Web aplikacija za recepte i kuhanje

Jurkić, Luka

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

Sveučilišni prijediplomski studij Računarstvo

WEB APLIKACIJA ZA RECEPTE I KUHANJE

Završni rad

Luka Jurkić

Osijek, 2024.

SADRŽAJ

1. UVOD

Ljudi od davnina teže zdravijoj prehrani. Zdrava prehrana iziskuje puno vremena, truda i novca. Neke studije provedene u Ujedinjenom Kraljevstvu pokazuju kako prakticiranje zdrave prehrane može produljiti život i do 10 godina [1]. Uz taj studij postoje i drugi koji pokazuju kako konzumiranje zdravih namirnica ima pozitivan učinak na zdravlje. Inspirirani zdravim načinom života ljudi su potaknuti razvijati kulinarstvo.

U posljednjih nekoliko godina može se primijetiti veliki porast nutricionista koji savjetuju ljude kako se zdravije hraniti. Nutricionistički načini prehrane zahtijevaju vrijeme, novac i strpljenje. Slijedno tome dolazi do rasta broja kuharica, tekstualnih i onih na internetu. Zbog povećanja broja recepata, kojih je svaki tjedan više, potrebno je digitalizirati kulinarstvo kao granu znanosti.

Kako bi se omogućio jednostavniji pristup receptima, ovaj rad pokazuje postupak izrade web aplikacije koja nudi rješenja navedenih problema, jednostavnost u izradi recepata, brzina dijeljenja i savjetovanja ljudi te međusobna povezanost. Cilj ove web aplikacije je jednostavnije objavljivanje recepata na Internet, komentiranje i poboljšanje recepata te naposljetku izvoz recepata prebacivajući ih u tekstualni oblik.

U drugom poglavlju rada opisana su postojeća rješenja aplikacije. Svaka od aplikacija koje su opisane prikazuju neka od rješenja koja ovaj rad nastoji objediniti u jedinstvenu web aplikaciju. Također je opisano korisničko sučelje te njegov dizajn.

Treće poglavlje pokriva zahtjeve aplikacije. Aplikacija ima dvije vrste zahtjeva: funkcionalne i nefunkcionalne. Napisan je niz raznih zahtjeva koje aplikacija treba ispunjavat te koje su njihove uloge u aplikaciji. Uz zahtjeve aplikacije, treće poglavlje daje pobliži uvid u arhitekturu koja je korištena za izradu web aplikacije.

Četvrto poglavlje obuhvaća cjeloviti rad. Opisuje se tok izrade aplikacije te daje kodove koji su korišteni za ostvarivanje zahtjeva. Četvrti dio opisuje izradu logike aplikacije i korisničkog sučelja.

1.1. Zadatak rada

Dati opis zahtjeva i funkcionalnosti koje treba imati web aplikacija s receptima i izračun makronutrijenata. Navesti nekoliko sličnih postojećih rješenja i usporediti izrađenu web aplikaciju s postojećim rješenjima. Projektirati i izraditi bazu podataka koja je potrebna za

ispravno funkcioniranje web aplikacije te opisati postupak izrade. Detaljno opisati postupak izrade kao i izrađene funkcionalnosti. Omogućiti registriranim korisnicima postavljanje recepata i ocjenjivanje tuđih recepata uz mogućnost prikaza najpopularnijih recepata. Za svaki recept potrebno je navesti sastojke te izračun makronutrijenata za zadani recept

