Spring framework u izgradnji aplikacije za unos i pregled biljnih vrsta

Jauk, Toni

VELEUČILIŠTE U RIJECI

Toni Jauk

SPRING FRAMEWORK U IZGRADNJI APLIKACIJE ZA UNOS I PREGLED BILJNIH VRSTA

(specijalistički završni rad)

Rijeka, 2020.

VELEUČILIŠTE U RIJECI
Poslovni odjel
Specijalistički diplomski stručni studij Informacijske tehnologije u poslovnim sustavima

SPRING FRAMEWORK U IZGRADNJI APLIKACIJE ZA UNOS I PREGLED BILJNIH VRSTA

(specijalistički završni rad)

MENTOR

Vlatka Davidović, viši predavač

STUDENT

Toni Jauk, bacc. Inf.

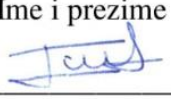
MBS: 0242036788

Rijeka, rujan 2020.

I Z J A V A

Izjavljujem da sam specijalistički završni rad pod naslovom **SPRING FRAMEWORK U IZGRADNJI APLIKACIJE ZA UNOS I PREGLED BILJNIH VRSTA** izradio samostalno pod nadzorom i uz stručnu pomoć mentora Vlatka Davidović, viši predavač.

Ime i prezime



(potpis studenta)

VELEUČILIŠTE U RIJECI
Poslovni odjel

Rijeka, 2. lipnja 2020.

ZADATAK
za specijalistički završni rad

Pristupnik TONI JAUK **MBS: 2422000113/18**

Studentu specijalističkog diplomskog stručnog studija Informacijske tehnologije u poslovnim sustavima izdaje se zadatak specijalističkog završnog rada – tema specijalističkog završnog rada pod nazivom:

SPRING FRAMEWORK U IZGRADNJI APLIKACIJE ZA UNOS I PREGLED BILJNIH VRSTA

Sadržaj zadatka: Kreirati *web*-aplikaciju za unos i pregled biljnih vrsta pri čemu bi se podaci o biljnim vrstama i njihovim karakteristikama bilježili u relacijsku bazu podataka. Aplikaciju razviti u Spring frameworku, koristiti Hibernate i JPA za kreiranje baze podataka. Opisati navedene tehnologije i dokumentirati izgradnju aplikacije kroz njih. *Web*-aplikaciju pripremiti za isporuku.

Preporuka _____

Rad obraditi sukladno odredbama Pravilnika o završnom radu Veleučilišta u Rijeci.

Zadano: 2.lipnja 2020. **Predati do:** 15.rujna 2020.

Mentor:
Vlatka Davidović, viši predavač

Pročelnica odjela:
mr. sc. Anita Stilin, viši predavač

Zadatak primio dana: 2. lipnja 2020.

Toni Jauk

Dostavlja se:
- **mentoru**
- **pristupniku**

Sažetak

Cilj ovog specijalističkog završnog rada je kreiranje *web*-aplikacije za unos i pregled biljnih vrsta, s naglaskom na izradu serverskog dijela aplikacije. Na serverskom dijelu aplikacije kreiran je REST API (eng. *Representational State Transfer Application Interface*) servis koji će pružati krajnje točke za dohvat i upravljanje podacima i klijentskog dijela aplikacije kojim će korisnik komunicirati s serverskom stranom aplikacije koristeći HTTP (eng. *HyperText Transfer Protocol*) pristup upita i odgovora. U razvoju serverskog dijela *web*-aplikacije korišteni su Spring i Hibernate razvojni okviri, a za klijentski dio aplikacije Vue.js. Klijentski dio aplikacije implementira SPA (eng. Single Page Application) pristup. Sigurnost i zaštita podataka postignuta je JWT (eng. *JSON Web Token*) autentifikacijom. Pisani dio ovoga rada opisuje pojmove, korištene tehnologije, komponente, pristup i proces kreiranja praktičnog dijela rada.

Ključne riječi: serverska strana aplikacije, klijentski dio aplikacije, Java, Spring, Vue.js

Sadržaj

1. Uvod

Web-aplikacija je računalni program koji ima sve funkcionalnosti klasične desktop aplikacije, s razlikom što se pokreće u *web*-pregledniku. *Web*-aplikacije su temeljene na klijent-server modelu. Klijentu se prikazuje grafičko sučelje preko kojeg on komunicira s serverskom stranom aplikacije (eng. *backend*).

Svaka se *web*-aplikacija sastoji od dva dijela pa tako i *web*-aplikacija ovog praktičnog dijela rada. Sastoji se od serverske strane aplikacije i klijentskog dijela aplikacije. Serverski dio praktičnog dijela ovog rada kreiran je koristeći Spring *framework* koji je baziran je na Java programskome jeziku. Spring Boot je njegova komponenta, te je korišten kako bi se izgradila struktura aplikacije i dodale potrebne ovisnosti (eng. *dependency*) koje serverska strana aplikacije koristi. Ovisnosti predstavljalju razne biblioteke koje nadograđuju aplikaciju svojim funkcionalnostima tj. dijelovi aplikacije naslijeđuju funkcionalnosti ovisnosti. Aplikacija koristi Maven kao SPM (eng. *Software Project Managment*) za upravljanje projektom. Ovisnosti koje se koriste u ovome radu su: Hibernate, Spring Data JPA, Spring Security i Lombok. Hibernate i Spring Data JPA kreiraju bazu podataka i relacije dok Lombok uređuje i otklanja nepotrebni dio koda unutar modela serverske strane aplikacije. Komponenta Spring Security dodaje autentifikaciju i autorizaciju nad krajnjim točkama (eng. *endpoint*).

Klijentski dio praktičnog dijela ovog rada kreiran je u Vue.js *frameworku* koji se bazira na JavaScript programskome jeziku. Vue-Material i MDB (eng. Material Bootstrap Design) korišteni su kao biblioteke koje nadograđuju dizajn samog klijentskog dijela aplikacije. Uz to koristi se i biblioteka Axios koja povezuje klijentski dio aplikacije s serverskom stranom. Axios poziva upitima krajnje točke iz serverske strane praktičnog dijela rada i prikazuje odgovor korisniku aplikacije. Nad tom komunikacijom primjenjen je JWT (eng. *JSON web token*). JWT je jednostavni standard koji definira način za sigurno komuniciranje između serverske strane i klijentskog dijela aplikacije. Uz Axios koristi se i Vuex biblioteka kako bi se omogučilo pohranjivanje tokena koji autorizira korisnika aplikacije.

2. Arhitektura sustava

Na primjeru računala, serversku stranu aplikacije se može svrstati u komponente računala koje obavljaju posao (RAM (eng. *Random Access Memory*), GPU (eng. *Graphic Processing Unit*), CPU (eng. *Computer Processing Unit*) i dr.), dok klijentski dio aplikacije računala se može gledati kroz komponente monitora, tipkovnice, miša, itd. On predočava ono što serverska strana aplikacije obavlja u pozadini. Korisnik komunicira s klijentskim djelom aplikacije, tj. daje naredbe serverskoj strani aplikacije preko klijentskog dijela. Radnje koje serverska strana aplikacije obavlja u pozadini nisu vidljive i korisnik ih nije svjestan. Serverska strana aplikacije je u principu arhitektura serverske strane *web*-aplikacije.

Serverska strana aplikacije je zadužena za procesiranje svakog korisnikova upita u tajnosti. Korištenjem serverske strane aplikacije korisniku se ne dopušta zloupotreba mašinerije aplikacije, baze podataka i osjetljivog sadržaja. Uključuje procesiranje, bazu podataka i svu ostalu mašineriju aplikacije. Serverska strana aplikacije se koristi i zato da korisnik ne mora dodatno instalirati i postavljati, osim *web*-preglednika, ako želi koristiti određenu *web*-aplikaciju.

Komponente serverske strane su: server, baza podataka, posrednik (eng. *middleware*), programski jezici, *frameworci*. *Framework* je komponenta koja se bazira na komponenti programskoga jezika teje ona ovisna o drugima. Serverska strana aplikacije se može isprogramirati u raznim jezicima i *frameworcima*. Za ovaj praktični dio rada koristi se Spring kao *framework* koji je razvijen na Java programskome jeziku i koji kreira REST API (eng. *Representational State Transfer Application Interface*) servis. Svaki programski jezik i njegov *framework* imaju razne specifikacije koje su drugačije od drugih jezika i *frameworka* (brzina, kompliciranost, kompatibilnost, potrebitost itd.).

Posrednik je bilo koji softwer koji spaja serversku i klijentsku stranu aplikacije, on se ponaša kao posrednik koji uzima upit s klijentskog dijela aplikacije te ga prosljeđuje serverskoj strani i vraća odgovor klijentu. U ovome radu to je Axios. Server se koristi kao lokacija za spremanje serverske strane aplikacije. U ovome radu za upravljanje serverskom stranom, Springom koristi se Maven, dok je server lokalni development server koji se vrlo jednostavno može postaviti korištenjem raznih aplikacija koje kreiraju lokalne development servere. Baza

podataka je zadnja komponenta koja služi za spremanje svih podataka koje koristi serverska strana. Baza podataka za ovaj rad rađena je u MySqlu. Baza podataka od statičke *web*-aplikacije kreira dinamičku *web*-aplikaciju s protokom podataka koji tu aplikaciju hrani.

Klijentski dio aplikacije je dio aplikacije koju korisnik vidi i koristi, a kreira se koristeći HTML (eng. *Hyper Text Markup Language*), CSS (eng. *Cascading Style Sheet)* i JavaScript za *web*-aplikacije. Korisnik koristi *web*-aplikaciju preko *web*-preglednika. Cilj klijentskog dijela je jednostavnost i dizajn. Korisniku se žele prikazati podaci i informacije na što jednostavniji način, tj. na način koji je razumljiviji korisniku. Uz to želi se postići konzistentnost dizajna i ljepota kako bi se privuklo korisnika na korištenje aplikacije. U današnje je vrijeme vrlo važna komponenta klijentskog dijela *web*-aplikacije responzivnost. Time se postiže korištenje *web*-aplikacije na raznim uređajima. Veliku ulogu u tome ima JavaScript. U ovom specifičnome primjeru se koristi Vue.js *framework*. Vue.js je SPA (eng. *Single Page Application*), koristi arhitekturu jedne stranice. Takav pristup učitava HTML te dohvaća JS (eng. *JavaScript*). Nakon što je JS pokrenut on izvršava sve upite premaserverskoj strani, a serverska strana aplikacije pruža podatke klijentu. Takav proces se većinom izvodi u JSON (eng. *JavaScript Object Notation*) formatu. JS pomaže pri učitavanju podataka sserverske strane, dok je DOM (eng. *Document Object Model*) izgrađen dinamički bazirajući se na podacima. DOM elementi spojeni s JS-om prikazuju se korisniku. U takvoj komunikaciji serverska strana aplikacije ne zna kako klijent izgleda i što radi s podacima, ali nije mu ni važno jer mu je cilj samo odgovoriti u JSON formatu.

REST (eng. Representational State Transfer) je set uputa i protokola koji opisuju komunikaciju, kako ona mora izgledati između aplikacija koje koriste internet. REST API koristi HTTP metode kako bi komunicirao. Upit REST servisa sadrži:

1. Krajnje točke URL-a (eng. *Uniform Resource Locator*)
2. HTTP metodu
3. HTTP zaglavlje
4. Tijelo.

Na serverskoj strani definiraju se REST API krajnje točke (eng. *endpoint*) koji sadrže URL kako bi im se moglo pristupiti. HTTP metode se definiraju na serverskoj strani aplikacije za svaku krajnju točku.

3. Analiza sustava za unos i pregled biljnih vrsta

Sustav biljnih vrsta se kreira u svrhu praćenja i bilježenja biljaka te stvaranja baze svih prikupljenih i analiziranih vrsta.

> ***sistematika*** *(prema grč. συστηματικός: zajedno sastavljen), grana biologije koja proučava biološke raznolikost organizama i njihovu međusobnu povezanost te ih svrstava u odgovarajuće srodstvene skupine. Uglavnom se podudara s taksonomijom, granom koja se bavi identifikacijom, klasifikacijom i nomenklaturom današnjih (recentnih) te izumrlih (fosilnih) organizama* (Hrvatska Enciklopedija, Pristupljeno: 8.8.2020).

Glavne smjernice koje su bile najvažnije za kreiranje baze podataka ovoga rada, i po kojima se programirala aplikacija, bile su sistematske kategorije u botanici. Sistematske se kategorije mogu najbolje opisati kroz tablicu. Važno je napomenuti da su sistematske kategorije ovisne o redoslijedu, stoga se njihov redoslijed ne smije mijenjati. Primjerice, porodica može imati više rodova, ali rod ne može imati više porodica. U sljedećoj se tablici takav redoslijed očituje silaznom putanjom.

Tablica 1: Sistematske kategorije biljke

Sistematske kategorije	**Završetak**	**Primjer: cvjetača**	**Primjer: dalmatinski crni bor**
Carstvo ***(regnum)***		*Plantae* (biljke)	*Plantae* (biljke)
Podcarstvo ***(subregnum)***	*-bionta*	*Cormobionta (Cormophyta)*	*Cormobionta*
Odjeljak ***(phylum)***	*-phyta*	*Spermatophyta* (sjemenjače)	*Spermatophyta* (sjemenjače)
Pododjeljak ***(subphylum)***	*-phytina*	*Magnoliophytina (Agniospermae)* (kritosjemenjače)	*Coniferophytina (Pinophyta)* (igličaste i rašljaste golosjemenjače)
Razred ***(classis)***	*-phyceae, -atae, -opsida*	*Magnoliatae (Dicotyledoneae)* (dvosupnice)	*Pinatae*

Podrazred (subclassis)	*-idae*	*Dilleniidae*	*Pinidae (Coniferae)*
Nadred (superordo)	*-anae*	*Dillenianae*	
Red ordo)	*-ales*	*Capparales*	*Pinales*
Porodica (familia)	*-aceae*	*Brassicaceae (Cruciferae)* (krstašice)	*Pinaceae* (borovi)
Potporodica (subfamilia)	*-oideae*		*Pinoidae*
Rod (genus)		*Brassica* (vrzina)	*Pinus* (bor)
Vrsta (species)		*Brassica oleracea* (vrtna vrzina)	*Pinus nigra* (crni bor)
Podvrsta (subspecies)		*Brassica oleracea subsp. oleracea*	*Pinus nigra subsp. dalmatica* (dalmatinski crni bor)
Odlika ili varijetet (varietas)		*Brassica oleracea subsp. oleracea var. botrytis* (cvjetača)	

Izvor: Hrvatska Enciklopedija

U današnje se vrijeme sistematika i taksonomija često izjednačavaju jer se sistematika temelji na filogenetskoj srodnosti organizama, koja je odraz njihova razvoja od zajedničkih predaka, a isti takav princip ima i taksonomija. Aristotel je napravio prvu specifikaciju živoga svijeta, a u 18. stoljeću C. Von Linne je biljke sistematizirao prema građi reproduktivnih organa, a životinje po anatomskim obilježjima. Takav sustav klasifikacije organizama zasniva se na sličnosti organizama, a ne na njihovoj srodnosti, stoga se može reći da je umjetan. Linnea se smatra utemeljiteljem znanstvene sistematike čiji se sustav imenovanja živoga svijeta i danas koristi. Takav sustav naziva se *binarna nomenklatura* prema kojem svaka vrsta ima svoje latinsko ili latinizirano prezime i ime. Zapravo, se može reći da svaka vrsta ima svoje znanstveno prezime i ime. Nomenklatura u ovome radu i izgradnji *web*-aplikacije ima veliku ulogu, što će se vidjeti u sljedećim poglavljima. Do značajnog je napretka u sistematici došlo objavom Darwinove teorije evolucije te se samim time danas srodni organizmi svrstavaju u sistematske to jest taksonomske jedinice koje su u hijerarhijskome sustavu.

Vrsta je osnovna taksonomska jedinica koja ima više taksonomske jedinice i podjedinice. Vrsta obuhvaća sve jedinke istoga podrijetla, izgleda i anatomske građe, koje međusobnim razmnožavanjem daju plodne potomke. Postoji više glavnih viših taksonomskih jedinica (tablica) dok su one koje su obrađene u ovome radu:

1. Rod
2. Porodica

Kako postoji veći broj viših jedinica tako postoji i veći broj nižih jedinica (tablica), a one koje su obrađene u ovome radu su:

1. Podvrsta
2. varijet.

4. Specifikacija zahtjeva

Aplikacija ima tri aktera koji se vežu i kao uloge (eng.*role*) koje korisnik ima, to jest koje su mu dodjeljene u bazi podataka. Akteri su:

1. Admin
2. Profesor
3. Student
4. Ostali – osnovni akter koji se ne bilježi u bazi.

Akter profesor je „glavni“ akter, i ima najviše ovlasti. Jedino akter, tj. korisnik s ulogom administratora ima veće ovlasti, administrator je programer ili korisnik koji održava sustav. Funkcionalnosti sustava koje će koristiti profesor su:

1. registracija korisnika kojima je rola student
2. prijava korisnika
3. unos podataka o biljci
4. ažuriranje svih unesenih podataka, tj. svakog zapisa
5. pregledavanje svih podataka, tj. svih zapisa
6. pretraga svih podataka
7. filtriranje svih podataka

8. pregled profila.

