Agilni razvoj web aplikacije za pomoć pri liječenju životinja zasnovane na automatiziranoj kontejnerskoj arhitekturi

Kramar, Filip

SVEUČILIŠTE JOSIPA JURJA STROSSMAYERA U OSIJEKU
FAKULTET ELEKTROTEHNIKE, RAČUNARSTVA I INFORMACIJSKIH TEHNOLOGIJA OSIJEK

Sveučilišni studij

Agilni razvoj web aplikacije za pomoć pri liječenju životinja zasnovane na automatiziranoj kontejnerskoj arhitekturi

Diplomski rad

Filip Kramar

Osijek, 2024.

FERIT
FAKULTET ELEKTROTEHNIKE, RAČUNARSTVA I INFORMACIJSKIH TEHNOLOGIJA OSIJEK

Obrazac D1: Obrazac za ocjenu diplomskog rada na sveučilišnom diplomskom studiju

Ocjena diplomskog rada na sveučilišnom diplomskom studiju

Ime i prezime pristupnika:	Filip Kramar
Studij, smjer:	Sveučilišni diplomski studij Računarstvo
Mat. br. pristupnika, god.	D1295aR, 07.10.2022.
JMBAG:	0165081880
Mentor:	prof. dr. sc. Goran Martinović
Sumentor:	
Sumentor iz tvrtke:	
Predsjednik Povjerenstva:	izv. prof. dr. sc. Ivica Lukić
Član Povjerenstva 1:	prof. dr. sc. Goran Martinović
Član Povjerenstva 2:	izv. prof. dr. sc. Mirko Köhler
Naslov diplomskog rada:	Agilni razvoj web aplikacije za pomoć pri liječenju životinja zasnovane na automatiziranoj kontejnerskoj arhitekturi
Znanstvena grana diplomskog rada:	**Programsko inženjerstvo (zn. polje računarstvo)**
Zadatak diplomskog rada:	U teorijskom dijelu diplomskog rada treba opisati izazove u liječenju životinja, arhitekture temeljene na kontejnerima, te mogućnosti agilnog razvoja web aplikacija. Također, treba analizirati stanje u području i slična rješenja. Nadalje, potrebno je definirati funkcionalne i nefunkcionalne zahtjeve na web aplikaciju, predložiti model i arhitekturu web aplikacije, prikladne metodologije agilnog razvoja i automatiziranu kontejnersku arhitekturu uz potporu pristupa CI/CD za ažuriranje inačica programskog koda na kontejneru. Web aplikacija treba omogućiti
Datum ocjene pismenog dijela diplomskog rada od strane mentora:	17.09.2024.
Ocjena pismenog dijela diplomskog rada od strane mentora:	Izvrstan (5)
Datum obrane diplomskog rada:	26.09.2024.
Ocjena usmenog dijela diplomskog rada (obrane):	Izvrstan (5)
Ukupna ocjena diplomskog rada:	Izvrstan (5)
Datum potvrde mentora o predaji konačne verzije diplomskog rada čime je pristupnik završio sveučilišni diplomski studij:	28.09.2024.

FERIT
FAKULTET ELEKTROTEHNIKE, RAČUNARSTVA I INFORMACIJSKIH TEHNOLOGIJA **OSIJEK**

IZJAVA O IZVORNOSTI RADA

Osijek, 28.09.2024.

Ime i prezime Pristupnika:	Filip Kramar
Studij:	Sveučilišni diplomski studij Računarstvo
Mat. br. Pristupnika, godina upisa:	D1295aR, 07.10.2022.
Turnitin podudaranje [%]:	4

Ovom izjavom izjavljujem da je rad pod nazivom**: Agilni razvoj web aplikacije za pomoć pri liječenju životinja zasnovane na automatiziranoj kontejnerskoj arhitekturi**

izrađen pod vodstvom mentora prof. dr. sc. Goran Martinović

i sumentora

moj vlastiti rad i prema mom najboljem znanju ne sadrži prethodno objavljene ili neobjavljene pisane materijale drugih osoba, osim onih koji su izričito priznati navođenjem literature i drugih izvora informacija.
Izjavljujem da je intelektualni sadržaj navedenog rada proizvod mog vlastitog rada, osim u onom dijelu za koji mi je bila potrebna pomoć mentora, sumentora i drugih osoba, a što je izričito navedeno u radu.

Potpis pristupnika:

SADRŽAJ

1. UVOD

Liječenje životinja suočava se s posebnim izazovima koji se razlikuju od onih u ljudskoj medicini. Nedostatak standardiziranih protokola za različite vrste životinja može otežati donošenje odluka i osiguranje dosljedne kvalitete skrbi. Komunikacija između veterinara i vlasnika može biti otežana zbog specifičnih potreba životinjskih pacijenata i složenih medicinskih pojmova. Osim toga, trenutni sustavi u veterinarskim stanicama često su zastarjeli i koriste se samo interno, što može otežati pristup informacijama iz drugih stanica i stvoriti izazove u dosljednosti i sigurnosti. Ovi problemi ukazuju na potrebu za modernizacijom sustava za prikupljanje i upravljanje podacima o pacijentima kako bi se poboljšala koordinacija i kvaliteta skrbi.

Cilj ovog diplomskog rada je napraviti web aplikaciju koja je namijenjena svim veterinarskim stanicama kako bi prilikom posjeta različitim stanicama u različitim gradovima imale isti sustav i isti pristup podacima. Aplikacija će uključivati funkcionalnosti poput centralizirane baze podataka o ljubimcima, praćenja povijesti liječenja, ispisa propisanih lijekova i terapija te automatizacije procesa isporuke proizvoda korištenjem automatizirane kontejnerske arhitekture. Za ostvarivanje funkcionalnosti web aplikacije koristit će se suvremeni alati: Spring Boot na poslužiteljskoj strani, Angular na klijentskoj strani, PostgreSQL i Liquibase za bazu podataka i praćenje verzija baze podataka te Docker i GitHub Actions za ostvarivanje automatizirane kontejnerske strukture.

Diplomski rad strukturiran je tako da je prvo opisano trenutno stanje područja veterinarstva, agilnog razvoja i automatizacije kontejnerske strukture, što je pokriveno u drugom poglavlju. Također, u drugom poglavlju prikazana su slična rješenja i aplikacije vezane za tematiku ovog diplomskog rada. Treće poglavlje prikazuje pojedine zahtjeve zadane za web aplikaciju koju ovaj rad treba napraviti te prikazuje model i strukturu razvijenog web sustava. U četvrtom poglavlju pokrivena je teorijska pozadina izvedenog web sustava, što uključuje alate, programske jezike, programske okvire, programe te prikazuje izvedeno programsko rješenje rada za prije opisane zahtjeve web aplikacije i implementaciju automatizirane kontejnerske strukture. Peto poglavlje opisuje način korištenja aplikacije nad različitim uobičajenim scenarijima i analizira rezultate ostvarenog web sustava i automatizirane kontejnerske arhitekture.

2. PREGLED STANJA U PODRUČJU I SLIČNA RJEŠENJA

U sljedećem poglavlju detaljnije će se opisati izazovi i problemi specifičnih područja koja su u fokusu ovog diplomskog rada, kao i relevantna rješenja sličnih tema. Svako od obrađenih područja nosi specifične izazove koji mogu uključivati tehničke, organizacijske ili metodološke poteškoće.

2.1. Izazovi i stanje u području liječenja životinja

Veterinari se suočavaju s problemima sličnima onima s kojima se susreću i ljudski liječnici, poput složenosti određivanja bolesti na temelju određenih simptoma, nepredvidljivosti reakcija na određene lijekove i slično. Međutim, prilikom liječenja životinja suočavaju se s dodatnim problemima. „Životinjski modeli povijesno su igrali ključnu ulogu u istraživanju i razumijevanju patofiziologije bolesti, identifikaciji ciljeva za liječenje i evaluaciji novih terapijskih sredstava“ [1]. Različite vrste životinja s kojima se veterinari susreću razlikuju se po veličini i vrsti, svaka sa svojom posebnom anatomijom i podložnošću određenim bolestima, što dodatno otežava određivanje točne dijagnoze i odabir odgovarajuće terapije. Za razliku od ljudi, životinje ne mogu dati izravnu povratnu informaciju o boli niti odgovoriti na pitanja koja bi olakšala tretman; npr., životinje kao što su mačke dobro skrivaju bol, pa vlasnici često niti ne primijete da nešto nije u redu s njihovim ljubimcem sve dok stanje ne postane ozbiljno.

U današnjem vremenu, tema unaprjeđenja veterinarske medicine postaje sve važnija, osobito u svjetlu nedavne epidemije COVID-19, uzrokovane zoonotskim virusom SARS-CoV-2. Mnogi članci, poput „SARS-CoV-2 Infection in Animals: A Review“, ističu da „detaljno proučavanje zaraze SARS-CoV-2 kod životinja ne samo da pomaže u razumijevanju patogeneze bolesti, već i pruža važne informacije za kontrolu i prevenciju zoonotskih bolesti“ [2], što dodatno dokazuje da je epidemija potaknula istraživanje životinjskih bolesti i njihovih tretmana, ističući potrebu za naprednim metodama i pristupima u veterinarskoj medicini.

2.2. Izazovi i stanje područja agilnoga razvoja

„Iako agilne metodologije postoje manje od dva desetljeća, njihova popularnost stalno raste“ [3]. Prije nego što su se pojavile agilne metode, za razvoj programskog rješenja koristile su se tradicionalne razvojne metode, koje su bile znatno sporije, imale manji kontakt s klijentom, uključivale duža vremena čekanja i kasnu detekciju grešaka u razvoju, što je često bila razlika između uspjeha i propadanja projekata. No, „nedostaje jasnosti u vezi s načinom na koji timovi donose i procjenjuju mnoštvo odluka od početka značajke programskog rješenja do isporuke i usavršavanja proizvoda“ [4].

Nejasni ili nedefinirani zahtjevi mogu dovesti do nesuglasica između klijenata i timova, što može rezultirati neodređenostima u prioritetima zadataka. Budući da je u agilnom razvoju fokus na timskom radu, ako timovi nisu usklađeni ili ako komunikacija nije jasna, to može uzrokovati nesporazume i neučinkovitost. Agilni razvoj zahtijeva visoku razinu samostalnosti, ali i suradnje unutar tima.

Zbog prirode agilnog razvoja, koja uključuje definiranje „manjih koraka“, često dolazi do prekomjernog fokusiranja na kratkoročne ciljeve, dok se veća slika projekta zanemaruje. Kada se koristi agilna metodologija za razvoj programa, puno vremena može otići na ceremonije kao što su planiranje sprintova, obrađivanje zadataka, određivanje težine pojedinog zadatka, dnevni sastanci, sprint retro i slično, te programerima ne ostane puno vremena da ispune svoje zadatke. Također, ponekad je teško unaprijed odrediti težinu pojedinog zadatka, što može rezultirati prekovremenim radom i nezadovoljstvom, kako s strane programera, tako i s strane klijenata zbog slabijih rezultata.

Dodatni izazov nastaje kada se projekt prethodno razvijao koristeći drukčiju metodologiju. Prema istraživanjima, „prilagodba tima na agilne metode može biti teška kada su prethodno koristili različite razvojne metodologije“ [3]. Ovi problemi ukazuju na potrebu za pažljivim planiranjem i upravljanjem tijekom prijelaza između metodologija kako bi se minimizirali negativni utjecaji i osigurao uspješan razvoj projekta.

Danas oko 90% IT tvrtki koristi agilne metodologije u nekom obliku tijekom razvoja proizvoda, s najčešće korištenim metodologijama kao što su Scrum i Kanban. Zbog same popularnosti agilnih metodologija i učestalosti korištenja, postoje razni alati koji podržavaju agilni razvoj, poput Jire, OpenProject, Trello itd. Svaki od alata ima svoje prednosti i mane, no svi alati primarno olakšavaju tvrtkama korištenje agilnih metodologija i prelazak s tradicionalnog ili hibridnog pristupa.

2.3. Izazovi i stanje u području automatizirane kontejnerske arhitekture

Neki od izazova ostvarivanja kontejnerske arhitekture su što zahtijevaju duboko razumijevanje DevOps praksi, rad s datotekama za stvaranje Docker slika i kontejnera te upravljanje cjevovodima pomoću alata kao što je Jenkinsfile. Također, jedan od problema je korištenje alata poput GitHub Actions. Iako su GitHub Actions fleksibilne i dopuštaju prilagodbu, može biti teško implementirati neke specifične značajke pod uvjetom da te značajke nije razvila zajednica ili nisu dokumentirane. To može zahtijevati puno vremena za čitanje, razumijevanje i možda prilagodbu nekih već

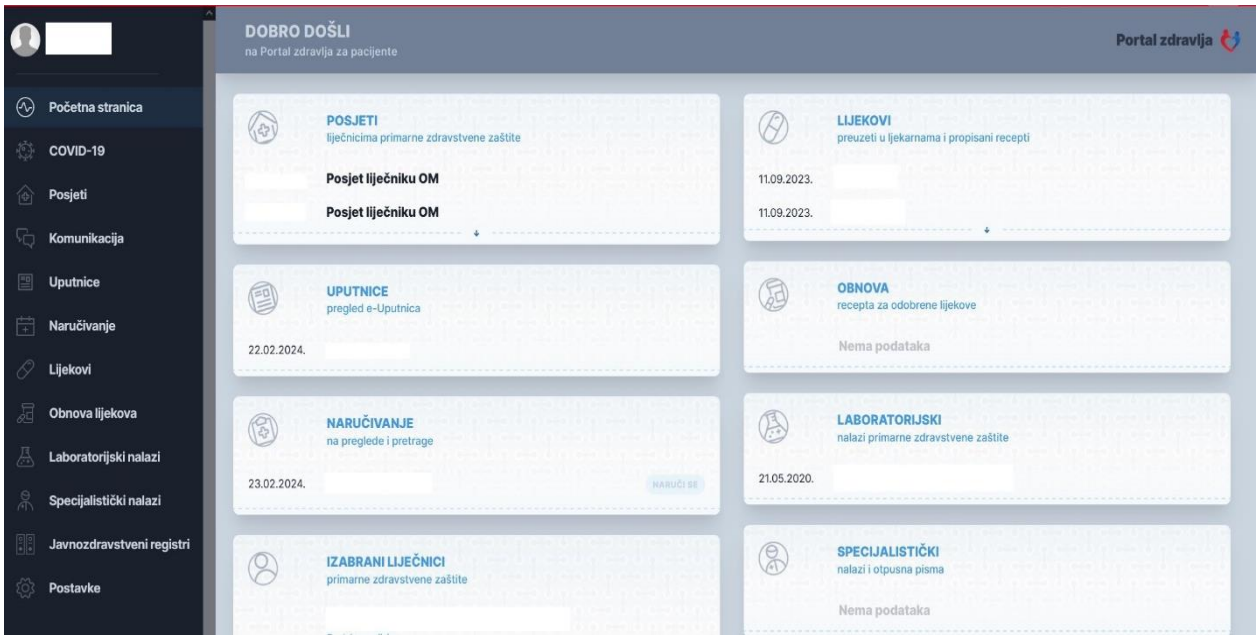
razvijenih rješenja ili čak za razvoj vlastitih rješenja ako je funkcionalnost vrlo složena. Osim dodatnog istraživanja, korištenje tuđih alata može značajno dovesti u rizik sigurnost poslovnih tajni [5].

Danas mnoge tvrtke koriste kontejnerske arhitekture kako bi poboljšale efikasnost i fleksibilnost u razvoju i isporuci aplikacija klijentima. Kontejnerske tehnologije, poput Dockera, omogućuju dosljedna razvojna okruženja, dok alati za upravljanje cjevovodima, poput Jenkinsa, Argoa i GitHub Actions, podržavaju automatsku integraciju i isporuku.

2.4. Postojeće slične aplikacije

Područje veterinarstva, kao i druge grane medicine, svakodnevno se poboljšava i digitalizira. Na tržištu postoje razne aplikacije koje se bave tematikom liječenja životinja, medicinom i automatiziranom kontejnerskom arhitekturom. U nastavku su predstavljena neka od postojećih rješenja koja se bave sličnim temama.

Portal zdravlja za pacijente u sklopu e-Građana omogućuje korisnicima da lako pristupe informacijama o zdravlju, zakazuju preglede i prate svoje medicinske podatke. S Portalom zdravlja smanjuje se nepotrebno čekanje u čekaonicama te se omogućuje uniformno sučelje za sve doktore unutar Hrvatske [6]. Na slici 2.1 prikazano je grafičko sučelje Portala zdravlja.



Slika 2.1 Grafičko sučelje Portala Zdravlja

Razvijena je internetska stranica i Android aplikacija koja korisnicima pruža uslugu prijavljivanja i udomljavanja ljubimaca s ciljem smanjenja broja lutalica i napuštenih ljubimaca u azilima i drugim udrugama [7].

Slična tematika obrađena je u diplomskom radu u kojem je izrađena internetska aplikacija u razvojnome okviru Laravel, namijenjena vlasnicima pasa. U tom radu pruža se platforma korisnicima koja omogućava pristup svim informacijama i uslugama vanjskih izvora za pse, kao što su šetači, dadilje pasa, udruge za zaštitu životinja i slično [8]. Za razliku od tog rada, ovaj rad nudi uniformnu platformu veterinarima i vlasnicima ljubimaca, ne samo pasa, nego i drugih životinja, te digitalni medicinski karton njihovih ljubimaca bez potrebe korištenja drugih stranica ili platformi.

Automatiziranu kontejnersku strukturu koristio je kolega Branimir Valentić, koji je dokumentirao korake kontinuirane integracije i kontinuirane dostave za tvrtku Base 58, koristeći GitLab, Gradle i druge alate. U svojim blogovima opisuje tijek rada koji provodi za automatiziranu isporuku programa te postupke i važnost verzioniranja izdanja. Također je pružio osnovne predloške koji se mogu prilagoditi drugim projektima ili koristiti kao inspiracija za lakše provođenje automatiziranih procesa u našim projektima [9].

PetDesk je internetska aplikacija koja služi za upravljanje zdravljem kućnih ljubimaca, dizajnirana da pojednostavi živote vlasnika ljubimaca [10]. Korisnicima pruža uniformno mjesto za praćenje medicinske dokumentacije svojih ljubimaca, upravljanje rasporedom cijepljenja, postavljanje podsjetnika za lijekove i rezerviranje veterinarskih termina. *PetDesk* je integriran s velikim brojem veterinarskih stanica, što omogućuje izravnu komunikaciju s veterinarima te pristup aktualnim zdravstvenim informacijama, no nije dostupan za hrvatsko tržište.

3. ZAHTJEVI, MODEL I GRAĐA WEB APLIKACIJE

U nastavku se detaljno opisuju zahtjevi za web aplikaciju, s fokusom na ključne funkcionalnosti kao što su prijava i registracija korisnika te stvaranje e-zdravstvenog kartona ljubimca. Također se razmatra arhitektura i struktura aplikacije na klijentskoj i poslužiteljskoj strani, kao i u bazi podataka, uz naglasak na raspodjelu zadataka putem alata Jira tijekom agilnog razvoja.

3.1. Zahtjevi za web aplikaciju

Prije započinjanja programske implementacije, potrebno je jasno definirati zahtjeve za web aplikaciju. Ovi zahtjevi dijele se na funkcionalne i nefunkcionalne. Funkcionalni zahtjevi obuhvaćaju specifične funkcionalnosti koje aplikacija mora ispuniti, poput prijave i registracije korisnika, stvaranja e-zdravstvenog kartona ljubimca i zakazivanja posjeta veterinaru. Nefunkcionalni zahtjevi uključuju aspekte kao što su performanse, sigurnost, korisničko sučelje i pristupačnost, koji su ključni za kvalitetu i korisničko iskustvo aplikacije.

3.1.1. Prijava i registracija korisnika

Prilikom učitavanja web aplikacije, korisniku se prikazuje forma za unos adrese e-pošte i lozinke. Nakon unošenja podataka, informacije se šalju s klijentske strane na poslužiteljsku stranu i obrađuju. Ako korisnik postoji u bazi podataka, prosljeđuje se na glavnu stranicu aplikacije; u suprotnom, prikazuje se odgovarajuća greška na korisničkom sučelju.

Ako korisnik ne postoji u bazi podataka ili prvi put pristupa aplikaciji i želi napraviti račun, prikazuje se forma s podacima kao što su ime, prezime, adresa e-pošte i lozinka. Nakon prikupljanja podataka, oni se spremaju u bazu podataka, a korisnik se prosljeđuje na stranicu za prijavu.

3.1.2. Stvaranje e-zdravstvenog kartona ljubimca

Prilikom učitavanja glavne stranice, korisniku se nude opcije izbora postojećeg ljubimca ili stvaranja novog kartona za ljubimca. Ako korisnik odabere opciju izbora postojećeg ljubimca, prikazuje mu se lista njegovih ljubimaca.

Ako korisnik odabere opciju stvaranja novog medicinskog kartona za ljubimca, prikazuje mu se obrazac za stvaranje novog kartona koji sadrži podatke o vrsti, pasmini, imenu ljubimca i opcionalno broj mikročipa.

3.1.3. Prikaz posjeta veterinaru i prikaz računa

Korisnik ima opciju prikazivanja posjeta veterinaru za odabranog ljubimca. Nakon odabira opcije prikaza posjeta veterinaru, prikazuje se lista posjeta s podacima o veterinaru, ljubimcu, propisanim lijekovima, primijenjenim terapijama, datumom posjete i razlogom posjete.

Korisnik može odabrati jedan od posjeta kako bi dobio detalje o posjetu veterinaru u obliku računa koji sadrži informacije o veterinaru, ljubimcu te primijenjenim terapijama i lijekovima iz posjete.