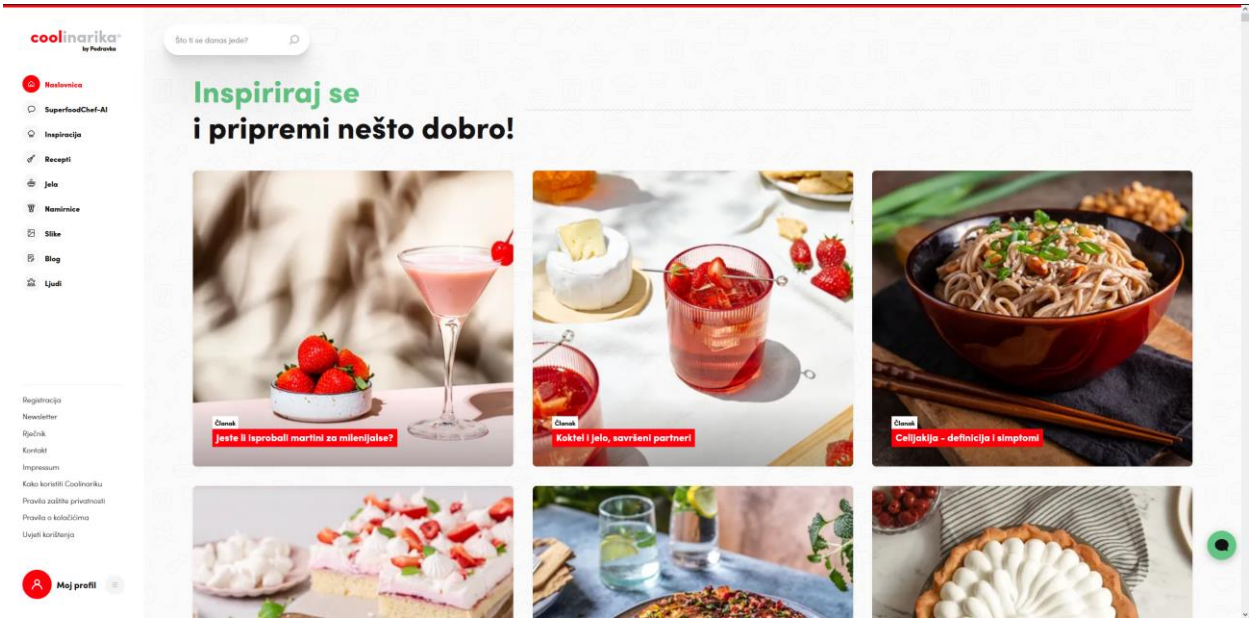
2. POSTOJEĆA RJEŠENJA

Porastom potrebe za zdravom prehranom raste i broj web stranica te blogova na kojima ljudi dijele recepte i znanje u području kulinarstva. Neki od najpopularnijih su: Coolinarika, Recepttura, Uvijekgladna, Minimalist baker i drugi.

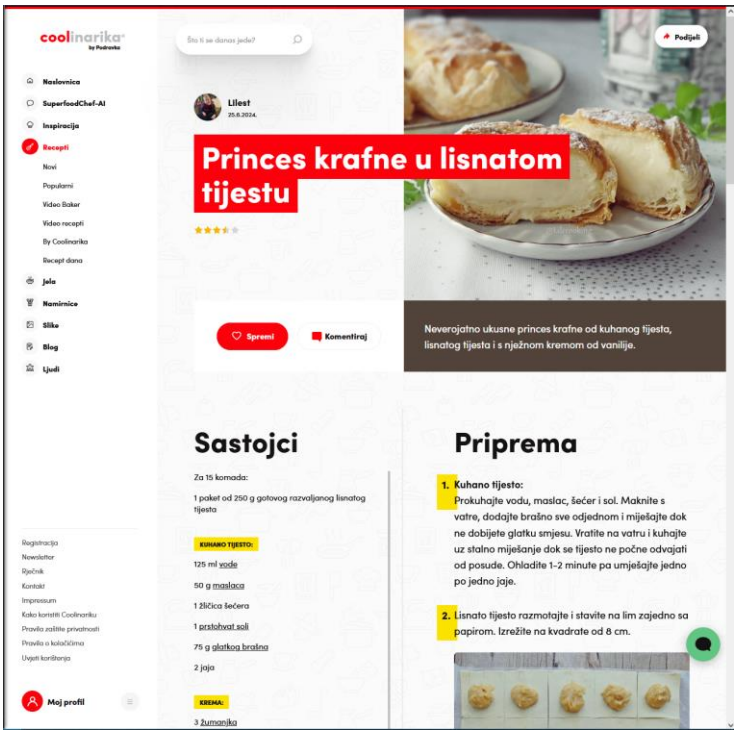
2.1. Coolinarika

Jedna od najpoznatijih stranica za pretraživanje recepata je stranica Coolinarika, čiju je početnu stranicu moguće vidjeti na 2.1. Coolinarika je stranica koja pod pokroviteljstvom i u vlasništvu Podravke. Coolinarika omogućuje pretraživanje recepata na razna načine. Opcije pretraživanja jela na Coolinariki su pretraživanje prema: kategorijama, namirnicama, slikama, sezonska jela, po popularnosti i tako dalje. Coolinarika omogućuje povezivanje ljudi preko blogova. Daje mogućnost komentiranja te dijeljenja svojih iskustava s drugim korisnicima.

Rastom i sve bržim razvojem umjetne inteligencije potrebno je pratiti potražnju i zahtjeve tržišta. Iz toga razloga Coolinarika također uvodi umjetnu inteligenciju na svoju stranicu. Pomoću umjetne inteligencije Coolinarika daje mogućnost jednostavnije pretrage raznih jela.



Sl. 2.1. Coolinarika početna stranica



Sl. 2.2. Coolinarika recept

Coolinarika ima intuitivno korisničko sučelje. Za svaki recept nudi opcije spremanja i komentiranja. Svaki recept ima jasno naznačene sastojke za svaki dio jela, te korake pripreme uz pripadajuće slike, primjer recepta koji se nalaze na stranici prikazan je na 2.2.

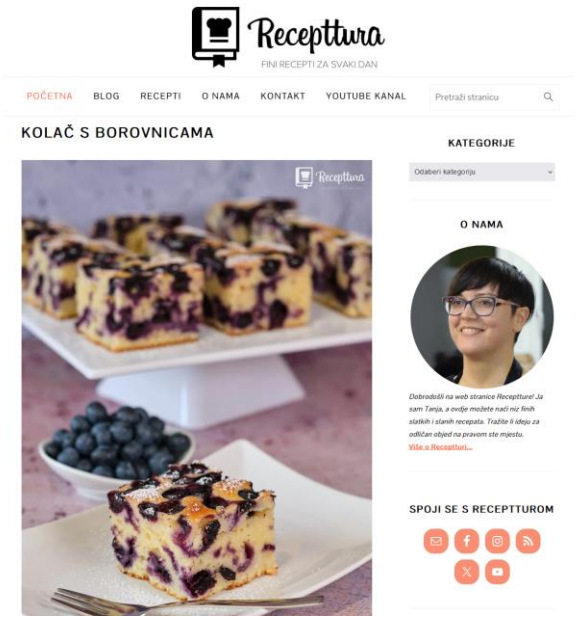
Coolinarika kao najpopularnija stranica za recepte je također i najbrže rastuća. Stranica svaki dan dobiva nove recepte te nove korisnike koji šire zajedništvo te promoviraju zdravu i kvalitetnu prehranu. Uz razne opcije pretrage recepata može se pronaći za svakog nešto.

