoDB-ju* i koristi se za interakciju s bazom podataka. Model definira strukturu podataka (ili 'shemu') za određeni skup podataka i pruža metode za čitanje, umetanje, ažuriranje i brisanje zapisa iz baze podataka. Shema modela služi kao plava skica koja detaljno opisuje oblik i sadržaj dokumenata unutar kolekcije, uključujući tipove podataka, obavezna polja, standardne vrijednosti i validacijska pravila.

Slika 17 Mongoose model za korisnika

models > JS User.js > ...

```
const mongoose = require('mongoose');
const bcrypt = require('bcryptjs');

const userSchema = new mongoose.Schema({
    firstName: {
        type: String,
        required: true,
    },

    lastName: {
        type: String,
        required: true
    },

    email: {
        type: String,
        required: true,
        lowercase: true,
        unique: true
    },

    password: {
        type: String,
        required: true
    },

    profile: {
        dateOfBirth: Date,
        gender: {
            type: String,
            enum: ['Male', 'Female', 'Other']
        },

        contact: {
            phone: { type: String, default: '' },
            address: {
                city: {type: String, default: '' },
                postalCode: { type: String, default: '' },
                street: { type: String, default: '' }
            }
        },

        nauticalLevel: { type: String, enum: ['Beginner', 'Intermediate', 'Experienced', 'Pro' ]},

        yachtLicenseHolder: { type: Boolean, default: false}
    }

}, { timestamps: true});
```

Izvor: Obrada autora

Kada se definira model korisnika, on služi kao apstrakcija koja pojednostavljuje manipulaciju korisničkim podacima. Definiranje modela omogućava jednostavno i sigurno upravljanje registracijom korisnika, autentifikacijom, profilima i ostalim operacijama koje su vezane uz korisničke podatke. Model osigurava da se svi podaci koji ulaze u bazu pridržavaju zadane sheme, čime se održava integritet i konzistentnost podataka.

U *userSchema*, koja je primjer *Mongoose* sheme, definirani su atributi i njihova pravila koja korisnički dokumenti moraju slijediti. Ime i prezime su obavezna polja, što znači da svaki zapis mora sadržavati te informacije. Email adresa također mora biti prisutna, jedinstvena i formatirana u malim slovima kako bi se osigurala njena jedinstvenost unutar aplikacije.

Dodatni osobni podaci korisnika grupirani su pod profile, koji sadrži podatke poput datuma rođenja i spola, a ovo polje koristi enumeraciju da ograniči unos na predefinirane vrijednosti. Kontakt informacije su strukturirane unutar zasebnog objekta koji uključuje telefon i adresu, omogućujući organiziranu pohranu i lak pristup ovim informacijama.

Polja kao što su *nauticalLevel* i *yachtLicenseHolder* pružaju dodatne informacije specifične za domenu *rent-a-boat* servisa, omogućujući aplikaciji da prilagodi iskustvo korisnika njihovim kvalifikacijama i pravima.

Lozinka je osjetljiv podatak koji se štiti korištenjem *hashiranja*, a model to zahtijeva kao obavezno polje.

Slika 18 Hashiranje passworda prije spremanja u bazu podataka

```
userSchema.pre('save', async function (next){
    if(!this.isModified('password')) return next();

    const salt = await bcrypt.genSalt(15);
    this.password = await bcrypt.hash(this.password, salt);
});
```

1 Izvor: Obrada autora

Skripta u *userSchema.pre('save', ...)* predstavlja *middleware* koji *Mongoose* pokreće prije spremanja svakog korisničkog dokumenta. U njemu se provodi provjera je li lozinka korisnika izmijenjena, a ako nije, koristi se *bcrypt* biblioteka za sigurnosno *hashiranje* nove lozinke. Skripta implementira sigurnosni postupak gdje se, u slučaju izmjene korisničke lozinke, generira nasumični niz znakova, poznat kao '*salt*', koji se kombinira s originalnom lozinkom prije procesa *hashiranja*. Ovaj korak dodaje sloj sigurnosti enkripciji lozinke, čime se značajno otežava njena *dekripcija* čak i u slučaju kompromitacije baze podataka.

4.5. Rute

U kontekstu web aplikacija, rute su definicije koje serveru govore kako reagirati na različite *HTTP* zahtjeve — poput *GET, POST, PUT, DELETE* — na specifičnim *URL-ovima*. Svaka ruta može imati jednu ili više funkcija obrade, koje se izvršavaju kada se ruta podudara s *URL-om* zahtjeva.

U *Node.js* i *Express.js* okruženju, rute su organizirane u posebnim datotekama i direktorijima kako bi se olakšala organizacija i održavanje koda. Direktorij *routes* obično sadrži skup datoteka koje grupiraju rute po logici ili domeni, na primjer, rute za autentifikaciju korisnika, upravljanje korisničkim profilima ili rukovanje informacijama o plovilima. Svaka datoteka unutar *routes* direktorija predstavlja modul koji izvozi definirane rute, koje *Express* aplikacija potom koristi za usmjeravanje zahtjeva do odgovarajućih kontrolera ili *middleware-a*.

Rute ovog projekta podijeljene su u auth.js, user.js i boat.js. Prateći konvencije REST API-ja ovom podjelom logički svrstavamo rute u vlastite kategorije.

Datoteka auth.js sadrži rute koje se koriste pri svim *autentifikacijskim* zahtjevima koji uključuju registraciju, prijavu i odjavu korisnika.

Slika 19 Ruta za registraciju korisnika

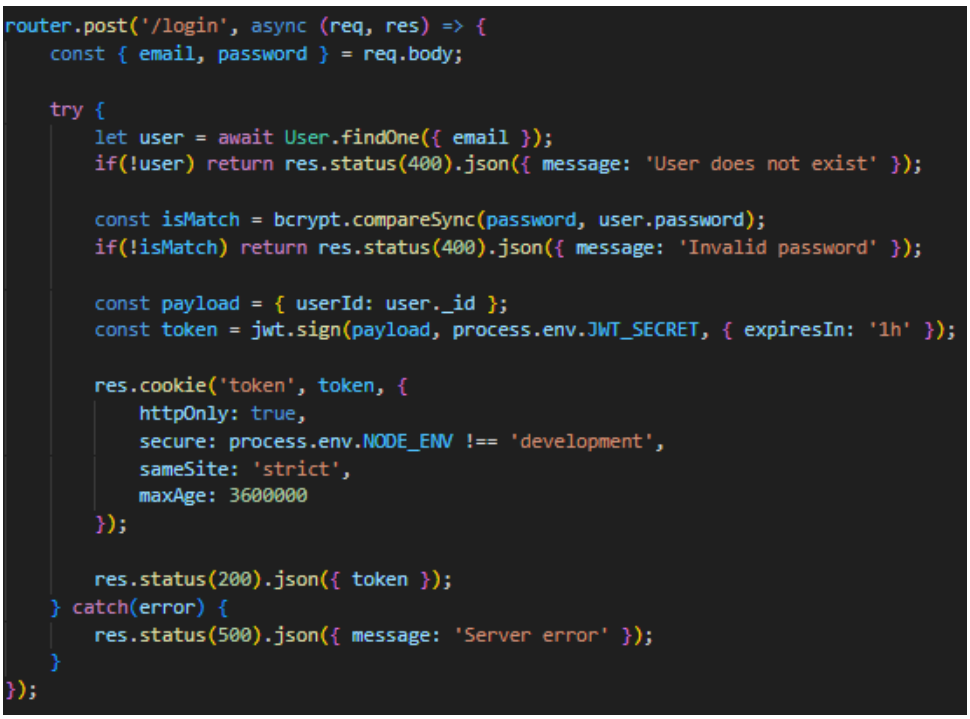
```
router.post('/register', [
    body('firstName').notEmpty().withMessage('Name is required'),
    body('lastName').notEmpty().withMessage('Surname is required'),
    body('email').isEmail().withMessage('Invalid email format'),
    body('password').isLength({ min: 3 }).withMessage('Password shoud be at least 3 charachters!')
], async (req, res) => {
    const errors = validationResult(req);
    if(!errors.isEmpty()) {
        return res.status(400).json({ errors: errors.array() });
    }

    const { firstName, lastName, email, password } = req.body;

    try {
        let user = await User.findOne({ email });
        if (user) return res.status(400).json({ message: 'User already exists'});

        user = new User({
            firstName,
            lastName,
            email,
            password
        });

        await user.save();

        res.status(201).json({ message: 'User registered successfully' });
    } catch(error) {
        res.status(500).json({ message: 'Server Error' });
    }
});
```

Izvor: Obrada autora

Ovaj kod definira Express.js rutu koja upravlja registracijom novih korisnika. Prilikom poziva *endpointa* POST */register,* prvo se izvršava niz validacijskih provjera na podacima koje je korisnik poslao. Ove provjere uključuju da li su polja *firstName, lastName*, *email i password* prisutna i ispravna. Ako podaci ne prođu validaciju, server odgovara statusnim kodom 400 i šalje popis grešaka.