3.1.4. Prikaz primijenjenih terapija i lijekova

Nakon odabira jednog od ljubimaca, korisniku se nudi opcija prikazivanja primijenjenih terapija i postupaka za odabranog ljubimca. Kada korisnik pritisne opciju za prikaz primijenjenih terapija i postupaka, ispisuje se lista s podacima o opisima terapija, tipu, cijeni i listi korištenih materijala. Korisnik može odabrati bilo koju od terapija kako bi dobio detaljniji opis odabrane terapije.

Osim prikazivanja primijenjenih terapija, korisnik ima mogućnost pregledavanja svih lijekova propisanih za odabranog ili unaprijed dogovorenog ljubimca. Nakon što pritisne opciju za prikaz lijekova, ispisuje se popis veterinarskih lijekova s podacima o nazivu lijeka, dozi, uputama, te datumu izdavanja ili datumu isteka recepta. Korisnik može odabrati jedan od lijekova kako bi dobio detalje za odabrani lijek.

3.1.5. Zakazivanje termina posjete veterinaru

Nakon odabira ljubimca, korisniku se pruža mogućnost zakazivanja termina za posjetu veterinaru. Kada korisnik pritisne opciju za zakazivanje posjete, prikazuje se formular s podacima za izbor veterinara, kao i mogućnost odabira vremena i datuma posjete.

3.1.6. Prijava veterinara

Ukoliko je korisnik veterinar, prikazuje mu se forma za unos adrese e-pošte i lozinke. Nakon što korisnik unese podatke, informacije se šalju s klijentske strane na poslužiteljsku stranu i obrađuju. Ako veterinar postoji u bazi podataka, korisnik se prosljeđuje na glavnu stranicu aplikacije; u suprotnom, na korisničkom sučelju prikazuje se odgovarajuća greška.

3.1.7. Dodavanje detalja posjete

Ukoliko je korisnik veterinar, prikazuje mu se lista posjeta. Nakon odabira pojedinog posjeta, veterinar može dodatno uređivati informacije o posjeti dodavanjem primijenjenih terapija i lijekova.

3.1.8. Nefunkcionalni zahtjevi

Aplikacija mora imati responzivan i intuitivan dizajn s kratkim vremenom odziva i minimalnim kašnjenjem. Također, aplikacija mora biti održiva kroz modularni dizajn i jasnu organizaciju koda, te pratiti SOLID (eng. *Single Responsibility Principle, Open/Closed Principle, Liskov Substitution Principle, Interface Segregation Principle, Dependency Inversion Principle*) načela. Uz to, aplikacija mora biti pouzdana s učinkovitim upravljanjem pogreškama.

S obzirom na složenost alata korištenih za izradu web aplikacije, nefunkcionalni zahtjevi nisu eksplicitno definirani u kodu, već su podržani raznim bibliotekama i funkcionalnostima koje okviri pružaju. Za responzivnost i intuitivan dizajn u programskom okviru Angular koriste se Angular Material i Flex Layout.

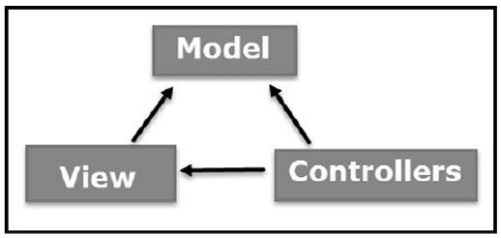
Kako bi se ostvario modularni dizajn i jasna organizacija koda, slijede se definirani standardi u industriji prilikom razvoja web aplikacija, uključujući arhitekturu model-pogled-kontroler, koja osigurava odvajanje poslovne logike, korisničkog sučelja i podataka.

3.2. Model i arhitektura aplikacije

Aplikacije na klijentskoj i poslužiteljskoj strani slijede uzorak model-pogled-kontroler, koji razdvaja aplikaciju u tri glavne komponente [11], kao što je prikazano na slici 3.1:

- Model – komponenta odgovorna za upravljanje podacima, uključujući spremanje, dohvaćanje i manipulaciju podacima. Također može sadržavati poslovnu logiku, poput validacije podataka i drugih logičkih operacija koje su potrebne prije prosljeđivanja podataka
- View - komponenta koja je odgovorna za prikazivanje podataka korisniku. Služi kao korisničko sučelje aplikacije i zadužena je za jednostavno prikazivanje podataka. Podaci koje View prikazuje primaju se od Modela
- Controller – komponenta koja povezuje View i Model. Obrađuje ulazne podatke, komunicira s Modelom za dohvaćanje i ažuriranje podataka

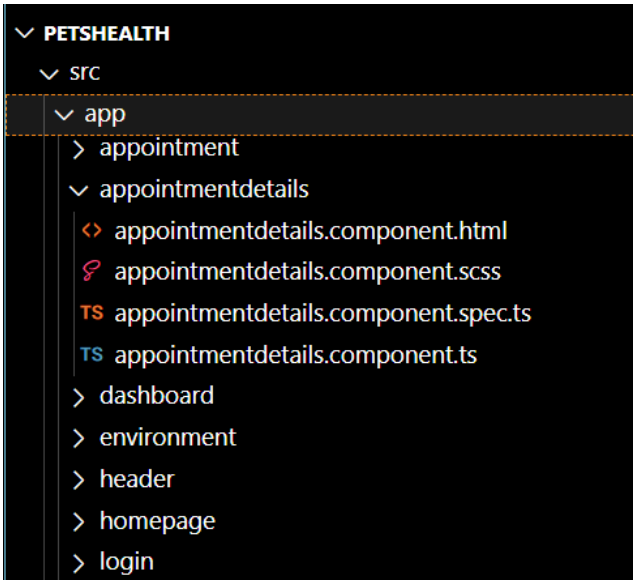
Uzorak model-pogled-kontroler omogućuje modularnost i fleksibilnost, kao i lakše održavanje i proširivanje aplikacije. Svaka komponenta može se neovisno razvijati jedna o drugoj. Budući da se koriste različiti programski okviri, imenovanje komponenti može se razlikovati od gore navedenih.



Slika 3.1 Dijagram toka unutar uzorka model-pogled-kontroler [11]

3.2.1. Struktura aplikacije na klijentskoj strani

„Struktura Angular aplikacije sastoji se od niza komponenti, pri čemu svaka komponenta uključuje HTML, SCSS, spec.ts i TypeScript datoteke, koje zajedno omogućuju razvoj modularnog korisničkog sučelja“ [12]. Na slici 3.2 prikazana je struktura pojedine Angular komponente unutar web aplikacije. Svaka od tih datoteka ima specifičnu ulogu u definiranju izgleda, funkcionalnosti i testiranja komponente:

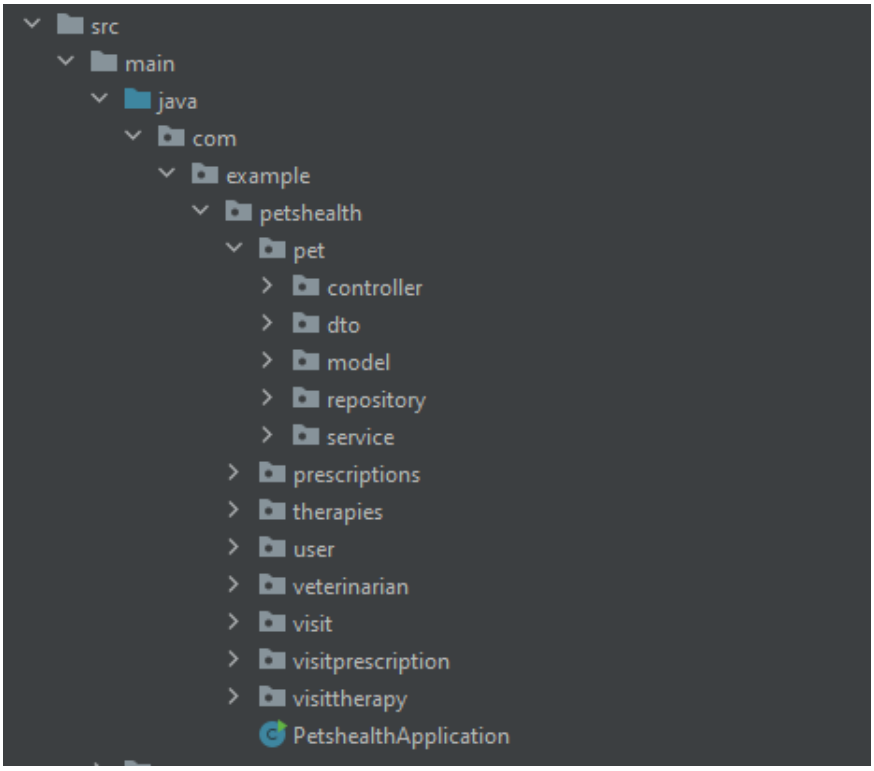


Slika 3.2 Struktura Angular komponente

- component.html – datoteka koja sadrži HTML predložak komponente, tj. kako će komponenta izgledati na korisničkom sučelju
- component.scss – datoteka koja opisuje stilove komponente i omogućuje prilagodbu dizajna i izgleda
- component.spec.ts – datoteka koja sadrži jedinične testove za provjeru funkcionalnosti komponente
- component.ts – datoteka koja sadrži TypeScript skripte za ostvarivanje logike komponente, u kojoj se definira klasa komponente, atributi i pozivaju usluge itd.

3.2.2. Struktura aplikacije na poslužiteljskoj strani

Slika 3.3 prikazuje zbirku entiteta koji čine aplikaciju na poslužiteljskoj strani. Svaki entitet se sastoji od sljedećih dijelova:



Slika 3.3 Prikaz strukture aplikacije na poslužiteljskoj strani

- **Controller** - ovdje su pohranjene krajnje točke koje obrađuju HTTP zahtjeve. Oni komuniciraju s uslugama kako bi dohvatili ili promijenili podatke te vraćaju odgovarajući odgovor korisniku
- **Model** - mapa koja sadrži entitete ili klase modela koje predstavljaju podatkovne strukture korištene unutar aplikacije. Oni obično odgovaraju zapisima u bazi podataka; jedan model predstavlja jednu tablicu, a jedan stupac tablice je jedan atribut modela
- **DTO (Data Transfer Object)** - definirane klase za prijenos podataka između različitih slojeva aplikacije. DTO-ovi pomažu u prenošenju samo potrebnih podataka i poboljšavaju sigurnost i performanse. Na primjer, prilikom prijave korisnika, nisu potrebni ime i prezime, nego samo adresa e-pošte i lozinka
- **Service** - uslužni sloj obavlja poslovnu logiku aplikacije. Ovdje se definiraju metode koje obavljaju operacije nad podacima i komuniciraju s repozitorijima

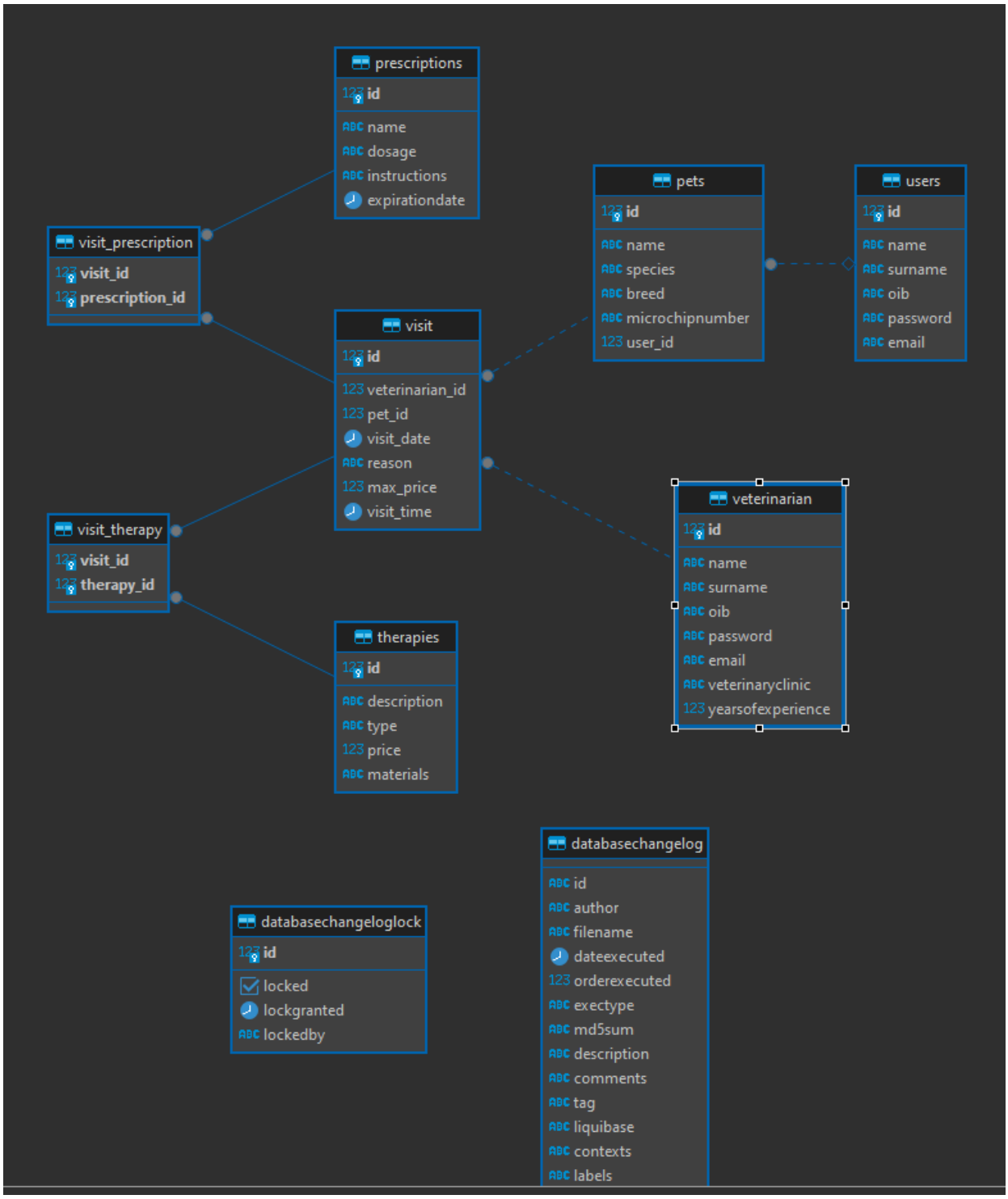
- **Repository** – sloj koji obuhvaća klase koje se bave pristupom podacima, a direktno komunicira s bazom podataka radi dohvaćanja, ažuriranja, brisanja i drugih logičkih operacija koje se izvode nad podacima

3.2.3. Struktura baze podataka aplikacije

Na slici 3.4 prikazan je dijagram entiteta i veza koji vizualno prikazuje tablice korištene unutar ove web aplikacije, njihove međusobne odnose i pojedine atribute. Baza podataka sastoji se od sljedećih tablica:

- *users* - sadrži podatke vezane za korisnike, kao što su ime, prezime, adresa e-pošte, kontakt informacije i drugi relevantni podaci potrebni za registraciju i prijavu u sustav
- *veterinarian* - predstavlja veterinara i sadrži informacije, kao što su ime, prezime, ime klinike u kojoj radi, godine iskustva, specijalizacije te kontakt podatke, što omogućuje lakšu identifikaciju i odabir veterinara od strane korisnika
- *pets* - sadrži podatke o ljubimcima, kao što su ime, vrsta ljubimca (npr. pas, mačka), porodica (npr. Labrador, Persijska mačka), vlasnik i broj mikročipa, što omogućuje praćenje i upravljanje podacima o svakom ljubimcu
- *prescriptions* - opisuje razne lijekove za ljubimce, uključujući doze lijekova, datume kada su propisani i eventualne napomene, čime se osigurava da su svi podaci o terapijama lako dostupni
- *therapies* - bilježi primijenjene terapije ljubimca, kao što su vrsta terapije, cijena terapije, trajanje terapije i druge važne informacije, čime se osigurava detaljan pregled svih tretmana koji su provedeni na ljubimcima
- visits - ova tablica integrira podatke iz većine drugih tablica jer sadrži gotovo sve strane ključeve, kao što su posjete veterinara, i omogućava praćenje svih aspekata skrbi o ljubimcima, od dijagnoze do terapije

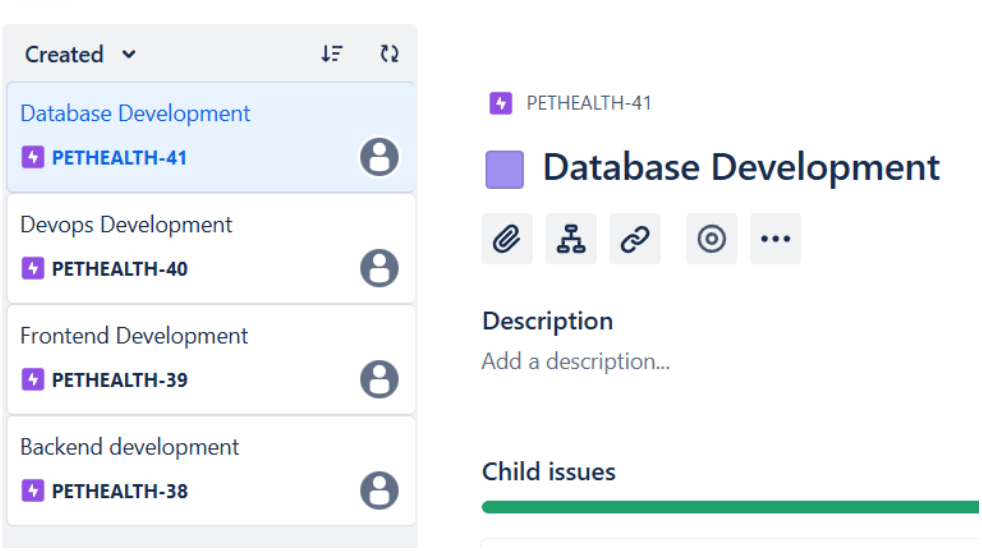
Kako bi se ostvario potreban odnos više-prema-više, uvedene su dodatne tablice visit_therapy i visit_prescription. Tablice *databasechangeloglock* i *databasechangelog* su automatski generirane od strane Liquibase alata za praćenje promjena primijenjenih na bazi podataka.



Slika 3.4 dijagram entiteta i veza baze podataka

3.2.4. Struktura zadataka u okviru agilnog razvoja

Jira ploča strukturirana je kroz četiri glavna epika: razvoj poslužiteljske strane, razvoj klijentske strane, razvoj baze podataka i razvoj kontejnerske arhitekture (slika 3.5). Svaki zadatak kategoriziran je unutar jednog od definiranih epika.



Slika 3.5 Struktura Jira *epika*

Osim pripadnosti određenom epiku, prilikom definiranja zadataka u Jira ploči unutar uglatih zagrada definira se oznaka koja dodatno opisuje područje koje pojedini zadatak pokriva (slika 3.6). Na primjer, oznaka *'frontend'* odnosi se na razvoj klijentske strane, *'backend'* označava razvoj poslužiteljske strane, *'db'* označava razvoj baze podataka, a *'devops'* se odnosi na razvoj kontejnerske arhitekture. Ove oznake pomažu u lakšem identificiranju specifičnih područja na koja se zadaci odnose, osobito kada dijele isto ime kao neki drugi zadatak.

PETHEALTH-9	[DB] Set Up Postgress and Liquibase
PETHEALTH-8	[BACKEND] User Registration
PETHEALTH-7	[FRONTEND] User Registration

Slika 3.6 Imena zadataka s njihovim oznakama

4. PROGRAMSKO RJEŠENJE OSTVARENOG WEB APLIKACIJE

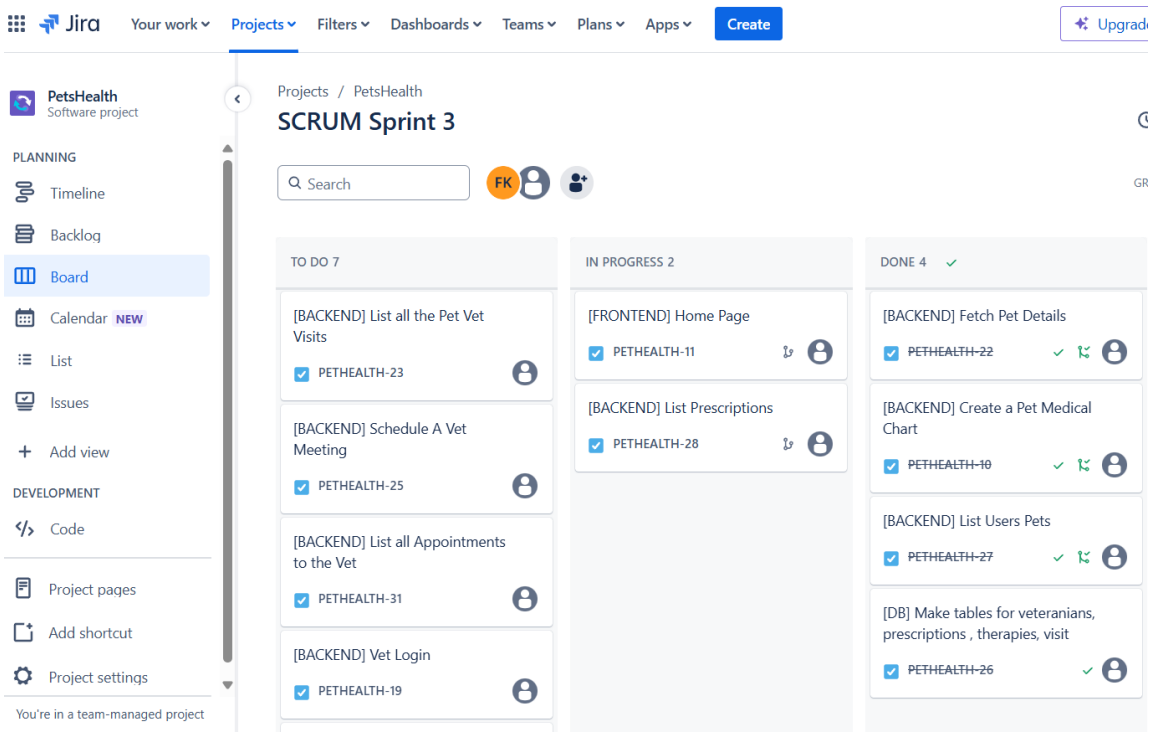
Nakon što su u prethodnom poglavlju teorijski opisani funkcionalni i nefunkcionalni zahtjevi za web aplikaciju, u ovom dijelu diplomskog rada proći će se kroz programsko rješenje ostvarene web aplikacije. Osim analize ostvarene funkcionalnosti, u poglavlju će se također opisati korišteni alati i programski okviri koji su primijenjeni za ostvarivanje rješenja.