Coolinarika postoji godinama te se razvila do razine gdje je novim korisnicima teško pohvatati sve mogućnosti. Korisničko sučelje više nije intuitivno jer ima previše opcija. Korisnik koji želi stranicu gdje može pohraniti sve svoje recepte na paru to može napraviti na Coolinariki, ali prije toga mora naučiti koristiti cijelo sučelje. Web aplikacija za recepte i kuhanje pojednostavljuje taj inicijali korak koje se odrađuje prilikom prvog korištenja aplikacije. Korisničko sučelje je jednostavnije i intuitivnije.

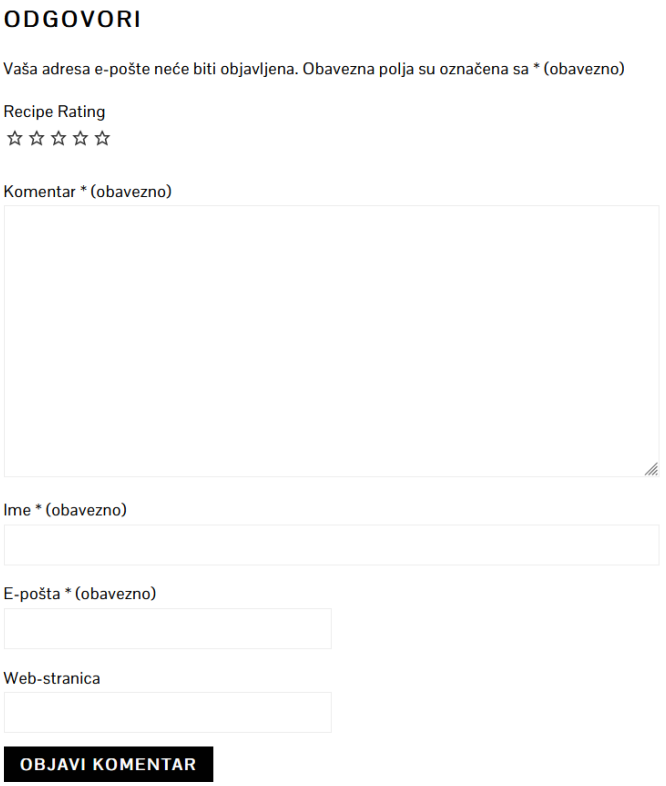
2.2. Recepttura

Recepttura, za razliku od Coolinarike je web stranica jedne osobe. Tanja, vlasnica stranice radi recepte te ih objavljuje. Recepttura sadrži razne načine pretraživanja. Pretraživati se može po kategoriji, po nazivu recepta te po datumu objave recepta. Na početnoj stranici Receptture se mogu pronaći recepti dana te razne novije recepte.

Stranica Receptture nudi mogućnost kontaktiranja te im je na taj način moguće poslati osobni recept. Kada se odabere neki recept stranica vodi na detalje o receptu. Tamo se mogu pronaći YouTube video o pripremi recepta. Uz video se nalaze recepti opisani u tekstualnom obliku. Na dnu stranice je popis sastojaka te popis koraka za izradu jela.



Sl. 2.3. Recepttura početna stranica



Sl. 2.4. Recepttura dodavanje komentara

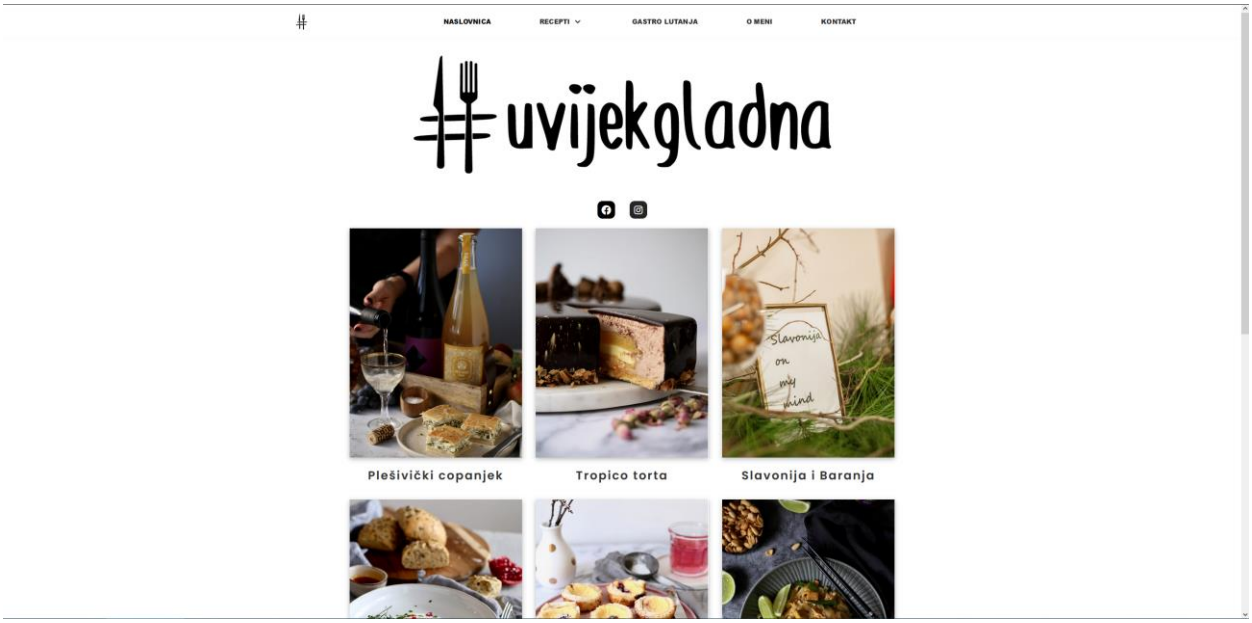
Za razliku od Coolinarike, stranica Receptture ne nudi mogućnost stvaranja profila, slijedno s time nije moguće spremiti neki recept za naknadni pregled. Iako ne nudi mogućnost stvaranja profila i dalje je moguće komentirati recepte te nudi mogućnost ispisa recepta u PDF formatu. Na 2.3. je prikazana početna stranica Receptture, a na 2.4. je prikaz podataka koje je potrebno unesti kako bi se ostavio komentar na nekom receptu.