Ako su svi podaci valjani, kod provjerava postoji li već korisnik s istom email adresom u bazi podataka. Ako korisnik postoji, odgovara se sa statusom 400 i porukom da korisnik već postoji. Ako ne postoji, stvara se novi korisnički dokument koristeći model *User* i poslani podaci se spremaju u bazu. Nakon uspješnog spremanja, korisniku se vraća odgovor sa statusom 201 koji označava uspješnu registraciju. U slučaju greške tijekom procesa, server odgovara sa statusom 500, signalizirajući unutarnju grešku servera.

Slika 20 Ruta za prijavu korisnka



```
router.post('/login', async (req, res) => {
    const { email, password } = req.body;

    try {
        let user = await User.findOne({ email });
        if(!user) return res.status(400).json({ message: 'User does not exist' });

        const isMatch = bcrypt.compareSync(password, user.password);
        if(!isMatch) return res.status(400).json({ message: 'Invalid password' });

        const payload = { userId: user._id };
        const token = jwt.sign(payload, process.env.JWT_SECRET, { expiresIn: '1h' });

        res.cookie('token', token, {
            httpOnly: true,
            secure: process.env.NODE_ENV !== 'development',
            sameSite: 'strict',
            maxAge: 3600000
        });

        res.status(200).json({ token });
    } catch(error) {
        res.status(500).json({ message: 'Server error' });
    }
});
```

2 Izvor: Obrada autora

Prilikom slanja zahtjeva na *POST /login endpoint*, proces autentifikacije se odvija u nekoliko koraka.

Prvo, iz tijela zahtjeva se izdvajaju email i password. Zatim se asinkrono pretražuje baza podataka za korisnikom koji odgovara poslanom emailu. Ako korisnik s tim emailom ne postoji, server vraća odgovor sa statusnim kodom 400, označavajući da korisnik ne postoji.

Ako korisnik postoji, koristi se *bcrypt* biblioteka za sinkronu usporedbu poslane lozinke s *hashiranom* lozinkom pohranjenom u bazi. U slučaju da lozinke nisu usklađene, odgovara se s statusom 400 i porukom o neispravnoj *lozinci*.

Kada se uspješno verificira lozinka, kreiranje JWT-a (JSON Web Tokena) se vrši pomoću *jwt.sign* metode, koja generira token na temelju korisničkog ID-a i tajnog ključa definiranog u okruženju. Token je konfiguriran da istječe nakon jednog sata, što znači da korisnik mora ponovno provesti postupak prijave nakon tog perioda.

Token se potom šalje korisniku kao dio *HTTP* odgovora, u obliku kolačića (*cookie*). Kolačić je konfiguriran da bude *httpOnly*, što znači da nije dostupan kroz klijentske skripte i smanjuje rizik od *XSS* napada. Također, kolačić je označen kao *secure* u proizvodnom okruženju, što znači da se šalje samo preko *HTTPS* veze.

Uspješan odgovor servera s statusom 200 uključuje i *JWT*, omogućujući klijentu da koristi token za autentifikaciju i autorizaciju kod nadolazećih zahtjeva.

Ruta *POST /logout* u kodu je definirana za omogućavanje korisnicima da se odjave iz aplikacije.

Slika 21 Ruta za odjavu korisnika

Izvor: Obrada autora

Kada se ova ruta pozove, prvo se izvršava *authenticate middleware* kako bi se osiguralo da je zahtjev autentificiran. *Middleware* provjerava postoji li valjani JWT u kolačićima zahtjeva, što je standardni postupak za verifikaciju korisničke sesije.

Nakon provjere autentifikacije, izvršava se asinkrona *callback* funkcija *res.clearCookie('token')* koja naređuje serveru da izbriše kolačić s imenom 'token', koji sadrži *JWT* korisnika. Time se efektivno uklanja sesijski token s korisnikova uređaja, čime se prekida sesija i onemogućava daljnje korištenje tokena za autorizaciju zahtjeva

Datoteka user.js sadrži sve rute koje su povezane sa korisnikom.
Ruta GET /me definirana u datoteci user.js služi za dohvat informacija o trenutno autentificiranom korisniku.

Slika 22 Ruta za dohvat informacija trenutnog korisnika

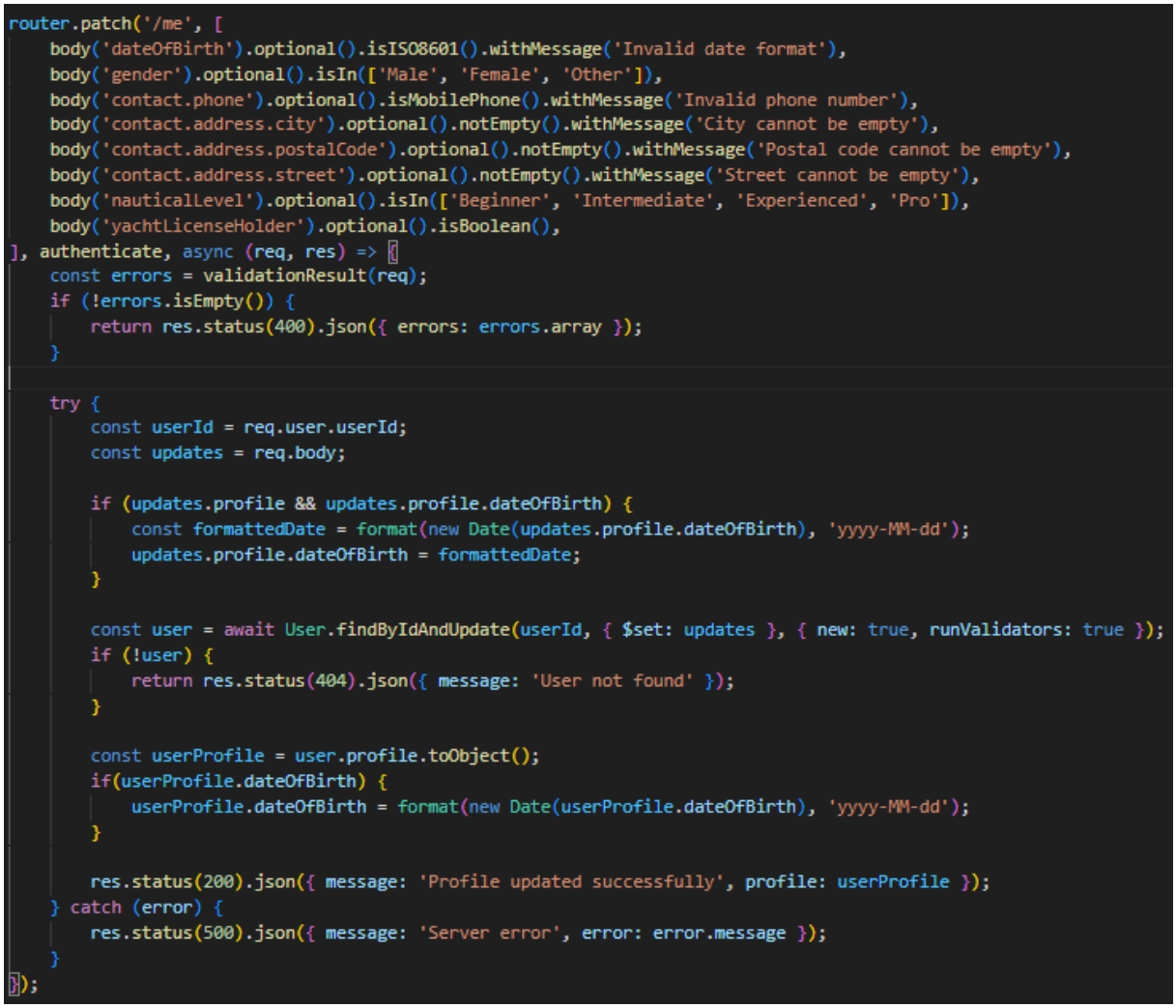
```
router.get('/me', authenticate, async (req, res) => {
    try {
        const user = await User.findById(req.user.userId)
        if (!user) {
            return res.status(404).json({ message: 'User not found' });
        }

        const { password, ...userWithoutPassword } = user.toObject();

        res.status(200).json(userWithoutPassword);
    } catch (error) {
        res.status(500).json({ message: 'Server error', error: error.message });
    }
});
```