Funkcionalnosti sustava koje će koristiti student su:

1. prijava
2. pregledavanje svih podataka o biljci
3. unos podataka o biljci
4. pretraga podataka o biljci
5. filtriranje podataka o biljci
6. pregled profila.

Funkcionalnosti sustava koje će koristiti korisnik su:

1. pregledavanje svih podataka o biljci
2. pretraga podataka o biljci
3. filtriranje podataka o biljci.

Funkcionalnosti sustava koje će koristiti administrator su sve one funkcionalnosti koje koristi i profesor, uz dodatne funkcionalnosti, koje su:

1. unos uporabnog dijela biljke
2. ažuriranje uporabnoga dijela biljke
3. registracija profesora
4. ažuriranje profesora
5. pregledavanje tablica registiranih korisnika
6. pretraživanje tablica registriranih korisnika.

Postoje tri različite funkcinonalnosti pregledavanja. Funkcionalnost *pregledavanje svih podataka o biljci* ima mogućnost pregledavanja samo porodice, roda, biljne vrste, pripadajućih slika i uporabnih dijelova svake bilje vrste, podvrste i varijeta. Funkcionalnost *pregledavanje svih podataka* ima mogućnost pregledavanja svih podataka o biljci uz pregledavanje dodatnih podataka kao što su podaci o uporabnom dijelu i podaci o studentima. Funkcionalnost *pregledavanja registriranih korisnika* ima mogućnosti prve dvije funkcionalnosti uz dodatnu funkcionalnost pregledavanja zapisa profesora.

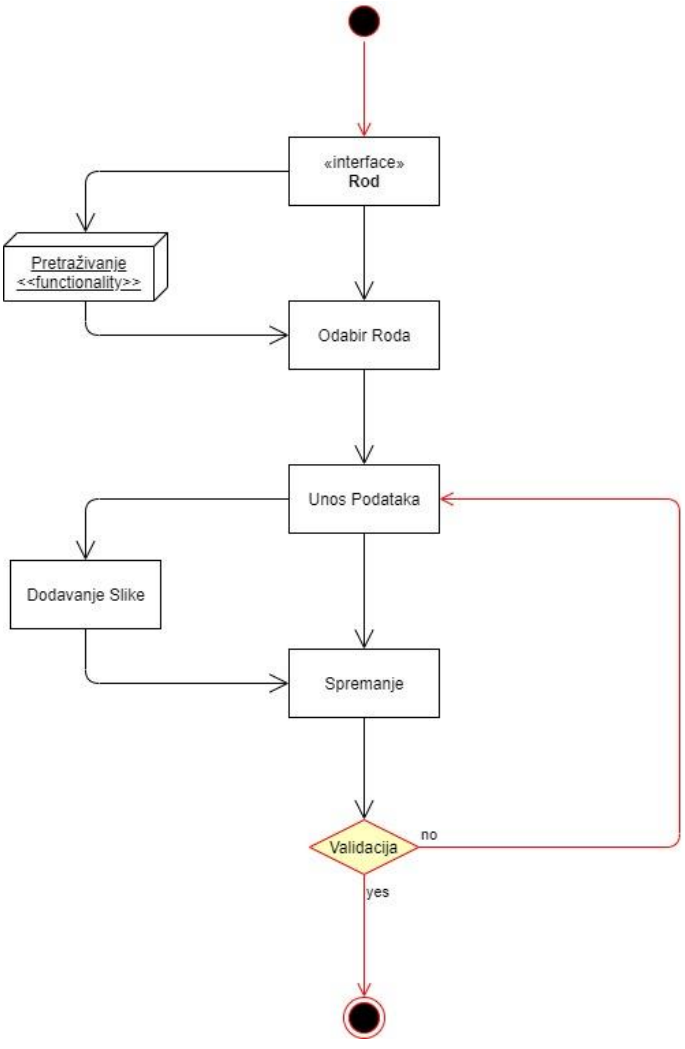
Funkcionalnost *unos podataka o biljci* kreira biljku, tj. kreira porodicu, rod, biljnu vrstu, podvrstu, varijet, dodaje uporabni dio biljnoj vrsti i dodaje slike biljnoj vrsti. Funkcionalnost *ažuriranje svih unesenih podataka* uređuje i briše sve podatke, neovisno o tome jesu li to podaci o biljci, slici ili studentu.

Postoje tri funkcionalnosti pretrage i filtriranja. *Pretraga i filtriranje biljke* odnosi se samo na one podatke iz funkcionalnosti iz pregleda svih podataka o biljci. *Pretraga i pretraživanje podataka* odnosi se na one podatke iz funkcionalnosti pregleda svih podataka, dok se *pretraga i filtriranje dodatnih tablica* odnosi na one podatke funkcionalnosti pregledavanja dodatnih tablica. Funkcionalnost je *pregled profila* i ona u sebi sadrži pregled i uređivanje vlastitoga profila.

4.1. Dijagram aktivnosti

U ovome će poglavlju biti opisana tri osnovna korištenja sustava na klijentskom dijelu aplikacije od strane korisnika prema funkcionalnostima sustava. Prvi primjer korištenja sustava je *unos biljne vrste*. Prvi korak je odabir sučelja u kojem su prikazani svi rodovi. Drugi korak je odabir željenog roda za koji se unosi biljna vrsta. Akose rodnije pronađen, koristi se pretraživanje ili filtriranje svih dostupnih rodova. Nakon što je rod odabran, unose se podaci o biljnoj vrsti. Nakon unesenih podataka, korisnik dodaje slike rodu i sprema unos, ili ga sprema bez dodavanja slika. Nakon što je odabrano spremanje roda, aplikacija provjerava jesu li podaci pravilno unijeti. Ako su podaci pravilno unijeti, proces unosa se završava uspješno, a ako podaci nisu pravilno unijeti, korisnik je dužan popraviti nepravilne unose te ponovno pokrenuti spremanje podataka u bazu podataka.

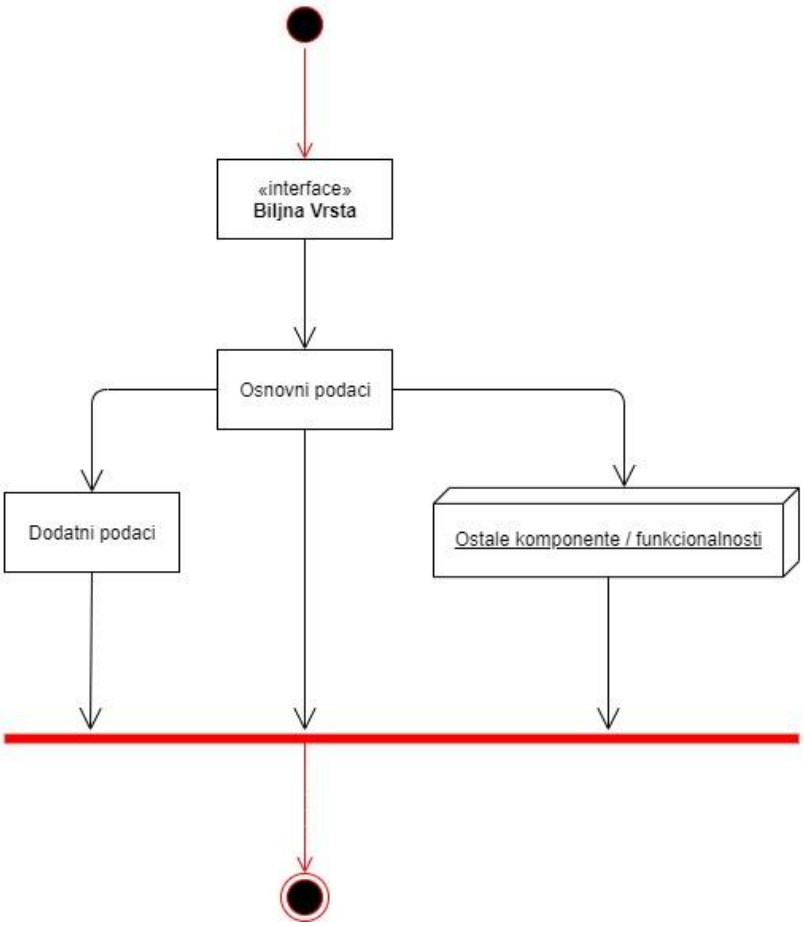
Slika 1: Dijagram- unos biljne vrste



Izvor: Izrada autora

Drugi glavni primjer korištenja je *pregledavanje zapisa*. *Pregledavanje zapisa* je prilično jednostavno. Na primjeru *pregledavanje zapisa biljne vrste* vidi se da je prvi korak odabir sučelja za pregledavanje biljne vrste u kojem su prikazani osnovni podaci. Nakon pregleda osnovnih podataka, korisnik ima tri mogućnosti. Prva mogućnost je završiti pregledavanje i samim time završiti ovaj primjer korištenja. Druga mogućnost je pregled dodatnih podataka te završavanje ovog primjera korištenja. Posljednja je, treća, mogućnost prelazak na neku drugu funkcionalnost korištenja, a to je: pregled podvrsta odabrane biljne vrste, unos podvrste, ažuriranje odabrane biljne vrste i završavanje ovog primjera korištenja.

Slika 2: Dijagram- pregledavanje zapisa biljne vrste

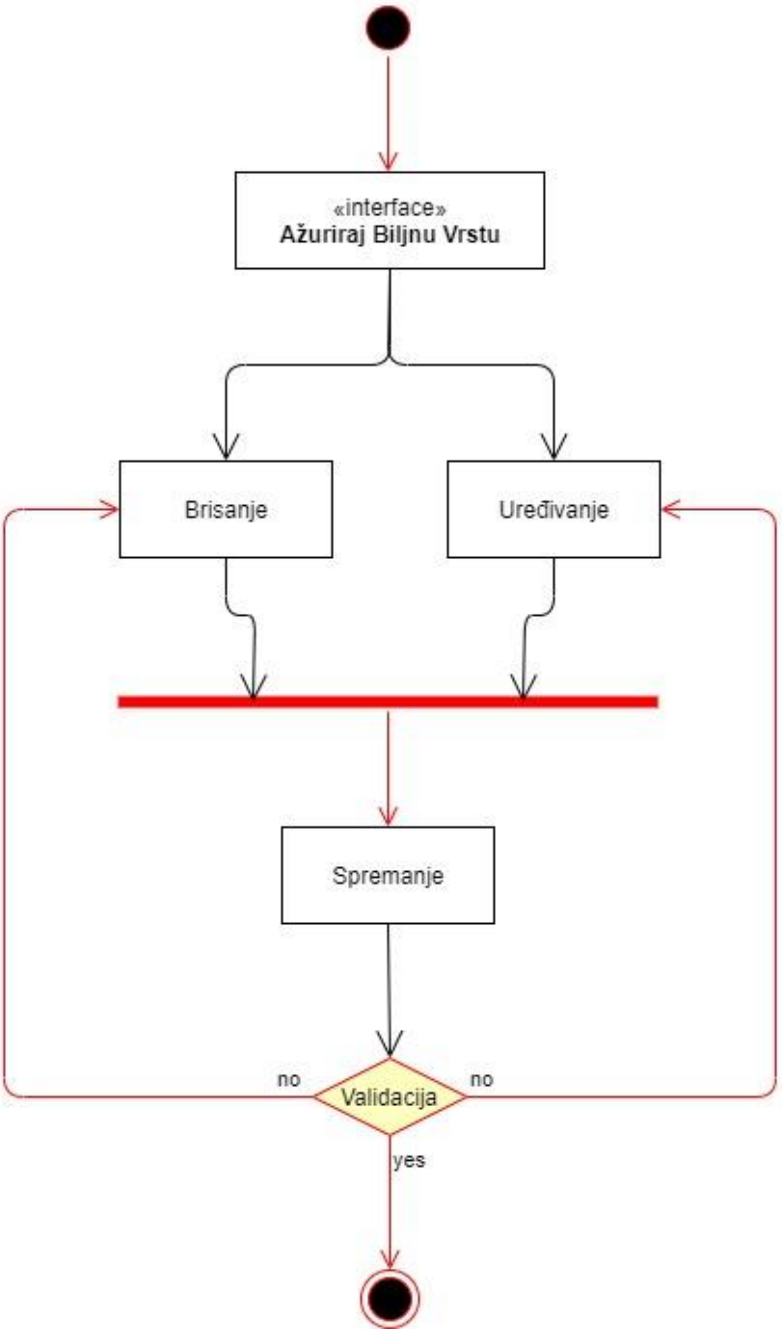


Izvor: Izrada autora

Treći, i zadnji, primjer korištenja je *ažuriranje podataka*. *Ažuriranje podataka* u prvome koraku ima odabir određenoga sučelja, u ovome primjeru sučelja *ažuriranje biljne vrste* to jest zapisa. Nakon što je odabrano sučelje, korisnik odabire želi li obrisati ili urediti podatke o tom zapisu. Nakon što odabere korak, sprema promjene. Nadalje, validacija provjerava

ažuriranje, a ako je ažuriranje uspješno izvršeno, proces ažuriranja se završava. U suprotnome slučaju korisnika se vraća na druga dva koraka, ovisno o tome koji je korak korisnik htio obaviti.

Slika 3: Dijagram- ažuriranje podataka



Izvor: Izrada autora

4.2. Relacijski model

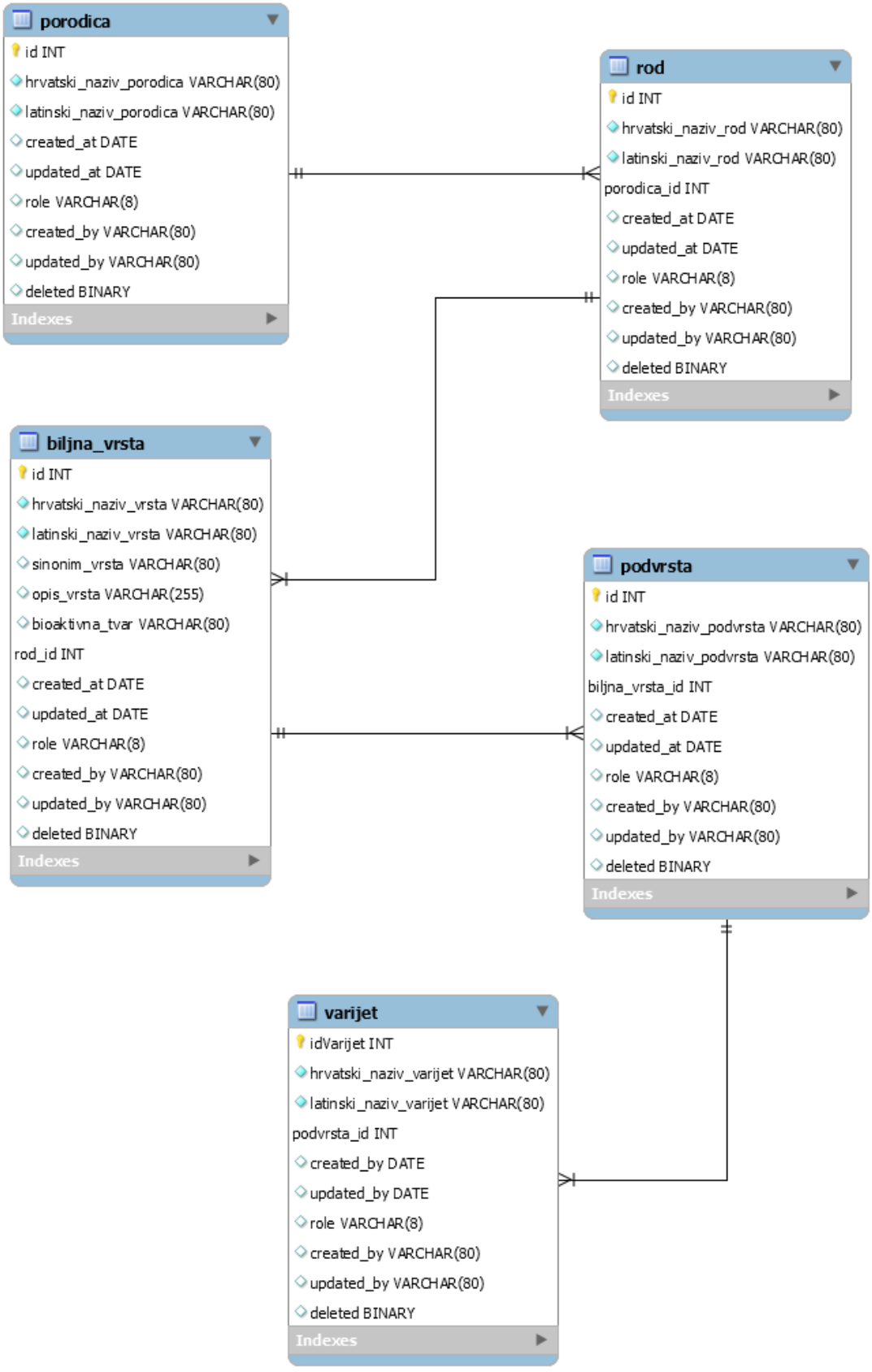
Relacijski model po kojemu je kreirana baza podataka sastoji se od dvanaest tablica od kojih su dvije agregacije. Relacijski model ima i 10 relacija, a podijeljen je u dva zasebna, međusobno nepovezana modela. Prvi model je model koji sadrži tablice s glavnim podacima aplikacije, dok drugi model sadrži tablice za autorizaciju. Prvi model, *Biljka*, koji je zadužen za osnovni rad aplikacije sadrži devet tablica, osam relacija te jednu agregaciju, a podijeljen je u dva zasebna podmodela. Jedan model prikazuje osnovnu podjelu biljke po relacijama, dok je drugi model zadužen za stvaranje relacija i podataka koji povezuju uporabni dio i sliku s odabranom biljkom. Drugi model je model *Autorizacija* te je on prvenstveno, i jedino, zadužen za obradu podataka korisnika aplikacije, tj. sustava. Sadrži tri tablice, od toga jednu agregaciju i dvije relacije. U nastavku će ovoga poglavlja relacijski model biti prikazan pomoću tri dijagrama, kako je prethodno opisano.