4.1. Korištene tehnologije i alati

U ovom podpoglavlju bit će opisane tehnologije i alati koji su korišteni tijekom razvoja web aplikacije. Fokusirat će se na programske okvire, jezike i alate koji su pridonijeli izradi aplikacije.

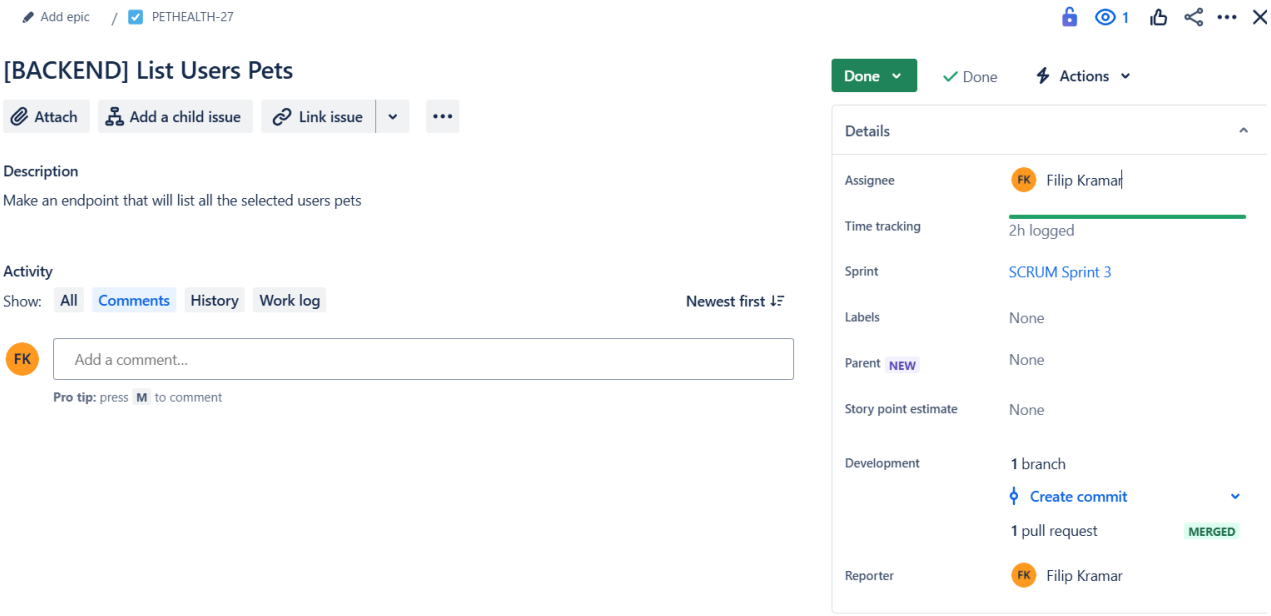
4.1.1. Jira

Jira je jedan od najpoznatijih alata za upravljanje projektima. „Alat je nastao 2002. godine, a danas se koristi u preko 300.000 tvrtki zato što podržava bilo koji tip projekta, kao i široku potporu vanjskih aplikacija (Confluence, Trello, Slack, GitHub i mnogi drugi)“ [13]. Također, neovisan je o broju korisnika i podržava manje timove, ali i velike kompleksne organizacije. Alat podržava razne metodologije, bilo to agilne metodologije kao što su Scrum i Kanban ili tradicionalne (model vodopada i V-model). Na slici 4.1 nalazi se jedna od Scrum ploča na Jiri s definiranim zadacima.



Slika 4.1 Primjer Scrum ploče u Jiri

Najveća prednost alata je što poboljšava timsku efektivnost, praćenje napretka i koordinaciju aktivnosti unutar tima, tj. svi članovi tima u bilo kojem trenutku mogu vidjeti što se radi, tko je zadužen za što i koji su sljedeći koraci, kao što je vidljivo na slici 4.2, gdje je prikazan jedan od zadataka na Jiri, iz kojega se može očitati tko radi na zadatku, koliko je vremena potrošeno itd. Jira je besplatna za timove do 10 ljudi i pruža osnovne funkcije za upravljanje projektima, ali za veće timove se plaća.



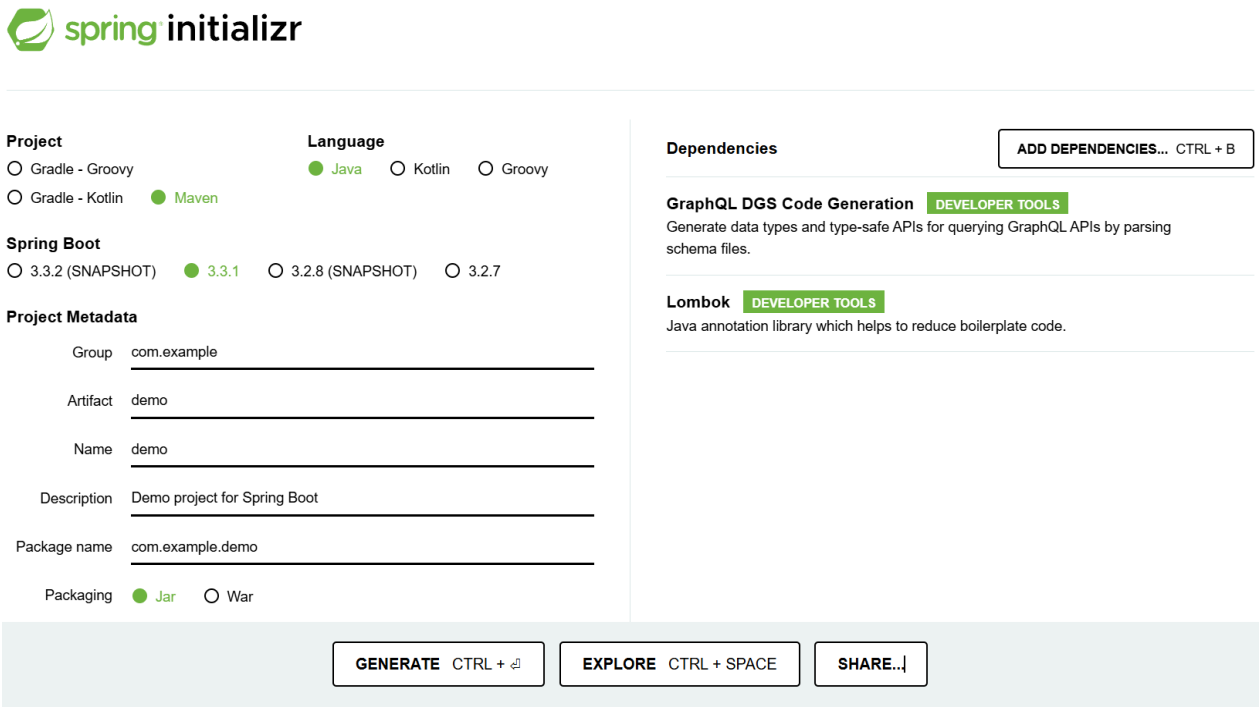
Slika 4.2 Primjer Jira zadatka

4.1.2. Spring Boot

„Spring Boot je programski okvir temeljen na programskom jeziku Java. Dizajniran je kako bi ubrzao proces razvoja i konfiguracije poslužiteljskih aplikacija te za izgradnju arhitektura mikrousluga“ [14]. Omogućuje automatsku konfiguraciju komponenata aplikacija na temelju obrazaca korištenja i dolazi s ugrađenim web poslužiteljem Tomcat.

Podržava alate poput Mavena, Gradlea i Groovyja, koji olakšavaju upravljanje ovisnostima i sadrže veliku količinu biblioteka kao što su Spring Security za sigurnost, Spring Data JPA za rad s bazama podataka, Lombok za pojednostavljenje Java koda i mnoge druge. Kompatibilan je s raznim bazama podataka (MySQL, PostgreSQL, MongoDB…).

Dolazi s alatom Spring Initializr (slika 4.3) koji omogućuje jednostavno generiranje početnih projekata s konfiguracijom po izboru. Ovaj alat je dostupan i kao web aplikacija i kao integrirani dio raznih razvojnih okruženja.

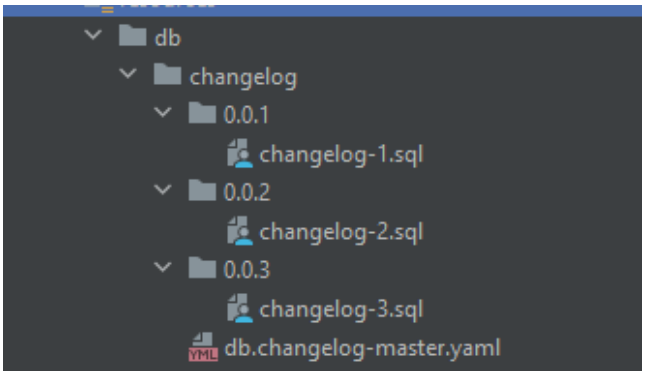


Slika 4.3 Spring Initializr web stranica

4.1.3. Liquibase

Liquibase je alat za upravljanje promjenama sheme baze podataka koji omogućuje bržu i sigurniju reviziju i implementaciju promjena od razvoja do proizvodnje. Dizajniran je kako bi timovi mogli jednostavno i sigurno upravljati promjenama u bazi podataka. Kao i Spring Boot, Liquibase podržava različite baze podataka.

Tijekom prvog pokretanja, Liquibase primjenjuje promjene definirane u datotekama navedenim u *db.changelog-master.yaml* (slika 4.5) i generira potrebne tablice prema SQL skriptama. Ovo uključuje kreiranje, modificiranje, popunjavanje i druge operacije nad tablicama (slika 4.4). Ako se dogodi promjena u datotekama nakon inicijalnog pokretanja, Liquibase detektira promjene i sprječava direktnu modifikaciju postojećih tablica u bazi podataka. Umjesto toga, potiče generiranje nove verzije datoteke, čime se osigurava da sve promjene budu dokumentirane i upravljane na siguran način.



Slika 4.4 Liquibase skripte unutar web aplikacije

```
databaseChangeLog:

 - include:
     file: db/changelog/0.0.1/changelog-1.sql

 - include:

     file: db/changelog/0.0.2/changelog-2.sql

 - include:

     file: db/changelog/0.0.3/changelog-3.sql
```

Slika 4.5 db.changelog-master.yaml datoteka

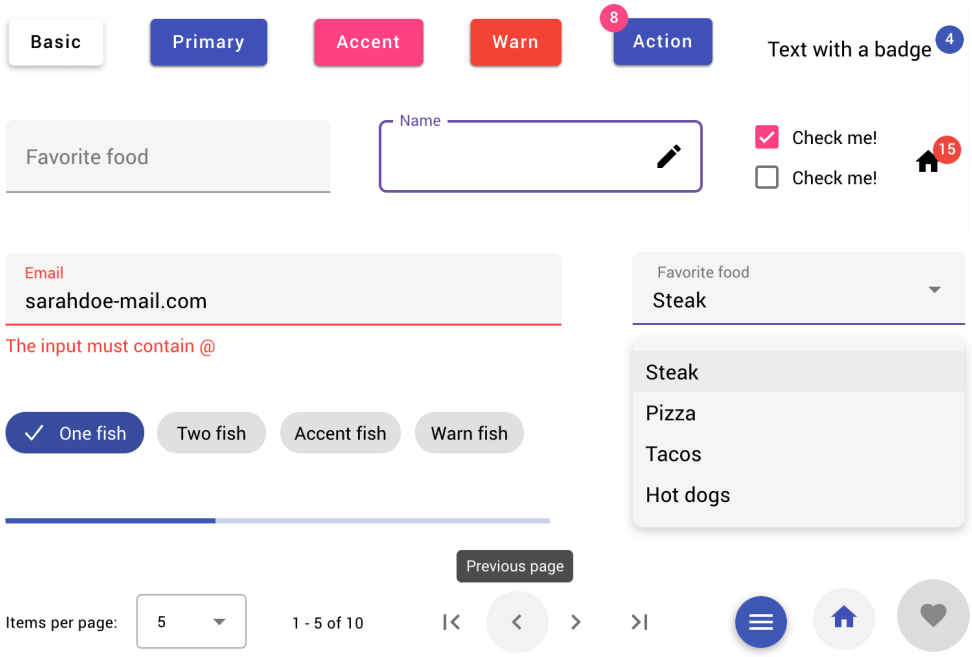
4.1.4. Angular

Angular je okvir otvorenog koda, originalno napisan u jeziku JavaScript, koji je razvio i održava Google, a pruža standardnu strukturu i dodatne značajke za pojednostavljenje razvoja web i mobilnih aplikacija. „U današnjem vremenu, Angular stoji kao robustan okvir na razini poduzeća, omiljen zbog svojih snažnih značajki, opsežnog alata i snažne podrške zajednice, što podcrtava njegovo značenje u krajoliku modernog web razvoja“ [12].

Angular koristi komponente gdje je korisničko sučelje podijeljeno u višekratno upotrebljive samostalne dijelove, što olakšava upravljanje i održavanje složenih aplikacija. Okvir uključuje ugrađeni sustav ubrizgavanja ovisnosti koji pomaže u upravljanju životnim ciklusom komponenti i usluga.

Piše se u jeziku TypeScript, jeziku koji je nadskup JavaScripta, ali dodaje statično tipiziranje te poboljšava produktivnost programera i kvalitetu koda. Podržava razne alate i dodatke koji olakšavaju korištenje okvira, kao što su Angular Material, koji nudi razne gotove komponente u

skladu s Material Design smjernicama (slika 4.6 preuzeta s [15]), FxLayout, koji omogućava jednostavno upravljanje rasporedom elemenata unutar komponente, i mnoge druge.



Slika 4.6 Primjeri gotovih komponenti unutar Angular Materiala [15]

4.1.5. PostgreSQL

„PostgreSQL je moćan objektno-relacijski sustav baze podataka otvorenog koda s više od 35 godina aktivnog razvoja, koji mu je priskrbio snažnu reputaciju pouzdanosti, robusnosti značajki i performansi“ [16]. Omogućuje korisnicima definiranje vlastitih tipova podataka, prilagođenih funkcija i pisanje koda iz različitih programskih jezika bez potrebnog ponovnog kompiliranja baze podataka.

PostgreSQL podržava značajke koje zahtijevaju SQL standard i podržan je u većini programskih jezika i programskih okvira. Ključna prednost PostgreSQL-a je podrška za napredne tipove podataka, kao što su JSON i XML, što omogućuje fleksibilno rukovanje strukturiranim i nestrukturiranim podacima. Također podržava transakcije s višestrukim verzijama, čime se osigurava visoka razina izolacije transakcija i omogućava paralelno izvršavanje bez zaključavanja.

PostgreSQL je podržan od mnogih alata za prikazivanje baza podataka, poput DBeavera, pgAdmin, DataGrip i mnogih drugih.

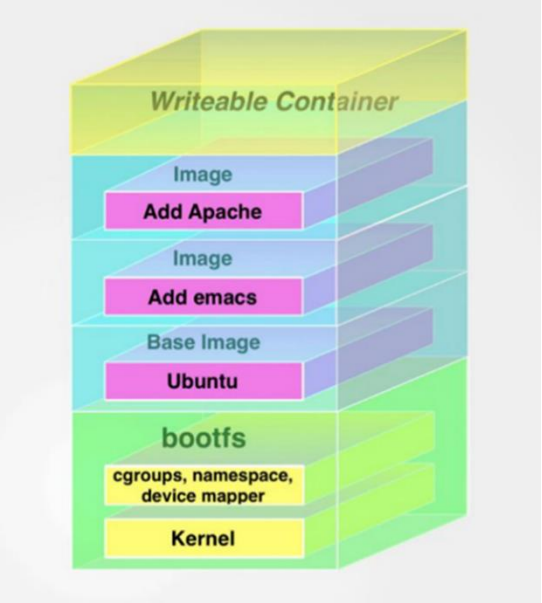
4.1.6. Docker

Docker je open-source platforma koja programerima omogućuje izgradnju, implementaciju, pokretanje, ažuriranje i upravljanje kontejnerima. Kontejneri su virtualna okruženja koja

omogućuju pakiranje i izvršavanje aplikacija i njihovih ovisnosti neovisno o vanjskim postavkama i dostupnim alatima računala na kojima se pokreću.

Svaki kontejner temelji se na Docker slici, koja sadrži sve potrebne komponente za pokretanje aplikacije, uključujući sav kod i ovisnosti“ [17]. Slike se najčešće kreiraju pomoću datoteke Dockerfile i mogu se pohraniti u spremišta poput Docker Huba za ponovno korištenje. Docker slike se mogu koristiti i u alatima koji se koriste na oblaku, poput Kubernetesa, kao osnovna slika za kreiranje pojedinačnih Pod jedinica.

Kontejneri dijele jezgru operacijskog sustava s poslužiteljskim sustavom, te su zato manje resursno zahtjevni i imaju veću efikasnost i produktivnost nego virtualni strojevi (slika 4.7 preuzeta sa [17]).



Slika 4.7 Unutarnja struktura Docker kontejnera [17]

Uz to, Docker nudi alate kao što su Docker Compose za definiranje i upravljanje višestrukim kontejnerima kao jedinstvenom aplikacijom, te Docker Swarm za orkestraciju kontejnera u višestrukim poslužiteljskim okruženjima radi skaliranja i visoke dostupnosti.

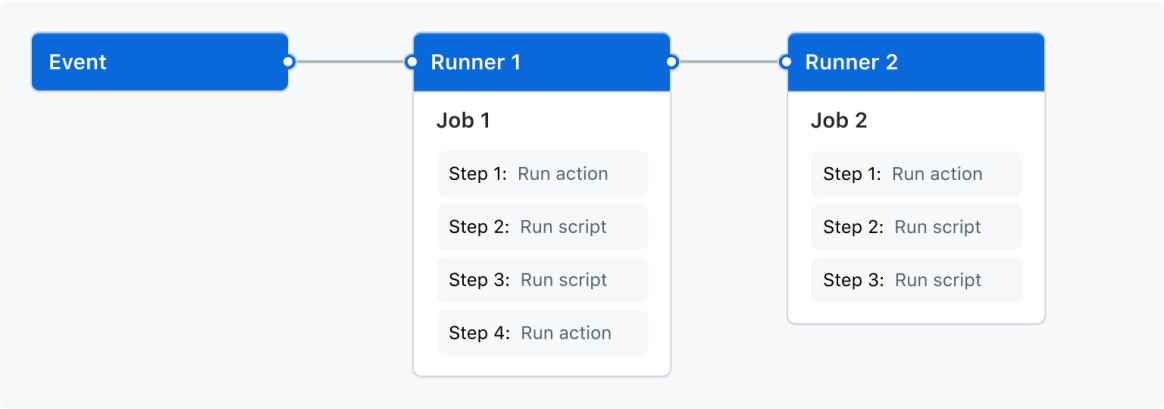
4.1.7. GitHub Actions

GitHub Actions je platforma za kontinuiranu integraciju i kontinuiranu isporuku koja omogućuje automatiziranje izgradnje, testiranja i implementacije. GitHub Actions nastao je 2019. godine, što ga čini „relativno novim“ alatom, a razvijen je kao alternativni pristup kontinuiranoj integraciji i dostavi na GitHub repozitorijima u usporedbi s naprednijim alatima kao što su Jenkins i Argo.

Integriran je u GitHub, te se uz pomoć GitHub Actions mogu definirati akcije prilikom događaja na specificiranoj GitHub grani. Budući da je integriran s GitHubom, nudi intuitivno korisničko sučelje koje jasno prikazuje radne tijekove, uključujući pojedine korake, njihove ispise i status.

Podržava širok spektar programskih jezika i okvira, što doprinosi fleksibilnosti i jednostavnosti korištenja. Može se definirati na različite okidače, na primjer, na *push* naredbu na određenoj grani ili čak ručnim pokretanjem. Nakon pokretanja radnog tijeka izvršavaju se definirani koraci pod sekcijom *jobs* slijedno (slika 4.8 preuzeta s [19]).

„Također, nudi predefinirane radne tijekove u sklopu GitHub Marketplacea za zadatke kao što su procesi za kontinuiranu integraciju i kontinuiranu isporuku, automatsko testiranje“ [18], te omogućuje korisniku da definira svoje vlastite radne tijekove i podijeli ih sa zajednicom.