Web aplikacija za recepte i kuhanje rješava problem korisnika. Recepttura ne nudi stvaranje korisnika, to uskraćuje korisniku stvaranje osobnih recepata. Recepttura nudi intuitivnije korisničko sučelje pod cijenu funkcionalnosti.

2.3. Uvijekgladna

Ova web stranica je malo drugačije osmišljena nego prethodne. Uvijekgladna je web stranica zamišljena kao blog jedne osobe na kojoj se objavljuju recepti te upute za izradu jela. Razlog zašto je ova web stranica drugačija je jer nudi pregled specijaliteta unutar Hrvatske po regijama.

Ova web stranica je jednostavnog izgleda i vrlo je intuitivna za korištenje. Na početnoj stranici, koju je moguće vidjeti na 2.5 se nalazi jednostavan izbornik te u njemu neke osnovne opcije za stranicu koja ima kulinarski sadržaj. Ispod naslova vide se kartice s naslovima. Te kartice mogu

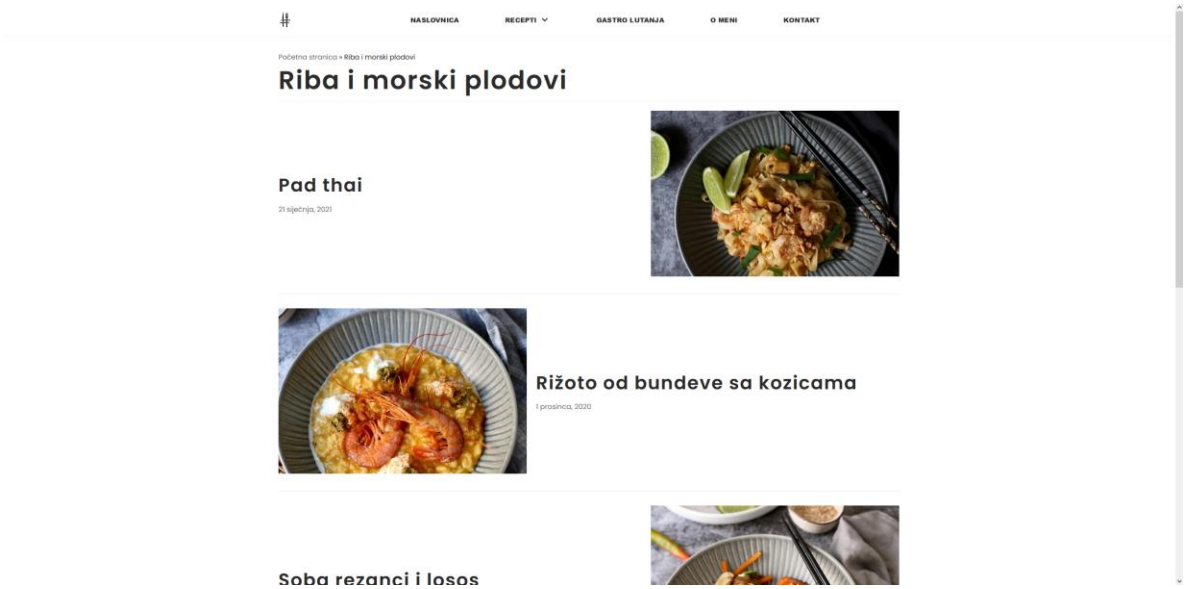


Sl. 2.5. Uvijekgladna početna stranica

predstavljati jela, mogu predstavljati neki proizvod u Hrvatskoj ili neku regiju te njezine specijalitete. Unutar izbornika pod opcijom „Recepti“ su svi tipovi recepata koje ova stranica nudi.

Sl. 2.7. Uvijekgladna izbornik

Prateći neku opciju od navedenih u izborniku, koji se nalazi na 2.6 se tada prikazuju razni recepti. Odabirući jedan od njih na početku se nalazi velika slika te opis toga jela. Na dnu te stranice su ispisani sastojci te postupak izrade jela.