Izvor: Obrada autora

Nakon što *authenticate middleware* potvrdi autentičnost zahtjeva, asinkrona funkcija pretražuje bazu podataka koristeći *Mongooseovu* metodu *findById* i identifikator korisnika dobiven iz zahtjeva.
Ako korisnik s pruženim ID-em ne postoji u bazi podataka, funkcija vraća odgovor s *HTTP* statusom 404, što označava da traženi resurs nije pronađen, zajedno s odgovarajućom porukom.
Kada se korisnik pronađe, njegov objekt se pretvara u *JavaScript* objekt i iz njega se izdvaja lozinka koristeći destrukturiranje objekta, čime se osigurava da se lozinka ne uključi u odgovor poslan klijentu. Ostatak korisničkih podataka, bez lozinke, šalje se natrag s odgovorom sa statusom 200, što znači da je zahtjev uspješno obradio.
Ako dođe do greške tijekom izvršavanja ove operacije, uhvati se iznimka i šalje se odgovor sa statusom 500, ukazujući na unutarnju grešku servera, zajedno s porukom greške kako bi korisnik bio obaviješten o problemu.

Ruta *PATCH /me* namijenjena je ažuriranju korisničkog profila.

Slika 23 Ruta za ažuriranje korisničkog profila



```
router.patch('/me', [
    body('dateOfBirth').optional().isISO8601().withMessage('Invalid date format'),
    body('gender').optional().isIn(['Male', 'Female', 'Other']),
    body('contact.phone').optional().isMobilePhone().withMessage('Invalid phone number'),
    body('contact.address.city').optional().notEmpty().withMessage('City cannot be empty'),
    body('contact.address.postalCode').optional().notEmpty().withMessage('Postal code cannot be empty'),
    body('contact.address.street').optional().notEmpty().withMessage('Street cannot be empty'),
    body('nauticalLevel').optional().isIn(['Beginner', 'Intermediate', 'Experienced', 'Pro']),
    body('yachtLicenseHolder').optional().isBoolean(),
], authenticate, async (req, res) => {
    const errors = validationResult(req);
    if (!errors.isEmpty()) {
        return res.status(400).json({ errors: errors.array });
    }

    try {
        const userId = req.user.userId;
        const updates = req.body;

        if (updates.profile && updates.profile.dateOfBirth) {
            const formattedDate = format(new Date(updates.profile.dateOfBirth), 'yyyy-MM-dd');
            updates.profile.dateOfBirth = formattedDate;
        }

        const user = await User.findByIdAndUpdate(userId, { $set: updates }, { new: true, runValidators: true });
        if (!user) {
            return res.status(404).json({ message: 'User not found' });
        }

        const userProfile = user.profile.toObject();
        if(userProfile.dateOfBirth) {
            userProfile.dateOfBirth = format(new Date(userProfile.dateOfBirth), 'yyyy-MM-dd');
        }

        res.status(200).json({ message: 'Profile updated successfully', profile: userProfile });
    } catch (error) {
        res.status(500).json({ message: 'Server error', error: error.message });
    }
});
```

Izvor: Obrada autora

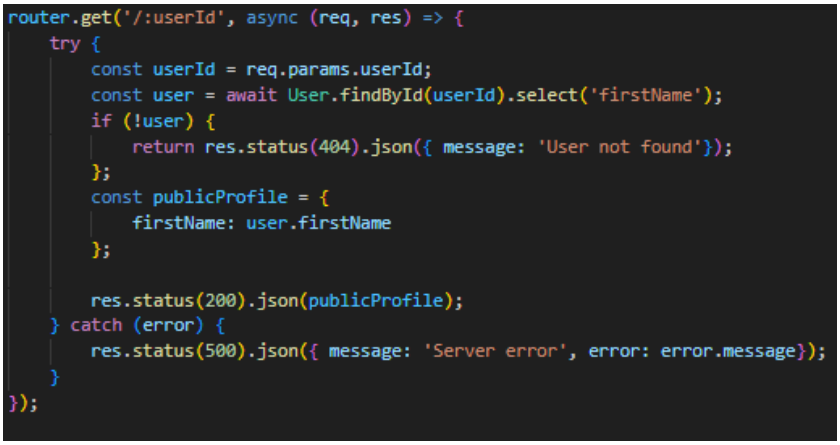
Provjerava se ispravnost podataka koji su poslani sa zahtjevom koristeći niza validatora. Validacija osigurava da su svi potrebni podaci prisutni i ispravnog formata, uključujući polja poput datuma rođenja, spola, kontaktnih informacija, nautičke razine vještine i statusa posjedovanja brodarske dozvole. Ukoliko podaci ne prođu validaciju, korisniku se vraća greška sa statusom 400 i detaljima o greškama.

Ukoliko su podaci validni, skripta ekstrahira korisnički ID iz zahtjeva i podatke koji se ažuriraju. Ako su prisutne promjene u profilnom dijelu koje uključuju datum rođenja, primjenjuje se formatiranje datuma. Nakon toga, koristi se *Mongooseova* metoda *findByIdAndUpdate* za pronalazak korisnika u bazi podataka pomoću ID-a i ažuriranje njegovih podataka. Opcija *{ new: true }* osigurava da se nakon ažuriranja

vrati ažurirani dokument, dok *runValidators: true* osigurava da se sva validacijska pravila definirana u shemi također primjenjuju tijekom ažuriranja.

Ruta *GET /:userId* omogućava asinkroni dohvat specifičnog korisnika iz baze podataka koristeći *Mongooseovu* funkciju *findById*

Slika 24 Ruta za dohvat specifičnog korisnika po ID-u



```
router.get('/:userId', async (req, res) => {
    try {
        const userId = req.params.userId;
        const user = await User.findById(userId).select('firstName');
        if (!user) {
            return res.status(404).json({ message: 'User not found'});
        };
        const publicProfile = {
            firstName: user.firstName
        };

        res.status(200).json(publicProfile);
    } catch (error) {
        res.status(500).json({ message: 'Server error', error: error.message});
    }
});
```

Izvor: Obrada autora

U ovoj funkciji, *req.params.userId* preuzima ID korisnika iz URL-a zahtjeva. Nakon dohvaćanja korisnika, kôd koristi metodu *.select('firstName')* za dohvaćanje samo imena korisnika iz baze podataka, umjesto da dohvati cijeli objekt korisnika, što pomaže u očuvanju privatnosti korisnika ograničavajući informacije koje se dijele.

Ako korisnik s tim ID-em ne postoji, odgovor se vraća sa statusom 404 *(Not Found*), informirajući klijenta da traženi korisnik nije pronađen. U suprotnom, stvara se objekt *publicProfile* koji sadrži samo ime korisnika, koji se zatim šalje natrag klijentu sa statusom 200 (OK), predstavljajući uspješan odgovor.