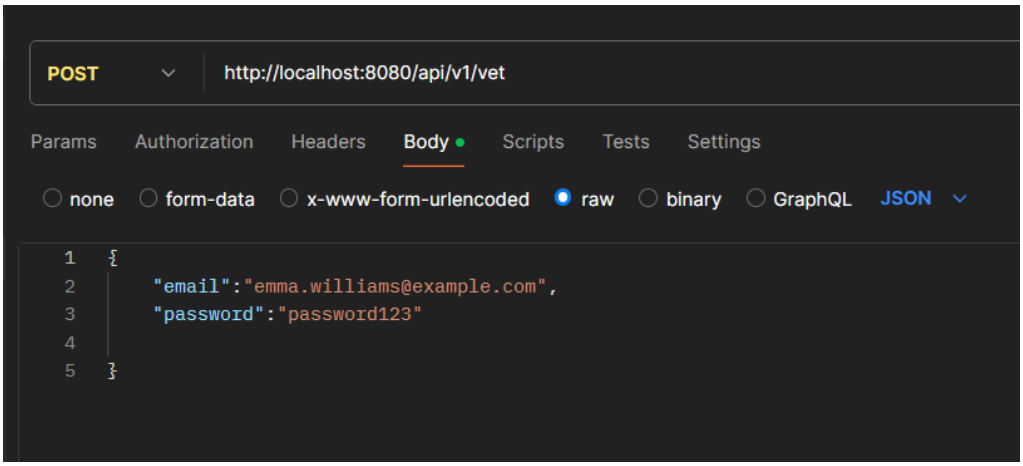
Slika 4.8 Grafički prikaz izvođenja jednog tijeka rada [19]

4.1.8. Postman

„Postman je besplatni API razvojni alat koji pomaže u izradi, testiranju i modificiranju API-ja“ [20]. Omogućuje postavljanje različitih vrsta HTTP i HTTPS zahtjeva (GET, POST, PUT, PATCH), spremanje okruženja za kasniju upotrebu te pretvaranje API-ja u kod za različite jezike (kao što su JavaScript i Python) uz automatizaciju testova putem kolekcija testova.

Iako se primarno koristi za slanje REST zahtjeva, podržava i SOAP protokol. Postman omogućuje rad s različitim formatima podataka poput JSON-a i XML-a, generira dokumentaciju na temelju kolekcija, te nudi mogućnost kreiranja poslužitelja za simulacije s definiranim odgovorima i zahtjevima. Osim osnovnih funkcija, Postman podržava i značajke kao što su pretraživanje unutar kolekcija, praćenje promjena i dijeljenje okruženja među članovima tima, što dodatno poboljšava suradnju.

Korisničko sučelje alata Postman (slika 4.9) intuitivno je i lagano za korištenje, što ga čini prikladnim čak i za početnike. Dodatno, mogućnost izvoza i uvoza kolekcija olakšava suradnju između različitih projekata, a integracija s drugim alatima za automatizaciju i procese kontinuirane integracije i dostave omogućuje bržu i efikasniju izradu i održavanje API-ja.



Slika 4.9 Sučelje Postman aplikacije

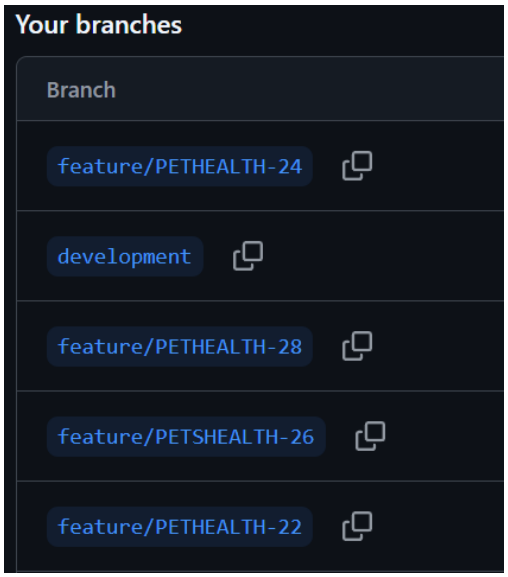
4.2. Programsko rješenje web aplikacije

U ovom poglavlju predstavljaju se programska rješenja koja su implementirana za web aplikaciju koristeći odabrane tehnologije. Fokus je stavljen na konkretne dijelove programskog koda, radne tijekove i metode koje su korištene za izvođenje i optimizaciju svake funkcionalnosti.

4.2.1. Integracija agilnog razvoja u sklopu programskog rješenja

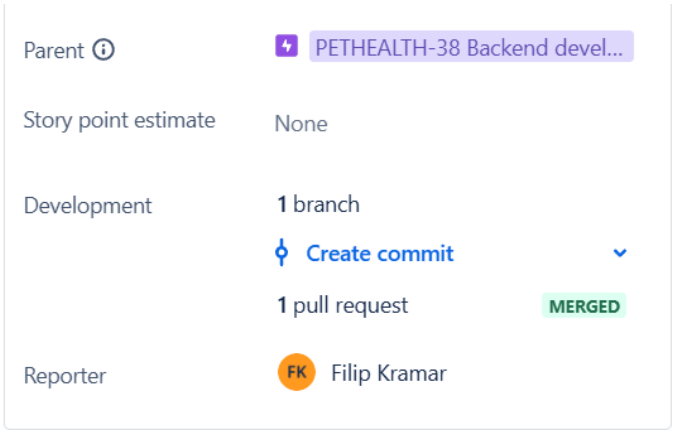
S obzirom na korištenje agilne metodologije i alata Jira, važno je povezati grane na GitHubu s odgovarajućim Jira zadacima. Povezivanjem grana s Jira zadacima omogućuje se automatsko ažuriranje statusa zadatka na temelju promjena u GitHubu, čime se poboljšava preglednost i koordinacija unutar tima. Ova integracija ne samo da pomaže u održavanju točnosti informacija o napretku, već i povećava efikasnost timskog rada, omogućujući brže reakcije na promjene.

Prilikom kreiranja grane na GitHubu, potrebno je uključiti identifikacijski ključ zadatka (slika 4.10). Ovaj ključ služi kao referenca koja povezuje konkretne promjene u kodu s određenim zadatkom u Jira sustavu, čime se olakšava praćenje napretka. Iako nije obvezno dodavati prefiks „feature/“ prilikom imenovanja grane, preporučuje se to raditi kako bi se uskladilo s pravilima imenovanja grana na GitHubu.



Slika 4.10 Prikaz *GitHub* grana projekta

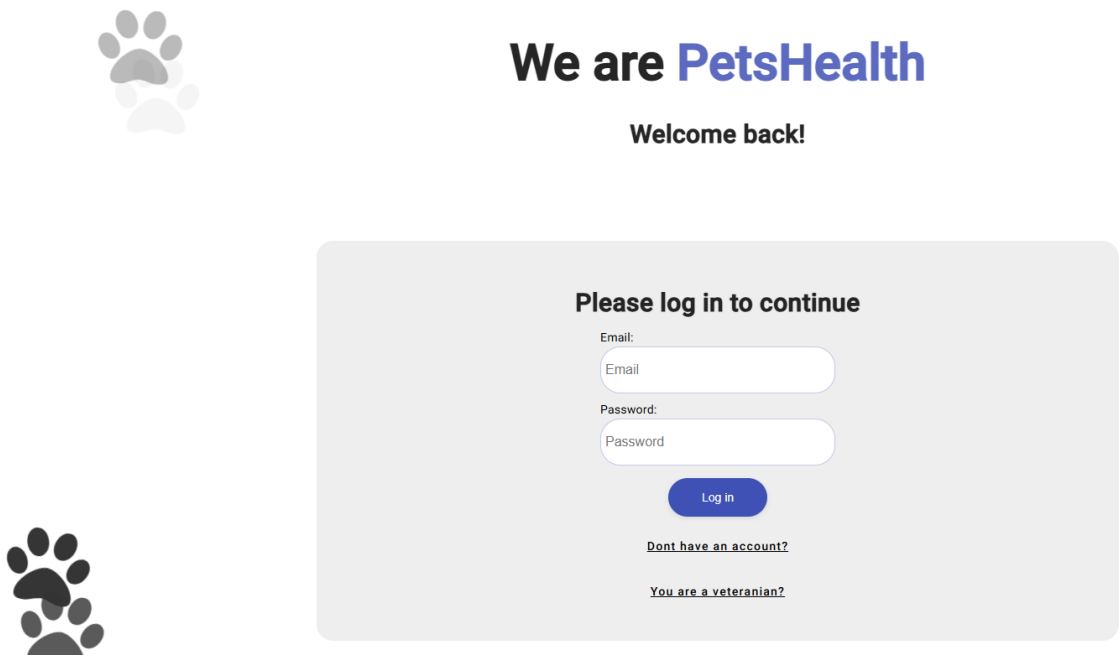
Nakon što je grana kreirana, trebala bi biti vidljiva unutar odgovarajućeg Jira zadatka. Osim prikazivanja povezane grane, sučelje Jira zadatka prikazuje broj promjena primijenjenih na granu, kao i stanje zahtjeva za integraciju (eng. *Pull Request*) (slika 4.11).



Slika 4.11 Jira prikaz grana i promjena

4.2.2. Programsko rješenje prijave i registracije korisnika

Prilikom pokretanja web aplikacije, korisniku se prikazuje forma za prijavu u sustav koja se može vidjeti na slici 4.12. Korisnik upisuje svoju adresu e-pošte i lozinku, te se pritiskom na dugme za prijavu poziva funkcija *collectFormData* (slika 4.13) koja prikuplja podatke. Funkcija *collectFormData* prosljeđuje podatke usluzi kako bi se podaci proslijedili s klijentske strane na poslužiteljsku stranu preko REST protokola (slika 4.14).



Slika 4.12 Početna stranica web aplikacije

```
@Component({
  selector: 'app-login',
  templateUrl: './login.component.html',
  styleUrls: ['./login.component.scss']
})
export class LoginComponent {
  constructor(private apiRequestService:ApirequestService){}
  loginFormGroup: FormGroup = new FormGroup({
    email: new FormControl('', Validators.required),
    password: new FormControl('', Validators.required),
  });
  collectFormData() {
    this.apiRequestService.loginUser(this.loginFormGroup.value);
  }
}
```

Slika 4.13 Definicija *collectFormData* funkcije

```
loginUser(data: any) {
  const url = `${apiUrl.key}users`;
  this.http.post<number>(url, data).subscribe(
    (Id: number) => {
      this.router.navigate(['/home/dashboard']);
      sessionStorage.setItem('isUser', 'true');
      sessionStorage.setItem('userid', Id.toString());
    },
    (error) => {
      alert('Wrong email or password');
    }
  );
}
```

Slika 4.14 Metoda *loginUser* na klijentskoj strani

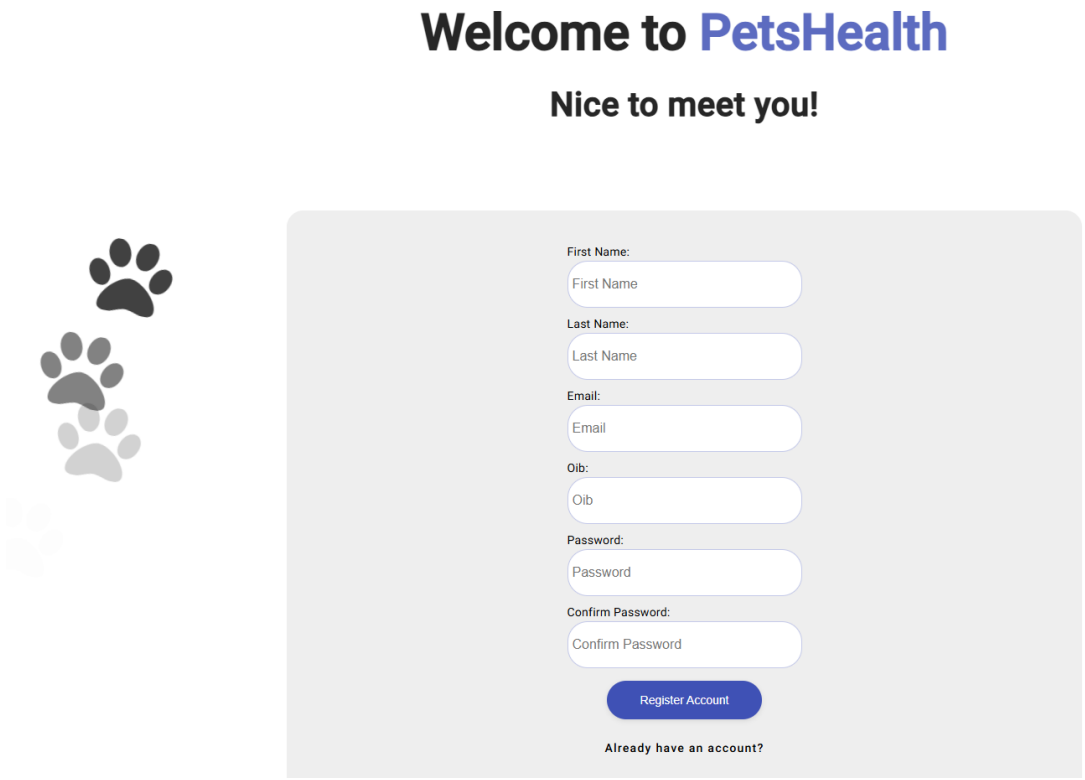
Na poslužiteljskoj strani definirana je krajnja točka na upravljaču koja prima podatke (slika 4.15). Podaci koje krajnja točka primi zatim se prosljeđuju usluzi koja ih obrađuje (slika 4.16). Prilikom obrade prvo se dohvaća korisnik iz baze podataka putem unesene adrese e-pošte, ukoliko korisnik postoji.

Ako korisnik ne postoji, izbacuje se iznimka, što rezultira greškom na klijentskoj strani, tj. zabranom prijave u sustav. U suprotnom, korisnik se dohvati, a na klijentsku stranu vraća se korisnikov identifikacijski broj koji se koristi u drugim krajnjim točkama i prosljeđuje se na izbornik prijavljenih ljubimaca.

Slika 4.15 Programski kod *UserController*-a za prijavu korisnika

Slika 4.16 Programski kod *UserService* logike za prijavu korisnika

Ukoliko korisnik nema račun na web aplikaciji, pritiskom na dugme za registraciju korisniku otvara se forma za registraciju novog računa (slika 4.17).



Slika 4.17 Forma za registraciju korisnika

Slično kao i za prijavu, pritiskom na dugme za registraciju, podaci se prikupljaju pomoću funkcije *collectFormData* (slika 4.18) i prosljeđuju usluzi kako bi se podaci prenijeli s klijentske strane na poslužiteljsku stranu putem REST protokola (slika 4.19).

```
export class RegisterComponent {
  constructor(private apiRequestService:ApirequestService){}
  registerFormGroup: FormGroup = new FormGroup({
    oib: new FormControl('', Validators.required),
    password: new FormControl('', Validators.required),
    name: new FormControl('', Validators.required),
    surname: new FormControl('', Validators.required),
    email: new FormControl('', Validators.required),
  });

  collectFormData() {
    this.apiRequestService.registerUser(this.registerFormGroup.value);
  }
}
```

Slika 4.18 Funkcija *CollectFormData* za registraciju korisnika

```
registerUser(data: any) {
  const url = `${apiUrl.key}users/register`;
  this.http.post<void>(url, data).subscribe(
    () => {
      alert('User created sucessfully');
      this.router.navigate(['/login']);
    },
    (error) => {
      alert('Cant register');
    }
  );
}
```

Slika 4.19 Metoda *registerUser* na klijentskoj strani

Na poslužiteljskoj strani primljeni podaci obrađuju se u metodi usluge (slika 4.20) gdje se koristi obrazac ponašanja graditelj koji mapira primljene podatke na atribute modela definiranog za entitet User. Nakon mapiranja, novonastali korisnik se sprema u bazu podataka i vraća na klijentsku stranu. Ukoliko je korisnik uspješno napravljen, korisnik se prosljeđuje na stranicu za prijavu.

```
1 usage   Filip Kramar
public User registerUsers(UserRegistrationDto registrationDTO) {

        User user = User.builder()
            .password(registrationDTO.getPassword())
            .name(registrationDTO.getName())
            .surname(registrationDTO.getSurname())
            .email(registrationDTO.getEmail())
            .oib(registrationDTO.getOib())
            .build();

        userRepository.save(user);
        System.out.println(user);
        return user;

}
```

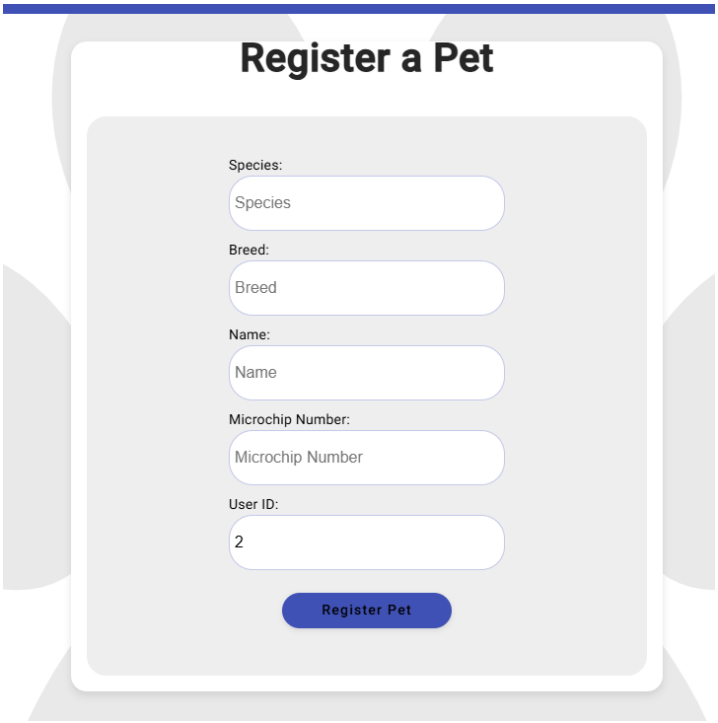
Slika 4.20 Programski kod *UserService* logike za registraciju korisnika

4.2.3. Programsko rješenje za upravljanje ljubimcima i stvaranje e-zdravstvenog kartona

Nakon uspješne prijave korisnika, prikazuju se svi njegovi ljubimci (slika 4.21). Ukoliko korisnik želi odabrati već kreiranog ljubimca, pritiskom na željenog ljubimca vodi se na stranicu s detaljima ljubimca. Ako korisnik ne vidi ljubimca, ima mogućnost registracije novog ljubimca pritiskom na dugme za registraciju ljubimca.

Nakon pritiska na dugme za registraciju ljubimca, korisniku se prikazuje forma (slika 4.22) u kojoj unosi detalje novog ljubimca, kao što su vrsta, ime i mikročip ljubimca. Zatim se prikupljeni podaci prosljeđuju preko REST protokola na poslužiteljsku stranu, na krajnju točku prikazanu na slici 4.23.

Slika 4.21 Komponenta s korisnikovim ljubimcima



Slika 4.22 Forma za registraciju ljubimca

Na poslužiteljskoj strani prvo se provjerava postoji li korisnik na kojeg se pokušava prijaviti novi ljubimac. Ako korisnik ne postoji, izbacuje se greška; u suprotnom, kreira se novi ljubimac koji se sprema u tablicu ljubimaca u bazi podataka i ažurira se lista ljubimaca pojedinog korisnika (slika 4.24).

Svaki novoprijavljeni kućni ljubimac automatski dobiva svoju elektroničku zdravstvenu iskaznicu u kojoj su pohranjeni osnovni podaci, poput pasmine, imena i mikročipa, te svi ostali zdravstveni

podaci uneseni prilikom registracije. Ti se podaci mogu ažurirati dodatnim zdravstvenim informacijama, kao što su povijest cijepljenja, liječnički pregledi i tretmani.

```
no usages  Filip Kramar
@PostMapping()
public ResponseEntity<Pet> createAPetChart(@RequestBody PetCreationDTO petCreationDTO){

    return ResponseEntity.ok(petService.createAPetChart(petCreationDTO));
}
```

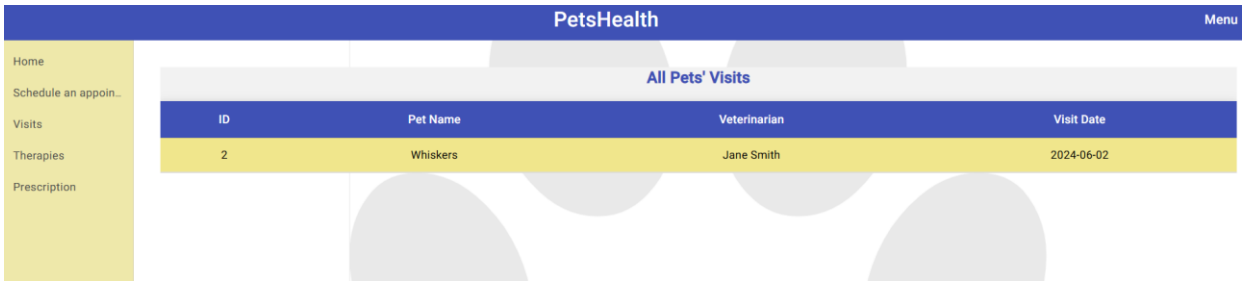
Slika 4.23 Programski kod krajnje točke za kreiranje novog kartona za životinju

```
1 usage  Filip Kramar
public Pet createAPetChart(PetCreationDTO petCreationDTO) {
    if (userRepository.findById(petCreationDTO.getUserId()).isPresent()) {
        User user= userRepository.findById(petCreationDTO.getUserId()).get();
        Pet pet = Pet.builder()
                .name(petCreationDTO.getName())
                .species(petCreationDTO.getSpecies())
                .breed(petCreationDTO.getBreed())
                .microchipnumber(petCreationDTO.getMicrochipNumber())
                .user(user)
                .build();

        petRepository.save(pet);
        user.getPets().add(pet);
        userRepository.save(user);
        return pet;
    }
    return null;
}
```

Slika 4.24 Metoda *createAPetChart* za kreiranje novog kartona za životinju

4.2.4. Programsko rješenje praćenja stanja i tijeka liječenja

Nakon što je odabran jedan od ljubimaca, korisnik može odabrati posjete veterinaru pritiskom na tipku *Visits* (slika 4.25). Dugme zatim pokreće rutu i korisnika vodi do stranice koja navodi sve prethodne posjete odabranom ljubimcu.