Sl. 2.6. Uvijekgladna recept

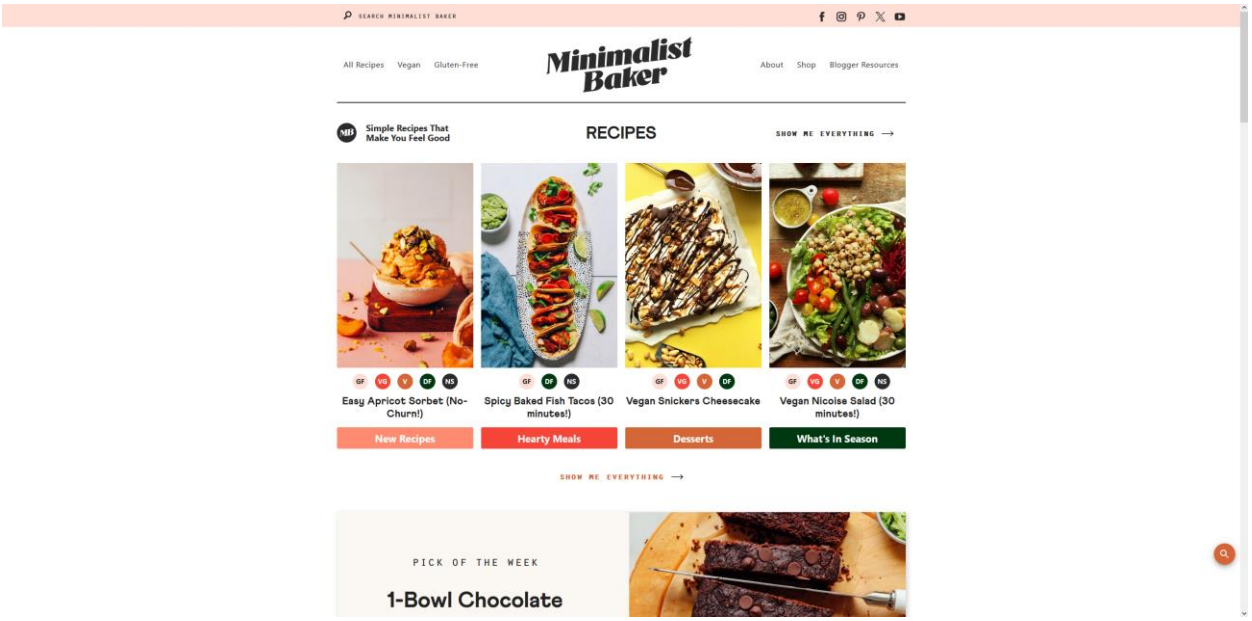
Ova web stranica ne nudi mogućnost izrade profila. Nudi mogućnost kontaktiranja vlasnice stranice ostavljajući poruku. Iako Recepttura nema opciju izrade profila i dalje nudi, za razliku od stranice Uvijekgladna, opciju ostavljanja komentara na pojedini recept. Za razliku od ostalih

stranica, stranica Uvijekgladna ne nudi opciju pretraživanja recepata. Primjer nekoliko recepata odabranih po nekoj kategoriji prikazano je na 2.7.

Stranica Uvijekgladna ima jednaki problem kao i stranica Recepttura, nedostaje joj mogućnost prijave korisnika. Web aplikacija za recepte i kuhanje rješava problem stvaranje osobnih recepata i dijeljenja s drugim korisnicima

2.4. Minimalist baker

Minimalist baker je web stranica za recepte koja ne dolazi s Hrvatskog govornog područja. Ova



Sl. 2.8. Minimalist baker početna stranica

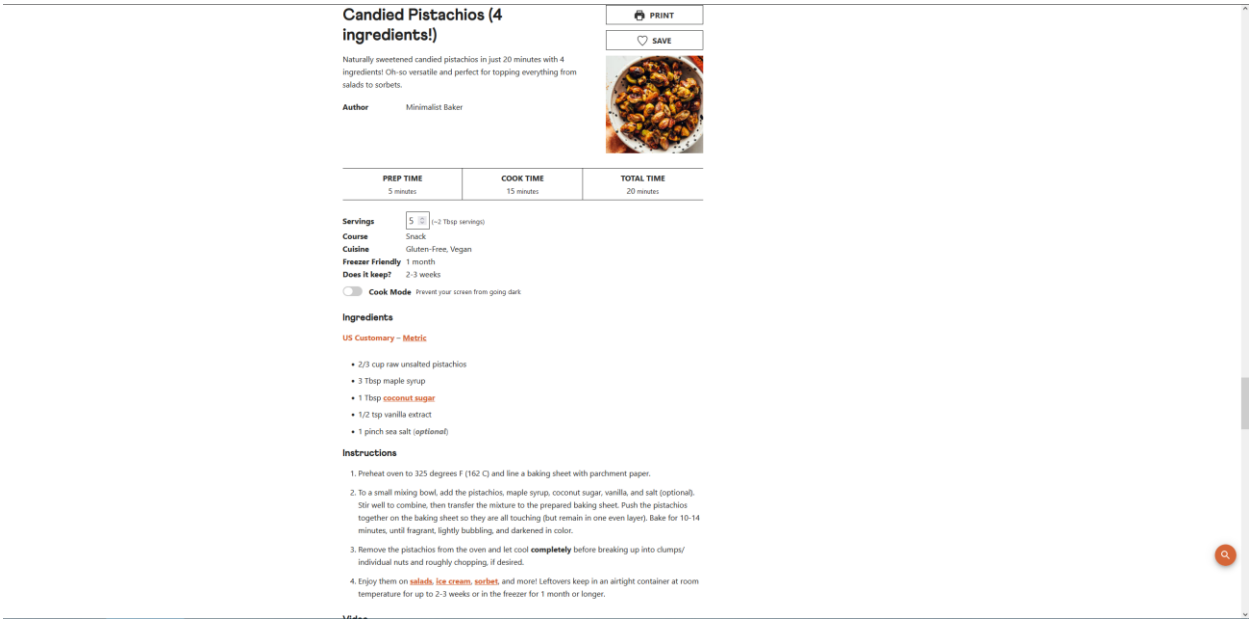
stranica nudi puno širi spektar recepata iz svijeta. Minimalist baker je stranica koja je, kao i Uvijekgladna, napravljena poput bloga, ali s više mogućnosti. Na početnoj stranici su opcije za recepte te osnovni izbornik koji se nalazi s lijeve i desne strane naslova stranice. Također na dnu 2.8. vidi se izbor tjedna kojeg nudi stranica. Kada se ode na opciju „*All Recipes*“ otvara se dodatan izbornik u gdje se dodatno filtriraju recepte na razne načine i tako se pronalaze specifični recepti.