5. FRONTEND

U kontekstu *frontend* razvoja web aplikacije "*Rent a Boat*", React.js se ističe svojom komponentnom arhitekturom i sposobnošću efikasnog upravljanja korisničkim sučeljima. U ovom dijelu rada, fokus je na detaljnom objašnjenju kako React.js doprinosi izgradnji interaktivnog i dinamičkog korisničkog iskustva.

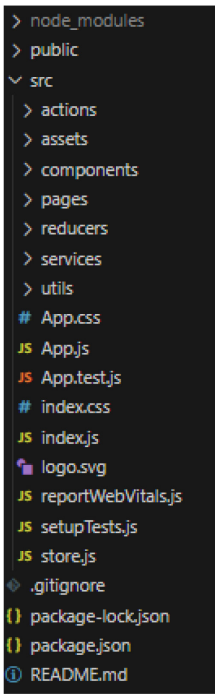
U *Reactu,* razvoj aplikacije provodi se kroz modularne i neovisne jedinice poznate kao komponente. Svaka komponenta u *Reactu* može biti samostalna i sadržavati vlastito stanje i logiku, čime se omogućava ponovna upotreba i bolja organiziranost koda. Ovaj pristup olakšava održavanje i skaliranje aplikacije, jer promjene mogu biti izolirano primijenjene na pojedinačne komponente bez utjecaja na ostatak aplikacije.

Hooks, uvedeni u *React*, omogućuju korištenje stanja i drugih *Reactovih* značajki unutar funkcionalnih komponenata. *Hooks* kao *useState* i *useEffect* pružaju fleksibilnost za upravljanje internim stanjem komponenti i reagiranje na promjene u njihovom životnom ciklusu bez potrebe za klasnim komponentama. Ovo pojednostavljuje kod i omogućuje lakše upravljanje stanjem i efektima unutar aplikacije.

Daljnja poglavlja pružat će detaljan pregled primjene *React.js-a* u razvoju aplikacije "*Rent a Boat*", uključujući strukturu projekta, upotrebu *Reduxa* za upravljanje stanjem, te prikaz odabranih komponenata i stranica koje čine sučelje aplikacije. Ovaj pregled pružit će uvid u tehničke aspekte i odluke koje stoje iza izrade korisničkog sučelja aplikacije "*Rent a Boat*".

Struktura *Frontend* Direktorija

Slika 25 Sadržaj direktorija Frontend-a



Izvor: Obrada autora

Struktura *frontend* projekta podijeljena je u različite direktorije, omogućujući jasnu organizaciju koda i resursa:

- *actions*: Sadrži skup *Redux* akcija koje definiraju logiku za slanje podataka iz aplikacije u *store*.
- *reducers:* Direktorij koji sadrži *Redux reducere*, funkcije koje određuju kako će stanje aplikacije reagirati na akcije poslane *storeu*.
- *assets:* služi kao središte za sve statičke resurse koji se koriste unutar aplikacije. Ovo uključuje slike, stilove, fontove i druge resurse poput ikona i grafika koje su vizualni dio korisničkog iskustva.
- *components*: Sadrži *React* komponente, ponovno upotrebljive dijelove korisničkog sučelja.
- *pages*: Skup komponenti koje predstavljaju cijele stranice unutar aplikacije.
- *services*: Uključuje logiku za komunikaciju s *backendom*, kao što su *API* pozivi.
- *utils*: Direktorij s pomoćnim funkcijama koje pružaju dodatne funkcionalnosti poput formata datuma, validacije i slično.

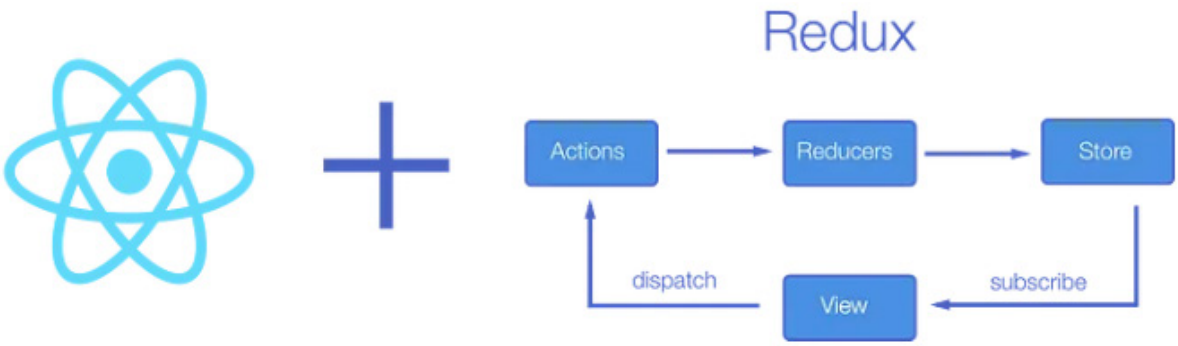
5.1. Redux

Redux je evolucija arhitekture *Flux*, koju je razvio Facebook, a služi za centralizirano upravljanje stanjem u *JavaScript* aplikacijama, posebno u složenim *React* aplikacijama. Omogućuje aplikaciji da ima jedan izvor istine u obliku '*storea'*, koji sadrži cjelokupno stanje aplikacije. To olakšava praćenje promjena unutar aplikacije, *debugiranje* i testiranje. *Redux* također uspostavlja stroga pravila oko promjena stanja, što dovodi do više predvidljivog koda kroz 'akcije' i '*reducere*'.

Unutar konteksta "*Rent a boat*" aplikacije, *Redux* omogućava efikasno upravljanje stanjem poput autentifikacije korisnika, upravljanja podacima o plovilima, te rezervacija. To je posebno korisno kada se radi o ažuriranju korisničkog sučelja u realnom vremenu kao odgovor na korisničke interakcije ili pristizanje podataka s poslužitelja.

Akcije u *Reduxu* su objekti koji prenose informacije o tome kako bi stanje trebalo biti promijenjeno, a *reducere* su čiste funkcije koje određuju kako se aktualno stanje, u odnosu na akcije, transformira u novo stanje. Ovaj pristup omogućuje rad s aplikacijom kao s deterministčkim strojem stanja, gdje svaka promjena stanja može biti jasno definirana i predvidljiva.

Slika 26 React Redux

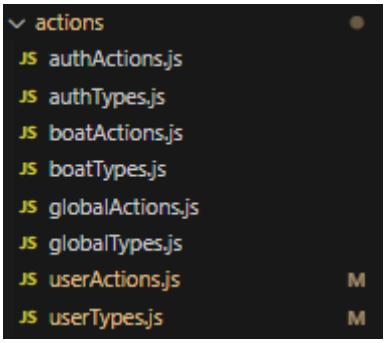


Izvor: https://medium.com/the-web-tub/managing-your-react-state-with-redux-affab72de4b1

5.2. Actions

U direktoriju '*actions*' se nalaze datoteke podijeljene u akcije za odgovarajuće kategorije.

Slika 27 Actions direktorij



Izvor: Obrada autora

Datoteka *authTypes.js* sadrži definirane konstante koje se unose u odgovarajuću datoteku *authActions.js*.

5.2.1. Registracija

Slika 28 Akcija za registraciju

```
// REGISTRACIJA

export function register(userData) {
    return async function (dispatch) {
        dispatch({
            type: types.REGISTER_REQUEST
        });

        try{
            await registerService(userData);
            dispatch({
                type: types.REGISTER_SUCCESS,
                successMessage: "Registration successful! Please log in to continue."
            });
        } catch (error) {
            dispatch({
                type: types.REGISTER_ERROR,
                payload: getErrorMessage(error)
            });
        }
    }
}
```

Izvor: Obrada autora

Kroz ovu akciju, *Redux* upravlja cijelim tokom registracije korisnika, od početnog zahtjeva do uspješnog završetka ili hvatanja i obrade grešaka. Ova *Redux* akcija, nazvana *register*, zadužena je za obradu registracije novog korisnika. Funkcija prima *userData* kao argument, koji predstavlja podatke korisnika potrebne za registraciju, kao što su ime, email i lozinka.