Tablice prvog modela *Biljka* su:

1. porodica
2. rod
3. biljna_vrsta
4. podvrsta
5. varijet
6. uporabni_dio
7. uporabni _dio_vrste (agregacija)
8. slika.

Prvih pet tablica prati i predstavlja povezanost sa sistematikom same biljke te njezine relacije (jedan prema više) to očituju. Preostale tri tablice imaju relaciju prema tablici *biljna_vrsta* te služe kao tablice s dodatnim podacima o određenoj biljnoj vrsti, a ako gledamo i po sistematici, tada one predstavljaju povezanost s čitavom biljkom, s obzirom na to da je centar biljke u sistematici biljna vrsta. Uz to, valja napomenuti da je u prvome dijelu modela jedina agregacija *uporabni_dio_vrste* koja je povezana s tablicom *uporabni_dio* i tablicom *biljna_vrsta* te nam govori da biljna vrsta može imati više uporabnih dijelova kao i da uporabni dio može pripadati jednoj ili više biljnih vrsti.

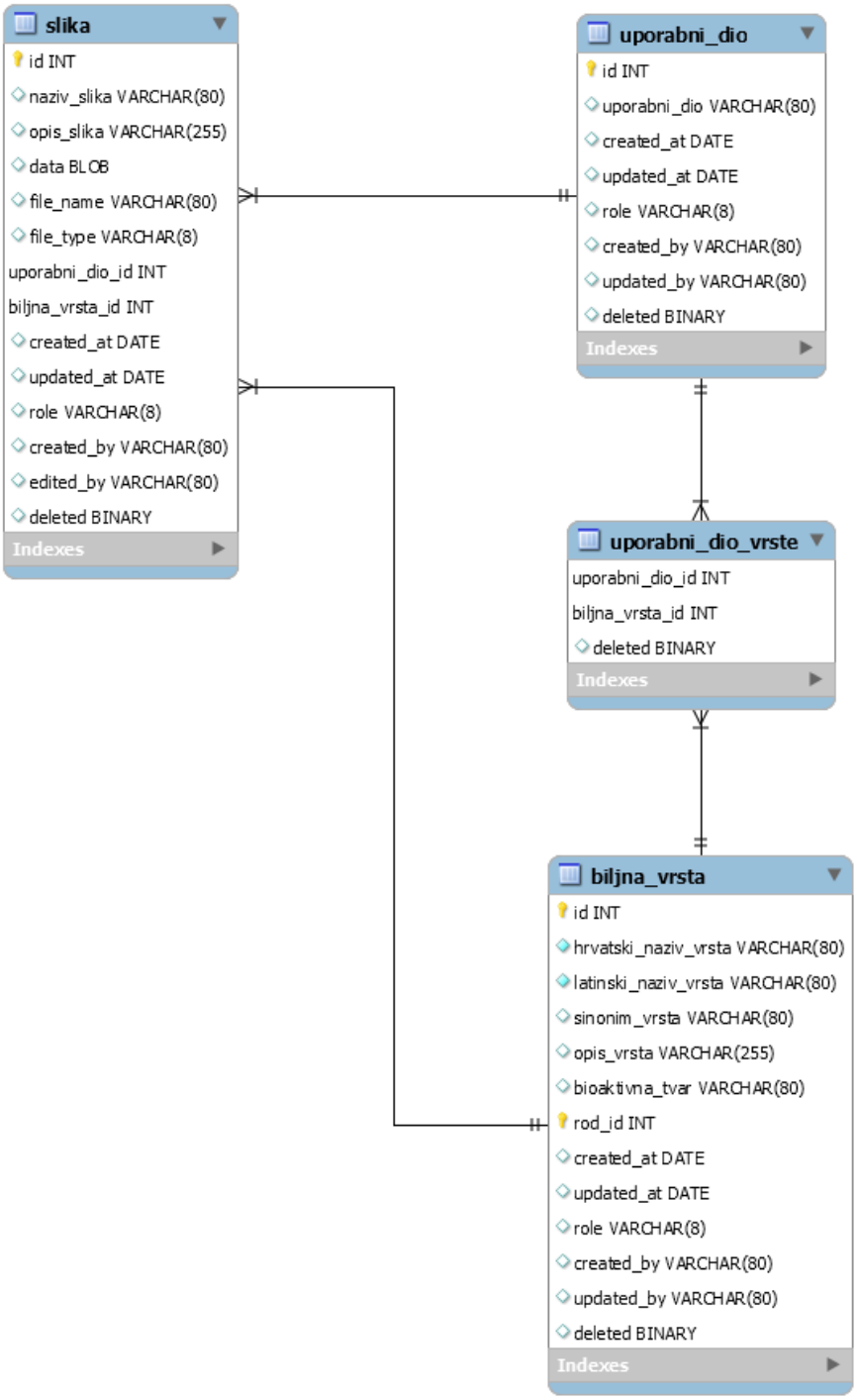
Slika 4: Relacijski model- Biljka dio 1



Izvor: Izrada autora

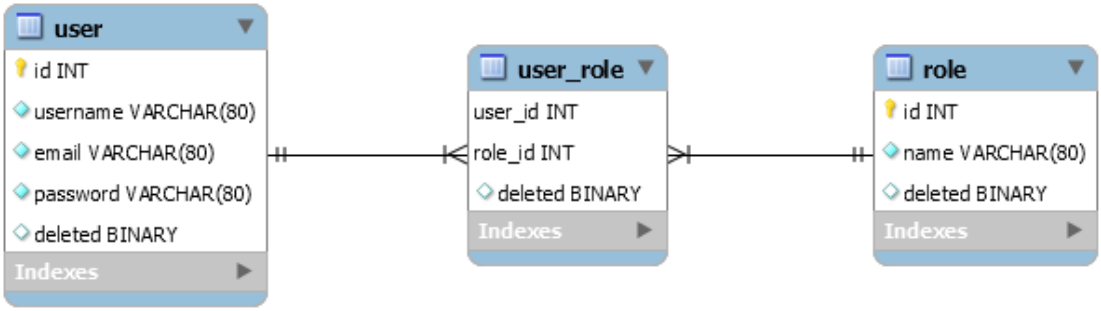
U svakoj tablici jedini atributi koji su obavezni za unos su hrvatski i latinski nazivi, dok ostali atributi nisu obavezni prilikom unosa. Takva je praksa dobra, zato što su glavni atributi vrlo važni i moraju se znati prilikom bilježenja biljke, dok se ostali atributi mogu saznati kasnije prilikom bilježenja, ili prilikom trenutnog bilježenja nisu relevantni.

Slika 5: Relacijski model- Biljka dio 2



Izvor: Izrada autora

Slika 6: Relacijski model- Autorizacija



Izvor: Izrada autora

Drugi dio relacijskog modela *Autorizacija* odnosi se na tablice koje sadrže podatke o korisnicima i služi za autorizaciju i autentifikaciju, a sastoji se od tri tablice od kojih je jedna agregacija i dvije relacije. Tablice su:

1. user
2. user_role
3. role.

Korisnik može imati više uloga, dok jednoj ulozi može pripadati više korisnika. Uz gore navedeno, i prikazane dijagrame, svaka tablica u modelu *Biljka* ima i šest, to jest svaka tablica u modelu *Autorizacija* ima jedan, atribut/a koji se nalaze/i u svakoj tablici. Atributi su generirani od strane autora, a oni su:

1. created_at
2. updated_at
3. created_by
4. updated_by
5. role
6. deleted.

Prva dva atributa su datumskoga tipa i oni predstavljaju vrijeme dodavanja zapisa i vrijeme uređivanja zapisa u svakoj tablici te se automatski dohvaćaju. Druga dva atributa predstavljaju korisnika koji je dodao, ili uredio, atribut i njihovi podaci se dohvaćaju s aplikacijske strane za svaki zapis automatski. Oni na određeni način predstavljaju vanjske ključeve. Atribut *role* predstavlja stanje tog zapisa, dohvaća se s aplikacijske strane i može poslužiti kao određeni filter. Atribut *deleted* predstavlja u binarnome obliku privremeno

brisanje (eng.*soft delete)* i služi kao zamjena za brisanje zapisa iz tablice. Svi podaci u svim atributima svakog zapisa mogu se uređivati osim podataka u atributima *updated_at* i *updated_by*., ta dva atributa predstavljaju prvo stanje zapisa, i kao takvi se ne smiju uređivati.

4.2.1. Shema relacije

Shema relacije prikazuje popis svih tablica, atributa tablica, primarnih ključeva i relacija, tj. vanjskih ključeva. Shema relacije je dodatni zapis koji u potpunosti prikazuje relaciju na čitljiv način i pomaže u razumijevanju samog relacijskoga modela, a samim time i baze podataka kao i jednosmjerne (eng. *unidirectional*), i dvosmjerne (eng. *bidirectional*) veze.

porodica (**id (PK)**, hrvatski_naziv_porodica, latinski_naziv_porodica, created_at, updated_at, role, created_by, updated_by, deleted)

rod (**id (PK)**, hrvatski_naziv_rod, latinski_naziv_rod, *porodica_id (FK)*, created_at, updated_at, role, created_by, updated_by, deleted)

biljna_vrsta (**id (PK)**, hrvatski_naziv_vrsta, latinski_naziv_vrsta, sinonim_vrsta, opis_vrsta, bioaktivna_tvar, *rod_id (FK)*, created_at, updated_at, role, created_by, updated_by, deleted)

podvrsta (**id (PK)**, hrvatski_naziv_podvrsta, latinski_naziv_podvrsta, *biljna_vrsta_id (FK)*, created_at, updated_at, role, created_by, updated_by, deleted)

varijet (**id (PK)**, hrvatski_naziv_varijet, latinski_naziv_varijet, *podvrsta_id (FK)*, created_by, updated_y, role, created_by, updated_by, deleted)

slika (**id (PK)**, naziv_slika, opis_slika, data, file_name, file_type, *uporabni_dio_id (FK)*, *biljna_vrsta_id (FK)*, created_at, updated_at, role, created_by, edited_by, deleted)

uporabni_dio (**id (PK)**, uporabni_dio, created_at, updated_at, role, created_by, updated_by, deleted)

uporabni_dio_vrste (***uporabni_dio_id (PK, FK)***, ***biljna_vrsta_id (PK, FK)***, deleted)

user (**id (PK)**, username, email, password, deleted)

user_role (***user_id (PK, FK)***, ***role_id (PK, FK)***, deleted), role (***id (PK)***, name, deleted)

5. Spring framework

Spring je *framework* za kreiranje Java *enterprise* aplikacija. On je *framework* otvorenoga koda (eng. *open source*) Kroz godine izgradnje i poboljšanja postao je jedan od najboljih *frameworkova* za izgradnju *enterprise* aplikacija. Pokreće se, i ima potporu, za JDK (eng. *Java Development Kit*) 8 pa nadalje. Sadrži sve što nam treba za izgradnju projekta te podržava različit broj scenarija za izgradnju *web*-aplikacija. U velikim poduzećima aplikacije se većinom kreiraju i koriste na duže vrijeme te se zbog toga trebaju pokretati na JDK-u i aplikacijskom serveru čiji je krug nadogradnje izvan kontrole developmenta. Glavne tehnologije Springa su ugradnja ovisnosti (eng. *dependency*), *eventi*, resursi, validacija, povezivanje podataka i ostali dok se *web framework* Springa bazira na MVC (eng. *Model View Controll*) i Spring *Web*Flux pristupu.

Spring je u početku bio jedna cjelina, tj. *framework*, no s njegovom nadogradnjom rascijepan je na projekte koji imaju različite funkcionalnosti (Spring Boot, spring Security, spring Dana JPA i ostali). Ovaj praktični dio rada bavi se prvenstveno Spring *frameworkom* koji u sebi ima implementiranih par Spring projekata, ali i ostale projekte, biblioteke i tehnike programiranja. Prilikom prvog kreiranja projekta, dobro je krenuti s Spring Bootom. Spring Boot omogućuje brzo i prilagođeno kreiranje projekta baziranog na Spring *frameworku*.

Kako su svi projekti nastali iz Spring *frameworka*, tako se može reći da je on osnova ili baza čitavog sustava. Tu do izražaja dolazi znanje o osnovnim Spring tehnologijama, tj. ugradnji ovisnosti (eng. *dependency*). Kroz ovisnosti se ugrađuju razne biblioteke koje nasljeđivanjem i korištenjem aplikaciji pružaju svoje funkcionalnosti. Spring *framework* je kontejner za komponente koje grade aplikaciju. Komponente su međusobno slabo povezane (engl. loosly coupled) a kod razvoja aplikacije povezuju se mehanizmom nazvanim ubacivanje ovisnosti (engl. dependency injection). Navedeni mehanizam je jedan od uzoraka dizajna koji se često koristi kako bi se aplikacija mogla lako nadograditi i izmijeniti.

5.1. Maven

Prilikom kreiranja ovog projekta koristimo Maven za upravljanje serverskom stranom aplikacije kao SPM (eng. *Software Project Managment*). Uz Maven za kreiranje Spring projekta može se koristi i Gradle, ali Gradle se više koristi prilikom izgradnje mobilnih aplikacija dok je Maven primarni SPM za izgradnju *web*-aplikacije, REST-API-ja baziranih na Javi i Springu.

Mavenov cilj je dopustiti i pomoći developeru da shvati kompletno stanje izgradnje projekta u najkraćem mogućem vremenskom razdoblju:

- Pojednostavljivanje procesa izgradnje projekta
- Pružanje uniformiranog sustava izgradnje
- Pružanje kvalitetne informacije o projektu
- Izgradnja bolje prakse izgradnje projekta ili developmenta

Prvi korak je inicijaliziranje projekta. Nakon što se struktura projekta inicijalizira, *pom.xml* document sadrži sve odabrane ovisnosti. U ovome slučaju odabran je Maven te je *pom.xml* pisan u XML jeziku dok za odabir Gradleaa kao alata izgradnje ovaj dokument izgleda i zove se drugačije.

Slika 7: Primjer koda- dependency

```
<dependencies>

        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-data-jpa</artifactId>
        </dependency>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-validation</artifactId>
        </dependency>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-web</artifactId>
        </dependency>
        <dependency>
            <groupId>io.jsonwebtoken</groupId>
            <artifactId>jjwt</artifactId>
            <version>0.9.1</version>
        </dependency>
        <dependency>
            <groupId>mysql</groupId>
            <artifactId>mysql-connector-java</artifactId>
            <scope>runtime</scope>
        </dependency>
        <dependency>
            <groupId>org.projectlombok</groupId>
            <artifactId>lombok</artifactId>
            <optional>true</optional>
        </dependency>
        //Ostali dependency

</dependencies>
```

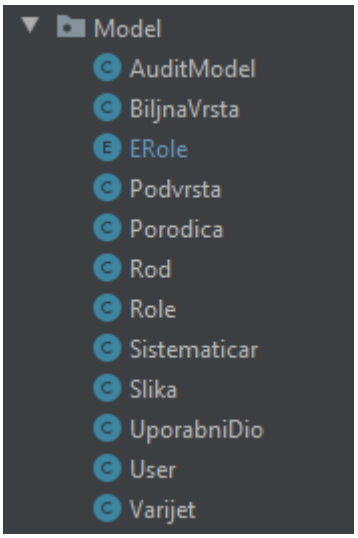
Izvor: Izrada autora

Nakon što je projekt inicijalizrian može se krenuti s njegovom izgradnjom. Pokretanje projekta je jednostavno i vrši se naredbom unutar CLI-a (eng. *Comand Line*) : *mvn spring-boot:run*. Prvi korak je postavljanje postavki samog projekta unutar *application.properties* dokumenta. U postavkama je definirana povezanost s bazom, Hibernate i Spring Data JPA postavke i multipartfile postavke.

5.2. Model

Prvi korak u programiranju Java aplikacije po MVC dizajnu nakon postavljanja projekta je kreiranje modela. Model se sastoji od klasa, a svaka klasa ima određeni broj atributa. Atributi mogu biti tipa: *String*, *Integer*, objekta i ostalih te oni sadrže informacije o aplikaciji. Za potrebe ovog rada modeli se kreiraju prema relacijskome modelu iz kojeg će kasnije automatski, Spring uz pomoć Hibernatea, JPA-a i Lomboka, napraviti migraciju na bazu podataka.