Nakon odabira recepta na stranici recepta je niz slika koji detaljnije opisuju recept, kao što se može vidjeti na 2.9. Uz slike nalazi se i tekstualni opis slike. Na dnu stranice se pronalazi popis sastojaka te postupak izrade za to jelo. Značajka koju Minimalist baker nudi, a stranice do sada

nisu nudile je opcija biranja broja serviranja koje radimo, te se prema tome korigiraju količine sastojaka. Ispod svega još se nalazi video za izradu jela te popis nutrijenata.

Iako Minimalist baker ne nudi opciju stvaranja profila, ostavlja mogućnost pisanja komentara na pojedini recept. Prednost ove stranice nad ostalima je u tome što je svaki recept označen na način ima li glutena ili nema, je li vegetarijansko jelo te sadrži li laktozu. Primjer toga može se vidjeti na 2.10.

Minimalist baker je stranica čije je korisničko sučelje jako šareno i pretrpano opcijama koje nisu potrebne korisnicima koji započinju svoju repozitorij recepata. Korisnici koji dolaze na novu stranicu očekuju osnovne funkcionalnosti na lako dostupnim lokacijama na stranici. Web aplikacija za recepte i kuhanje rješava taj problem uklanjajući nepotrebne funkcionalnosti.



Candied Pistachios (4 ingredients!)

PRINT

SAVE

Naturally sweetened candied pistachios in just 20 minutes with 4 ingredients! Oh-so versatile and perfect for topping everything from salads to sorbets.

Author Minimalist Baker

PREP TIME	COOK TIME	TOTAL TIME
5 minutes	15 minutes	20 minutes

Servings 5 (~2 Tbsp servings)
Course Snack
Cuisine Gluten-Free, Vegan
Freezer Friendly 1 month
Does it keep? 2-3 weeks

Cook Mode Prevent your screen from going dark

Ingredients

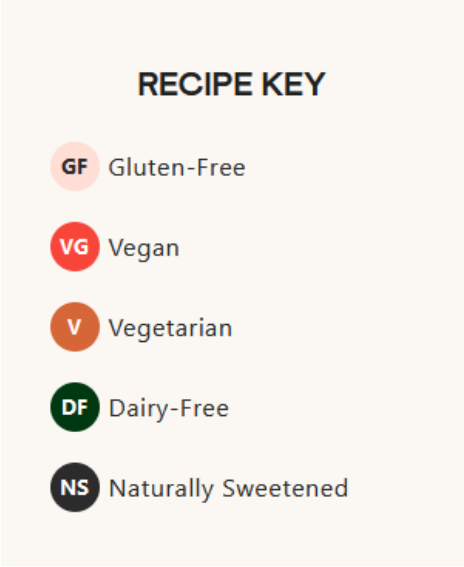
US Customary – Metric

- 2/3 cup raw unsalted pistachios
- 3 Tbsp maple syrup
- 1 Tbsp coconut sugar
- 1/2 tsp vanilla extract
- 1 pinch sea salt (*optional*)

Instructions

1. Preheat oven to 325 degrees F (162 C) and line a baking sheet with parchment paper.
2. To a small mixing bowl, add the pistachios, maple syrup, coconut sugar, vanilla, and salt (optional). Stir well to combine, then transfer the mixture to the prepared baking sheet. Push the pistachios together on the baking sheet so they are all touching (but remain in one even layer). Bake for 10-14 minutes, until fragrant, lightly bubbling, and darkened in color.
3. Remove the pistachios from the oven and let cool **completely** before breaking up into clumps/ individual nuts and roughly chopping, if desired.
4. Enjoy them on salads, ice cream, sorbet, and more! Leftovers keep in an airtight container at room temperature for up to 2-3 weeks or in the freezer for 1 month or longer.