Pri pozivu, akcija prvo pokreće asinkronu funkciju koja dispatchuje početni akcijski tip *REGISTER_REQUEST*. Ova akcija signalizira početak procesa registracije i može se koristiti za aktiviranje indikatora učitavanja u korisničkom sučelju, obavještavajući korisnika da je njegov zahtjev za registracijom u tijeku.

Unutar *try* bloka, izvršava se asinkroni poziv *registerService* funkciji, koja komunicira s *backendom* kako bi pohranila *userData* u bazu podataka. Ako je poziv uspješan, dispatchuje se akcija *REGISTER_SUCCESS*, označavajući da je korisnik uspješno registriran. Uz ovu akciju, može se poslati i poruka o uspjehu koja će biti prikazana korisniku, npr. "*Registration successful! Please log in to continue*."
Ako proces registracije naiđe na grešku, izvršava se catch blok koji hvata iznimku i dispatchuje akciju *REGISTER_ERROR* s odgovarajućom greškom kao *payload*. Funkcija *getErrorMessage* može formatirati ili izabrati odgovarajuću poruku greške za prikaz korisniku.

5.2.2. Prijava

Slika 29 Akcija za prijavu

```
// LOGIN

export function login (email, password) {
    return async function (dispatch) {
        try {
            const response = await loginService(email, password);
            dispatch({
                type: types.LOGIN_SUCCESS,
                payload: { token: response.data.token },
                successMessage: "Log in successfull!"
            });
        } catch (error) {
            dispatch({
                type: types.LOGIN_ERROR,
                payload: getErrorMessage(error)
            });
        }
    }
}
```

Izvor: Obrada autora

Akcija login prima korisničko ime i lozinku te pokreće asinkronu operaciju. U okviru ove operacije, prvo se pokušava *autentificirat*i korisnik putem *loginService* funkcije koja prima email i lozinku, a vraća JWT ako su vjerodajnice ispravne. Uspješan odgovor dovodi do dispatchovanja akcije *LOGIN_SUCCESS*, gdje se *JWT (token)* dobiven od servera šalje kao dio *payloada*, a korisniku se može prikazati poruka uspješne prijave. U slučaju greške tijekom procesa prijave, uhvati se iznimka i dispatchuje se akcija *LOGIN_ERROR* s odgovarajućom porukom o grešci dobivenom iz funkcije *getErrorMessage.*

5.2.3. Odjava

Slika 30 Akcija za odjavu

```
// LOGOUT

export function logout() {
    return async function (dispatch) {
        try {
            await logoutService();
            dispatch({
                type: types.LOGOUT_SUCCESS,
                successMessage: 'You have been logged out'
            });
        } catch (error) {
            dispatch({
                type: types.LOGOUT_ERROR,
                payload: getErrorMessage(error)
            });
        }
    }
}
```

Izvor: Obrada autora

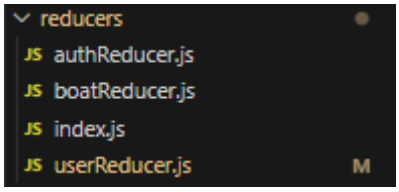
Akcija *logout* inicira proces odjave bez potrebe za argumentima. Asinkrona funkcija poziva *logoutService*, koji bi trebao izvršiti potrebne korake na serveru za prekid sesije. Nakon toga se dispatchuje *LOGOUT_SUCCESS*, signalizirajući da je korisnik uspješno odjavljen, i korisniku se može prikazati odgovarajuća poruka.

Ako se tijekom odjave dogodi greška, izvršava se *catch* blok koji dispatchuje *LOGOUT_ERROR* s porukom o grešci, informirajući korisnika o problemu.

5.3. Reducers

U direktoriju *'reducers'* datoteke su podijeljene u *'reducere'* za odgovarajuće kategorije.

Slika 31 Reducers direktorij



Izvor: Obrada autora

Datoteka index.js unutar direktorija *reducers* u *Redux* arhitekturi aplikacije služi kao središnje mjesto za kombiniranje više *reducera* u jedan glavni *reducer*, poznat kao *rootReducer. Reduxova* funkcija *combineReducers* omogućava da se pojedinačni reduceri, koji upravljaju određenim segmentima stanja aplikacije, združe u jedan skup pravila za ažuriranje cijelog stanja.

Dok *authActions* definiraju kako aplikacija šalje podatke i komunicira s *backendom* za potrebe autentifikacije, *authReducer* direktno reagira na te akcije i odgovorno je za ažuriranje stanja aplikacije. To uključuje praćenje korisničkog tokena, autentifikacijskog statusa, bilježenje bilo kakvih grešaka tijekom procesa autentifikacije, te uspjeha operacija.

Slika 32 Inicijalna stanja auth reducera

```
const initialState = {
    token: null,
    isAuthenticated: false,
    error: null,
    successMessage: null,
};
```

Izvor: Obrada autora

U authReducer.js datoteci, *initialState* objekt definira početno stanje vezano za autentifikaciju u *Reduxu*. Ovaj objekt služi kao osnova za praćenje autentifikacijskih podataka unutar aplikacije. *Reduceri* u *Reduxu* koriste ovakvo početno stanje kako bi definirali kako se stanje aplikacije treba mijenjati kao odgovor na različite akcije. *authReducer* će slušati za akcije koje se odnose na autentifikaciju, kao što su prijava, registracija, ili odjava, te će ažurirati *initialState* u skladu s tim, čime se osigurava da aplikacija uvijek ima točan pregled trenutnog autentifikacijskog stanja korisnika.

Slika 33 Sadržaj authReducer.js

```
const authReducer = (state = initialState, action) => {
    switch (action.type) {
        case types.LOGIN_SUCCESS:
            return {
                ...state,
                token: action.payload.token,
                isAuthenticated: true,
                successMessage: action.successMessage
            };

        case types.LOGIN_ERROR:
            return {
                ...state,
                error: action.payload.message,
            };

        case types.LOGOUT_SUCCESS:
            return {
                ...initialState,
                successMessage: action.successMessage
            };

        case types.LOGOUT_ERROR:
            return {
                ...state,
                error: action.payload.message,
            };

        case types.REGISTER_REQUEST:
            return {
                ...state,
                error: null,
                successMessage: null,
            };

        case types.REGISTER_SUCCESS:
            return {
                ...state,
                error: null,
                successMessage: action.successMessage
            };

        case types.REGISTER_ERROR:
            return {
                ...state,
                error: action.payload.message,
            };

        case CLEAR_MESSAGES:
            return {
                    ...state,
                    error: null,
                    successMessage: null
                };

        default:
            return state;
    }
};

export default authReducer;
```

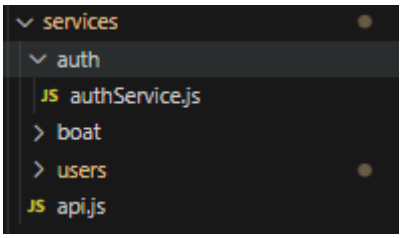
Izvor: Obrada autora

Reducer authReducer u aplikaciji "*Rent a boat*" služi kao funkcija koja preslušava za akcije vezane uz autentifikaciju te na temelju njih ažurira stanje aplikacije. Kad se izvrši neka od akcija autentifikacije, *authReducer* prima tu akciju i analizira njen tip pomoću *switch* izraza. Za svaki tip akcije*, reducer* određuje kako će se ažurirati *initialState*.
Na primjer, za proces registracije, akcija *REGISTER_REQUEST* resetira sve poruke o greškama i uspjehu na *null* kako bi se osiguralo da se ne prikazuju zastarjele informacije. Ako je registracija uspješna, akcija *REGISTER_SUCCESS* ažurira stanje s porukom o uspjehu, dok *REGISTER_ERROR* postavlja poruku o grešci.

5.4. Services

Direktorij *'services*' u strukturi projekta *React* aplikacije "*Rent a boat*" igra ulogu u organizaciji i izolaciji logike komunikacije s poslužiteljem. Ovaj direktorij tipično sadrži skripte koje se bave *API* pozivima, rukovanjem zahtjevima i odgovorima te upravljanjem podacima koji se šalju ili primaju od *backend* sustava.