Slika 8: Struktura modela



Izvor: Izrada autora

Osim osnovnog modela koji je karakterističan za Javu, služi i kao migracija na bazu podataka te koristi Hibernate i Spring Data JPA koji su dodani kao ovisnost, te Lombok biblioteka koja je također dodana kao ovisnost. Hibernate i Spring Data JPA su biblioteke koje prije svega mapiraju modele u projektu prema relacijskome modelu. Lombok bibliotekaotklanja pisanje nepotrebnog koda koristeći svoje anotacije.

Za sljedeći primjer kreiranja klase modela se može uzeti primjer modela porodice, odnosno klasu *Porodica* (*Porodica.class*). U prvome je koraku potrebno kreirati klasu, a nakon što je klasa kreirana potrebno ju je popuniti atributima prema relacijskome modelu. Dakle, potrebno je dodati samo atribute, konstruktore, gettere i settere, a kasnije će se model porodice, kao i ostali modeli, nadograđivati. Ovakav nam model neće biti od prevelike koristi prilikom kreiranja *web* servisa i spajanja s bazom koristeći Spring *framework*, stoga on treba biti

nadograđen korištenjem prvenstveno Springa i njegovog načina kreiranja modela (*mapping*, migracije, anotacije) kao i Lombok biblioteka koja olakšava nepotrebno ponavljanje koda.

Slika 9: Klasa Porodica

```
//Importi

//Klasa
public class Porodica extends AuditModel {

    //Varijable
    private Long id;
    private String hrvatskiNazivPorodica;
    private String latinskiNazivPorodica;
    private Boolean deleted = false;

    //Konstruktori
    public Porodica(){
    }
    public Porodica(String hrvatskiNazivPorodica,
                    String latinskiNazivPorodica){
        this.hrvatskiNazivPorodica=hrvatskiNazivPorodica;
        this.latinskiNazivPorodica=latinskiNazivPorodica;
    }

    //Getteri i Setteri
    public Long getId() {
        return id;
    }
    public void setId(Long id) {
        this.id = id;
    }
    //Ostali Getteri i Setteri

}
```

Izvor: Izrada autora

Istaknuti primjer pokazuje da klasa nasljeđuje (eng. *extend*) klasu *AuditModel*. Klasa *AuditModel* sadrži sve ostale atribute koji se konstantno ponavljaju kroz sve tablice u relacijskome modelu. Na takav način ta klasa, kod nadogradnje svake klase, dodaje sve preostale atribute koji se nalaze u toj klasi (koja nasljeđuje), i na taj se način klasi smanjuje nepotrebno pisanje istoga koda.

Ako određena tablica u relacijskome modelu ima vanjski ključ, on se u modelu prezentira kao varijabla tog tipa objekta. Na primjeru modela roda – to bi bila porodica. Za takav tip varijable također se kreiraju getteri i setteri, a ako je u projektu potrebno, kreiraju se i konstruktori koji je sadrže.

Slika 10: Objekt Porodica

```
private Porodica porodica;
```

Izvor: Izrada autora

Nakon što se generira potpuni model sa svim anotacijama, JPA entitetima, Lombok anotacijama i mapiranjima, potrebno ga je migrirati, kako bi se napravila baza podataka. Migraciju se može vrlo jednostavno napraviti naredbom unutar CMD-a (eng. *Comand Prompt*) s pozicijom unutar projekta: *mvn spring-boot:run*.

5.2.1. Spring Data JPA i Anotacije

Ovo poglavlje objedinjuje Hibernate, JPA, Spring Data JPA, definiranje JPA entiteta i bean validatore. Sve su ove definicije vrlo značajne u kreiranju modela, tj. u definiranju samog JPA entiteta ili dodavanja JPA specifikacija modelu. Ovo poglavlje opisuje kako od običnog modela, odnosno entiteta, nastaje model koji preko anotacija definira atribute entiteta onakvima kakvi su u relacijskome modelu.

Java Persistance API ili skraćeno JPA je Java standard za mapiranje Java objekata na relacijsku bazu podataka. Mapiranje Java objekata u bazu podataka, i obratno, naziva se objektno relacijsko mapiranje ili skraćeno ORM (eng. *Object Relational Mapping*) . JPA je jedan od pristupa korištenja ORM-a, a koristeći JPA mapiramo, pohranjujemo, ažuriramo i dohvaćamo podatke iz relacijskih baza podataka u Java objekte i obratno. JPA je specifikacija, a dostupno je više njezinih implementacija; Hibernate i Spring Data JPA.

Hibernate je *framework* koji implementira JPA anotaciju. Hibernate se ponaša kao dodatni sloj iznad JDBC-a (eng. *Java Database Connectivity*) te nam omogućava implementaciju neovisnog sloja baze podataka. ORM implementacija mapira zapise baze

podataka u Java objekte i generira SQL upite kako bi zamijenila sve operacije nad bazom. Hibernate se može postaviti kao komunikatora između određenoga modela dijagrama klase i određene tablice relacijskoga modela. Model i tablica su svojim atributima i njihovim tipovima isti, a Hibernate služi kaoposrednik u komunikaciji.

Spring Data JPA je biblioteka Spring *frameworka* koja dodaje dodatni sloj apstrakcije na vrh JPA pružatelja usluge kao što je Hibernate. Repozitorj Spring Data JPA sastoji se od tri sloja:

1. Spring Data JPA – pruža podršku za kreiranje JPA repozitorija proširivajući Spring Data sučelja (eng. *interface*) repozitorija
2. Spring Data Commons – pruža infrastrukturu koja se djeli s specifičnim Spring Data projektima
3. JPA Provider – implementira JPA.

Definiranje JPA entiteta je postupak dodavanja anotacija nad klase i njihove atribute, kreiranje reprezentacija tablica i atributa u projektu. Svaka instanca entiteta predstavlja jedan zapis u tablici u bazi podataka. bean validacija je standardna validacijska specifikacija koja omogućuje jednostavnu provjeru objekata koristeći set anotacija. Bean validacija implementirana kroz ovisnost koju koristi Hibernate validator i prije svega koristi anotaciju *@NotNull*.

Slika 11: Model BiljnaVrsta

```
@Entity
@Table(name = "biljnaVrsta")
public class BiljnaVrsta extends AuditModel {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    //@Column(name = "idBiljnaVrsta")
    private Long id;

    @NotNull
    @Size(max = 80)
    @Column(unique = true)
    private String hrvatskiNazivVrsta;

    @NotNull
    @Size(max = 80)
    @Column(unique = true)
    private String latinskiNazivVrsta;

    @NotNull
    @Size(max = 80)
    private String sinonimVrsta;

    @NotNull
    @Lob
    private String opisVrsta;

    @NotNull
    @Size(max = 80)
    private String bioaktivnaTvar;
```

Izvor: Izrada autora

U primjeru se može vidjeti klasu *BiljnaVrsta* koja je reprezentacija podataka biljne vrste u bazi podataka. Anotacije predstavljaju opis podataka kao što je to u bazi podataka. Prvi korak je definiranje entiteta kako bi JPA uopće bio svjestan da reprezentacija postoji, a to se postiže kreiranjem anotacije *@Entity* na razini klase. Potrebno je kreirati i osnovni *no-args* konstruktor kao i primarni ključ. Kroz atribut anotacije *@Table* dodali smo naziv tablice kojoj pripada ova klasa *BiljnaVrsta*. Ako atribut *name* nije definiran, naziv entiteta će biti naziv klase. Razne JPA implementacije će probati raditi potklasiranje entiteta kako bi pružili sve funkcionalnosti, te zbog toga entiteti klase ne smiju biti deklariran kao *final*.

Nakon štoje entitet definiran, svaki JPA entitet mora imati primarni ključ koji ga identificira. *@Id* anotacija definira primarni ključ koji može biti kreiran na različite načine pomoću *@GeneratedValue* anotacije. U ovome slučaju odabrana strategija, tj. tip primarnog ključa je *IDENTITY* koji kreira primarni ključ kao obični *integer*. Uz *IDENTITY*, mogu se odabrati još tri strategije: *AUTO, TABLE, SEQUENCE*.

Anotacija *@Column* daje detalje atributu u tablici. *@Column* anotacija može sadržavati više elemenata kao što su *name, lenght, nullable* i *unique*. U primjeru je *@Column* postavljen kao jedinstven (eng. *unique*) na točno (eng. *true*) što znači da može postojati samo jedan zapis s jednom jedinstvenom vrijednošću. Anotacija *@Lob* je skraćenica za veliki objekt (eng. *large object*), te služi za spremanje veliki količina podataka kao što su: slike, video, razni dokumenti, velike količine teksta itd. Takav tip podataka ima dvije variante CLOB (eng. *Character Large Object*, tekst podaci) i BLOB (eng. *Binary Large Object*, binarni podaci).

Anotacija *@Size* određuje maksimalnu i minimalnu veličinu vrijednosti za određeni atribut. U ovome slučaju postavljena je samo maksimalna veličina od 80 znakova. Anotacija *@NotNull* je bean validacija i zabranjuje atributu zapisa u tablici da bude *null*. Anotacija *@NotNull* može imati i atribut poruke (eng. *message*) koja vraća odgovor u slučaju da je vrijednost atributa *null*. Uz Anotaciju *@NotNull*, postoje i druga dva bean validatora, a to su *@NotEmpty* i *@NotBlank*. Njihova je zadaća izrečena u njihovim imenima.

Uz navedene anotacije, postoje i neke druge, koje nisu korištene u projektu, ali mogu biti korisne, primjeri su takvih anotacija: *@Transient, @Temporal, @Enumerated*.

5.2.2. Mapiranje

Kao što smo u prethodnome poglavlju objasnili, Hibernate je najpopularniji objektno relacijski alat za mapiranje (ORM) za Javu. On implementira *Java Persistance* API (JPA) specifikacije, a njegova je primjena masovna. Asocijativno mapiranje je jedno od glavnih funkcionalnosti unutar JPA i Hibernatea, a u projektu je dodano pomoću *dependencya* Spring Data JPA. Ono modelira relacije, odnosno povezanosti između dvije tablice u bazi podataka pomoću atributa u modelu projekta. To omogućava jednostavnu navigaciju između relacija u modelu i JPQL-u (eng. *Jakarta Persistance Query Language*) ili kriterijskim upitima. Mapiranje se vrši na isti način kao i kreiranje relacija u bazi podataka, a u ovome se radu radi upravo po relacijskome modelu korištenjem Spring Data JPA.

JPA i Hibernate pružaju iste asocijacije/relacije koje proizlaze iz relacijskog modela, a to su:

1. Jedan na jedan relacija
2. Više naprama jedan relacija
3. Više naprama više relacija.

Svaka od navedene tri relacije može biti mapirana kao:

1. jednosmjerna (eng. *unidirectional*) ili,
2. dvosmjerna (eng. *bidirectional*) relacija.

Mapiranje nema nikakav utjecaj na postavljanje relacija unutar baze podataka, nego služi za određivanje smjera korištenja relacije unutar modela projekta i kriterijskih upita. Prilikom programiranja neki koriste jednosmjerno, a neki dvosmjerno mapiranje. Radom na ovome projektu najviše se koristi jednosmjerno mapiranje, no koristi se i dvosmjerno mapiranje, ovisno o slučaju korištenja. Oba mapiranja su korisna, ovisno o tome što želimo postići. Jednosmjerno mapiranje ili mapiranje u jednome smjeru je jednostavnije za razumjeti i može se objasniti kao relaciju *više naprama jedan*. Jedan rod pripada jednoj porodici, dok jednoj porodici može pripadati više rodova. Samim time na primjeru u obliku JSON-a to znači da određeni rod ima svoj zapis kao i zapis od porodice kojoj pripada.

Slika 12: JSON rod


```
{
    "createdAt": "2020-08-02T18:17:45.000+0000",
    "updatedAt": "2020-08-02T18:17:45.000+0000",
    "createdBy": "Toni",
    "editedBy": "Toni",
    "role": "a",
    "id": 63,
    "hrvatskiNazivRod": "Rod",
    "latinskiNazivRod": "Rod",
    "porodica": {
        "createdAt": "2020-07-09T19:03:12.000+0000",
        "updatedAt": "2020-08-13T11:57:26.000+0000",
        "createdBy": "Toni",
        "editedBy": "Toni",
        "role": "Admin",
        "id": 64,
        "hrvatskiNazivPorodica": "Porodica",
        "latinskiNazivPorodica": "Porodica",
        "deleted": null
    }
}
```


Izvor: Izrada autora

Ako nam dvosmjerno mapiranje nije potrebno, najbolje je u početku koristiti jednosmjerno mapiranje prilikom mapiranja relacija jer je jednostavnije. Dvosmjerno mapiranje koristi se kako bi se sačuvala kolekcija dječjih (eng. *child*) entiteta unutar roditelja (eng. *parent*) i dopustilo vraćanje i čuvanje dječjih entiteta unutar roditelja entiteta. Rod se primjenjuje na relaciju između slike i biljne vrste kao i biljne vrste i uporabnoga dijela. U tom slučaju, biljna vrsta sadrži relacije svih slika koje joj pripadaju. Mapiranje se kreira pomoću anotacija koje opisuju stanje atributa.

Slika 13: JSON Porodica

```
{
    "id": 64,
    "hrvatskiNazivPorodica": "Porodica",
    "latinskiNazivPorodica": "Porodica",
    //Ostali podaci
    "rod": [
        {
            "id": 63,
            "hrvatskiNazivRod": "Rod2",
            "latinskiNazivRod": "Rod2",
            //Ostali podaci
        },
        {
            "id": 19,
            "hrvatskiNazivRod": "Rod1",
            "latinskiNazivRod": "Rod1e",
            //Ostali podaci
        }
    ]
}
```

Izvor: Izrada autora

5.2.2.1. *Više naprama jedan*

Praktični dio ovog rada sadrži najviše jednosmjernih – *više naprama jedan* mapiranja. Na primjeru roda vidimo jednosmjerno *više naprama jedan* mapiranje. Tablica roda u relacijskome modelu ima relaciju s porodicom te sadrži vanjski ključ porodice. Takvoj relaciji potrebno je dodati atribut klase koja predstavlja vanjski ključ te sve potrebne anotacije koje opisuju *više naprama jedan* mapiranje.

Slika 14: Više naprama jedan mapiranje

```
    @ManyToOne(fetch = FetchType.EAGER, optional = false)
    @JoinColumn(name = "porodica_id", nullable = false)
    @OnDelete(action = OnDeleteAction.CASCADE)
    private Porodica porodica;
```

Izvor: Izrada autora

Gornji primjer je primjer jednosmjernog *više naprama jedan* mapiranja koji sadrži tri anotacije, i koji je glavni način mapiranja. Uz ove anotacije prilikom mapiranja *više naprama jedan*, ali i prilikom bilo kojeg tipa mapiranja, mogu se koristiti i druge anotacije, koje uz anotaciju mapiranja, opisuju stanje atributa.

@ManyToOne anotacija je osnovna anotacija koja se koristi prilikom *više naprama jedan* mapiranja te ona deklarira da rod ima *više naprama jednu* relaciju s porodicom, a sadrži *fetch* parametar i *optional* parametar u ovome slučaju. *FetchType* (hrv. tip dohvata) može biti tipa *EAGER* ili *LAZY*. *FetchType* predstavlja način na koji se dohvaćaju relacije, tj. dohvaćaju li se one, ili ne. Ako je *FetchType* postavljen na *LAZY*, relacija se neće dohvaćati. Na ovome primjeru to bi značilo da prilikom dohvaćanja roda, zapis pojedinog roda u sebi neće sadržavati atribut relacije porodice. U suprotnome, ako je *FetchType* postavljen kao *EAGER* sa zapisom će biti dohvaćena i njegova relacija. U ovome primjeru, uz zapis roda, bit će dohvaćena i njegova relacija porodica (slika). Ako se *FetchType* ne definira, bit će postavljen po *defaultu* na *LAZY*. Parametar *optional* govori nam je li obavezno unijeti relaciju prilikom unosa zapisa, odnosno može li se unijeti određeni zapis roda, a da mu se ne pridoda porodica? Ako je *optional* parametar postavljen na *false*, odgovor je ne, a ako je postavljen na *true*, odgovor je da.

@JoinColumn anotacija deklarira vanjski ključ u tablici. U ovome slučaju, vanjski ključ porodice u tablici roda bit će naziva *porodica_id* te mu vrijednost neće moći biti *NULL*, tj. uvijek će prilikom kreiranja zapisa stupac *porodica_id* morati biti popunjen. Ako je vrijednost parametra *nullable = true*, tada vrijednost stupca *porodica_id* može biti *NULL*.

@OnDelete anotacija predstavlja akciju koja se želi izvršiti nad vanjskim ključem ako se želi obrisati zapis iz baze podataka. Akcija *OnDeleteAction* može biti postavljena kao *CASCADE* ili *NO_ACTION*. Ako je postavljena kao *CASCADE*, poduzimaju se *cascade* mogućnosti prilikom brisanja, a ako je postavljena kao *NO_ACTION* onda se ne poduzimaju nikakve akcije.