Sl. 2.9. Minimalist baker recept



Sl. 2.10. Minimalist baker opcije recepta

3. FUNKCIONALNI I NEFUNKCIONALNI ZAHTJEVI PREMA APLIKACIJI

Prilikom izrade svake aplikacije potrebno je jasno definirati zahtjeve koje ta aplikacija treba ispunjavati. Zahtjevi aplikacije se dijele na funkcionalne i nefunkcionalne. Zahtjevi su ključni elementi u razvoju web aplikacije. Oni jasno definiraju što aplikacija treba raditi i kako treba funkcionirati.

3.1. Funkcionalni zahtjevi

Funkcionalni zahtjevi odnose se na točne funkcionalnosti i ponašanja koja aplikacija treba imati. Funkcionalni zahtjevi definiraju „što“ aplikacija treba raditi. Kod aplikacije za recepte i kuhanje osnovi funkcionalni zahtjevi su sljedeći:

- Autentikacija i registracija korisnika: Aplikacija omogućuje korisnicima stvaranje njihovog osobnog račun. Na taj način je aplikacija personalizirana za svakog korisnika. Svaki korisnik ima podatke sačuvane o svom računu. Aplikacija nudi opciju stvaranja računa upisivanjem osnovnih podataka. Nakon stvaranja računa korisnik se prijavljuje upisivanjem korisničkog imena i zaporke.
- Autorizacija korisnika: svaki korisnik prilikom stvaranja računa dobiva ovlasti USER. Web aplikacija sadrži jedan račun s ovlastima ADMINISTRATOR koji ima pristup svim funkcionalnostima. Viša razina pristupa od USER razine i manja od ADMINISTRATOR razine je ADMIN razina. U aplikaciji može biti više ADMIN i USER korisnika.
- Upravljanje korisničkim profilima: svakom korisniku je omogućen pregled i promjena osobnih podataka. Korisnici mogu obrisati svoje korisničke račune.
- Stvaranje recepta: korisniku je omogućeno stvaranja recepata. Korisnik dodaje sastojke u recepte odabirući sastojak, količinu i mjernu jedinicu.
- Brisanje i ažuriranje recepata: korisnik može obrisati svoje recepte te im može promijeniti sastojke ili naziv.
- Stvaranje sastojaka: korisnik može stvoriti nove sastojke, tako postavljajući im jedinstveno ime te ih postavi u kategoriju.
- Brisanje i ažuriranje sastojaka: ovoj funkcionalnosti imaju pristup korisnici s ADMIN te ADMINISTRATOR privilegijama. Korisnici mogu obrisati bilo koji sastojak te mu mogu promijeniti naziv i kategoriju.

- Stvaranje komentara: svaki korisnik ima mogućnost postavljanja komentara na bilo koji recept, tako što se odlazi na stranicu toga recepta, odabire gumb za komentiranje te unosi naziv i tekst komentara. Komentari na nekome receptu su vidljivi svima.
- Brisanje i ažuriranje komentara: korisnici su u mogućnosti promijeniti svoj komentar ili ga u potpunosti obrisati.

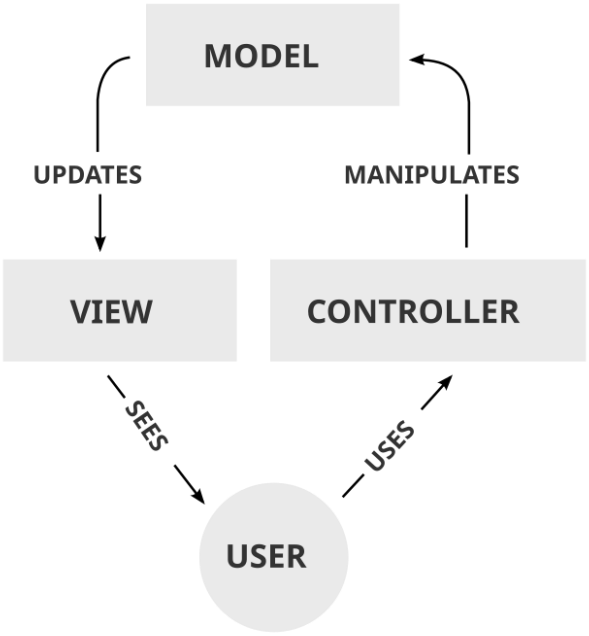
3.2. Nefunkcionalni zahtjevi

Nefunkcionalni zahtjevi odnose se na karakteristike i ograničenja sustava, odnosno „kako“ sustav treba raditi. Kod aplikacije za recepte i kuhanje nefunkcionalni zahtjevi su sljedeći:

- Sigurnost: sve zaporke koje se spremaju trebaju biti kriptirane da ih programer ili bilo tko drugi s pristupom bazi podataka ne može vidjeti njihovu pravu vrijednost
- Validacija: svi podaci koji dolaze u aplikaciju trebaju biti na ispravan način provjereni, Potrebno je postaviti provjere zaporke, praznih polja, duljine unosa ostale moguće validacije.
- Testiranje: za sve metode koje se bave logikom trebaju postojati dva tipa testa. Jedan test koji prolazi i jedan test koji pada.
- Upotrebljivost: grafički dizajn treba biti jednostavan i korisnički intuitivan. Novi korisnici trebaju biti u mogućnosti jednostavno razumjeti funkcionalnosti aplikacije.