Slika 34 Services direktorij



Izvor: Obrada autora

Unutar *services*, poddirektoriji i datoteke su obično razdvojeni prema njihovim funkcionalnostima ili entitetima s kojima se interakcija odvija. Na primjer, authService.js će upravljati svim pozivima *API-ja* vezanim uz autentifikaciju korisnika, uključujući prijavu, registraciju i odjavu. Slično tome, može postojati zasebni servis za upravljanje plovilima i korisničkim podacima, koji bi tretirali specifične zahtjeve vezane za te domene, kao što su dodavanje novih plovila, ažuriranje korisničkih profila ili dohvat informacija o postojećim korisnicima.

Datoteka api.js služi kao centralna točka za konfiguraciju *HTTP* klijenta (*Axios*) i definiranje osnovnih postavki kao što su *URL* poslužitelja, zaglavlja zahtjeva, token za autentifikaciju i sl. To omogućava aplikaciji da ima konzistentan pristup za sve *API* zahtjeve, olakšavajući upravljanje promjenama i održavanje koda.

Slika 35 Sadržaj api.js

```
const api = axios.create({
    baseURL: 'http://localhost:5000',
    withCredentials: true
});

api.interceptors.request.use(
    config => {
        const state = store.getState();
        const token = state.auth.token;
        if (token) {
            config.headers['Authorization'] =  `Bearer ${token}`
        }
        return config;
    },
    error => Promise.reject(error)
);

export default api;
```

Izvor: Obrada autora

Datoteka *authService.js* sadrži asinkrone funkcije koje se koriste za interakciju s autentifikacijskim *endpointima* definiranim na *backendu* aplikacije "*Rent a boat*". Svaka funkcija u ovoj datoteci upravlja specifičnim dijelom procesa autentifikacije.

Slika 36 Sadržaj authService.js

```
export const registerService = async (userData) => {
    try {
        const response = await api.post(`/auth/register`, userData);
        return response.data;
    } catch (error) {
        throw error.response.data;
    }
};

export const loginService = async (email, password) => {
    try {
        return await api.post(`/auth/login`, { email, password });
    } catch (error) {
        throw error;
    }
};

export const logoutService = async () => {
    try {
        const response = await api.post(`/auth/logout`);
        return response.data;
    } catch (error) {
        throw error.response.data;
    }
};
```

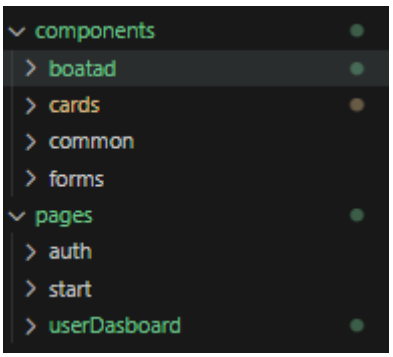
Izvor: Obrada autora

- *registerService* je funkcija koja upravlja registracijom novih korisnika. Ona šalje korisničke podatke, koje dobije kao argument, na *endpoint* za registraciju */auth/register* koristeći *HTTP POST* metodu. Ako je zahtjev uspješan, funkcija vraća podatke dobivene od servera. U slučaju greške, hvata iznimku i propušta odgovor greške dalje u tok izvršenja.
- *loginService* funkcija rukuje procesom prijave korisnika. Primajući email i lozinku kao argumente, šalje ih na *endpoint /auth/login*. Ako je autentifikacija uspješna i korisnik dobije token, funkcija vraća taj token za daljnju upotrebu u aplikaciji. Ako autentifikacija nije uspješna, funkcija baca iznimku sa greškom.
- *logoutService* upravlja odjavom korisnika. Funkcija šalje zahtjev na *endpoint /auth/logout* bez potrebe za dodatnim podacima. Kao i u prethodnim funkcijama, u slučaju uspjeha vraća podatke od servera, a u slučaju greške baca iznimku s podacima greške.

5.5. Komponente i stranice

U *React* aplikacijama, direktorij *components* i *pages* predstavljaju važne strukture kojima se organizira i definira korisničko sučelje.

Slika 37 Direktorij components i pages



Izvor: Obrada autora

Direktorij *components* sadrži modularne i ponovno upotrebljive dijelove koda, poznate kao komponente, koje se mogu koristiti na više mjesta unutar aplikacije. Te komponente obuhvaćaju specifične funkcionalnosti ili *UI* segmente kao što su dugmad, kartice, forme i navigacijski elementi.

S druge strane, direktorij *pages* sadrži komponente koje predstavljaju cijele stranice unutar aplikacije. Ove stranice su često povezane s određenim rutama u aplikaciji i služe kao kontejneri za grupiranje više komponenti kako bi se formirala kompletna korisnička stranica. Na primjer, *auth* direktorij sadrži stranice za prijavu i registraciju.

Ova organizacija omogućuje jasno razdvajanje *UI-a* po značajkama i funkcionalnostima, što pridonosi boljoj organizaciji projekta, lakšem održavanju i razvoju.

5.5.1. Komponente i Stranice: Prijava

Komponenta *LoginForm* u *React* aplikaciji *"Rent a boat"* predstavlja formular za prijavu korisnika. Komponenta prima funkciju *onSubmit* kao *prop*, koja se poziva kada korisnik pošalje formular. Ova funkcija je zadužena za daljnju obradu podataka za prijavu, poput komunikacije s *backendom.*

Slika 38 Sadržaj LoginForm.js

```
function LoginForm({ onSubmit, message }) {
  return (
    <div>
      <h1>Log In</h1>
      <Form
        onSubmit={onSubmit}
        validate={validateLogin}
        render={({ handleSubmit, submitting }) => (
          <form onSubmit={handleSubmit}>

            <Field name="email">
              {({ input, meta }) => (
                <div>
                  <input {...input} type="email" placeholder="Email" />
                  {meta.error && meta.touched && <span>{meta.error}</span>}
                </div>
              )}
            </Field>

            <Field name="password">
              {({ input, meta }) => (
                <div>
                  <input {...input} type="password" placeholder="Password" />
                  {meta.error && meta.touched && <span>{meta.error}</span>}
                </div>
              )}
            </Field>

            <button type="submit" disabled={submitting}>Log In</button>
          </form>
        )}
      />
    </div>
  );
}

export default LoginForm;
```

Izvor: Obrada autora

Formular koristi *<Form>* komponentu iz *react-final-form* biblioteke koja omogućava integraciju validacije i upravljanja stanjem formulara. *validate={validateLogin}* je funkcija koja se koristi za provjeru ispravnosti unesenih podataka prije slanja formulara. U formularu se nalaze dva polja: jedno za email i jedno za lozinku, definirana pomoću *<Field>* komponenti.

Komponenta stranice Login odgovorna je za proces prijave korisnika u aplikaciji "*Rent a boat*". Ova komponenta koristi *React hooks* kako bi omogućila interakciju sa *Redux store-om* i navigacijskim sposobnostima aplikacije.