Slika 4.25 Grafički prikaz ispisa posjeta ljubimca

Podaci o posjetima dohvaćaju se iz baze podataka korištenjem HTTP GET krajnje točke prosljeđivanjem identifikacijskog broja ljubimca (slika 4.26) i prosljeđuju se sa strane poslužitelja na stranu klijenta.

```
no usages  Filip Kramar
@GetMapping("/{id}")
public ResponseEntity<List<Visit>> listPetsVisits(@PathVariable Long id){
    return ResponseEntity.ok(visitService.listPetsVisits(id));
}
```

Slika 4.26 Krajnja točka za ispisivanje posjeta veterinaru

Prilikom dohvaćanja podataka prvo se mora provjeriti postojanje ljubimca unutar baze podataka. Ukoliko ljubimac postoji, dohvaćaju se svi njegovi posjeti; suprotno, izbacuje se greška da traženi ljubimac ne postoji (slika 4.27).

Nakon dohvaćanja podataka s poslužiteljske strane, podaci se mapiraju prema HTML-u prikazanom na slici 4.28. Korisnik može odabrati bilo koji od navedenih posjeta kako bi se aktivirala metoda *navigateToVisitDetails* i ispisali detalji odabranog posjeta u obliku računa.

```
3 usages  Filip Kramar
public List<Visit> listPetsVisits(Long id) {

    Optional<Pet> pet= petRepository.findById(id);

    return visitRepository.findByPet(pet.get());

}
```

Slika 4.27 Metoda za dohvaćanje podataka posjete ljubimca na poslužiteljskoj strani

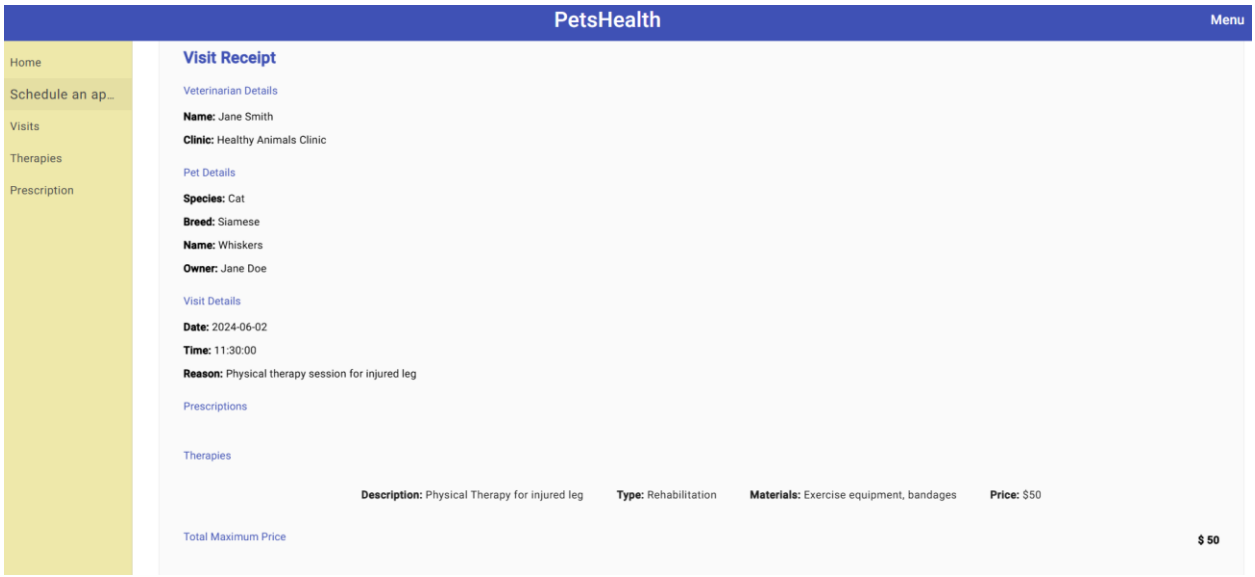
```
Go to component
<div class="grid-item item">
  <h2 class="heading">All Pets' Visits</h2>
    <table mat-table [dataSource]="visits" class="mat-elevation-z8">
      <ng-container matColumnDef="id">
        <th mat-header-cell *matHeaderCellDef>ID</th>
        <td mat-cell *matCellDef="let visit">{{ visit.id }}</td>
      </ng-container>

      <ng-container matColumnDef="petname">
        <th mat-header-cell *matHeaderCellDef>Pet Name</th>
        <td mat-cell *matCellDef="let visit">{{ visit.pet.name }}</td>
      </ng-container>

      <ng-container matColumnDef="vetname">
        <th mat-header-cell *matHeaderCellDef>Veterinarian</th>
        <td mat-cell *matCellDef="let visit">{{ visit.veterinarian.name }} {{ visit.veterina
      </ng-container>

      <ng-container matColumnDef="visitdate">
        <th mat-header-cell *matHeaderCellDef>Visit Date</th>
        <td mat-cell *matCellDef="let visit">{{ visit.visitDate }}</td>
      </ng-container>

      <tr mat-header-row *matHeaderRowDef="['id', 'petname', 'vetname', 'visitdate']"></tr>
      <tr mat-row *matRowDef="let visit; columns: ['id', 'petname', 'vetname', 'visitdate']"
        (click)="navigateToVisitDetails(visit.id)">
      </tr>
    </table>
</div>
```

Slika 4.28 HTML datoteka za prikazivanje liste posjeta

Na slici 4.30 prikazan je izgled stranice nakon odabira jednog od posjeta veterinaru, gdje se tijekom učitavanja stranice šalje zahtjev HTTP GET poslužiteljskoj strani. Zahtjev sadrži identifikacijski broj odabranog ljubimca i posjeta veterinaru (slika 4.29). Budući da entitet *Visits* sadrži strane ključeve iz drugih tablica, dohvaćanje podataka iz baze zahtijeva dodatnu logiku za pristup entitetima iz međutablica koje omogućuju više-prema-više odnose (slika 4.31) te dohvaćanje terapija i lijekova preko tih međutablica. Tijekom dohvaćanja pojedinih terapija iz posjeta izračunava se ukupni trošak.

```
no usages  Filip Kramar *
@GetMapping("/{id}/{visitId}")
public ResponseEntity<VisitDetailsDto> listPetVisitDetails(@PathVariable Long id, @PathVariable Long visitId){
    return ResponseEntity.ok(visitService.getPetsVisit(id,visitId));
}
```

Slika 4.29 Krajnja točka za dohvaćanje detalja posjete



Slika 4.30 Grafički prikaz detalja posjete

```
2 usages  Filip Kramar
public VisitDetailsDto getPetsVisit(long id, Long visitId) {
    List<Visit> visits = listPetsVisits(id);

    Visit visit = visits.stream() Stream<Visit>
            .filter(fetchedVisitId -> fetchedVisitId.getId().equals(visitId))
            .findFirst() Optional<Visit>
            .orElse( other: null);

    if (visit == null) {
        throw new NoSuchElementException("Visit not found");
    }

    List<VisitTherapy> visitTherapyList = visitTherapyRepository.findByVisit(visit);
    List<Therapies> therapyList = visitTherapyList.stream() Stream<VisitTherapy>
            .map(VisitTherapy::getTherapies) Stream<Therapies>
            .collect(Collectors.toList());

    BigDecimal totalTherapyPrice = therapyList.stream() Stream<Therapies>
            .map(Therapies::getPrice) Stream<BigDecimal>
            .reduce(BigDecimal.ZERO, BigDecimal::add);

    List<VisitPrescription> visitPrescriptionsList = visitPrescriptionsRepository.findByVisit(visit);
    List<Prescriptions> prescriptionList = visitPrescriptionsList.stream() Stream<VisitPrescription>
            .map(VisitPrescription::getPrescriptions) Stream<Prescriptions>
            .collect(Collectors.toList());

    visit.setMaxPrice(totalTherapyPrice);
    VisitDetailsDto visitDetailsDto = new VisitDetailsDto();
    visitDetailsDto.setVisit(visit);
    visitDetailsDto.setTherapies(therapyList);
    visitDetailsDto.setPrescriptions(prescriptionList);

    return visitDetailsDto;
}
```

Slika 4.31 Metoda *getPetsVisit* za dohvaćanje detalja posjete

4.2.5. Programsko rješenje prikaza primijenjenih terapija i prepisanih lijekova

Nakon što korisnik odabere jednog od ljubimaca, omogućuje mu se odabir primijenjenih terapija pritiskom na dugme *Therapies* (slika 4.32). Zatim se prikazuje tablica sa svim primijenjenim terapijama životinje, uključujući cijenu svake terapije i kratak opis. Podaci se dohvaćaju prilikom učitavanja stranice u tablicu putem metode HTTP GET, prosljeđivanjem identifikacijskog broja ljubimca krajnjoj točki na poslužiteljskoj strani (slika 4.33).

Slika 4.32 Grafičko sučelje liste terapija životinje

```
@GetMapping("/{id}")
public ResponseEntity<List<Therapies>> listPetsTherapies(@PathVariable Long id){
    return ResponseEntity.ok(therapiesService.listPetsTherapies(id));
}
```

Slika 4.33 Krajnja točka za dohvaćanje terapija životinje

Na poslužiteljskoj strani najprije se provjerava postoji li životinja unutar baze podataka pomoću metode *findById(id)* iz *petRepository*-ja. Ukoliko je životinja uspješno pronađena i dohvaćena iz baze podataka, dohvaćaju se sve posjete veterinaru te životinje korištenjem metode *findByPet(pet)* iz visitRepository-ja. Zatim se za svaku od posjeta se dohvaćaju sve primijenjene terapije koje su

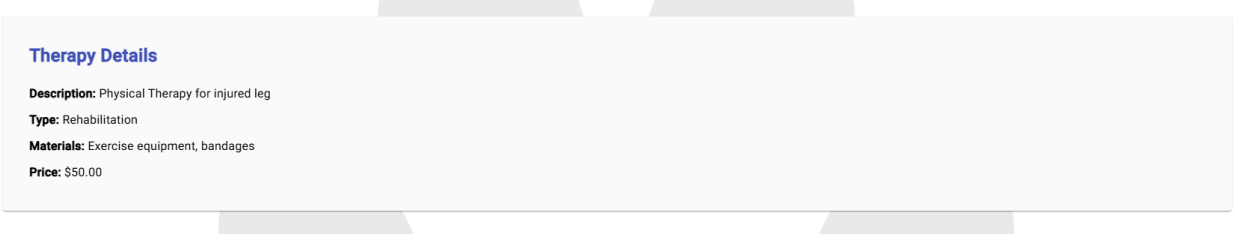
```
public List<Therapies> listPetsTherapies(Long id) {
    if (petRepository.findById(id).isPresent()) {

        List<Therapies> therapies = new ArrayList<>();
        if (!visitRepository.findByPet(petRepository.findById(id).get()).isEmpty()) {

            List<Visit> visits = visitRepository.findByPet(petRepository.findById(id).get());

            for (Visit visit : visits) {
                List<VisitTherapy> visitTherapies = visitTherapyRepository.findByVisit(visit);
                for (VisitTherapy visitTherapy : visitTherapies) {
                    therapies.add(visitTherapy.getTherapies());
                }
            }
            return therapies;
        } else {
            return Collections.emptyList();
        }
```

Slika 4.34 Programski kod dohvaćanja liste terapija životinja

propisane u posjeti. Na kraju, korisniku se vraća lista svih terapija životinje, a ako ljubimac ne postoji ili nema zabilježenih terapija, korisniku se vraća prazna lista. Primijenjeni postupak može se vidjeti i u programskom obliku na slici 4.34.

Korisnik može odabrati bilo koju primijenjenu terapiju pritiskom na stavku unutar tablice na web stranici. Nakon odabira pojedine terapije korisnika se vodi na prikaz detalja pojedine terapije (slika 4.35). Detalji se sastoje od opisa terapije, tipa terapije, cijene i materijala korištenih unutar terapije.

Tijekom učitavanja stranice s klijentske strane prosljeđuju se identifikacijski brojevi životinje i odabrane terapije krajnjoj točki na poslužiteljskoj strani (slika 4.36) kako bi se dohvatili podaci iz baze podataka. Na poslužiteljskoj strani dohvaća se lista terapija životinja korištenjem metode *listPetsTherapies* opisane na slici 4.34, zatim se pronalazi pojedinačna terapija koja je odabrana. Korisniku se vraća objekt terapije sa svim podacima ukoliko je terapija unutar liste terapija.



Slika 4.35 Grafičko sučelje detalja terapije

```
no usages  Filip Kramar *
@GetMapping("/{id}/{therapiesid}")
public ResponseEntity<Therapies> getPetsTherapy(@PathVariable Long id, @PathVariable Long therapiesid ){
    return ResponseEntity.ok(therapiesService.getPetsTherapy(id,therapiesid));
}
```

Slika 4.36 Krajnja točka za dohvaćanje detalja terapije

Ukoliko korisnik želi vidjeti prepisane lijekove životinje, pritiskom na gumb *Prescriptions* korisnika se vodi na tablicu koja sadrži listu lijekova (slika 4.38). Unutar tablice je za pojedini lijek prikazan njegov identifikacijski broj, ime i datum isteka. Postupak dohvaćanja podataka koji se potom mapiraju unutar tablice sličan je već opisanome za terapije, tj. krajnjoj točki (slika 4.37) se prosljeđuje identifikacijski broj životinje.

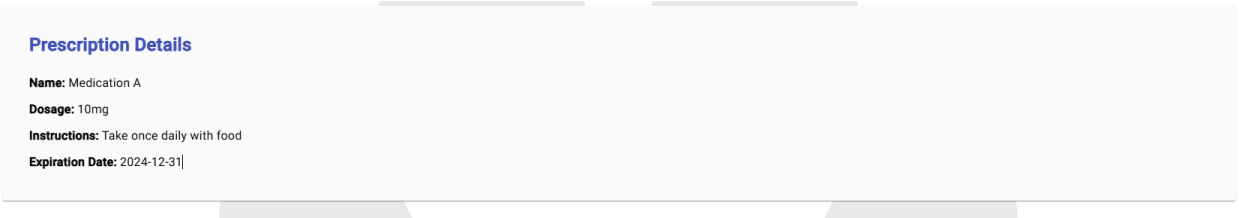
```
no usages  Filip Kramar
@GetMapping("/{id}")
public ResponseEntity<List<Prescriptions>> listPetsPrescriptions(@PathVariable Long id){
    return ResponseEntity.ok(prescriptionService.listPetsPrescriptions(id));
}
```

Slika 4.37 Krajnja točka za dohvaćanje liste prepisanih lijekova

Kao i u slučaju terapija, životinja se dohvaća iz baze podataka korištenjem metode *findById(id)* iz *petRepository-ja*. Ako se uspješno dohvati životinja, dohvaćaju se svi prepisani lijekovi po posjetama korištenjem *findByPet(pet)* iz *visitRepository-ja* (slika 4.40). Korisniku se vraća lista lijekova ukoliko nije došlo do greške prilikom obrade podataka.

Slika 4.38 Grafički prikaz liste prepisanih lijekova životinje

Korisnik može odabrati bilo koji od lijekova za životinje kako bi mu se prikazale dodatne informacije o odabranom lijeku, kao što su upute za korištenje i doza pojedinog lijeka (slika 4.39).



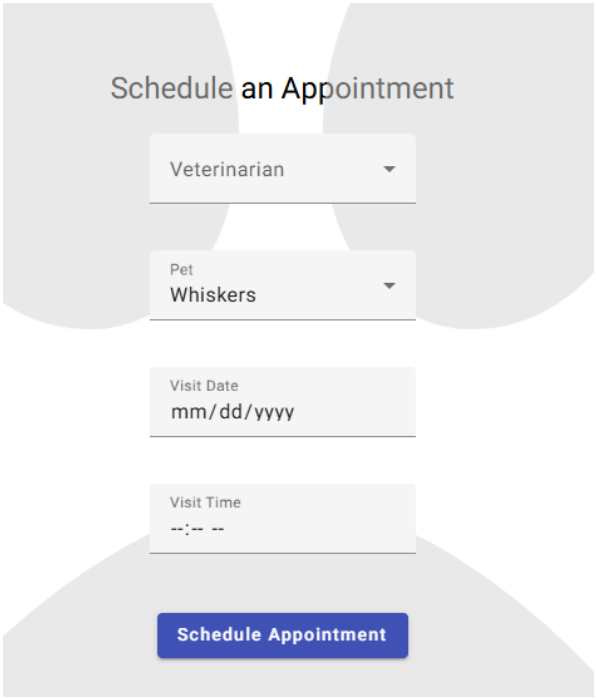
Slika 4.39 Grafički prikaz detalja lijekova životinje

```
1 usage   FilipKramar *
public List<Prescriptions> listPetsPrescriptions(Long id) {
    if (petRepository.findById(id).isPresent()) {

        List<Prescriptions> prescriptions = new ArrayList<>();
        if (!visitRepository.findByPet(petRepository.findById(id).get()).isEmpty()) {

            List<Visit> visits = visitRepository.findByPet(petRepository.findById(id).get());

            for (Visit visit : visits) {
                List<VisitPrescription> visitPrescriptions = visitPrescriptionsRepository.findByVisit(visit);
                for (VisitPrescription visitPrescription : visitPrescriptions) {
                    prescriptions.add(visitPrescription.getPrescriptions());
                }
            }
            return prescriptions;
        } else {
            return Collections.emptyList();
        }
    }
    return Collections.emptyList();
}
}
```

Slika 4.40 Programski kod PrescriptionService-a logike za izlistavanje odabranog ljubimca

4.2.6. Programsko rješenje zakazivanja termina

Korisnik može zakazati termin kod veterinara odabirom gumba *Schedule an appointment* u navigacijskoj traci. Nakon pritiska gumba, korisnik se vodi na stranicu gdje može popuniti formular za zakazivanje termina, koji se sastoji od odabira veterinara, datuma posjete i vremena posjete. Na slici 4.41 nalazi se grafičko sučelje komponente te pojedinih formi koje korisnik mora popuniti.



Slika 4.41 Grafički prikaz zakazivanja termina

Nakon pritiska na gumb *Schedule Appointment*, podaci se prikupljaju i prosljeđuju s klijentske strane na poslužiteljsku stranu u tijelu zahtjeva HTTP POST na krajnju točku (slika 4.42). Budući da klasa *Visit* sadrži dodatne podatke koji nisu potrebni za zakazivanje termina, koristi se DTO (eng. *Data Transfer Object*) *VisitScheduleDto* (slika 4.43).

```
no usages  Filip Kramar *
@PostMapping()
public ResponseEntity<Visit> scheduleVisit(@RequestBody VisitScheduleDto visitScheduleDto){
    return ResponseEntity.ok(visitService.scheduleAVisit(visitScheduleDto));
}
```

Slika 4.42 Krajnja točka za zakazivanje termina

```
4 usages  Filip Kramar *
@Getter
@Setter
@AllArgsConstructor
@NoArgsConstructor
public class VisitScheduleDto {

    Long vetId;

    Long petId;

    LocalDate visitDate;

    LocalTime visitTime;

    }
```

Slika 4.43 *VisitScheduleDto* klasa

Na poslužiteljskoj strani unutar metode *scheduleAVisit()* kreira se i zakazuje nova posjeta veterinaru (slika 4.44). Metodi se predaje DTO (eng. *Data Transfer Object*) objekt iz kojeg se dohvaćaju odabrani ljubimac i veterinar iz baze podataka, zatim se postavlja predani datum i vrijeme posjete. Za kreiranje objekta *Visit* koristi se programski obrazac graditelj, a kreirani objekt sprema unutar baze podataka i vraća korisniku.

```
1 usage  Filip Kramar *
public Visit scheduleAVisit(VisitScheduleDto visitScheduleDto) {

    Visit visit = Visit.builder()
            .veterinarian(veterinarianRepository.findById(visitScheduleDto.getVetId()).get())
            .pet(petRepository.findById(visitScheduleDto.getPetId()).get())
            .visitDate(visitScheduleDto.getVisitDate())
            .visitTime(visitScheduleDto.getVisitTime())
            .build();
    visitRepository.save(visit);
    return visit;
}
```

Slika 4.44 scheduleAVisit metoda za zakazivanje termina

4.2.7. Programsko rješenje prijave za veterinara

Ako je korisnik web stranice veterinar, a ne vlasnik životinje, prilikom zaslona za prijavu treba pritisnuti na *You are a veteraniran* hiperlink. Veterinar je potom preusmjeren na zaslon za prijavu (slika 4.45) gdje unosi svoju adresu e-pošte i lozinku u predviđena polja. Nakon unošenja podataka, pritiskom na gumb *Log in* pokreće se metoda *collectFormData*() koja prikuplja i provjerava točnost podataka te ih skuplja i prosljeđuje na poslužiteljsku stranu.