Slika 15: Više naprama jedan mapiranje 2

```
//Porodica
@OneToMany(mappedBy = "porodica",
           fetch = FetchType.LAZY,
           cascade = CascadeType.ALL,
           orphanRemoval = true)
@JsonIgnore
private List<Rod> rod;

//Rod
@ManyToOne(fetch = FetchType.LAZY, optional = false)
@JoinColumn(name = "porodica_id", nullable = false)
@OnDelete(action = OnDeleteAction.CASCADE)
private Porodica porodica;
```

Izvor: Izrada autora

Uz jednosmjerno, tu je i dvosmjerno *više naprama jedan* mapiranje. Iz primjera se vidi da se mapiranje sastoji od dva dijela:

1. Strana asocijacije naprama više (rod) sastoji se, tj. vlasnik je mapiranja relacije.
2. Strana asocijacije naprama jedan (porodica) je samo poveznica na mapiranje.

U gornjem primjeru roda, dvosmjerno mapiranje je identično onome kao i kod jednosmjernog mapiranja. Kako bi dvosmjerno mapiranje bilo potpuno, potrebno je samo kreirati poveznicu prema vlasniku/rodu. To se kreira unutar *@OneToMany* anotacije, dodavajući atributu *mappedBy* ime atributa mapirane relacije koji je kreiran unutar klase roda, a to je atribut *porodica*.

Uz jednosmjerno i dvosmjerno *više naprama jedan* mapiranje, postoji i *jedan naprama više* mapiranje. Primjer i funkcionalnost njegova koda izgleda jednako kao i strana asocijacije *naprama jedan* unutar dvosmjernog *više naprama jedan* primjera koda.

5.2.2.2. Jedan na jedan

Jedan na jedan relacije su rijetko korištene u odnosu na *više naprama jedan* relacije. Zbog toga i *jedan na jedan* mapiranje nije previše zastupljeno.

Slika 16: Jedan na jedan mapiranje

```
//Sistematičar
@OneToOne(fetch = FetchType.LAZY, optional = false)
@JoinColumn(name = "biljnaVrsta_id", nullable = false)
private BiljnaVrsta biljnaVrsta;
```

Izvor: Izrada autora

Gornji primjer je primjer jednosmjernog *jedan na jedan* mapiranja. Iz njega je vidljivo da je *jedan na jedan* mapiranje vrlo slično *više naprama jedan* mapiranju. Jedina razlika je u anotaciji; kod *jedan na jedan* mapiranja ta anotacija je *@OneToOne* te ona samim time ima i drugačiju funkcionalnost kao i relacija u relacijskome modelu. Kod jednosmjernog *jedan na jedan* mapiranja navigacija se vrši u jednome smjeru, npr. od sistematičara prema biljnoj vrsti, dok se kod dvosmjernog mapiranja navigacija vrši u oba smjera te samim time na oba modela mora biti *@OneToOne* anotacija za mapiranje.

5.2.2.3. Više naprama više

Više naprama više relacija se implementira korištenjem treće tablice koja se zove agregacija. U relacijskome modelu imamo dvije agregacije te su stoga kreirana i dva dvosmjerna *više naprama više* mapiranja. Prilikom kreiranja *više naprama više* mapiranja nije potrebno kreirati agregaciju kao model u projektu.

Slika 17: Više naprama više mapiranje

```
//Biljna Vrsta
@ManyToMany(fetch = FetchType.EAGER)
@JoinTable(name = "uporabniDioVrste",
           joinColumns = { @JoinColumn(name = "biljnaVrsta_id") },
           inverseJoinColumns = { @JoinColumn(name = "uporabniDio_id") })
private List <UporabniDio> uporabniDio;

// Uporabni Dio
@ManyToMany(fetch = FetchType.LAZY,
            mappedBy = "uporabniDio")
@JsonIgnore
private List <BiljnaVrsta> biljnaVrsta;
```

Izvor: Izrada autora

Kao primjer agregacije može se uzeti biljnu vrstu i uporabni dio. Biljna vrsta može imati više uporabnih dijelova dok uporabni dio može pripadati više biljnim vrstama. Za mapiranje jednosmjernog *više naprama više* mapiranja potrebna nam je anotacija *@ManyToMany* te nam omogućuje navigiranje u jednome smjeru. Primjerice, pregled uporabnih dijelova biljne vrste, ali ne i obratno.

Dvosmjerno *više naprama više* mapiranje omogućuje nam navigiranje u oba smjera. Npr.: može se pregledati uporabne dijelove biljne vrste, ali može se i pregledati sve biljne vrste određenog uporabnog dijela. Također, postupak mapiranja prati isti koncept kao i kod dvosmjernog *više naprama jedan* mapiranja. Možemo reći da je *više naprama jedan* mapiranje okosnica svih mapiranja i kada se jedno mapiranje razumije, razumiju se sva. U ovome slučaju, jedan od dva entiteta je vlasnik asocijacije i pruža informaciju o mapiranju. Drugi entitet se samo povezuje na asocijacijsko mapiranje kako bi Hibernate znao otkuda povući potrebnu informaciju.

U ovome primjeru mapiranje na strani biljne vrste koristi tablicu *uporabni_dio_vrste* s ključem *uporabniDio_id* kao vanjskim ključem tablice i i ključem „biljnaVrsta_Id“ tablice *biljna_vrsta*. Sve što se treba sljedeće učiniti je povezati atribut koji je vlasnik asocijacije. Atribut *uporabni_dio* entiteta *biljnaVrsta* vlasnik je relacije to jest asocijacije, te je potrebno samo proslijediti string *uporabniDio* atributu *mappedBy* koji se nalazi u anotaciji *@ManyToMany*.

5.2.3. Lombok

Lombok je biblioteka koji u ovome diplomskome radu ima značajno mjesto u kreiranju modela serverske strane aplikacije. U nastavku će biti objašnjen projekt Lombok, a njegova će primjena biti objašnjena kroz kod modela.

Jedna od osnovnih mana Java programskog jezika ali i ostalih jezika je učestalo pisanje uobičajenog ponavljajućeg dijela koda kao što su getter i setter ili osnovni konstruktori. Tu do izražaja dolazi Lombok koji na prvo mjesto stavlja produktivnost prilikom pisanja koda.

Lombok radi na način da se kao ovisnost ubaci u projekt te samogenerira *bytecode* u *.class* dokument ovisno o broju anotacija projekta koje se dodaju u kod. Za korištenje Lomboka, potrebno ga je dodati kao ovisnost u projekt.

Slika 18: Lombok

```
package com.example.BiljkaRestApi.Model;

import lombok.*;

@Entity
@Table(name = "biljnaVrsta")
@Data
@Getter
@Setter
@ToString
@AllArgsConstructor
@NoArgsConstructor
@EqualsAndHashCode
public class BiljnaVrsta extends AuditModel {
    //Body klase
}
```

Izvor: Izrada autora

Anotacije koje pripadaju Lomboku i koje su korištene su:

1. *@Data* osnovna anotacija za jednostavni POJO (eng. *Plain Old Java object*) i beanove;
2. *@Getter* generira gettere za sve etitete klase.
3. *@Setter* generira settere za sve entitete klase.
4. *@ToString* generira *toString* metodu.
5. *@AllArgsConstructor* generira konstruktor sa svim argumentima klase.
6. *@NoArgsConstructor* generira prazni konstruktor klase.
7. *@EqualsAndHashCode* generira implementaciju *equals (Object other)* i *hashCode()* metoda.
8. *@RequiredArgsConstructor* kreira konstruktor s jednim poljem za svaki argument klase.

Anotacija *@Data* sastoji se od anotacija *@ToString*, *@EqualsAndHashCode*, *@Getter*, *@Setter* i *@RequiredArgsConstructor*. Anotacija *@Data* kreira osnovu za jednostavnije

pisanje koda. Ako se generira anotacija *@Data*, tada nije potrebno generirati anotacije koje anotacija *@Data* sadrži.

5.3. Repozitorij

Spring Data Commons je dio Spring Data modula koji pruža infrastrukturu dijeljenja unutar Spring projekta. On sadrži tehnologiju neutralnih sučelja repozitorija kao i *metadata* model za čuvanje Java klasa. Spring Data Commons pruža sljedeća sučelja:

1. Repozitorij sučelje
2. *CrudRepository* sučelje
3. *PagingAndSortingRepository* sučelje
4. *QueryDslPredicateExecutor* sučelje.

Uz ta sučelja, Spring Data JPA modul pruža dodatna sučelja za pristup bazi podataka:

1. *JpaRepozitory* sučelje
2. *JpaSpecificationExecutor* sučelje.

Repozitorij sučelje je osnovno sučelje koje ima svrhu:

1. Upravljanja nad entitetom.
2. Pomaže Spring spremniku da otkrije "konkretna" sučelja tijekom *classpath* skeniranja.

CrudRepository sučelje pruža CRUD (eng. *Create Read Update Delete*) operacije za upravljajući entitet. *CrudRepository* sučelje pruža sljedeće metode/API-je:

1. *Long count*() – vraća broj dostupnih entiteta.
2. *Void delete*(*Tentity*) – briše odabrani entitet.
3. *Void deleteAll*() – briše sve entitete unutar tog repozitorija.
4. *Void deleteAll(Iterable<? extends T> entities)* – briše dane entitete.
5. *Void deleteById(ID id)* – briše entitet s danim ID-om.
6. *Boolean existsById(ID id)* – vraća vrijednost postojanja entiteta s odabranim ID-om.
7. *Iterable findAll()* – vraća sve instance tipa.
8. *Iterable findAllById(Iterable ids)* – vraća sve instance tipa s danim ID-om.

9. *Optional findById(ID id)* – vraća entitet preko danog ID-a.
10. *Save(S entity)* – sprema dani entitet.
11. *Iterable saveAll(Iterable entities)* – sprema sve dane entitete.

PaginationAndSortingRepository sučelje je ekstenzija *CrudRepository* sučelja koje pruža dodatne metode za vraćanje entiteta koristeći paginaciju i apstrakciju sortiranja. *QueryDslPredicateExecutor* sučelje deklarira metode koje se koriste za vraćanje entiteta iz baze podataka koristeći *QueryDsl* predefinirane objekte. *JpaSpecificationExecutor* deklarira metode koje su korištenje za vraćanje entiteta iz baze podataka koristeći specifikacijske objekte koji koriste JPA kriterijski API. *JpaRepository* sučelje kombinira, koristi i nasljeđuje, odnosno nadograđuje se na metode iz *Jparepository* i *PagingAndSortingRepository* sučelja.

Slika 19: Biljna Vrsta Repozitorij

```
public interface BiljnaVrstaRepository
                      extends JpaRepository<BiljnaVrsta, Long> {

    Page<BiljnaVrsta> findByRodId(
                            Long rodId,
                            Pageable pageable);
    Optional<BiljnaVrsta> findByIdAndRodId(
                            Long id,
                            Long rodId);
    Page<BiljnaVrsta> findByHrvatskiNazivVrsta(
                            String hrvatskiNazivVrsta,
                            Pageable pageable);
    Page<BiljnaVrsta> findByLatinskiNazivVrstaContaining(
                            String latinskiNazivVrsta,
                            Pageable pageable);
    Page<BiljnaVrsta> findByUporabniDioUporabniDio(
                            String uporabniDio,
                            Pageable pageable);

}
```

Izvor: Izrada autora

U primjeru se vide razni načini dohvata podataka koristeći *JpaRepository*. Svaka *findBy* metoda dohvaća entitete po danome ID-u u ostatku naziva metode. Ako je u nazivu metode, i *containing* metoda će vraćati entitete, iako nije pronađen puni ID, već polovični. Prilikom kreiranja objekta *Pageable* unutar parametara metode, metoda će vraćati zapis s označenim stranicama.

Koraci prilikom kreiranja JPA sučelja su:

1. Kreiranje sučelja repozitorija koje nasljeđuje *JpaRepository* sučelje.
2. Dodavanje prilagođenih metoda upita.

Ako dane metode nisu dovoljne, pomoću anotacije *@Query* koja se kreira nad prilagođenom metodom, i koja kao parametar prosljeđuje SQL upit, može se vraćati u toj metodi rezultate prilagođene pomoću SQL upita. Metode *JpaRepository* sučelja bile su dovoljne za ovaj diplomski rad te nije bilo potrebno kreirati prilagođene upite koristeći anotaciju *@Query*.

5.4. Controller

U ovome poglavlju bit će objašnjene REST API krajnje točke kroz CRUD primjer. Uz REST API kranjnje točke bit će objašnjena i implementacija paginacije, pretraživanja, filtriranja unutar REST API krajnjih točki, *multipart* i JWT autorizacija.

Nakon što je implementiran model u praktičnome dijelu rada, te je za svaki model kreiran JPA repozitorij, potrebno je kreirati i kontroler, tj. osnovu za REST API krajnje točke koje će obrađivati HTTP CRUD upite.Na primjeru porodice to su:

1. *POST - /api/porodica* – kreira novu porodicu.
2. *GET - /api/porodica* – dohvaća sve porodice,
3. *PUT - /api/porodica/id* – ažurira porodicu po ID-u.
4. *DELETE - /api/porodica/id* – briše odabranu porodicu.
5. *DELETE /api/porodica* – briše sve porodice.

Slika 20: Get ALL Porodica

```
@RestController
@CrossOrigin(origins = "*", allowedHeaders = "*")
@RequestMapping("/api")
public class PorodicaControll {

    @Autowired
    PorodicaRepository porodicaRepository;

    // Get All Porodica
    @GetMapping("/porodica")
    public Page<Porodica> getAllPorodica(Pageable pageable,
                            @RequestParam(defaultValue = "0") int page,
                            @RequestParam(defaultValue = "10") int size) {
    return porodicaRepository.findAll(pageable);
}
```

Izvor: Izrada autora

U gornjem primjeru može se vidjeti kreiranje *PorodicaControllera* kao i metodu za dohvat svih porodica. Anotacija *@CrossOrigin* konfigurira dozvoljene*krajnje točke za podjelu resursa*. Anotacija *@RestController* se koristi za definiranje kontrolera kako bi vraćena vrijednost metoda bila povezana s tijelom *web* odgovora. Anotacija *@RequestMapping* deklarira da svi URL-ovi API-ja u kontroleru počinju s parametrom anotacije. Anotacija *@Autowired* se koristi kako bi se unio *PorodicaRepository* bean u lokalnu varijablu.

Anotacija *@GetMapping* definira tip metode te ga dodjeljuje URL metodi, tj. krajnjoj točki koju ta metoda predstavlja. Uz Anotaciju *@GetMapping* postoje i anotacije *@PostMapping* koja definira *POST* upit, *@PutMapping* koja definira *PUT* upit, *@DeleteMapping* koja definira *DELETE* upit i ostale anotacije.

Metoda *getAllPorodica* kao parametar metode sadrži paginaciju, te predefinirane parametre upita koji daju vrijednost atributima paginacije. Tijelo metode poziva JPA metodu *findAll* koja dohvaća sve vrijednosti porodice.

Slika 21: GET Porodica with id

```
// Get Porodica with id
@GetMapping("/porodica/{id}")
public ResponseEntity<Porodica> getPorodicaById(@PathVariable("id") long id) {
    Optional<Porodica> porodicaData = porodicaRepository.findById(id);

    if (porodicaData.isPresent()) {
        return new ResponseEntity<>(porodicaData.get(), HttpStatus.OK);
    } else {
        return new ResponseEntity<>(HttpStatus.NOT_FOUND);
    }
}
```

Izvor: Izrada autora

Ako unutar URL-a želimo proslijediti varijablu, potrebno je prilikom definiranja krajnje točke, unutar anotacije zadužene za definiranje krajnje točke, dodati parametar određenog naziva u vitičastim zagradama kao što je to prikazano na gornjoj slici. U parametrima metode je također potrebno pomoću anotacije *@PathVariable* proslijediti tu varijablu. Gornji primjer u tijelu metode pretražuje bazu podataka pomoću danog ID-a te ako je zapis porodice s danim ID-om nađen, vraća taj zapis, a ako nije pronađen, vraća definirani odgovor greške.

Slika 22: POST Porodica

```
// Create a new Porodica
@PostMapping("/porodica")
public Porodica createPorodica(@Valid @RequestBody Porodica porodica) {
    return porodicaRepository.save(porodica);
}
```

Izvor: Izrada autora

Za kreiranje krajnje točke koja sprema zapis koristi se anotacija *@PostMapping*. Metoda *createPorodica* koja kao parametar ima objekt porodice prosljeđuje taj parametar JPA metodi koja se poziva pomoću repozitorija porodice. Anotacija *@RequestBody* definira parametar porodica kao tijelo upita, dok anotacija *@Valid* provjerava samu validaciju tijela upita.

Slika 23: PUT Porodica with id

```
// Edit porodica with specific id
@PutMapping("/porodica/{porodicaId}")
public Porodica updatePorodica(@PathVariable Long porodicaId,
                               @Valid @RequestBody Porodica porodicaRequest) {
    return porodicaRepository.findById(porodicaId).map(porodica -> {
        porodica.setHrvatskiNazivPorodica(porodicaRequest.getHrvatskiNazivPoro
dica());
        porodica.setLatinskiNazivPorodica(porodicaRequest.getLatinskiNazivPoro
dica());
        porodica.setEditedBy(porodicaRequest.getEditedBy());
        porodica.setRole(porodicaRequest.getRole());
        return porodicaRepository.save(porodica);
    }).orElseThrow(() -
> new ResourceNotFoundException("idPorodica " + porodicaId + " not found"));
}
```

Izvor: Izrada autora

Za kreiranje krajnje točke koja uređuje odabranu porodicu, potrebno je koristiti anotaciju *@PutMapping*. Metoda *updatePorodica* kao parametre ima ID odabrane porodice za uređivanje, i tijelo upita porodice. Proces je vrlo jednostavan te se u prvom koraku pretražuje zapis s odabranim ID-om, dok je drugi korak postavljanje novih vrijednosti iz tijela upita. Nakon što su nove vrijednosti postavljene, zapis porodice se sprema.