Slika 39 Sadržaj stranice za prijavu

```
function Login() {
    const navigate = useNavigate();
    const dispatch = useDispatch();
    const { isAuthenticated } = useSelector(state => state.auth);

    const handleSubmit = values => {
        dispatch(login(values.email, values.password));
    };

    useEffect(() => {
        if (isAuthenticated) {
            navigate('/userwelcome');
        }

    }, [ isAuthenticated, navigate]);

    return (
        <LoginForm onSubmit={handleSubmit} />
    );
}

export default Login;
```

Izvor: Obrada autora

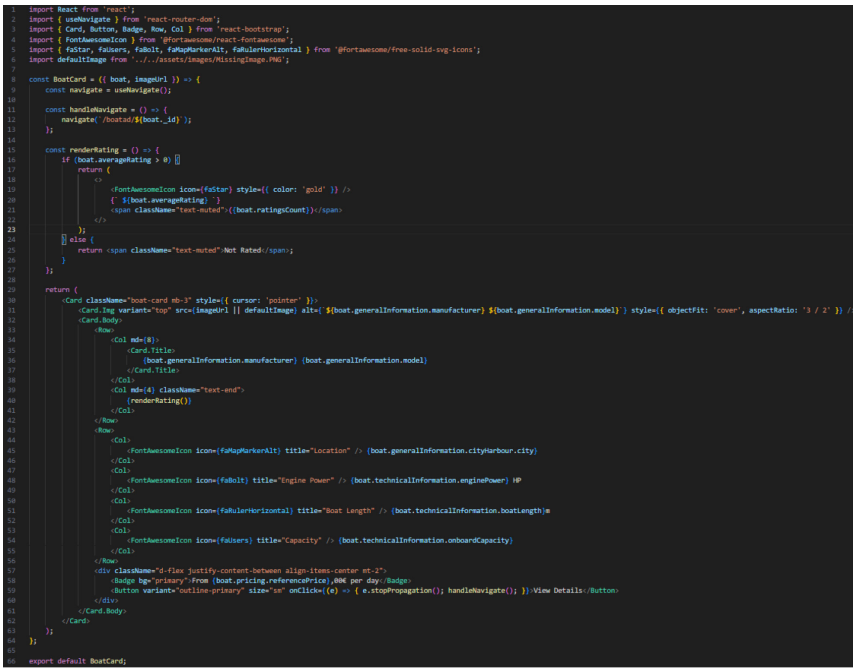
Hook useDispatch se koristi za slanje akcija u *Redux store*, dok *useSelector* omogućuje pristup stanju unutar *store-a*, u ovom slučaju posebno provjeravajući je li korisnik autentificiran. Ako je korisnik već autentificiran, *useEffect hook* će reagirati i preusmjeriti korisnika na stranicu dobrodošlice *(/userwelcome)*, koristeći *navigate* funkciju iz *hooka useNavigate*. Ovo osigurava da već prijavljeni korisnici ne mogu pristupiti formularu za prijavu.

Funkcija *handleSubmit,* koja se poziva prilikom slanja formulara, dispatchuje akciju login sa emailom i lozinkom koje korisnik unosi. Akcija login potom pokreće proces autentifikacije koji je definiran unutar *Redux* akcija.

Komponenta *LoginForm*, koja se renderira unutar Login komponente, prima *handleSubmit* kao prop za svoj *onSubmit* događaj. To znači da kada se formular u *LoginForm* komponenti pošalje, pokrenut će se logika definirana u *handleSubmit* unutar Login komponente.

5.5.2. Komponente i Stranice: Pretraživanje ponude plovila

Slika 40 Sadržaj BoatCard.js komponente



```
import React from 'react';
import { useNavigate } from 'react-router-dom';
import { Card, Button, Badge, Row, Col } from 'react-bootstrap';
import { FontAwesomeIcon } from '@fortawesome/react-fontawesome';
import { faStar, faUsers, faBolt, faMapMarkerAlt, faRulerHorizontal } from '@fortawesome/free-solid-svg-icons';
import defaultImage from '../../assets/images/MissingImage.PNG';

const BoatCard = ({ boat, imageUrl }) => {
    const navigate = useNavigate();

    const handleNavigate = () => {
        navigate(`/boatad/${boat._id}`);
    };

    const renderRating = () => {
        if (boat.averageRating > 0) {
            return (
                <>
                    <FontAwesomeIcon icon={faStar} style={{ color: 'gold' }} />
                    {` ${boat.averageRating} `}
                    <span className="text-muted">({boat.ratingsCount})</span>
                </>
            );
        } else {
            return <span className="text-muted">Not Rated</span>;
        }
    };

    return (
        <Card className="boat-card mb-3" style={{ cursor: 'pointer' }}>
            <Card.Img variant="top" src={imageUrl || defaultImage} alt={`${boat.generalInformation.manufacturer} ${boat.generalInformation.model}`} style={{ objectFit: 'cover', aspectRatio: '3 / 2' }} />
            <Card.Body>
                <Row>
                    <Col md={8}>
                        <Card.Title>
                            {boat.generalInformation.manufacturer} {boat.generalInformation.model}
                        </Card.Title>
                    </Col>
                    <Col md={4} className="text-end">
                        {renderRating()}
                    </Col>
                </Row>
                <Row>
                    <Col>
                        <FontAwesomeIcon icon={faMapMarkerAlt} title="Location" /> {boat.generalInformation.cityHarbour.city}
                    </Col>
                    <Col>
                        <FontAwesomeIcon icon={faBolt} title="Engine Power" /> {boat.technicalInformation.enginePower} HP
                    </Col>
                    <Col>
                        <FontAwesomeIcon icon={faRulerHorizontal} title="Boat Length" /> {boat.technicalInformation.boatLength}m
                    </Col>
                    <Col>
                        <FontAwesomeIcon icon={faUsers} title="Capacity" /> {boat.technicalInformation.onboardCapacity}
                    </Col>
                </Row>
                <div className="d-flex justify-content-between align-items-center mt-2">
                    <Badge bg="primary">From {boat.pricing.referencePrice},00€ per day</Badge>
                    <Button variant="outline-primary" size="sm" onClick={(e) => { e.stopPropagation(); handleNavigate(); }}>View Details</Button>
                </div>
            </Card.Body>
        </Card>
    );
};

export default BoatCard;
```

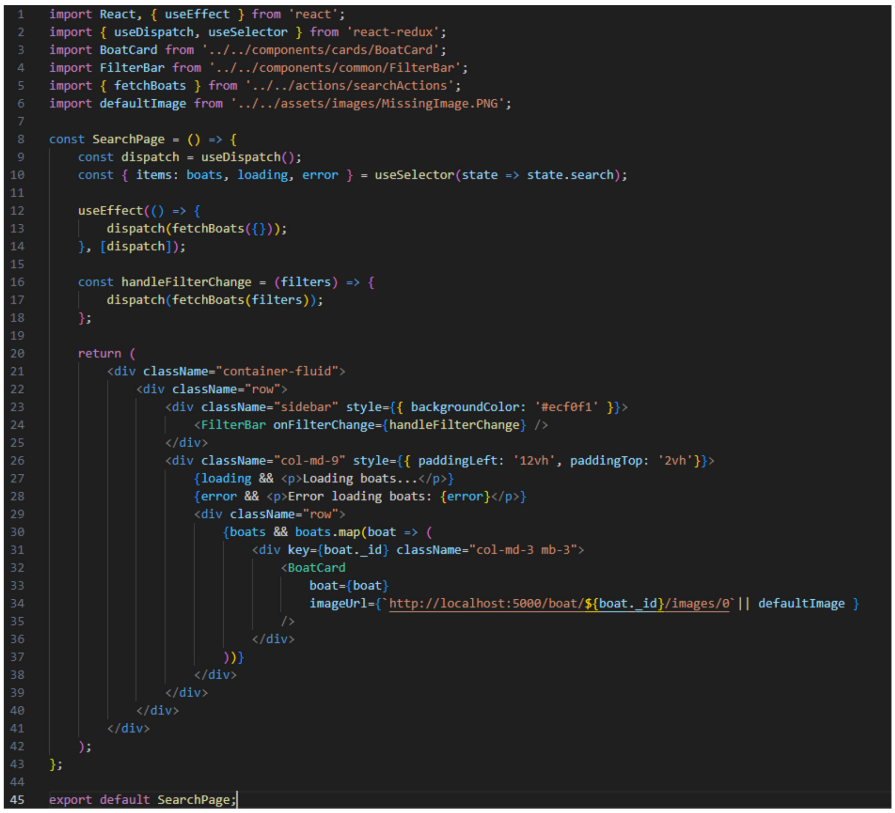
Izvor: Obrada autora

BoatCard je funkcionalna komponenta izgrađena koristeći *React.js*, koja uključuje integraciju s *React Routerom* putem *hooka useNavigate* za programsko usmjeravanje korisnika na detalje o plovilu. Kartica koristi *Bootstrap* za stiliziranje i *FontAwesome* za ikone, čime se postiže vizualno privlačan i informativan prikaz. Komponenta prima dva *propsa: boat*, koji sadrži detaljne informacije o plovilu, i *imageUrl*, putanju do slike plovila. Ukoliko slika nije dostupna, koristi se zadanu sliku iz direktorija '*assets*'. Unutar *BoatCard*, funkcija *renderRating* dinamički generira zvjezdani rejting plovila koristeći ikone *FontAwesome*, prikazujući prosječnu ocjenu i broj ocjena. Ako plovilo nije ocijenjeno, prikazuje se poruka *"Not Rated".* Glavni vizualni dio kartice je slika plovila, koja je stilizirana tako da zauzima proporcije 3:2, dok je sama kartica interaktivna - klik na nju vodi na stranicu s detaljima plovila. Informacije o plovilu su raspoređene u dva retka: prvi redak sadrži naziv proizvođača i model plovila, a drugi redak nudi brzi pregled ključnih tehničkih karakteristika plovila kao što su lokacija, snaga motora, dužina plovila i kapacitet.