Na poslužiteljskoj strani, slično kao i za vlasnika, unutar metode *authorizeAVet()* (slika 4.46) pokušava se dohvatiti veterinar iz baze podataka. Ukoliko veterinar postoji, na klijentsku stranu se prosljeđuje njegov identifikacijski broj i veterinar je uspješno prijavljen u sustav. U suprotnom, ispisuje se greška da je prijava neuspjela.

```
<div class="content" fxLayout="column" fxLayoutAlign="space-around center">
  <h1>We are <span>PetsHealth</span></h1>
  <h2>Welcome back!</h2>
  <form [formGroup]="loginFormGroup" fxLayout="column" fxLayoutAlign="space-around center">
    <h2>Veterinarian log in</h2>
    <div class="floating-label" fxLayoutAlign="space-between start" fxLayout="column">
      <div fxLayoutAlign="space-around center">
        <div fxLayoutAlign="center center">Email:</div>
      </div>
      <input placeholder="Email" type="text" [formControlName]="'email'"/>
    </div>
    <div class="floating-label" fxLayoutAlign="space-between start" fxLayout="column">
      <div fxLayoutAlign="space-around start">
        <div fxLayoutAlign="center center">Password:</div>
      </div>
      <input placeholder="Password" type="password" [formControlName]="'password'"/>
    </div>
    <button fxLayout="column" fxLayoutAlign="center center" (click)="collectFormData()">Log in</button>
    <div class="login">
      <a mat-button routerLink="/"> <u> Not a veterinarian?</u></a>
    </div>
  </form>
</div>
```

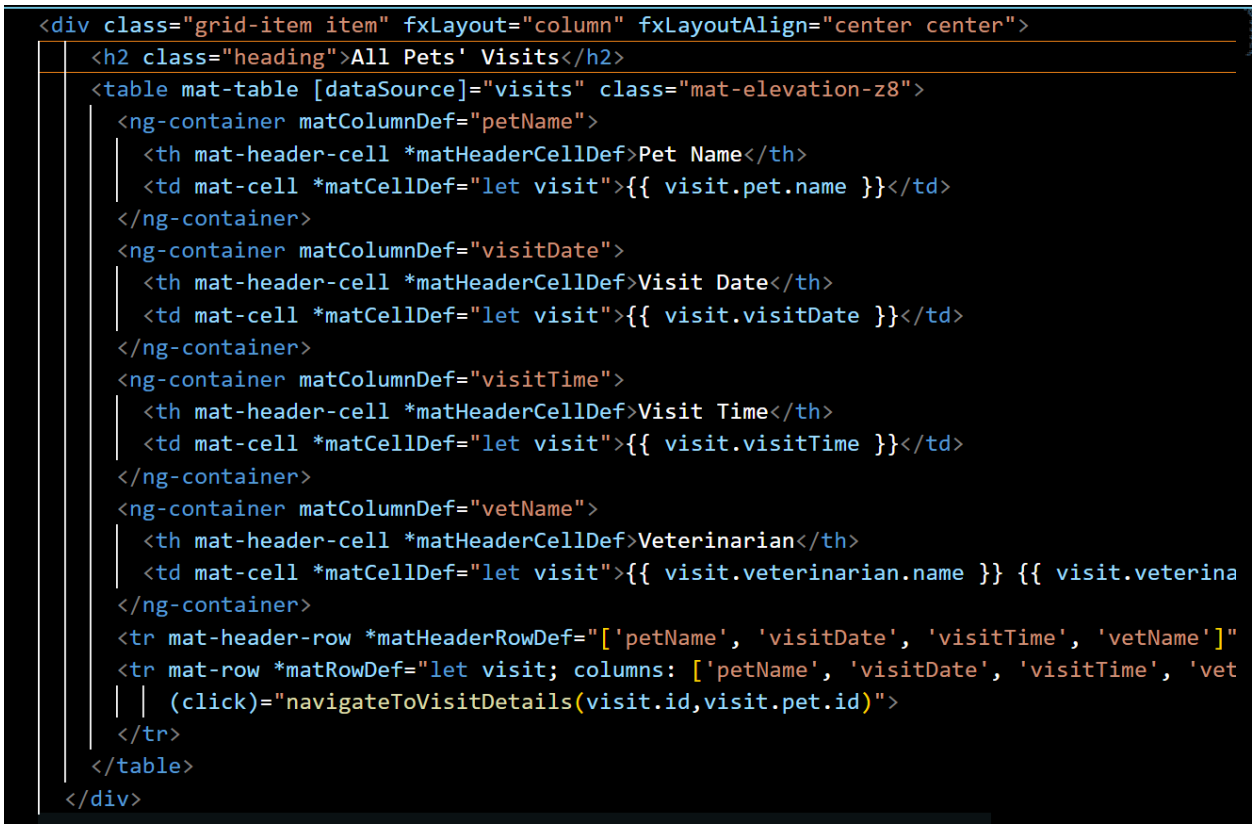
Slika 4.45 HTML struktura prijave veterinara

```
1 usage   Filip Kramar *
public Long authorizeAVet(VeterinarianLoginDto veterinarianLoginDto) {

    Optional<Veterinarian> fetchedVet = veterinarianRepository.findByEmail(veterinarianLoginDto.getEmail());

    if (fetchedVet.isEmpty()) {
        throw new RuntimeException("Vet with the username: " + veterinarianLoginDto.getEmail() + "does not exist"
    }
    return fetchedVet.get().getId();
}
```

Slika 4.46 *authorizeAVet* metoda za prijavu

4.2.8. Programsko rješenje dodavanja detalja posjete

Nakon uspješne prijave, veterinar pritiskom na tipku *Appointments* unutar navigacijske trake može pristupiti svim posjetima. Pritiskom na gumb, veterinaru se ispisuju svi posjeti u obliku tablice s podacima o imenu ljubimca, datumu posjete, vremenu posjete, te njegovim imenom i prezimenom. Struktura HTML-a koja se koristi za ostvarivanje izlistavanja posjeta nalazi se na slici 4.47.



Slika 4.47 HTML struktura *vetappointments* komponente

Veterinar može odabrati bilo koju posjetu s popisa, nakon čega mu se prikazuje isto sučelje kao i korisniku, kako je opisano u poglavlju 4.2.4. Unutar komponente *visitdetails* nalazi se gumb koji je vidljiv samo ako korisnik web aplikacije nije vlasnik (slika 4.48).

Slika 4.48 Dugme za dodavanje detalja posjete

Pritiskom na gumb *Add Visit Details*, veterinar se preusmjerava na komponentu za dodavanje detalja posjete, *appointmentdetails*. Unutar komponente *appointmentdetails*, veterinar unosi primijenjene lijekove i terapije za posjetu. Podaci se zatim šalju na poslužitelj putem metode HTTP PUT pritiskom na gumb *Add Appointment Details*. Metodi *addVisitDetails* (slika 4.48) unutar usluge na poslužiteljskoj strani prosljeđuju se identifikacijski broj posjete, podaci o veterinaru i objekt *VisitDetailsAdditionDTO* (slika 4.49).

Slika 4.49 VisitDetailsAdditionDTO klasa

Metoda dohvaća sve posjete za predanog ljubimca i pronalazi posjetu s predanim identifikacijskim brojem. Pronađenoj posjeti dodaju se predani lijekovi i terapije iz DTO (eng. *Data Transfer Object*) objekta. Budući da je odnos između posjeta i lijekova, kao i između posjeta i terapija, više-na-više, za dodavanje lijekova i terapija potrebno je kreirati objekte međutablica te ih spremiti u odgovarajuće repozitorije. Na kraju se ažurirana verzija posjete sprema u repozitorij i vraća kao odgovor na zahtjev klijentskoj strani putem metode prethodno objašnjene u poglavlju 4.2.5.

```
@Transactional
public VisitDetailsDto addVisitDetails(Long id, Long visitId , VisitDetailsAdditionDto visitDetailsAdditionDto) {

    List<Visit> visits= listPetsVisits(id);

    Visit visit= visits.stream() Stream<Visit>
            .filter(fetchedVisitId -> fetchedVisitId.getId().equals(visitId))
            .findFirst() Optional<Visit>
            .orElse( other: null);

    List<Prescriptions> fetchedPrescriptionList= prescriptionRepository.findAllById(visitDetailsAdditionDto.getPrescr

    fetchedPrescriptionList.forEach(prescription -> {
       VisitPrescription visitPrescription= new VisitPrescription( visitRepository.findById(visitId).get(), prescript
       visitPrescriptionsRepository.save(visitPrescription);
       visit.getVisitPrescriptions().add(visitPrescription);
    });

    List<Therapies> fetchedTherapiesList= therapiesRepository.findAllById(visitDetailsAdditionDto.getTherapyId());

    fetchedTherapiesList.forEach(therapy -> {
        VisitTherapy visitTherapy= new VisitTherapy( visitRepository.findById(visitId).get(), therapy);
        visitTherapyRepository.save(visitTherapy);
        visit.getVisitTherapies().add(visitTherapy);
    });

    visitRepository.save(visit);
    return getPetsVisit(id,visitId);
```

Slika 4.50 *addVisitDetails* metoda na poslužiteljskoj strani

4.3. Implementacija automatizirane kontejnerske arhitekture

Za implementaciju automatizirane kontejnerske arhitekture koristi se Docker u kombinaciji s alatom GitHub Actions. U implementaciji su potrebne sljedeće komponente: *Dockerfile* za poslužiteljsku i klijentsku stranu, datoteka *docker-compose.yaml* te *GitHub Actions workflow* datoteka.

Svaka od ovih komponenti ima svoju specifičnu ulogu u ostvarenju automatizirane kontejnerske arhitekture. U nastavku poglavlja detaljnije će se objasniti svrha svake komponente i njihov način korištenja unutar web aplikacije.

4.3.1. Implementacija datoteke za kreiranje Docker slike

Dockerfile je posebna vrsta datoteke koja se koristi unutar alata Docker za kreiranje slike. Web aplikacija ovog diplomskog rada sadrži dvije datoteke *Dockerfile*: jedan na poslužiteljskoj strani (slika 4.51) i jedan na klijentskoj strani (slika 4.52).

Svaka datoteka *Dockerfile* započinje generiranjem svih ovisnosti i biblioteka korištenjem alata za izgradnju, odnosno *Maven* na poslužiteljskoj strani i *NodeJS* na klijentskoj strani, te generiranjem izvršive datoteke za odgovarajući dio projekta.

```
1   FROM maven:latest AS build
2
3   WORKDIR /app
4
5   COPY pom.xml .
6   COPY src ./src
7
8   RUN mvn clean package -DskipTests
9
10  FROM openjdk:21-slim AS runtime
11
12  WORKDIR /app
13
14  COPY --from=build /app/target/*.jar app/app.jar
15
16  CMD ["java", "-jar", "app/app.jar"]
```

Slika 4.51 Dockerfile datoteka na poslužiteljskoj strani

```
FROM node:18.17.1 AS angular
WORKDIR /app
COPY package*.json ./
RUN npm install
COPY . .
RUN npm run build

FROM nginx:1.24.0-alpine AS product
COPY --from=angular /app/dist/petshealth /usr/share/nginx/html
EXPOSE 80
```

Slika 4.52 Dockerfile datoteka na klijentskoj strani

Nakon što je artefakt generiran, kopira se u sliku i definira kao početna točka te slike. Pojedina datoteka *Dockerfile* može se „pokrenuti“ naredbom *docker build*, što rezultira stvaranjem Docker slike, iz koje se potom mogu generirati kontejneri.

4.3.2. Implementacija datoteke za kreiranje Docker kontejnera

Upravljanje Docker kontejnerima unutar projekta provodi se pomoću datoteke *docker-compose.yaml*. Ova datoteka omogućuje definiranje više usluga na jednom mjestu, a u sklopu ovog projekta definirane su usluge za klijentsku stranu, poslužiteljsku stranu i bazu podataka. Na slici 5.54 prikazana je *docker-compose.yaml* datoteka kojom se izgrađuju kontejneri projekta.

Za kontejner baze podataka koristi se službena Docker slika PostgreSQL-a, dok se za kontejner klijentske i poslužiteljske strane koriste Docker slike generirane iz datoteka Dockerfile prikazanih na slici 4.54. Svakom kontejneru potrebno je definirati sliku koju koristi, mapiranje portova i ostale postavke. Na primjer, za bazu podataka definiraju se ime baze, korisničko ime i lozinka, koji moraju odgovarati varijablama okruženja kontejnera poslužiteljske strane kako bi se uspješno uspostavila veza s bazom.

Kontejneri se mogu izgraditi pomoću docker-compose.yaml datoteke unosom naredbe docker-compose up. Naredba preuzima potrebne slike ako one već ne postoje te generira kontejnere s definiranom konfiguracijom. Nakon izvršenja naredbe, kontejneri se vizualno prikazuju unutar aplikacije Docker Desktop (slika 4.53).

Slika 4.53 Docker mreža nastala nakon pokretanja docker-compose.yaml datoteke

```yaml
version: '3.1'
services:
  angular:
    image: petshealth-frontend:latest
    ports:
      - "4200:80"

  api:
    image: petshealth-backend:latest
    ports:
      - "8080:8080"
    environment:
      - SPRING_DATASOURCE_URL=jdbc:postgresql://db:5432/petshealth
      - SPRING_DATASOURCE_USERNAME=myuser
      - SPRING_DATASOURCE_PASSWORD=mypassword
    depends_on:
      - db

  db:
    image: postgres:latest
    ports:
      - "5432:5432"
    environment:
      - POSTGRES_PASSWORD=mypassword
      - POSTGRES_USER=myuser
      - POSTGRES_DB=petshealth
```

Slika 4.54 docker-compose.yaml datoteka

4.3.3. Implementacija kontinuirane integracije

Za implementaciju kontinuirane integracije i dostave koriste se GitHub Actions workflow-ovi. Na slikama 4.55, 4.56 i 5.58 prikazana je datoteka *build-images.yaml*, koja se koristi za implementaciju ovih procesa. Struktura datoteke slična je onoj u *docker-compose.yaml* datoteci, jer definira cjeline koje se izvršavaju slijedno

Unutar datoteke GitHub Actions workflow-a definirani su koraci koji se izvršavaju prilikom pokretanja specifičnih GitHub Actions. Budući da se nova verzija web aplikacije ne gradi često, postavljen je radni tijek koji omogućuje ručno pokretanje generiranja slika, umjesto automatskog pokretanja prilikom svake promjene na definiranim granama

```yaml
on:
  workflow_dispatch:
    inputs:
      build_frontend:
        description: 'Build frontend image'
        required: true
        default: 'true'
        type: boolean
      build_backend:
        description: 'Build backend image'
        required: true
        default: 'true'
        type: boolean
```

Slika 4.55 Definiranje *workflow-a* unutar .yaml datoteke

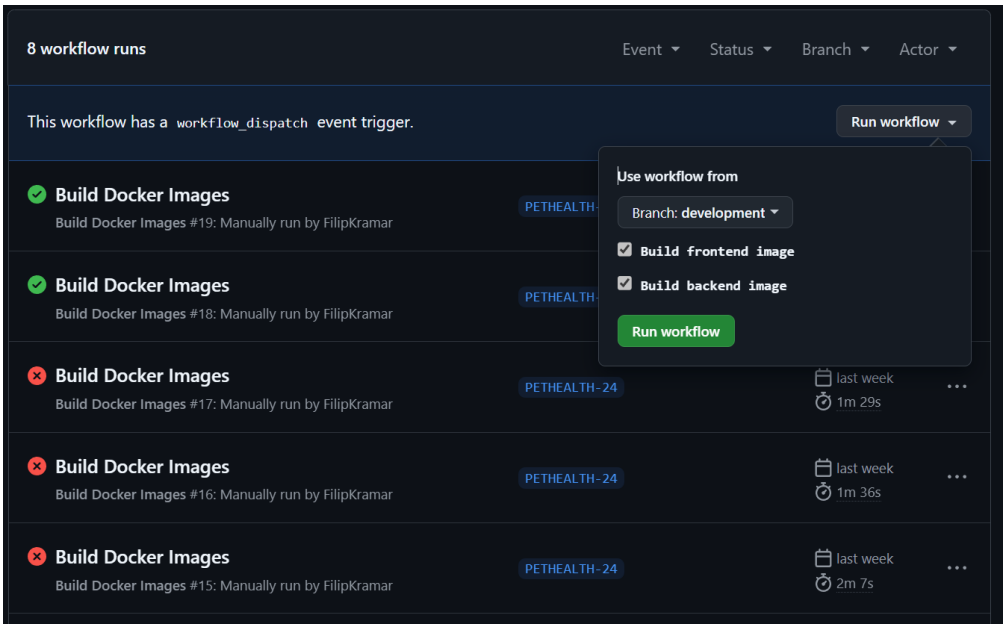
Na slici 4.57 prikazan je padajući izbornik koji omogućuje ručno pokretanje procesa kreiranja slike. Prilikom ručnog kreiranja slika odabire se grana s koje se želi generirati slika, kao i željeni dio aplikacije za koji se treba napraviti nova slika. Ovaj pristup omogućuje selektivno generiranje slika samo za dijelove aplikacije koji su izmijenjeni, što smanjuje nepotrebno kreiranje slika za nepromijenjene dijelove i skraćuje vrijeme generiranja. Na primjer, ako je otkrivena kritična greška na klijentskoj strani aplikacije, poslužiteljska strana ne mora biti ponovno generirana.

Kako bi se pokrenuo radni tijek, potrebno je kliknuti na gumb *Run Workflow,* koja automatski pokreće okruženje za generiranje slika na stranici GitHub, uključujući instalaciju potrebnih alata poput operacijskog sustava Ubuntu i alata Docker, čime se osiguravaju sve potrebne komponente za izgradnju aplikacije.

Nakon instalacije potrebnih alata, tijek nastavlja s definiranim slijedom koraka, provjeravajući koje slike treba generirati (slika 5.56). Ako je odabrana slika koja se provjerava, pokreće se naredba *docker build* nad datotekom *Dockerfile* i generira nova Docker slika.

```yaml
jobs:
  build:
    runs-on: ubuntu-latest
    steps:
    - name: Checkout code
      uses: actions/checkout@v4

    - name: Set up Docker Buildx
      uses: docker/setup-buildx-action@v3

    - name: Build frontend image
      if: ${{ github.event.inputs.build_frontend == 'true' }}
      run: |
        cd frontend/petshealth
        docker build -t petshealth-frontend:latest .

    - name: Build backend image
      if: ${{ github.event.inputs.build_backend == 'true' }}
      run: |
        cd backend/petshealth
        docker build -t petshealth-backend:latest .
```

Slika 4.56 Programsko rješenje logike datoteke build-images.yaml



Slika 4.57 Grafičko sučelje *workflow-a*

Na kraju procesa, slike se spremaju u datoteku koja se može preuzeti. Datoteku potom se preuzme i smješta u direktorij gdje se nalazi *unpackAndLoadImages.sh* shell skripta (slika 4.58). Skripta

raspakira datoteku, sprema ju u Docker i pokreće datoteku docker-compose.yaml, što rezultira kreiranjem kontejnera i pokretanjem naše aplikacije unutar okruženja Docker.

```yaml
- name: Save Docker images as tar files
  run: |
    if [ "${{ github.event.inputs.build_frontend }}" = "true" ]; then
      docker save petshealth-frontend:latest -o petshealth-frontend.tar
    fi
    if [ "${{ github.event.inputs.build_backend }}" = "true" ]; then
      docker save petshealth-backend:latest -o petshealth-backend.tar
    fi

- name: Create zip file of Docker images
  run: |
    zip -r petshealth-image.zip *.tar

- name: Upload zip file as artifact
  uses: actions/upload-artifact@v4
  with:
    name: petshealth-docker-image
    path: petshealth-image.zip
```

Slika 4.58 Programsko rješenje generiranja datoteke

```bash
#!/bin/bash

if ! command -v unzip &> /dev/null; then
  echo "unzip not found, installing..."
  sudo apt-get update && sudo apt-get install -y unzip
fi

ARTIFACT_ZIP="petshealth-docker-image.zip"
IMAGE_ZIP="petshealth-image.zip"
IMAGE_DIR="docker-images"

unzip -q "$ARTIFACT_ZIP"
unzip -q "$IMAGE_ZIP" -d "$IMAGE_DIR"
for tar_file in "$IMAGE_DIR"/*.tar; do
  docker load -i "$tar_file"
done
rm -rf "$IMAGE_DIR" "$IMAGE_ZIP"
docker-compose up -d
```

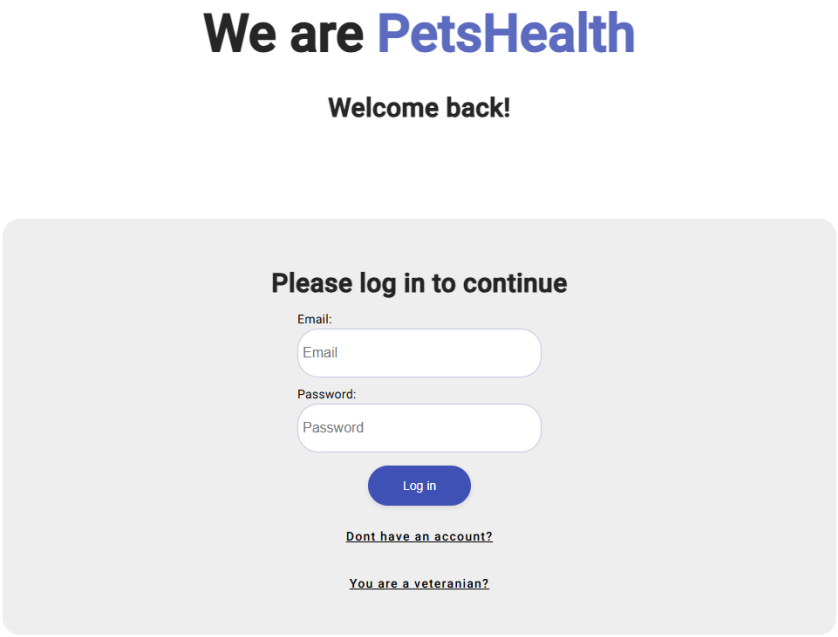
Slika 4.59 *unpackAndLoadImages*.sh shell skripta

5. NAČIN KORIŠTENJA WEB APLIKACIJE I ANALIZA EFIKASNOSTI KORIŠTENIH TEHNOLOGIJA

U sljedećem poglavlju bit će prikazan i opisan rad aplikacije te izgled korisničkog sučelja. Pratit će se „optimalan put“, odnosno kako bi aplikaciju koristila većina korisnika. Kroz aplikaciju će se proći s gledišta korisnika i veterinara.