Slika 24: DELETE Porodica with id

```
// Delete porodica with specific id
@DeleteMapping("/porodica/{porodicaId}")
public ResponseEntity<?> deletePorodica(@PathVariable Long porodicaId) {
    return porodicaRepository.findById(porodicaId).map(porodica -> {
        porodicaRepository.delete(porodica);
        return ResponseEntity.ok().build();
    }).orElseThrow(() -
> new ResourceNotFoundException("idPorodica " + porodicaId + " not found"));
}
```

Izvor: Izrada autora

Krajnja točka za brisanje porodice, koji nije privremeno brisanje (eng. *soft delete*), potrebno je koristiti anotaciju *@DeleteMapping*. Metoda *deletePorodica* pretražuje porodice po danom ID-u iz upita te briše odabranu porodicu ako je ima. Ako koristimo privremeno brisanje, potrebno je koristiti metodu *PUT* koja mijenja vrijednost u stupcu zapisa *deleted*.

Slika 25: DELETE ALL Porodica

```
// Delete ALL porodica
@DeleteMapping("/porodica")
public ResponseEntity<HttpStatus> deleteAllPorodica() {
    try {
        porodicaRepository.deleteAll();
        return new ResponseEntity<>(HttpStatus.NO_CONTENT);
    } catch (Exception e) {
        return new ResponseEntity<>(HttpStatus.EXPECTATION_FAILED);
    }
}
```

Izvor: Izrada autora

Ukoliko želimo obrisati sve zapise porodice, utoliko je dovoljno pozvati metodu koja u tijelu poziva pomoću repozitorija porodice JPA metodu *deleteAll*. Jedna od najvažnijih stvari je kreiranje *web*-aplikacije koja je *user-frendly*. Pod tim se podrazumijeva vremenski interval odgovora upita, stoga paginacija tu dolazi do izražaja. Paginacija dohvaća određeni broj zapisa po svakoj stranici i sortira zapis.

Slika 26: Krajnja točka

```
http://localhost:8080/api/porodica?search=Porodica2&page=0&size=2&sort=created
At,desc
```

Izvor: Izrada autora

Uz paginaciju važnu ulogu ima i pretraživanje. Pretraživanje se može promatrati kao pregled zapisa po ID-u. Jedina je razlika što se ID u URL-u nalazi kao varijabla putanje, dok pretraživanje kao ključ mora biti definirano u kontekstu parametra upita te sadrži pripadajuću vrijednost. To se postiže anotacijom *@RequestParam* koja se nalazi kao parametar metode koja obrađuje krajnju točku.

Prilikom prosljeđivanja slika, potrebno je kao parametar upita koristiti objekt tipa *MultipartFile*. Sva ostala logika je ista kao i kod slanja običnog parametra upita. *MultipartFile* je Spring *web* sučelje koje obrađuje HTTP *multi-part* upite. Diplomski rad pruža API-je za:

1. Upload slika u MySql bazu podataka.
2. Preuzimanje slike pomoću linka.
3. Dohvat liste slika.

Slika je pohranjena u bazu podataka kao BLOB tip (eng. *Binary Large Object* ili Objekt Binarnog tipa). Uz gornje kontrolere imamo i kontroler za *soft delete*. Prilikom korištenja *soft deletea* nad određenim zapisom, mijenja mu se stanje atributa *deleted* u bazi podataka, te se prilikom dohvata svih zapisa, dohvaćaju samo oni koji imaju istu vrijednost atributa *deleted*, dok se za ostale smatra da su obrisani.

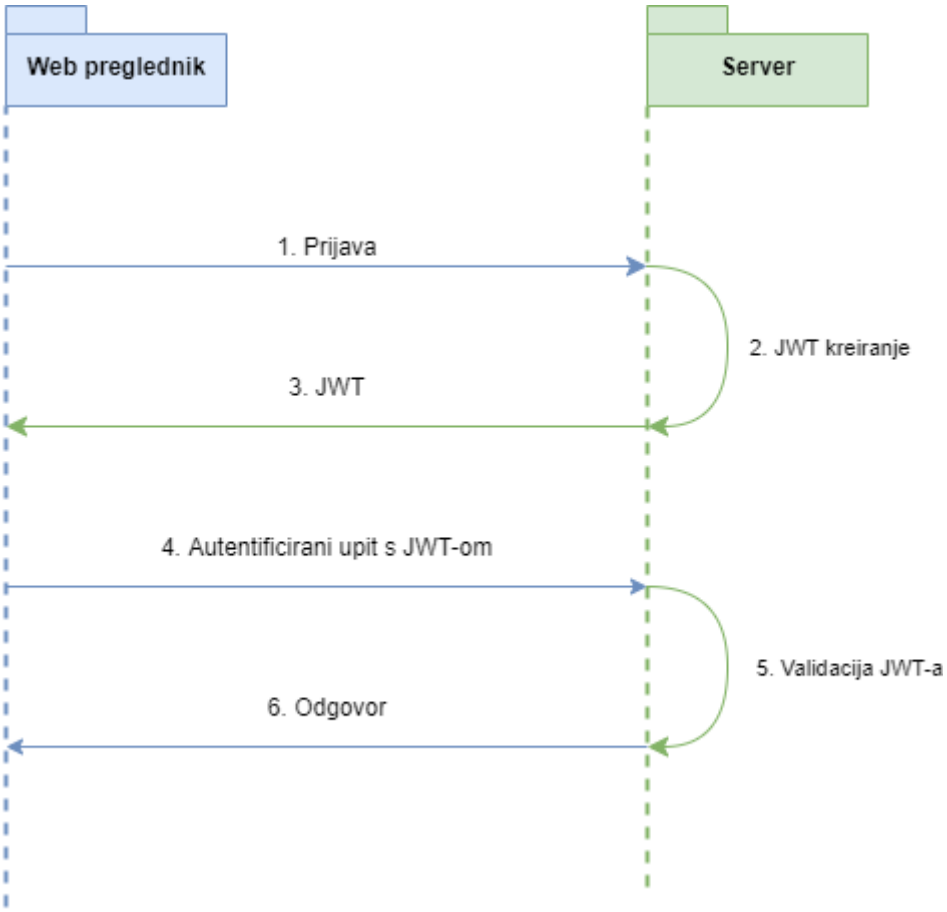
5.5. Autentifikacija i autorizacija serverske strane

Autentifikacija je jedan od najvažnijih dijelova svake aplikacije, bilo da se radi o *web*-aplikaciji, desktop aplikaciji ili mobilnoj aplikaciji. Ovaj rad koristi JWT (eng. *JSON Web Token*) token baziranu autentifikaciju. U većini aplikacija, pa tako i u aplikaciji ovog rada, potrebno je kreirati račun te se prijaviti unutar aplikacije za korištenje dodatnih usluga. Takva se usluga zove autentifikacija i autorizacija.

Autentifikacija bazirana na sesiji sastoji se od početne prijave u web-aplikaciju, prilikom koje server generira sesiju za tog korisnika i pohranjuje ju. Pohranjivanje se može obraditi u memoriji ili bazi podataka. Server vraća i ID sesije za klijenta koji se sprema u *browseru* kao kolačić (eng. *cookie*). Sesija na serverskoj strani ima rok trajanja, a nakon što istekne, korisnik se mora opet prijaviti u *web*-aplikaciju kako bi kreirao novu sesiju. Ako se korisnik ulogirao, a sesija nije još istekla, kolačić uvijek odlazi s HTTP upitom do servera. Server uspoređuje taj ID sesije s pohranjenom sesijom kako bi autentificirao i vratio odgovor.

Autentifikacija bazirana na sesiji je sasvim dovoljna za *web*-aplikacije, ali proširivanjem aplikacije s primjerice, mobilnom aplikacijom koja ne koristi kolačiće, potrebno je koristiti autentifikaciju baziranu na tokenu. U tom slučaju imamo jednu serversku stranu aplikacije koja može komunicirati s više različitih tipova aplikacija, tj. klijentskih dijelova aplikacija koje krajnji korisnik koristi. Pomoću te metode korisnikovo stanje prijave je enkodirano u JSON *Web* Token (JWT) od strane servera i poslano klijentu. Danas većina REST servisa koristi takav način autentifikacije.

Slika 27 JWT Autentifikacja



Izvor: Izrada autora

Iz primjera se može vidjeti da je JWT također jednostavan za korištenje. Umjesto kreiranja sesije, server generira JWT iz podataka prijave korisnika te ga šalje nazad klijentu. Klijent sprema JWT, a prilikom svakog HTTP upita kojeg klijent šalje serveru, šalje se i JWT. Nadalje, server validira JWT i šalje odgovor klijentu.

Važni dijelovi JWT-a su zaglavlje (eng. *header*), *payload* i potpis. Unutar zaglavlja, definira se tip autentifikacije i šalje se JWT token serveru. U *payloadu* se šalju podaci koji se pohranjuju uz JWT. U većini slučajeva to može biti korisničko ime i lozinka. Potpis je dio JWT-a u kojem se koristi *hash* algoritam. Enkodiraju se zaglavlje i *payload* te se pridružuju jedan drugome. Nakon toga kreira se *hash* podataka, koristeći *hash* algoritam koji smo definirali u *headeru* s tajnim ključem. U zadnjem koraku enkodira se rezultat *hashinga* kako bi se dobio potpis. Nakon što smo skupili sve dijelove JWT-a, kombiniramo ih u JWT strukturu *header.payload.signature.*

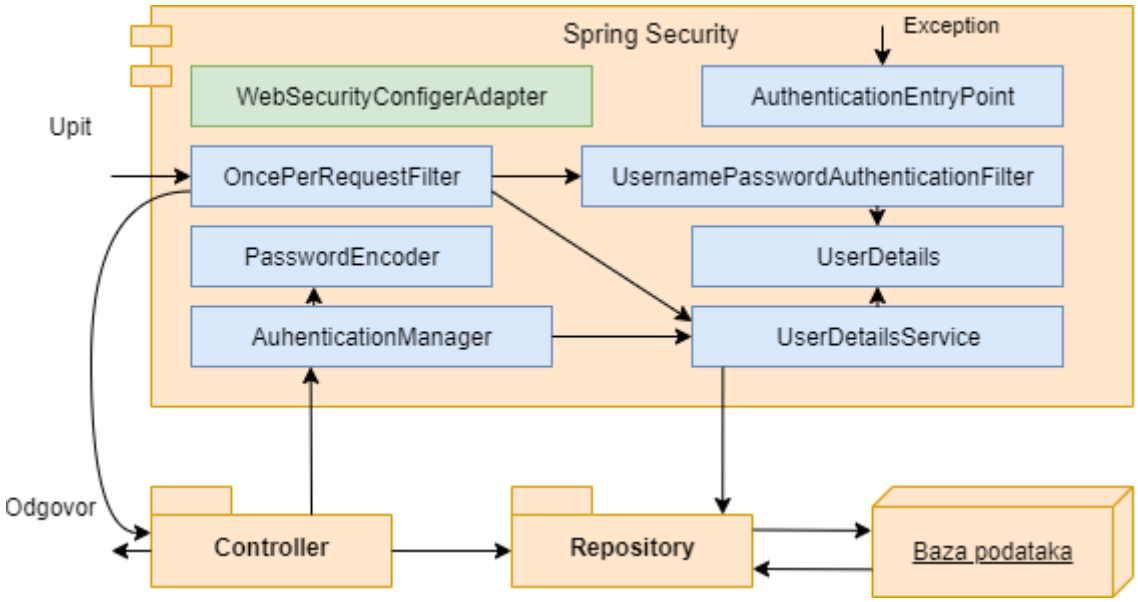
JWT ne skriva, i osigurava, podatke koje šaljemo. Može se vidjeti da proces generiranja JWT-a samo enkodira i *hashira* podatke, ne enkriptira ih. Svrha JWT-a je autentificirati podatke s korisnikom. Zbog toga je uvijek potrebno unutar aplikacije imati HTTPS (eng. *HyperText Transfer Protocol Secure*) enkripciju. Nakon što server primi JWT od klijenta, provjerava je li potpis pravilno *hashiran*. Ako se poklapa s tajnim ključem, tada je JWT ispravan i validiran. Svaka aplikacija ima tajni ključ koji mora biti sigurno pohranjen na serverskoj strani.

Implementirani JWT sadržava i podržava:

1. Registraciju i prijavu – JWT
2. Korištenje Spring Securitya
3. Konfiguriranje Spring Securitya s JWT-om
4. Definiranje modela za autorizaciju i autentifikaciju
5. Interakcija Spring Data JPA za pohranjivanje podataka autorizacije i autentifikacije.

Kao što je već objašnjeno u prethodnim poglavljima, postoje četiri tipa korisnika. Za tri tipa se radi autentifikacija i autorizacija, a jedan je tip korisnika javni i ima pristup određenim podacima bez autentifikacije i autorizacije.

Slika 28: Spring Security Arhitektura



Izvor: Izrada autora

Spring serverska arhitektura je vrlo jednostavna sa Spring Security modulom nakon što se pravilno implementira i objasne svi njezini elementi:

1. *WebSecurityConfigurerAdapter* – glavni dio. Pruža *HttpSecurity* za konfiguriranje CORS-a (eng. *Cross-Origin Resource Sharing*), CSRF-a (eng. *Cross-Origin Resource Sharing*) , upravljanja sesijom, pravila za sigurnost resursa.
2. *UserDetailsService* – sučelje za dohvaćanje korisnika po korisničkome imenu i vraćanje njegovih detalja za autentifikaciju i validaciju.
3. *UserDetails* – informacije o korisniku za izgradnju objekta autentifikacije.
4. *UsernamePasswordAuthenticationToken* – dohvaća korisničko ime i lozinku iz upita prijave u sustav.
5. *AuthenticationManager* – validira *UsernamePasswordAuthenticationToken* objekt.
6. *OncePerRequestFilter* – izvršava svaki upit za naš API posebno. Pruža metodu za parsiranje i validiranje JWT-a, učitavanje detalja korisnika, provjeru autorizacije.
7. *AuthenticationEntryPoint* – hvata neautorizirane greške i vraća kod greške 401 ako klijent pristupa neautoriziranim podacima.
8. *Controlleri* primaju i obrađuju upite nakon što su filtrirani kroz *OncePerRequestFilter*.
9. Repozitoriji – komuniciraju s bazom podataka.

Gornja struktura je implementirana na isti način i u projektu, te je uz osnovne dijelove serverske strane paketa (*Model*, *Controller*, *Repository*) može se dodati i u sljedeće pakete koji dopunjavaju Spring REST API sa Spring Security infrastrukturom:

1. Security
2. Payload.

Unutar *Security* paketa *WebSecurityConfig* proširuje *WebSecurityConfigurerAdapter*, *UserDetaildsServiceImpl* klasa implementira *UserDetailsService*, *UserDetailsImpl* klasa implementira *UserDetails*, *AuthEntryPointJwt* klasa implementira *AuthenticationEntryPoint*, *AuthTokenFilter* klasa proširuje *OncePerRequestFilter* i *JwtUtils* pruža metode za generiranje, parsiranje i validaciju JWT-a.

Slika 29: JWT Tajni Ključ

```
# App Properties
BiljkaRestApi.app.jwtSecret = BiljkaRestApiSecretKey
BiljkaRestApi.app.jwtExpirationMs = 20000000
```

Izvor: Izrada autora

Payload definira klase za objekte upita i odgovora. Uz gore postavljenu infrastrukturu, potrebno je u application.properties unijeti JWT tajni ključ kao što smo unijeli i postavke baze podataka i Spring Data JPA-a. Za ovo poglavlje potrebni su: Spring Boot Starter Security i io.json*Web*token *dependency*.

U nastavku će biti objašnjeni određeni koraci i klase, odnosno dijelovi određenih klasa koje implementiraju JWT autorizaciju. Nakon što se postave svi potrebni preduvjeti, može se krenuti s implementiranjem JWT autorizacije. Prvo se kreira model, a nakon modela kreira se repozitorij za taj model. Model se kreira po relacijskome modelu kao i ostatak projekta. Nakon kreiranja modela i repozitorija, kreiraju se klase po strukturi projekta, tj. po strukturi Spring Securitya koja je opisana gore u tekstu. Anotacija *@EnableWebSecurity* dopušta Springu da automatski nađe i primijeni klasu u globalnu *web* sigurnost. Anotacija *@EnableGlobalMethodSecurity* pruža AOP sigurnost nad metodama, omogućava *@PreAuthorize* i *@PostAuthorize*. *Configure(HttpSecurity http)* metoda se nadjačava iz *WebSecurityConfigurerAdapter* sučelja i govori Springu kako konfiguriramo CORS i CSRF, kada želimo dohvatiti sve korisnike koji su autentificirani ili ne, kada želimo da se to radi (*UsernamePasswordAuthenticationFilter*) i koji je *Exception Handler* odabran.