Na dnu kartice, prikazana je cijena najma po danu unutar *Bootstrap* oznake, te gumb koji poziva funkciju *handleNavigate* prilikom klika, *inicijalizirajući* navigaciju prema stranici s detaljima plovila.

Nakon detaljnog pregleda *BoatCard* komponente, valja se osvrnuti na stranicu unutar aplikacije koja služi kao roditeljski kontejner za niz ovih kartica, a to je *SearchPage*. Ova stranica predstavlja dio aplikacije "*Rent a Boat*" gdje korisnici mogu pretraživati i pregledavati dostupna plovila.

Slika 41 Sadržaj SearchPage.js stranice



```
import React, { useEffect } from 'react';
import { useDispatch, useSelector } from 'react-redux';
import BoatCard from '../../components/cards/BoatCard';
import FilterBar from '../../components/common/FilterBar';
import { fetchBoats } from '../../actions/searchActions';
import defaultImage from '../../assets/images/MissingImage.PNG';

const SearchPage = () => {
    const dispatch = useDispatch();
    const { items: boats, loading, error } = useSelector(state => state.search);

    useEffect(() => {
        dispatch(fetchBoats({}));
    }, [dispatch]);

    const handleFilterChange = (filters) => {
        dispatch(fetchBoats(filters));
    };

    return (
        <div className="container-fluid">
            <div className="row">
                <div className="sidebar" style={{ backgroundColor: '#ecf0f1' }}>
                    <FilterBar onFilterChange={handleFilterChange} />
                </div>
                <div className="col-md-9" style={{ paddingLeft: '12vh', paddingTop: '2vh'}}>
                    {loading && <p>Loading boats...</p>}
                    {error && <p>Error loading boats: {error}</p>}
                    <div className="row">
                        {boats && boats.map(boat => (
                            <div key={boat._id} className="col-md-3 mb-3">
                                <BoatCard
                                    boat={boat}
                                    imageUrl={`http://localhost:5000/boat/${boat._id}/images/0` || defaultImage }
                                />
                            </div>
                        ))}
                    </div>
                </div>
            </div>
        </div>
    );
};

export default SearchPage;
```

Izvor: Obrada autora

SearchPage je *React* komponenta koja koristi *hookove* iz *React-Reduxa* za pristup stanju aplikacije i izvršavanje akcija. Korištenjem *useSelector hooka*, komponenta dohvaća stanje pretrage, koje uključuje popis plovila, informaciju o učitavanju i potencijalne greške. *Hook useEffect* se koristi za pokretanje akcije *fetchBoats* kada se komponenta učita, što pokreće zahtjev za dohvaćanje podataka o plovilima. Stranica također sadrži funkciju *handleFilterChange* koja se poziva kada korisnik promijeni filtere za pretraživanje. Ova funkcija dispatchuje akciju *fetchBoats* s novim filterima, omogućavajući da se rezultati pretrage dinamički ažuriraju bez potrebe za ponovnim učitavanjem stranice. U *JSX* dijelu *SearchPage* komponente, korišten je *Bootstrap grid* sustav za raspored elemenata. Sa strane je implementirana bočna traka (sidebar) sa *FilterBar* komponentom koja omogućuje korisnicima da specificiraju kriterije pretrage. Glavni dio stranice sadrži redove (*Row*) i stupce (*Col*) gdje svaki *BoatCard*

komponenta zauzima određeni prostor u *gridu*, čineći sučelje preglednim i odgovarajućim za različite veličine ekrana.

U slučaju da se podaci još uvijek učitavaju ili je došlo do greške, prikazuju se odgovarajuće poruke korisnicima, osiguravajući da korisničko iskustvo ostane netaknuto. Kad su podaci dostupni, korištenjem metode mapiranja *(map)* svako plovilo predstavljeno je kroz *BoatCard* komponentu koja prima relevantne *propse* kao što su informacije o plovilu i *URL* slike.

Primjer *SearchPage* komponente demonstrira kako se u *Reactu* komponente mogu učinkovito povezati i organizirati za stvaranje složenih korisničkih sučelja. Ova komponenta ne samo da pruža funkcionalnost pretraživanja i pregleda plovila, već i doprinosi konzistentnom i intuitivnom korisničkom iskustvu unutar "*Rent a Boat*" aplikacije.

6. Zaključak

Razvoj web aplikacije za najam plovila u Hrvatskoj, predstavljen u ovom radu, jasno ilustrira kako se inovativne tehnologije mogu efikasno primijeniti u turističkoj industriji, posebno u segmentu usluga najma plovila. Aplikacija je u skladu s početnom hipotezom o stvaranju konkurentne i korisnički orijentirane platforme na hrvatskom tržištu najma plovila.

Kroz rad, detaljno je istraženo kako kombinacija *frontend* i *backend* tehnologija - *React.js, Redux, Node.js, Express.js* i *MongoDB* - doprinosi stvaranju efikasnog, skalabilnog i intuitivnog web sustava. *Frontend*, razvijen uz upotrebu React.js i *Reduxa*, pruža dinamično i responzivno korisničko sučelje, dok *backend*, zasnovan na *Node.js, Express.js i MongoDB*, osigurava pouzdanost i efikasnost sustava, ključne za upravljanje i pohranu podataka.

Ostvareni su zadani ciljevi: dizajniranje funkcionalne web aplikacije, uspješna implementacija tehnologija te optimizacija za izvanredno korisničko iskustvo. Agilna metodologija razvoja, upotreba *UML* dijagrama i *REST* arhitekture omogućili su efikasan razvoj i implementaciju aplikacije, dok je *Redux*, kao centralizirani sustav za upravljanje stanjem, bio neophodan za održavanje konzistentnosti i predvidljivosti aplikacije.

U konačnici, "Rent a Boat" aplikacija ne samo da zadovoljava trenutne potrebe tržišta, već je i dobro pozicionirana da se prilagodi budućim trendovima i inovacijama u industriji najma plovila. Ovaj rad ne samo da demonstrira napredne tehnološke prakse u razvoju web aplikacija, već i jasno pokazuje kako se takve aplikacije mogu uspješno implementirati i koristiti u stvarnom poslovnom svijetu, čime potvrđuje važnost i relevantnost tehnoloških inovacija u turističkom sektoru.

7. Literatura

1. Node.js (pristupljeno 20.10.2023)

Web pristup: https://nodejs.org/docs/latest/api/

2. Express.js (pristupljeno 20.10.2023)

Web pristup: https://expressjs.com/en/guide/

3. MongoDB (pristupljeno 20.10.2023)

Web pristup: https://www.mongodb.com/docs/

4. React Redux (pristupljeno 18.11.2023)

Web pristup: https://react-redux.js.org/using-react-redux/

5. React.js (pristupljeno 18.11.2023)

Web pristup: https://react.dev/learn

6. React Final Form (pristupljeno 19.11.2023)

Web pristup: https://final-form.org/docs/react-final-form/getting-started

7. React Router (pristupljeno 18.11.2023)

Web pristup: https://reactrouter.com/en/main/start/tutorial (pristupljeno 18.11.2023)

8. Chrome DevTools (pristupljeno 18.11.2023)

Web pristup: https://developer.chrome.com/docs/devtools/resources

8. Popis Slika