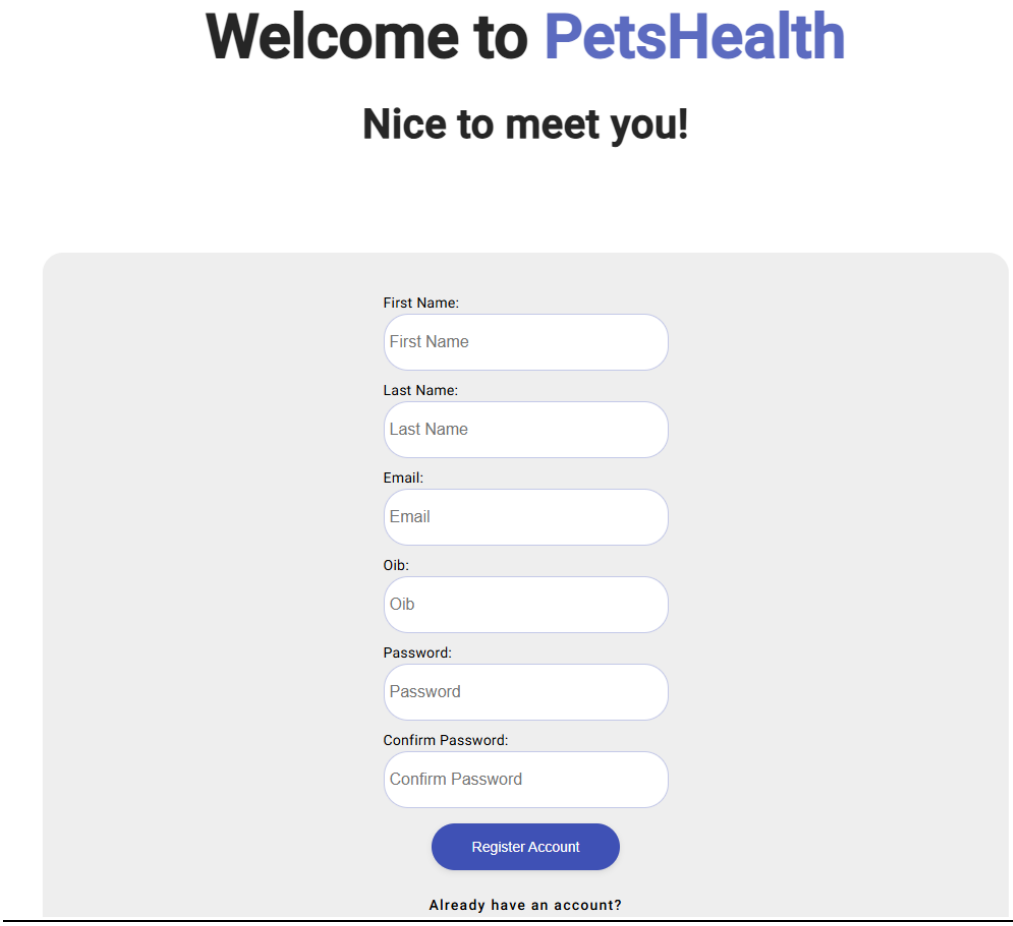
5.1. Način korištenja web aplikacije kao korisnik

Prilikom učitavanja stranice korisnika se vodi na zaslon za prijavu (slika 5.1).



Slika 5.1 Zaslon za prijavu korisnika

Korisnik se može prijaviti u aplikaciju unošenjem svoje adrese e-pošte i lozinke u predviđena polja te pritiskom na gumb *Log In*. U slučaju da korisnik nema račun u aplikaciji, može pritisnuti hiperlink *Don't have an account?*, koji će ga odvesti na zaslon za registraciju (slika 5.2).

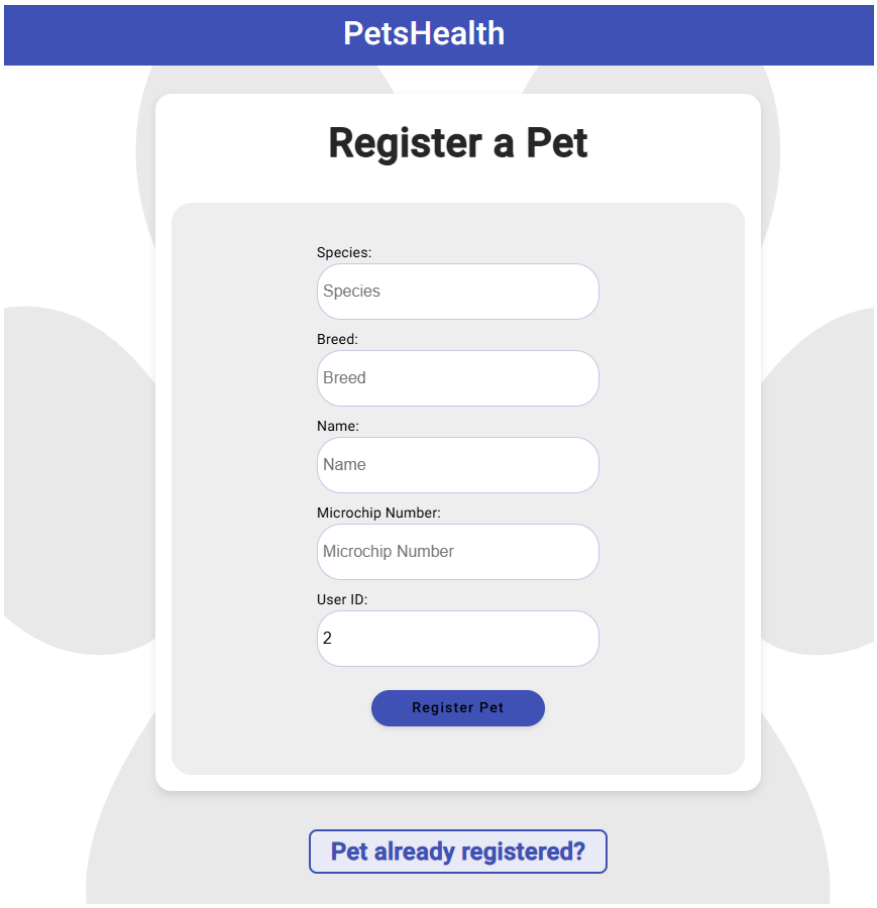


Slika 5.2 Zaslon za registraciju korisnika

Na zaslonu za registraciju korisnik može unijeti svoje podatke, poput imena, prezimena, adrese e-pošte, OIB-a i lozinke. Nakon unošenja podataka provodi se validacija polja, i ako su uneseni validni podaci, pritiskom na gumb *Register Account* korisnik se uspješno registrira i vraća na zaslon za prijavu (slika 5.1), gdje se može prijaviti u sustav.

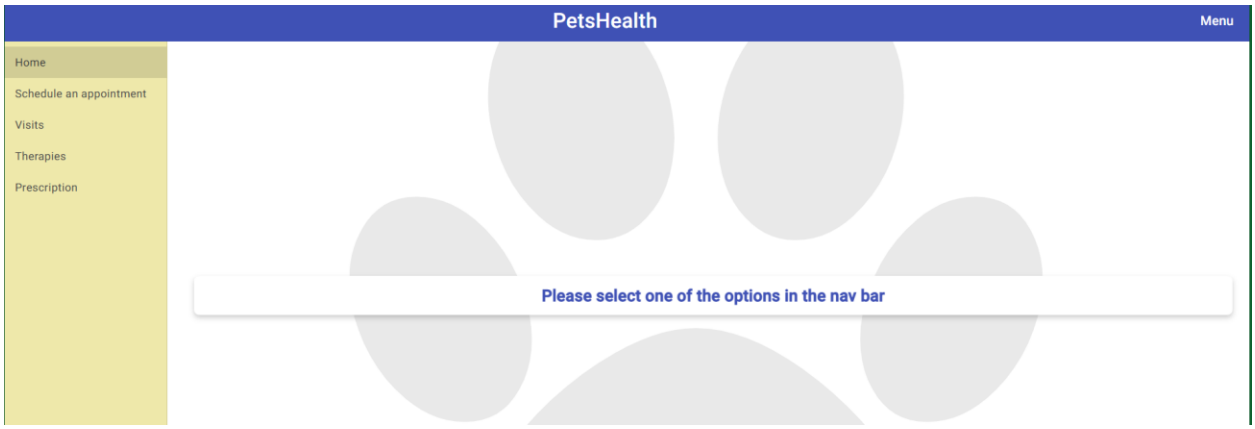
Nakon prijave unutar sustava, korisniku se prikazuje lista životinja koje su prijavljene pod njegovim imenom (slika 5.3). Korisnik može odabrati jednu od životinja s liste ili odabrati gumb *Register your pet* kako bi registrirao novu životinju, što ga vodi na zaslon za registraciju životinja (slika 5.4). Na tom zaslonu korisnik unosi podatke životinje koju želi registrirati, što uključuje vrstu životinje, pasminu, ime i mikročip. Polje za korisnički identifikacijski broj automatski se popunjava, tako da će novokreirana životinja biti pod prijavljenim korisnikom. Pritiskom na gumb *Register Pet*, korisniku se prikazuje poruka o uspješnosti i vraća se na listu životinja (slika 5.3).

Slika 5.3 Zaslon liste korisnikovih ljubimaca

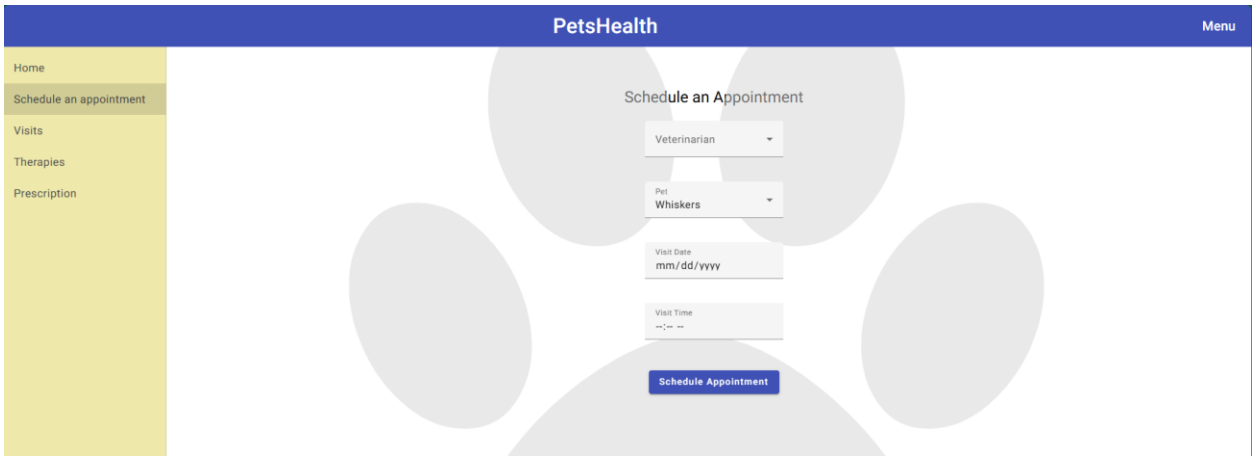


Slika 5.4 Zaslon registracije nove životinje

Kada korisnik odabere jednu od životinja, vodi ga se na nadzornu ploču (slika 5.5), gdje može odabrati jednu od opcija u navigacijskoj traci. Pritiskom na gumb *Schedule an appointment*, korisnik se preusmjerava na zaslon za prijavu posjete veterinaru (slika 5.6). Unutar zaslona za prijavu, korisnik odabire željenog veterinara, datum i vrijeme posjete. Ako su uneseni validni podaci, korisnik se vraća na zaslon naslovne ploče.

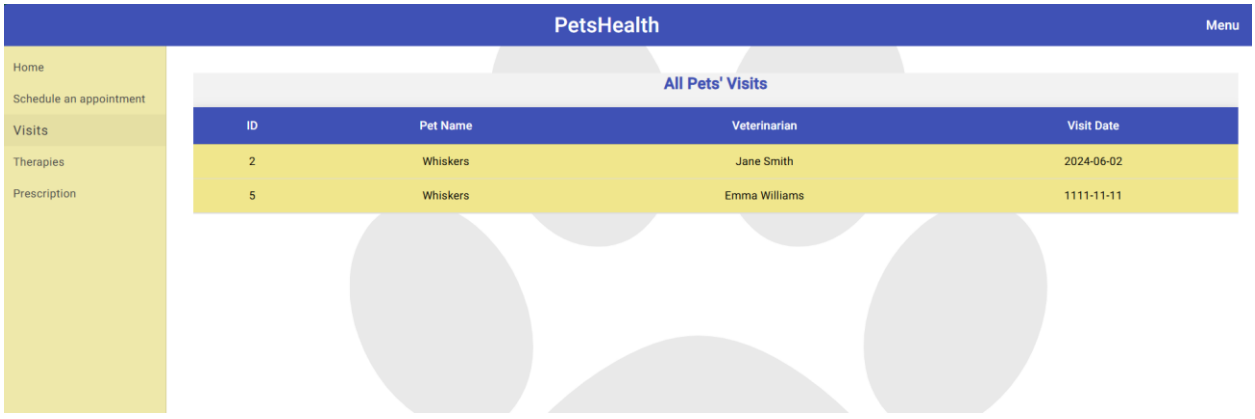


Slika 5.5 Prikaz zaslona nadzorne ploče

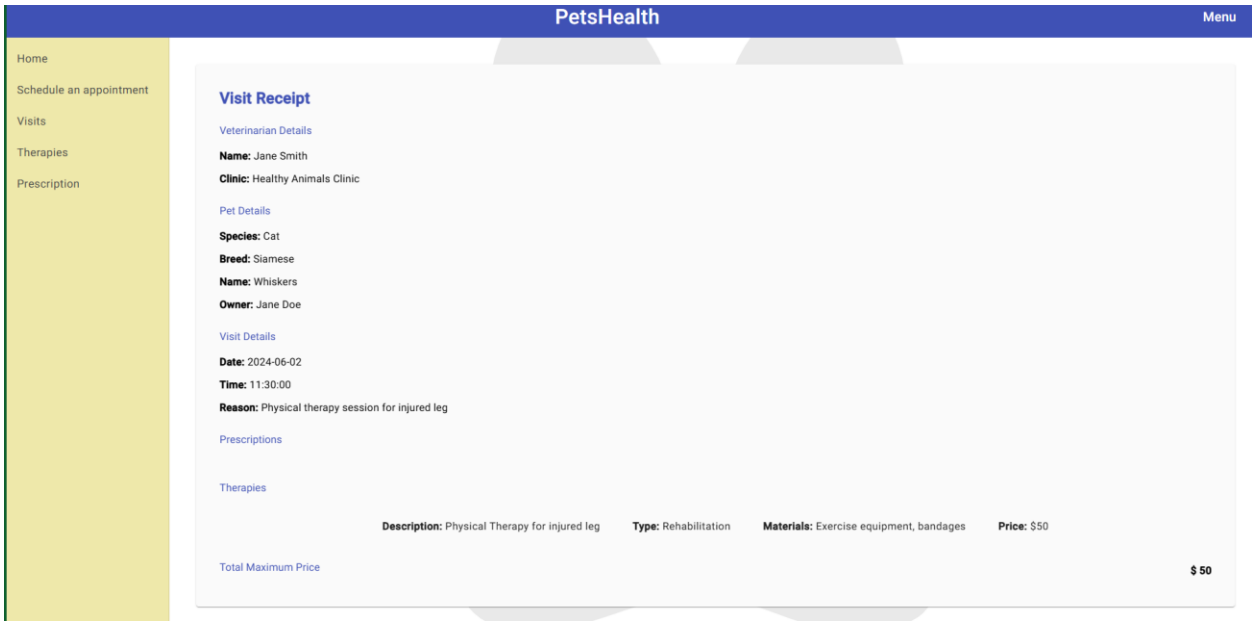


Slika 5.6 Prikaz zaslona prijave posjete veterinaru

Pritiskom na gumb *Visits* unutar navigacijske trake, korisnik se preusmjerava na listu svih posjeta za odabranu životinju (slika 5.7). Korisnik može odabrati bilo koju od posjeta, a zatim mu se prikazuju detalji posjete (slika 5.8), koji uključuju informacije o veterinaru kod kojeg je bila posjeta, detalje o životinji i posjeti, primljene terapije, prepisane lijekove te ukupnu cijenu tog posjeta.



Slika 5.7 Zaslon prikaza liste posjeta životinje

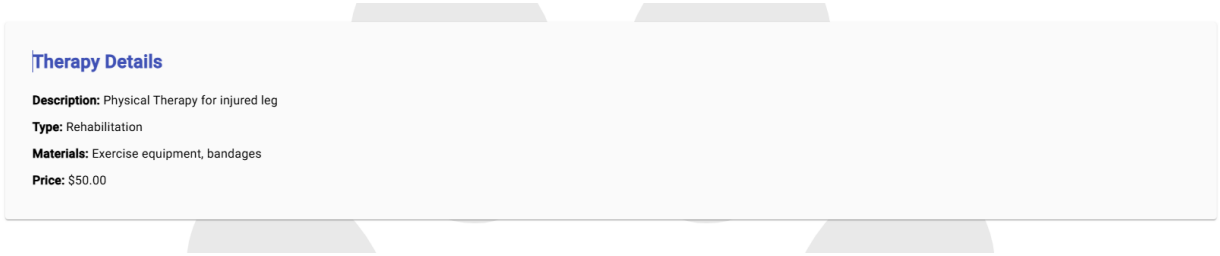


Slika 5.8 Prikaz detalja odabranog posjeta

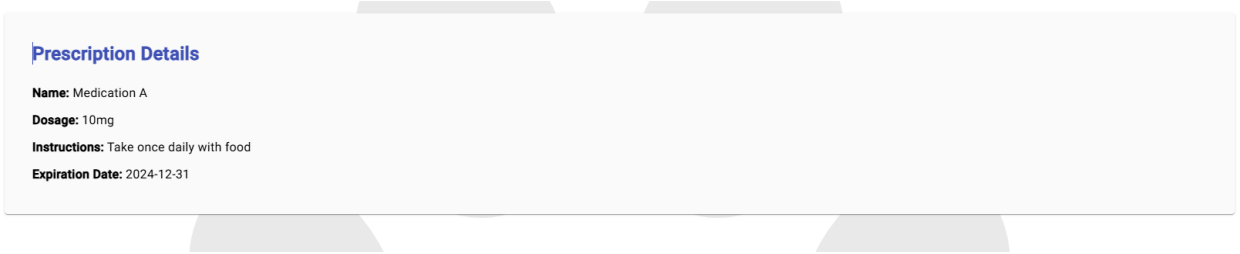
Odabirom tipke *Therapies* ili *Prescription* na navigacijskoj traci, korisnik se preusmjerava na listu svih prepisanih lijekova (slika 5.9) ili primijenjenih terapija (slika 5.10). Korisnik može odabrati bilo koju stavku s liste kako bi dobio detaljniji opis terapije (slika 5.11) ili lijeka (slika 5.12).

Slika 5.9 Zaslon liste prepisanih lijekova životinje

Slika 5.10 Zaslon liste primijenjenih terapija životinje



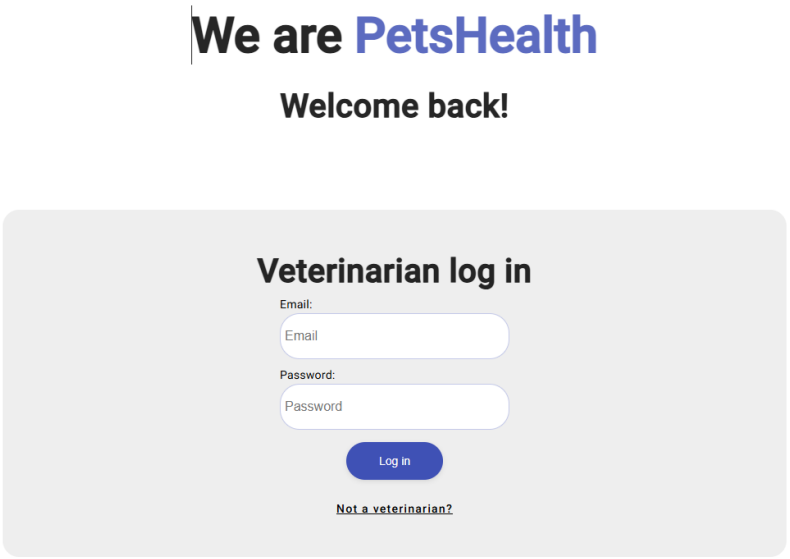
Slika 5.11 Prikaz detalja primijenjene terapije životinje



Slika 5.12 Prikaz zaslona detalja prepisanog lijeka

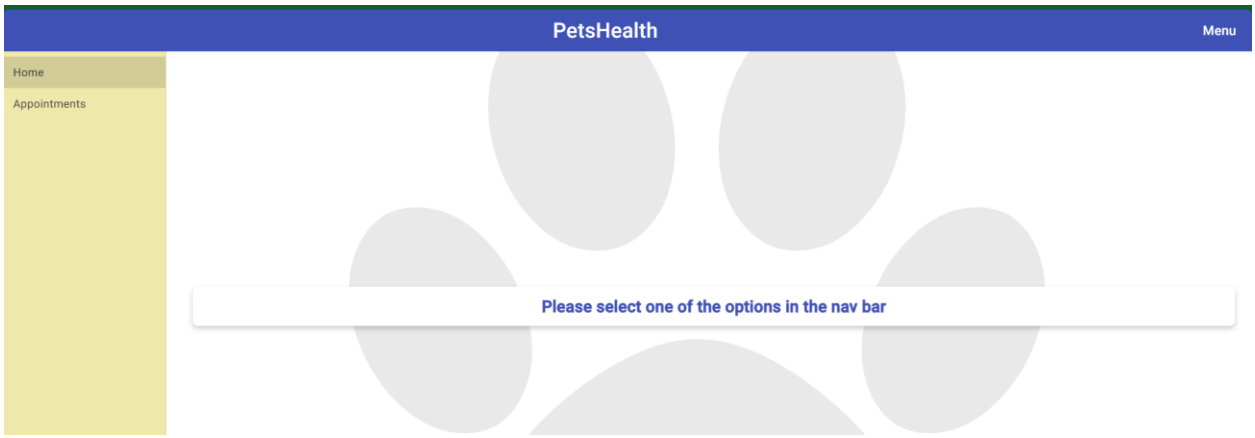
5.2. Način korištenja web aplikacije kao veterinar

Prilikom učitavanja stranice, veterinar se preusmjerava na zaslon za prijavu korisnika (slika 5.1). Na stranici za prijavu, veterinar treba pritisnuti hiperlink *You are a veterinarian?* kako bi došao do zaslona za prijavu veterinara (slika 5.13), gdje unosi svoju adresu e-pošte i lozinku kako bi pristupio sustavu.