Slika 30 Klasa WebSecurityConfig

```
@Configuration
@EnableWebSecurity
@EnableGlobalMethodSecurity(
        // securedEnabled = true,
        // jsr250Enabled = true,
        prePostEnabled = true)
public class WebSecurityConfig extends WebSecurityConfigurerAdapter {

    //Body

    @Override
    protected void configure(HttpSecurity http) throws Exception {
        http.cors().and().csrf().disable()
                .exceptionHandling()
                .authenticationEntryPoint(unauthorizedHandler).and()
                .sessionManagement()
                .sessionCreationPolicy(SessionCreationPolicy.STATELESS).and()
                .authorizeRequests().antMatchers("/api/auth/**").permitAll()
                .antMatchers("/api/**").permitAll()
                .anyRequest().authenticated();

        http.addFilterBefore(authenticationJwtTokenFilter(),
        UsernamePasswordAuthenticationFilter.class);
    }
}
```

Izvor: Izrada autora

AuthTokenFilter.java klasa definira se filter koji se izvršava jednom po upitu. Klasa se nadograđuje na *OncePerRequestFilter* i *overrida doFilterInternal()* metodu. Nakon postavljanja, svaki put kada želimo dohvatiti detalje korisnika, može se koristiti *SecurityContext*. Unutar *doFilterInternal()* metode obrađuje se sljedeće:

1. Dohvat JWT-a iz autorizacijskog zaglavlja.
2. Validiranje JWT-a i prosljeđivanje korisničkog imena iz njega.
3. Kreiranje autentifikacijskog objekta.
4. Ostavljanje trenutnih korisničkih detalja u *SecurityContextu*, koristeći *setAuthentication*(*authentication*) metodu.

Slika 31: Funkcija setAuthentication(authentication)

```
SecurityContextHolder.getContext().setAuthentication(authentication);
```

Izvor: Izrada autora

JwtUtils klasa ima tri funkcije:

1. Generira JWT iz korisničkog imena, datuma, roka trajanja i tajnog ključa.
2. Dohvaća korisničko ime iz JWT-a.
3. Validira JWT.

Slika 32: JWT Odgovor

```
{
    "id": 2,
    "username": "toni",
    "email": "toni@toni.hr",
    "roles": [
        "ROLE_ADMIN"
    ],
    "accessToken": "eyJhbGciOiJIUzUxMiJ9.eyJzdWIiOiJtbzMyZCIsImlhd
                    CI6MTU5OTUwNzA2MywiZXhwIjoxNTk5NTkzNDYzfQ.w9ob
                    Zy9uKxAPHUMhtBxrFCln58wCEbHqSoJm2lwI_KU-nlThmN
                    8Uf2IbWMER-P4hIvt5SnKwq4pf6gwci9PYKA",
    "tokenType": "Bearer"
}


{
    "timestamp": "2020-09-07T19:27:32.755+0000",
    "status": 401,
    "error": "Unauthorized",
    "message": "Error: Unauthorized",
    "trace": "org.springframework.security.access.AccessDeniedException: ...",
    "path": "/api/mod"
}
```

Izvor: Izrada autora

Slika 33: Klasa JwtUtils

```
@Component
public class JwtUtils {
    private static final Logger logger =
    LoggerFactory.getLogger(JwtUtils.class);

    @Value("${BiljkaRestApi.app.jwtSecret}")
    private String jwtSecret;

    @Value("${BiljkaRestApi.app.jwtExpirationMs}")
    private int jwtExpirationMs;

    public String generateJwtToken(Authentication authentication) {

        UserDetailsImpl userPrincipal =
        (UserDetailsImpl) authentication.getPrincipal();

        return Jwts.builder()
                .setSubject((userPrincipal.getUsername()))
                .setIssuedAt(new Date())
                .setExpiration(new Date((new Date()).
                getTime() + jwtExpirationMs))
                .signWith(SignatureAlgorithm.HS512, jwtSecret)
                .compact();
    }

    public String getUserNameFromJwtToken(String token) {
        return Jwts.parser().setSigningKey(jwtSecret).
        parseClaimsJws(token).getBody().getSubject();
    }

    public boolean validateJwtToken(String authToken) {

        // Try Catch

        return false;
    }
}
```

Izvor: Izrada autora

Nakon kreiranih glavnih klasa koje obrađuju JWT, potrebno je kreirati *payload* REST API-ja za JWT. Za upite imamo dva *payloada*: *LoginRequest* i *SignupRequest*, dok za odgovor imamo *payloadove*: *JwtResponse* i *MessageResponse*. Zadnji korak je kreiranje kontrolera za prijavu i registraciju te dodavanje JWT autorizacije, tj. dopuštenje pristupa za svaku metodu, krajnju točku REST API-ja. Kontrolu pristupa pomoću JWT-a vršimo anotacijom

@PreAuthorize. Anotacija *@PreAuthorize* dopušta dohvat krajnje točke onome korisniku kojemu se *rola* podudara s definiranom *rolom* nad tom krajnjom točkom.

Slika 34: Klasa PorodicaControll

```
@CrossOrigin(origins = "*", allowedHeaders = "*")
@RequestMapping("/api")
public class PorodicaControll {

    // Get All Porodica
    @PreAuthorize("hasRole('STUDENT')
                  or hasRole('PROFESOR')
                  or hasRole('ADMIN')")
    @GetMapping("/porodica")
    public Page<Porodica> getAllPorodica(
                    Pageable pageable,
                    @RequestParam(defaultValue = "0") int page,
                    @RequestParam(defaultValue = "10") int size) {
        return porodicaRepository.findAll(pageable);
    }

}
```

Izvor: Izrada autora

6. Vue.js

Klijentski dio ovoga rada rađen je u Vue.js *frameworku* uz pomoć Vue-Materiala i MDB biblioteke. Ostale biblioteke koji se koriste, a relevantniji su za projekt su: vue-router, axios, vuex. Vue.js je progresivni *web framework* baziran na JavaScript programskome jeziku. Vue.js kreira SPA (eng. *Single Page Applications*). Takve stranice su jednostavnije za održavanje, koriste komponente i reaktivne su. Uz kreiranje SPA aplikacija Vue se koristi i za kreiranje UI-a (eng. *User Interface*) to jest korisničkih sučelja. Modularizacija biblioteka, koristeći *framework* je vrlo česta u dizajniranju i programiranju klijentskog dijela aplikacije. Uz Vue, Angular i React imaju modularizaciju. Od ta tri *frameworka,* Vue je najjednostavniji u proširivanju funkcionalnosti i radu s više modula istodobno. U slučaju da želimo osnovnu i malu aplikaciju, dosta je koristiti samo Vue osnovnu biblioteku. Kako se kompleksnost aplikacije povećava, tako je potrebno implementirati i dodatne biblioteke u samu aplikaciju. Vue-router je biblioteka za upravljanje rutama u aplikaciji, vuex je biblioteka za upravljanje globalnim stanjem komponenata, axios je biblioteka koja obrađuje HTTP komunikaciju, dok su Vue-Material i MDB biblioteka koji izgrađuju responzivne *web*-aplikacije. Samim time može se podijeliti SPA-aplikaciju u razine ovisno o kompleksnosti:

1. Mala SPA – koristi osnovni Vue.
2. Prosječna SPA – koristi uz Vue i Vue ruter.
3. Velika SPA – uz Vue i Vue ruter koristi i Vuex, Axios, Vue-Material, MDB.
4. MPA – uz sve navedeno, koristi i VUE Server renderiranje.

Aplikacija kreirana u ovome radu spada pod velike SPA-je te uz navedene biblioteke koristi i ostale koje su potrebne kako bi se obradili pojedini dijelovi kao što je paginacija, validacija, *multiselect* i ostali. Uz Vue i njegove biblioteke imamo i Vue-Material kao i MDB. One su također biblioteke, no mnogo su kompleksnije i njihova je važnost u vue projektu mnogo veća. Uz Vue elemente, sadrže komponente i funkcionalnosti, i svoje vlastite elemente i funkcionalnosti, koje se prije svega nadovezuju na Vue kako bi poboljšali dizajn same aplikacije, ali isto tako uveli određene funkcionalnosti specifične za određenu biblioteku.

Vue-Material i MDB (eng. *Material Design Bootstrap*) svojim komponentama nadograđuju dizajn Vue,js *frameworka*. Vue Material je baziran na Material Dizajnu od Googlea, dok je MDB baziran na Bootstrapu. MDB je jedan od najpopularnijih i

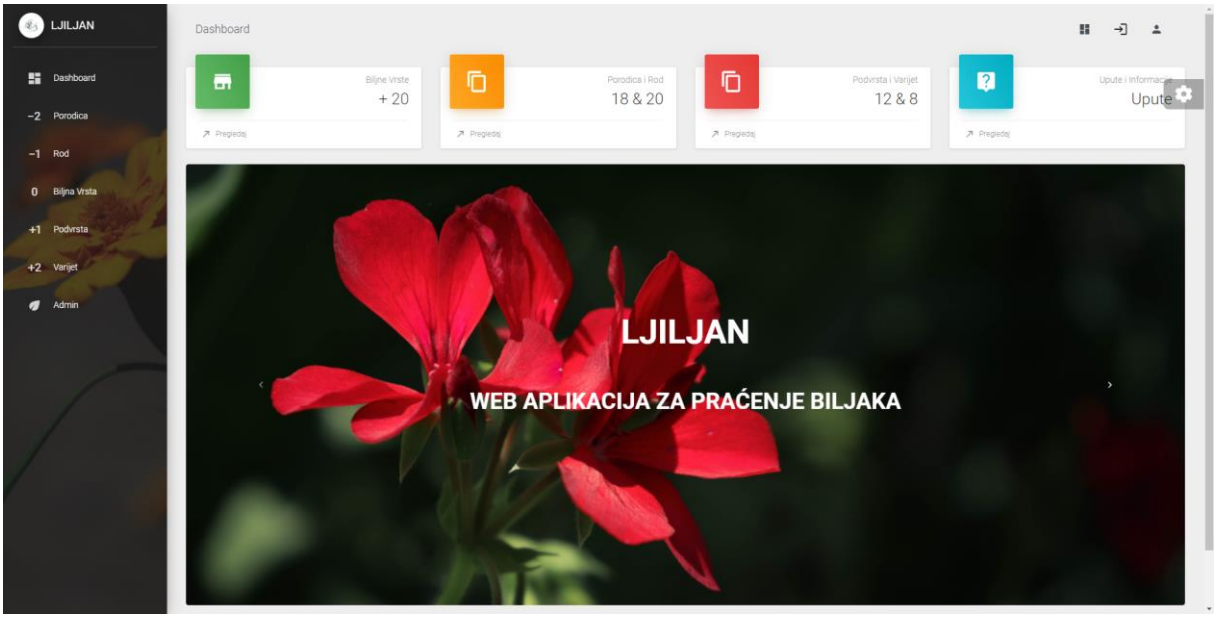
najjednostavnijih Bootstrap nadogradnji koja svojim izgledom izgleda kao Material Dizajn. U ovome projektu korišten je *dashboard* kreiran pomoću Vue-Materiala baziran na Vue.js-u, dok su neke funkcionalnosti i komponente korištene prije svega od MDB-a. U nastavku će biti opisan osnovni dizajn, integracija klijentske i serverske strane aplikacije pomoću Axios-a te će biti prikazana JWT autorizacija primijenjena preko Vuex-a. Što se tiče dohvata podataka, svi podaci su dohvaćeni s serverske strane praktičnog dijela ovoga rada.

6.1. Dizajn

U ovome će poglavlju prije svega biti prikazan osnovni dizajn aplikacije kroz sučelje *Porodica*. No prvenstveno će biti objašnjen dizajn, tj. kompozicija *web*-aplikacije na primjeru početne stranice. Stranicu se može podijeliti u četiri dijela što se tiče kompozicije:

1. Glavni prozor – podaci
2. Meni – navigacija
3. Zaglavlje – naslov trenutne stranice, autorizacija, pretraga, profil
4. Podnožje – završni kostur koji kompozira glavni prozor.

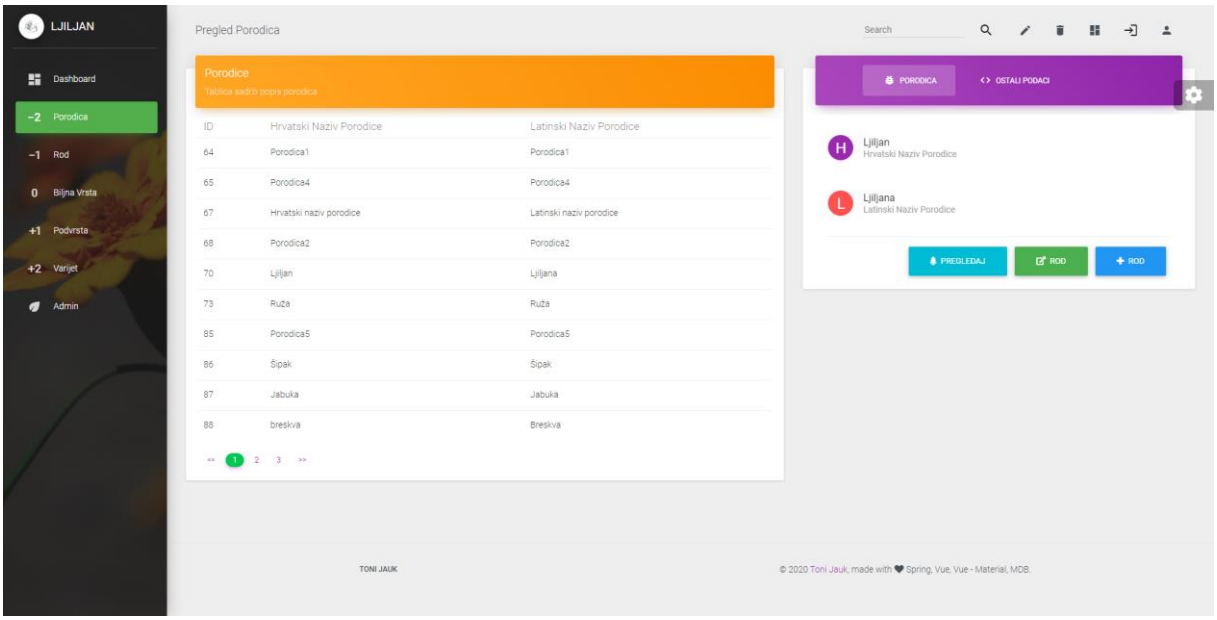
Slika 35: Dashboard



Izvor: Izrada autora

Navigacija aplikacije nalazi se na lijevoj strani prozora te, uz naziv i ikonu, sadrži šest glavnih ruta koje vode do ostalih prozora i koje su definirane po bazi podataka hijerarhijski s glavnim prozorom *Biljna Vrsta*. Zaglavlje se nalazi u gornjem dijelu prozora te na svojoj lijevoj strani sadrži naziv sučelja u kojem se trenutno korisnik nalazi, dok na desnoj strani sadrži tipke za prijavu, tj.odjavu, prikaz profila, *dashboard* i u određenim sučeljima pretragu, unos i brisanje. Podnožje se nalazi na dnu prozora te sadrži opis aplikacije i projekta. Glavni prozor je u sredini i, kao takav, sadrži osnovne podatke svakog sučelja, odnosno glavne funkcionalnosti sustava. U gornjem primjeru se može vidjeti kartice s pregledom podataka te slike koje opisuju aplikaciju.

Slika 36: Pregled Porodica

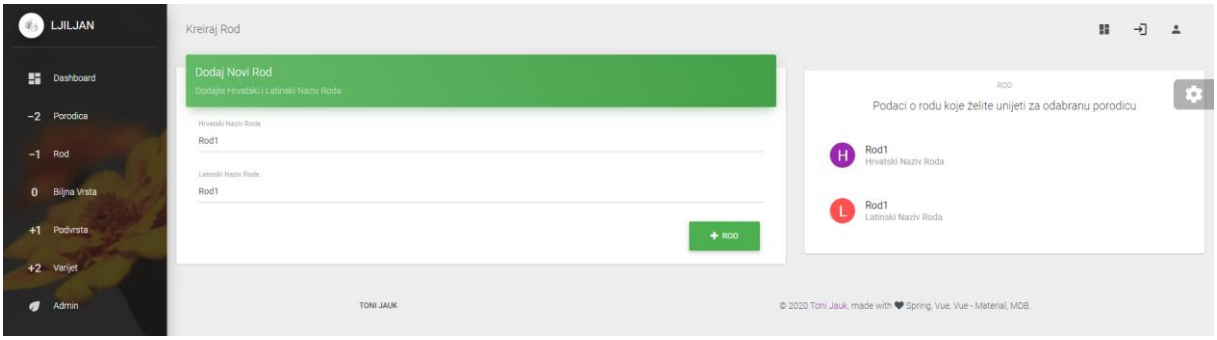


Izvor: Izrada autora

Pregled podataka unutar aplikacije je vrlo jednostavan i intuitivan. Prozor pregleda podataka, u ovome slučaju porodice, sastoji se od dvije kartice. Lijeva kartica je tablica sa svim zapisima unutar tablice *Porodica*. Sastoji se od rednoga broja, te ID-a i hrvatskog i latinskog naziva porodice. Pretraživati se može po svakom atributu u tablici. Na desnoj strani nalazi se kartica sa svim podacima koje odabrani zapis sadrži. Prilikom odabira zapisa iz tablice prikazuju se podaci o tom zapisu. Kartica trenutnog zapisa sadrži dva tab-a. Prvi tab daje najvažnije podatke o tom zapisu porodice, dok drugi tab daje dodatne podatke, tj. metapodatke o zapisu (tko ga je kreirao, kad je kreiran, uređen i dr.). Uz podatke o porodici kartica sadrži i

akcije koje se obavljaju prilikom pritiska na određenu tipku. To je glavni podmeni koji prolazi kroz sve CRUD akcije nad porodicom, ali i pripadajućom relacijom.

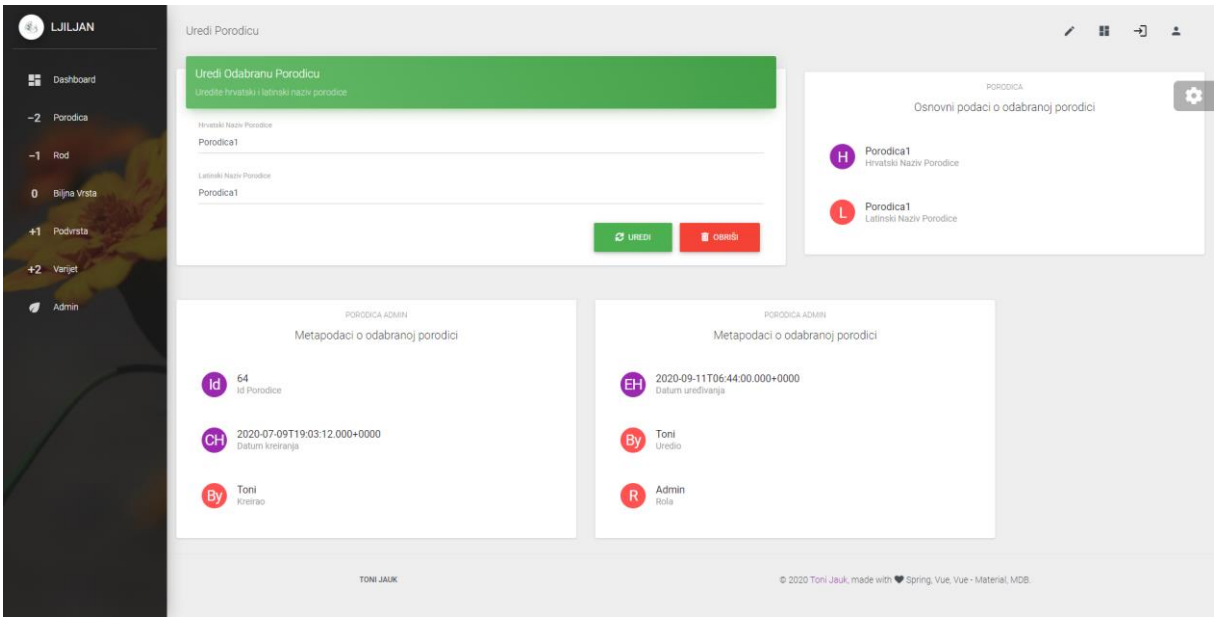
Slika 37: Dodaj Rod



Izvor: Izrada autora

Klikom na tipku + *Rod* prelazimo na stranicu u kojoj se dodaje rod za odabranu porodicu. Na lijevoj strani nalazi se kartica s formom za unos podataka, polja za unos validirana su prema bazi podataka kao i na serverskoj strani aplikacije. Klikom na tipku dodaj rod, rod se pomoću upita šalje u bazu podataka i sprema ako su svi podaci pravilno unijeti. Na desnoj strani nalazi se kartica s opisom unesenih vrijednosti koja prikazuje podatke koje unosimo, ali i podatke koji su vraćeni, tj. unesene podatke odgovorom prilikom uspješnog unosa roda.

Slika 38: Uredi Porodicu



Izvor: Izrada autora

Zadnja, ali ne i nevažna, stranica koja je opisana u pisanome je pregled i ažuriranje zapisa. Stranica sadrži n broj kartica od koji jedna kartica vrši ažuriranje nad odabranim podacima, dok ostale kartice prikazuju sve unesene podatke o tome zapisu i njenim pripadajućim relacijama. Podaci odabrane relacije za ažuriranje proslijeđuju se u polja forme te njihovom izmjenom i klikom na tipku – uredi rod oni se uređuju i spremaju u bazu. Odabirom tipke obriši otvara se modul kojim potvrđujemo brisanje. Ako je zapis obrisan, aplikacija nas prebacuje na pregled zapisa rodova.

6.2. Komunikacija

Najvažniji dio je spajanje i komunikacija s serverskom stranom, tj. dohvat, slanje i manipulacija nad podacima. Ako nema komunikacijse s serverom, klijentski dio diplomskoga, a i svakog drugog projekta, je nepotreban i nebitan. Klijent s serverom u ovome praktičnom dijelu rada komunicira pomoću Axios biblioteke, dohvaćajući HTTP krajnje točke. On šalje upit serverskoj strani aplikacije te mu ona nazad odgovara.

Kako bismo pozvali krajnju točku te poslali podatke, potrebno je kreirati servis s funkcijama koje sadrže URL-ove tih krajnjih točki. Uz pozivanje URL-ova krajnjih točki, šalju se varijable putanja, parametri i podaci u tijelu upita.

Slika 39: Klasa PorodicaDataService

```
import http from "../http-common";

class PorodicaDataService {
  getAll(pagee) {
    return http.get("/porodica", { params: { size: 10, page: pagee } });
  }

  get(id) {
    return http.get(`/porodica/${id}`);
  }

  create(data) {
    return http.post("/porodica", data);
  }

  update(id, data) {
    return http.put(`/porodica/${id}`, data);
  }

  delete(id) {
    return http.delete(`/porodica/${id}`);
  }

  deleteAll() {
    return http.delete(`/porodica`);
  }
```

Izvor: Izrada autora

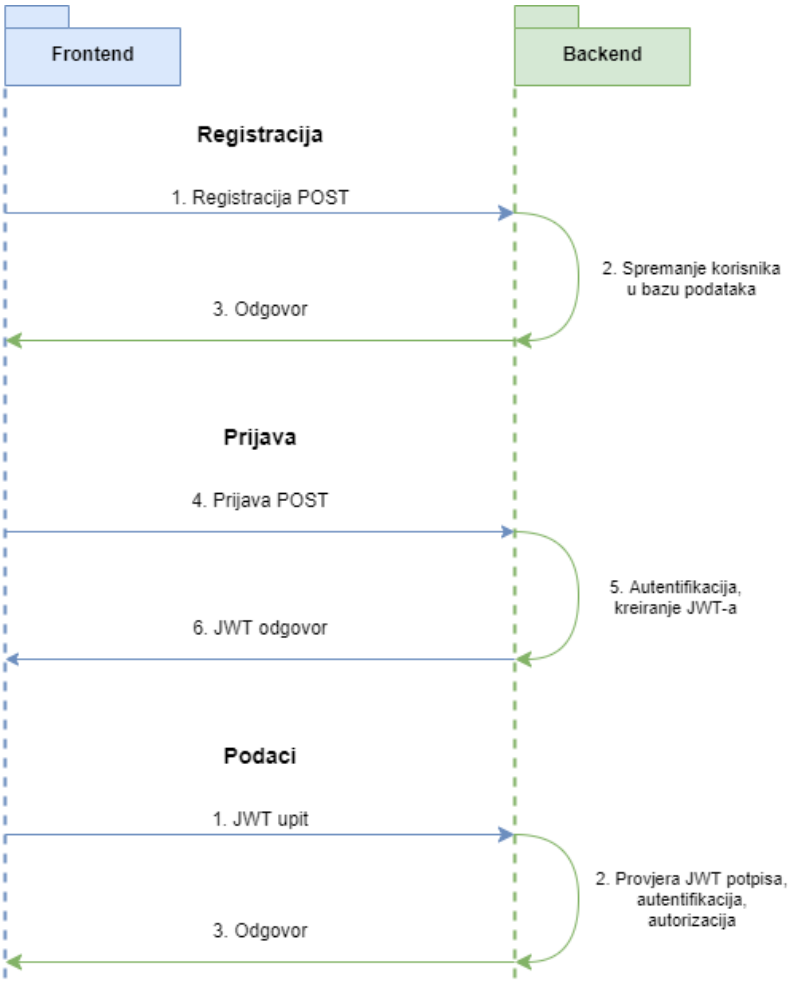
Korištenje axiosa vrši se na način da se pozove krajnja točka pomoću tipa metode. U slučaju da želimo poslati *GET* http metodu potrebno je pozvati Axios funkciju *http.get* te proslijediti URL parametre te dodatne podatke, ako je to potrebno. Nakon što je upit poslan, vraća odgovor, koji se kasnije također obrađuje. Svaki odgovor mora biti obrađen, ako je odgovor uspješan, pokazuju se u većini slučajeva podaci koje koristi klijentski dio, a ako je odgovor neuspješan, potrebno je obraditi poruku koja opisuje neuspješnost kako bi korisnik znao što je krivo te kako postupati dalje. Prosljeđivanje podataka kroz tijelo upita je najjednostavnije te je to u primjeru vidljivo nad varijablom *data*. Korištenje varijabli putanje, u većini slučajeva to je ID, se može vidjeti na primjeru varijable ID, dok za prosljeđivanje parametara, može se za primjer uzeti prvu funkciju *getAll* koja proslijeđuje parametre. Za pozivanje *POST* krajnje točke koristimo Axios funkciju *http.post*, za pozivanje *PUT* krajnje

točke koristimo Axios funkciju *http.put*, a za pozivanje *DELETE* krajnje točke koristimo Axios funkciju *http.delete.*

6.3. Autorizacija i autentifikacija nad klijentskim dijelom

Klijentski dio praktičnog dijela ovoga rada sadrži JWT autorizaciju, autentifikaciju i validaciju. Sadrži forme za prijavu i registraciju za postojeće i nove korisnike. Formi prijave može svatko pristupiti, dok registracijskoj formi mogu prisupiti samo profesor i administrator. Uz to raznim dijelovima aplikacije pristupa se i po ulogama korisnika.

Slika 40: JWT upit odgovor



Izvor: Izrada autora

JWT autorizacija se može podijeliti po komponentama koristeći Vuex i Vue Router:

1. Komponenta aplikacije – obrađuje podatke i prikazuje one koje može vidjeti određeni korisnik. Dohvaća stanje apliakcije iz Vuex-a i prosljeđuje podatke komponentama.
2. Komponente prijave i registracije imaju formu za prihvat podataka kao i validaciju forme. Vuex *dispatch()* funkcija kreira akcije za prijavu/registraciju.
3. Vuex akcije pozivaju autorizacijski servis, metode koje koriste axios za komunikaciju s serverskom stranom aplikacije. Unutar tih metoda, pohranjuje se, ili se dohvaća JWT.
4. Komponenta profila prikazuje profil autentificiranog i autoriziranog korisnika.
5. Servis korisnika koristi funkciju *auth-header()* za dodavanje JWT-a u HTTP autorizacijsko zaglavlje.

7. Zaključak

Ovaj specijalistički završni rad prvenstveno se bavi serverskom stranom aplikacije praktičnog dijela ovog rada. Kroz serversku stranu tekstualnog dijela rada opisan je prvenstveno proces kreiranja REST API servisa koristeći Spring *framework*. U praktičnome dijelu kreiran je potpuno funkcionalan REST API servis koji može poslužiti i kao okosnica po kojoj se kreiraju novi projekti unutar Springa. Takav je princip vrlo dobar jer smanjuje vrijeme programiranja i dizajniranja aplikacija. Uz to, glavna svrha ovog rada bilo je kreiranje glavnih CRUD funkcija koje omogućavaju studentima, profesorima ali i javnosti, pregled i ažuriranje podataka o biljkama. Aplikacija ima JWT autorizaciju koja za takav tip *web*-aplikacija prolazi bolje od autorizacije bazirane na sesiji. Uz to JWT je bolji ako bismo ikada htjeli proširiti primjenu REST Api servisa i na druge aplikacije kao što su mobilne aplikacije. Klijentski dio praktičnog dijela rada rađen je u Vue.js-u uz Vue-Material i MDB te on prikazuje klijentsku stranu rada. Serverska strana aplikacije ima definirane korisnike svojih krajnjih točki, dok klijentski dio može biti korišten od nasumičnog korisnika. Prvenstveno su to studenti i profesori, no određeni su dijelovi w*eb*-aplikacije namijenjeni i javnosti . Uz gore spomenute *frameworke*, u današnje se vrijeme aplikacije ne može izgraditi i bez dodatnih biblioteka i *frameworka* koji su korišteni i u ovome radu.

Svaki projekt, pa tako i ova w*eb*-aplikacija, može biti kreirana na više različitih načina i pomoću više različitih tehnologija. Za serversku stranu aplikacije je prije svega odabran Spring, jer je baziran na Javi, kao i većina glavnih predmeta na Veleučilištu u Rijeci kojima je primarni cilj učenje programiranja.

Prije svega, želja autora je bila napisati i izgraditi ovaj rad u programskome jeziku koji najviše preferira (Java) i koji ga je potaknuo da se dublje upusti u programiranje, ali upustiti se i u nove razvijajuće tehnologije (JavaScript) koje zauzimaju veliki prostor na tržištu. *Web*-aplikacija ovoga rada u potpunosti je iskoristiva u trenutnoj formi ali i u formi prilagođenoj određenim zahtjevima.

Literatura

1. Babić, S., Finka, B., Moguš, M., Hrvatski pravopis, osmo izdanje, Školska knjiga, Zagreb, 2004.
2. Baeldung, Build Your API with Spring, Baeldung, 2019.
3. Gutierrez F., Pro Spring Boot 2: An Authoritative Guide to Building Microservices, *Web* and Enterprise Applications, and Best Practices, Apress, Albuquerque, NM, USA, 2019.
4. Jauk T, Izgradnja dijela informacijskog sustava prijama pošte, Veleučilište u Rijeci, Rijeka, Hrvatska, 2018.
5. Java Code Geeks, Spring Framework Cookbook, Java Code Geeks, 2019
6. Zwitserloot R., Spilker R., Documentation, https://projectlombok.org/features/all, 2020 (24.7.2020).
7. StartupFlow S.C., Documentation, https://mdbootstrap.com/, 2019 (15.8.2020).
8. Pecinovský R., OOP – Learn Object Oriented Thinking and Programming, Bruckner T., Czech Republic, 2013.
9. Ravi K. S., Learning Spring Application Development, Packt Publishing, Birmingham, UK, 2015.
10. Leksikografski zavod Miroslav Krleža, Sistematika, Hrvatska enciklopedija, mrežno izdanje, 2020., http://www.enciklopedija.hr/Natuknica.aspx?ID=56261. (8.08.2020.).
11. Siva K., Beginning Spring Boot 2: Applications and Microservices with the Spring Framework, Apress, Hyderabad, India. 2017.
12. Johnson,R., Hoeller,J., Donald,K., Spring Framework Documentation, 2020, https://docs.spring.io/spring/docs/current/spring-framework-reference/ (12.8.2020.).
13. Rod Johnson, et al., Guides, https://spring.io/guides, 2016 (12.8.2020).
14. Van de Velde T., et al., Beginning Spring Framework 2, Wiley Publishing, Indianapolis, Indiana, 2008.
15. Vue.js, Documentation, https://vuejs.org/v2/guide/, 2020 (15.8.2020).
16. Vue-Material, Documentation, https://vuematerial.io/getting-started/, 2020 (15.8.2020).

Popis slika

Popis tablica

Popis kratica

API	Application Programming Interface
AOP	Aspect Oriented Programming
BLOB	Binary Large Object
CPU	Computer Processing Unit
CSS	Cascading Style Sheet
CORBA	Common Object Request Broker Arhitecture
CLI	Comand Line
CMD	Comand Prompt
CLOB	Character Large Object
CORS	Cross-Origin Resource Sharing
CSRF	Cross-Site Request Forgery
CSV	Coma Separated Values
CRUD	Create Read Update Delete
DOM	Document Object Model
EE	Enterprise Edition
FK	Foreign Key
GPU	Graphic Processing Unit
HTTP	Hyper Text Transfer Protocol
HATEOAS	Hypermedia as the Engine of Application State
HTTPS	HyperText Transfer Protocol Secure
JS	JavaScript
JWT	JSON *Web* token
JDK	Java Development Kit

JMS	Java Message Service
JSON	JavaScript Object Notation
JPA	Java persistance API
JDBC	Java Database Connectivity
MVC	Model View Controll
MDB	Material Design Bootstrap
ORM	Object Relational Mapping
PK	Primary Key
POJO	Plain Old Java object
REST	Representational State Transfer
RAM	Random Access Memory
RM	Relacijski Model
RPC	Remote Procedure Call
SQL	Structured Query Language
SPA	Single Page Application
SOAP	Simple Object Access Protocol
SPM	Software Project Manager
URL	Uniform Resource Locator
UML	Unified Model Language
XML-RPC	Extensible Markup Language-Remote Procedure Call
XML	Extensible Markup Language