Slika 5.13 Zaslon prijave veterinara

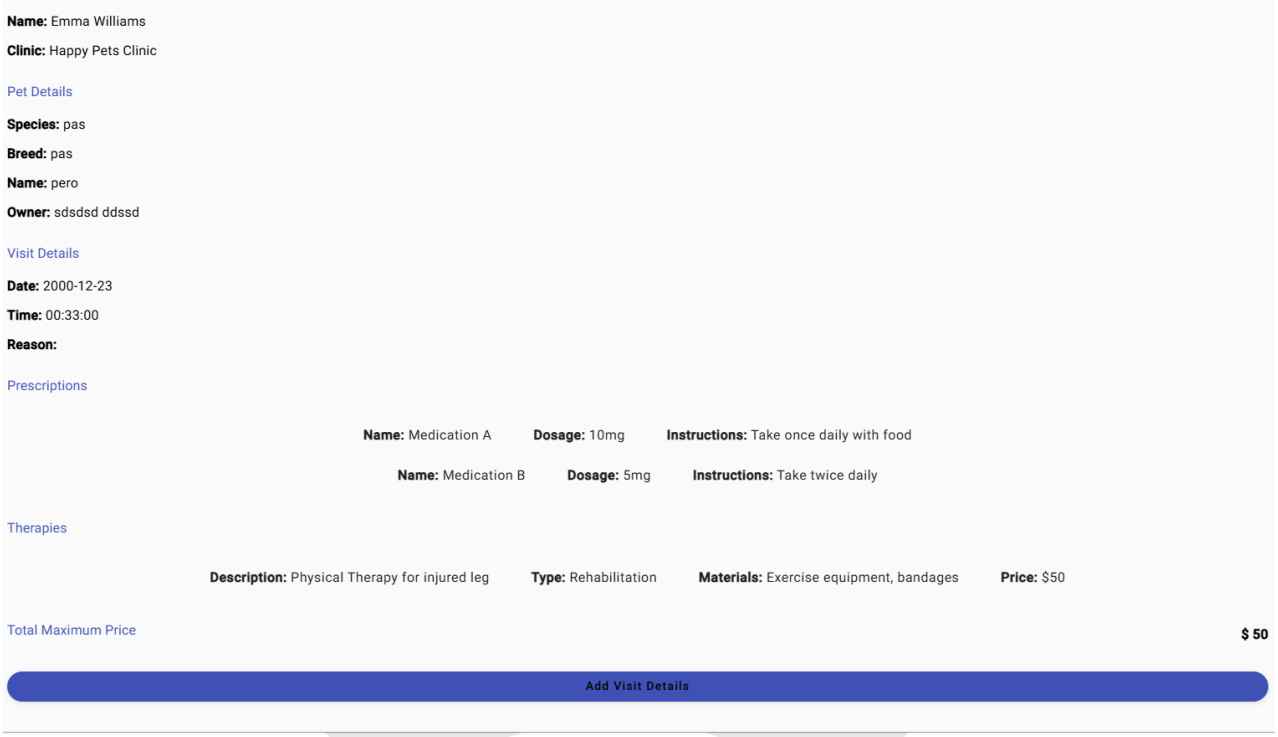
Za razliku od korisnika, veterinara se nakon prijave preusmjerava izravno na nadzornu ploču (slika 5.14). Pritiskom na gumb *Appointments* na navigacijskoj traci, veterinar se preusmjerava na listu svojih posjeta (slika 5.15).



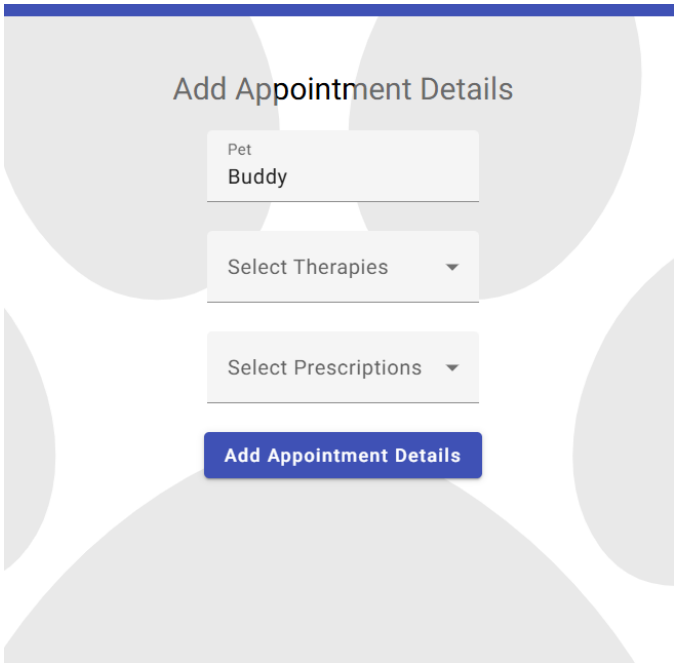
Slika 5.14 Zaslon nadzorne ploče veterinara

Slika 5.15 Zaslon liste veterinarovih posjeta

Veterinar može, kao i korisnik, odabrati pojedinu posjetu i dobiti izlistane detalje posjete (slika 5.16). Međutim, veterinar na zaslonu detalja posjete ima opciju dodati informacije o odabranoj posjeti pritiskom na tipku *Add Visit Details* (slika 5.16). Na zaslonu za dodavanje detalja posjeti (slika 5.17), veterinar u padajućem izborniku odabire primijenjene terapije i prepisane lijekove, te ih sprema pritiskom na gumb *Add Appointment Details*. Detalji posjete automatski se ažuriraju nakon dodavanja informacija kako bi se izbjegla nekonzistentnost.



Slika 5.16 Zaslon detalja posjete s pogleda veterinara



Slika 5.17 Zaslon dodavanja detalja posjete

5.3. Analiza učinkovitosti agilnog razvoja u funkcionalnosti aplikacije

Tijekom razvoja web aplikacije korištena je agilna metodologija uz alat Jira. Na kraju razvoja, od ukupno trideset i tri zadatka, šest je bilo nedovršeno, dok je dvadeset i sedam zadataka završeno. Razvoj je bio organiziran kroz šest sprintova, svaki u trajanju od dva do tri tjedna.

Planiranje sprintova obuhvaćalo je stalno praćenje napretka zadataka. Na kraju svakog sprinta analizirani su dovršeni zadaci, njihova razina dovršenosti, kao i nedovršeni zadaci i razlozi njihovog nedovršavanja, poput nedostatka vremena ili ovisnosti o drugim zadacima. Također, zadaci su se po potrebi dodavali ili uklanjali iz sprinta ako su bili blokirani ili je njihovo uključivanje u sprint naknadno ocijenjeno kao korisno.

Korištenje agilne metodologije dovelo je do značajnog povećanja dokumentacije i proceduralnih koraka u usporedbi s tradicionalnim metodologijama. Svaki zadatak imao je svoju GitHub granu, kao što je objašnjeno u poglavlju 4.2.1, što je poboljšalo preglednost i organizaciju. Ipak, uporaba agilnog pristupa rezultirala je većim vremenskim ulaganjem u procedure poput ažuriranja statusa zadatka, bilježenja provedenih sati i opisivanja svakog zadatka. Također, odluke o raspodjeli zadataka po sprintovima zahtijevale su dodatno vrijeme i resurse.

5.4. Analiza učinkovitosti automatizirane kontejnerske arhitekture

U poglavlju 4.3 opisani su koraci korištenja kontinuirane dostave i integracije putem *GitHub Actions* za postizanje automatizirane kontejnerske arhitekture. Korištenjem alata *GitHub Actions* omogućena je automatizacija procesa kontejnerizacije i implementacije aplikacije putem definiranih radnih tijekova, što je značajno unaprijedilo razvojni proces.

Radni tijekovi se inicijaliziraju ručno na web stranici GitHub repozitorija, što omogućuje lakše ažuriranje verzije kada je potrebna nova verzija aplikacije. Automatsko stvaranje Docker slika dodatno je smanjilo vrijeme potrebno za isporuku i potrebu za manualnim radom, čime je proces postao manje podložan ljudskim pogreškama, kao što su pogrešna imenovanja slika, slučajno prepisivanje postojeće Docker slike i slično.

Međutim, inicijalna konfiguracija *GitHub Actions* zahtijevala je prilagodbu, uključujući rješavanje problema s konfiguracijom, optimizaciju vremena i dodavanje različitih slučajeva korištenja. Osim toga, bilo je potrebno uložiti dodatno vrijeme u istraživanje prethodno dostupnih „akcija“ koje se nalaze na *GitHub Marketplace-u.*

6. ZAKLJUČAK

Cilj ovog diplomskog rada bio je izraditi aplikaciju koja pomaže veterinarima i korisnicima prilikom liječenja ljubimaca. Aplikacija pruža uniformnu platformu te omogućuje korisnicima registraciju i prijavu, omogućuje vlasnicima registraciju svojih ljubimaca ako se ne nalaze u sustavu, zakazivanje termina posjeta veterinaru i pregled svih posjeta, uključujući primljene terapije i prepisane lijekove unutar pojedinog posjeta veterinaru. Osim funkcija za vlasnike, web aplikacija veterinarima omogućuje prijavu u sustav te dodavanje detalja za pojedini posjet. Kroz izradu aplikacije korištena je agilna metodologija uz pomoć alata Jira, te se radilo kroz sprintove kako bi se ostvarila web aplikacija sa svim njenim funkcionalnostima. Na kraju, uz pomoć alata GitHub Actions, koji je integriran s GitHub repozitorijem, napravljen je radni tijek koji omogućuje bilo kome s pristupom repozitoriju da napravi Docker sliku web aplikacije i kroz skriptu unutar repozitorija implementira aplikaciju na svom računalu, u oblak okruženju i slično. U radu su objašnjeni alati korišteni unutar aplikacije poput Spring Boota na poslužiteljskoj strani, Angulara na klijentskoj strani, PostgreSQL-a za bazu podataka te je predstavljen model i struktura pojedinog dijela aplikacije. Korištenje biblioteka poput JPA na strani poslužitelja, Liquibase i Angular Materials uvelike je olakšalo postupak kreiranja aplikacije i smanjilo potrebu za manualnim radom. Za ostvarivanje kontinuirane kontejnerske arhitekture korišteni su alati Docker za kreiranje pojedinih slika i kontejnera te GitHub Actions kako bi se ostvarila kontinuirana isporuka i dostava. U bilo kojem trenutku može se napraviti Docker slika poslužiteljske strane i/ili klijentske strane te se dobije artefakt koji se može instalirati putem Dockera.

Iako su u aplikaciji obrađeni neki slučajevi korisnika koji nisu prvobitno bili definirani, uvijek ima mjesta za daljnje poboljšanje i dodavanje novih funkcionalnosti. Aplikacija može poslužiti kao temelj za izgradnju još naprednije aplikacije. Neka od područja koja se mogu poboljšati uključuju dodavanje novih funkcionalnosti, dodavanje DTO-ova (eng. *Data Transfer Object*) na sve krajnje točke te dodavanje JWT tokena ili sličnog tipa sigurnosti prilikom prijave. S ovim unapređenjima bi se znatno poboljšala sigurnost aplikacije, korisničko iskustvo, kao i drugi aspekti aplikacije..

LITERATURA

[1] P. McGonigle, B. Ruggeri, Animal models of human disease: Challenges in enabling translation, Biochemical Pharmacology, No. 86, Vol. 7, pp. 1027-1034, August 2013, dostupno na: https://doi.org/10.1016/j.bcp.2013.08.006

[2] C.-N. Lin, K. R. Chan, E. E. Ooi, M.-T. Chiou, M. Hoang, P.-R. Hsueh, P. T. Ooi, *Animal Coronavirus Diseases: Parallels with COVID-19 in Humans*, *Viruses*, vol. 13, no. 8,p. 1507, srpanj 2021. Dostupno na: https://doi.org/10.3390/v13081507 [pristupljeno: 1. rujna 2024]

[3] T. J. Gandomani, Obstacles in Moving to Agile Software Development Methods; At a Glance, *Journal of Computer Science*, vol. 9, no. 5, p. 620, 2013.

[4] G. Miller, Agile Problems, Challenges, Failures, *ResearchGate*, 2019. Dostupno na: https://www.researchgate.net/profile/GloriaMiller2/publication/335475075_Agile_problems_challenges_failures/links/5d683a6d299bf1d599449143/Agile-problems-challenges-failures.pdf

[5] H.O. Delicheh, T. Mens, Mitigating Security Issues in GitHub Actions, *EnCyCriS/SVM '24: Proceedings of the 2024 ACM/IEEE 4th International Workshop on Engineering and Cybersecurity of Critical Systems (EnCyCriS) and 2024 IEEE/ACM Second International Workshop on Software Vulnerability*, pp. 6-11, August 2024. Dostupno na: https://doi.org/10.1145/3643662.3643961 [pristupljeno: 6. rujna 2024].

[6] K. Fišter, I. Cerovečki, I. Babić, A. Šimunec-Jović, *Bilten Hrvatskog društva za medicinsku informatiku*, Svezak 30, Broj 2, Godina 2022. Dostupno na: https://hrcak.srce.hr/ojs/index.php/bilten-hdmi/issue/view/1289/406 [pristupljeno: 9. rujna 2024].

[7] J. Duraković, "Usluga za prijavljivanje i udomljavanje ljubimaca", Diplomski rad, Sveučilište Josipa Jurja Strossmayera u Osijeku, Fakultet elektrotehnike, računarstva i informacijskih tehnologija Osijek, Osijek, 2016. Dostupno na: https://urn.nsk.hr/urn:nbn:hr:200:777378

[8] I. Dujmenović, "Web aplikacija namijenjena vlasnicima pasa kao kućnih ljubimaca", Diplomski rad, Sveučilište Josipa Jurja Strossmayera u Osijeku, Fakultet elektrotehnike, računarstva i informacijskih tehnologija Osijek, Osijek, 2022. Dostupno na: https://urn.nsk.hr/urn:nbn:hr:200:740234

[9] B. Valentić, Automated Software Delivery with GitLab CI, *Base58*, dostupno na: https://base58.hr/en/stories/automated-software-delivery-with-gitlab-ci [pristupljeno: 6. rujna 2024].

[10] P.I. Sari, K. Nashirin, M. Arifudin, Y. Setiawan, Android Mobile Application System for Pet Care Services Using MVVM Architecture, *Journal of Technology and Innovation*, vol. 7, no. 2, pp. 1-12, August 31, 2023. Dostupno na: https://ijoms.internationaljournallabs.com/index.php/ijoms/article/view/637/837 [pristupljeno: 1. rujna 2024]

[11] A.-V. Catrina, *A Comparative Analysis of Spring MVC and Spring WebFlux in Modern Web Development*, Bachelor's thesis, Helsinki Metropolia University of Applied Sciences, Helsinki, August 2024. Dostupno na: https://www.theseus.fi/bitstream/handle/10024/812448/CatrinaAlexandru.pdf?sequence=2 [pristupljeno: 3. rujna 2024].

[12] S. Khati Chhetri, *Comparative Study of Front-End Frameworks: React and Angular*, Bachelor's thesis, Helsinki Metropolia University of Applied Sciences, Helsinki, August 2024. Dostupno na: https://www.theseus.fi/bitstream/handle/10024/865488/Khati%20Chhetri_Sanjay.pdf?sequence=2&isAllowed=y [pristupljeno: 3. rujna 2024].

[13] P. Li, *Jira Software Essentials: Plan, Track, and Release Great Applications with Jira Software*, Packt Publishing, Birmingham, 2017. Dostupno na: https://books.google.hr/books?hl=hr&lr=&id=sSZKDwAAQBAJ&oi=fnd&pg=PP1&dq=jira+tool&ots=sUlTPKjDEx&sig=sRBAQAbvPR1cZxZtKxL7IS0vo&redir_esc=y#v=onepage&q=jira%20tool&f=false [pristupljeno: 2. rujna 2024].

[14] Y. Tang, *Design and Implementation of Student Management System Based on Spring Boot Framework Technologies*, u: *ICSETPSD 2023: Proceedings of the First International Conference on Science, Engineering, and Technology for Post-pandemic Sustainable Development*, ICSETPSD, India 2023. Dostupno na: https://books.google.hr/books?hl=hr&lr=&id=1Lf3EAAAQBAJ&oi=fnd&pg=PA301&dq=spring+boot+framework&ots=kqG07Hsc16&sig=6nwcj8pqfzo0LTO_iTHNbCKBxU&redir_esc=y#v=onepage&q=spring%20boot%20framework&f=false [pristupljeno: 2. rujna 2024].

[15] G. Paiz, *Angular Material UI Kit* [online], Justinmind, n.p., 2023, dostupno na: https://www.justinmind.com/ui-kits/angular-material [Posjećeno: 14.09.2024.]

[16] J. Worsley, J. D. Drake, *Practical PostgreSQL*, O'Reilly Media, Sebastopol, 2005. Dostupno na:https://books.google.hr/books?hl=hr&lr=&id=G8dh95j5NgcC&oi=fnd&pg=PR4&dq=postgresql+database&ots=8U42NPH_g6&sig=6kOaIUz0aapKmsdbKJu63jiT1o&rediresc=y#v=onepage&q=postgresql%20database&f=false [pristupljeno: 2. rujna 2024].

[17] J. Turnbull, The Docker Book: Containerization Is the New Virtualization, James Turnbull, 2014. Dostupno na: https://books.google.hr/books?hl=hr&lr=&id=4xQKBAAAQBAJ&

oi=fnd&pg=PA1&dq=docker&ots=wy4Ift4iIV&sig=4sN5kzWmYZUFEZutAOVxVCLI5qk&redir_esc=y#v=onepage&q=docker&f=false [pristupljeno: 1. rujna 2024].

[18] T. Kinsman, M. Wessel, M. A. Gerosa, and C. Treude, "How Do Software Developers Use GitHub Actions to Automate Their Workflows?," *2021 IEEE/ACM 18th International Conference on Mining Software Repositories (MSR)*, Madrid, Spain, 2021, pp. 420-431. Dostupno na: How Do Software Developers Use GitHub Actions to Automate Their Workflows? | IEEE Conference Publication | IEEE Xplore

[19] GitHub Actions, dokumentacija, Dostupno na:Understanding GitHub Actions - GitHub Docs [pristupljeno 14. rujna. 2024.]

[20] Postman, dokumentacija dostupno na: https://www.postman.com/product/what-is postman/, [pristupljeno 3. rujna 2024.]

SAŽETAK

Cilj diplomskog rada je izrada internetske aplikacije za pomoć pri liječenju životinja, koja je zasnovana na automatiziranoj kontejnerskoj arhitekturi i agilnom razvoju. U aplikaciji su ostvarene funkcionalnosti: prijava i registracija korisnika, stvaranje e-zdravstvenog kartona životinje, praćenje stanja i tijeka liječenja, primijenjenih terapija, prikaz računa te zakazivanje termina. Kroz rad su opisane korištene tehnologije: Angular na klijentskoj strani, Spring Boot na poslužiteljskoj strani, PostgreSQL za bazu podataka, Docker i GitHub Actions za implementaciju kontejnerske arhitekture i kontinuiranu integraciju koristeći definirani radni tijek za automatiziranu izgradnju Docker slika, te alat Jira za agilni razvoj pomoću kojeg su postavljeni zadaci i poboljšana organizacija projekta.

Ključne riječi: agilna metodologija, github actions, kontinuirana integracija, kontejnerska arhitektura, liječenje životinja

ABSTRACT

The goal of the thesis is to develop a web application to assist in animal treatment while utilizing an automated container architecture and agile development. The application includes functionalities such as user login and registration, creating an e-medical record for the animal, tracking the condition and progress of treatment, applied therapies, invoice display, and scheduling appointments. The thesis describes the technologies used: Angular on the client side, Spring Boot on the server side, PostgreSQL for the database, Docker and GitHub Actions for implementing container architecture and continuous integration using a defined workflow for automated Docker image builds, and Jira for agile development, which was used to set tickets and improve project organization.

Keywords: agile methodology, animal treatment, container architecture, continuous integration, github actions

ŽIVOTOPIS

Filip Kramar rođen je 09.08.2000. u Osijeku. Od 2007. do 2015. pohađa OŠ Višnjevac. Nakon toga upisuje III. gimnaziju Osijek u Osijeku. Srednju školu završava 2019. te potom upisuje Fakultet elektrotehnike, računarstva i informacijskih tehnologija Osijek koji trenutno pohađa.

PRILOZI

Prilog 1. Diplomski rad u datoteci .docx

Prilog 2. Diplomski rad u datoteci .pdf

Prilog 3. Jira Ploča projekta PetsHealth

Prilog 4. GitHub repozitorij programskoga koda FilipKramar/DiplomskiRad