3.3. Arhitektura aplikacije

Najčešće korišteni obrazac za rješavanje prethodno navedenih zahtjeva je MVC arhitektura. *Model-View-Controller* je obrazac koji se često koristi u razvoju web aplikacija. MVC obrazac omogućava razdvajanje logike aplikacije u različite slojeve. Slojevi u koje MVC obrazac razdvaja logiku su sljedeći: sloj modela (engl. *Model*), sloj pogleda (engl. *View*) i sloj upravljača (engl. *Controller*). Taj način razdvajanja logike omogućuje brži razvoj aplikacije, jednostavnije testiranje i otkrivanje grešaka, omogućuje jednostavniji rad u timovima te jednostavnost prilikom proširenja aplikacije. Prikaz MVC obrasca se nalazi na 3.1.



Slika 3.1. MVC arhitektura

3.3.1. Model

Model predstavlja podatke i poslovnu logiku aplikacije. Model upravlja podacima bilo da su oni pohranjeni u bazi podataka, dolaze od korisnika ili se obrađuju unutar aplikacije. U sloju modela najčešće se pronalaze servisi, entiteti i repozitoriji. Repozitorij je sučelje koji upravlja direktnom interakcijom s bazom podataka. Tipično se koristi *Spring Data JPA* koji olakšava osnovne operacije nad podacima. Entiteti su klase koje predstavljaju tablice koje se nalaze u bazi podataka. Oni služe za povezivanje podataka koji se nalaze u bazi podataka s klasama koje se nalaze u programskom kodu. Slojevi servisa su implementacije sučelja koja sadrže svu logiku aplikacije i rada s podacima. Unutar servisa se nalazi poslovna logika za rukovanje podacima. Servisi često uključuju u validaciju te razne provjere i obradu podataka. Logika za rad s provjerom i validacijom može biti izdvojena u posebne klase.

3.3.2. Pogled

Pogled je sloj koji predstavlja korisničko sučelje aplikacije. On je zadužen za prikazivanje podataka korisniku te primanje podataka od korisnika. Ovaj sloj se nekada može ukloniti ako se radi o RESTful (engl. *REpresentational State Transfer*) aplikacijama. Takve aplikacije nemaju korisnički prikaz podataka nego obrađuju podatke te vraćaju odgovor u obliku JSON (engl.

JavaScript Object Notation) ili XML (engl. *eXtensible Markup Language*) datoteka. Ako je potreban prikaz podataka tada se mogu koristiti razne tehnologije za prikaz. Jedna od najpopularnijih je *thymeleaf*. *Thymeleaf* služi za izradu dinamičnih web stranica. On se jednom naredbom uključuje u HTML (engl. *HyperText Markup Language*) dokumente gdje se onda koristi posebna sintaksa za njegove mogućnosti.

3.3.3. Upravljač

Upravljač je sloj koji posreduje između sloja modela i sloja pogleda. On je zadužen za primanje korisničkih zahtjeva, njihovu obradu korištenjem sloja servisa. Vraća pogled kao odgovor na korisnički zahtjev. Upravljači se predstavljaju klasama pomoću posebnih anotacija. Postoje dvije vrste upravljača. MVC upravljači su upravljači koji primaju podatke i vraćaju poglede s podacima koji su potrebni. REST upravljači su upravljači koji primaju podatke i vraćaju objekte kao odgovor koji se zatim prebacuju u JSON ili XML oblik. Za pristup upravljačima koriste se URL-ovi (engl. *Uniform Resource Locator*). Koristi se anotacija @RequestMapping(‘/user’) s URL-om kojem se pristupa u zagradi. Na taj način upravljač zna kojoj metodi želi korisnik pristupiti, to jest koju akciju on želi aktivirati.

4. OPIS POSTUPKA IZRADE APLIKACIJE

Kompleksne web aplikacije se grade postepeno. Prvi koraci su dizajniranje baze podataka i određivanje entiteta u bazi. Ovom koraku je potrebno posvetiti najviše vremena jer su promjene na bazi 'skupe'. To znači ako dođe do greške u bazi kasnije ih možda neće biti moguće popraviti ili će promjene povlačiti još niz drugi programskih promjena. Kako bi se baza podataka dobro dizajnirala potrebno je analizirati zahtjeve korisnika. Potrebno je odrediti koje podatke korisnik želi čuvati, na koji način trebaju biti povezani te koliko će tih podataka biti. Potrebno je paziti prilikom stvaranja baze na zauzeće memorije.

Idući korak u izradi web aplikacije je izrada logike aplikacije te obrada podataka koji stižu u aplikaciju. Kod ovoga koraka je potrebno pripaziti kako bi aplikacija ispunjavala sve funkcionalne zahtjeve koji su zadani. U ovom koraku osim dizajniranja i izrade same logike aplikacije potrebno je i napisati testove te svu potrebnu dokumentaciju koja se piše u izvornom kodu. Pisanje kvalitetnih testova je ključna stvar prilikom izrade bilo koje aplikacije. Testovi kasnije omogućuju konzistentnost aplikacije. Moguće ih je pokrenuti nakon dodavanja ili uklanjanja funkcionalnosti kako ne bi došlo do oštećenja dijela aplikacije koji ostaje netaknut.

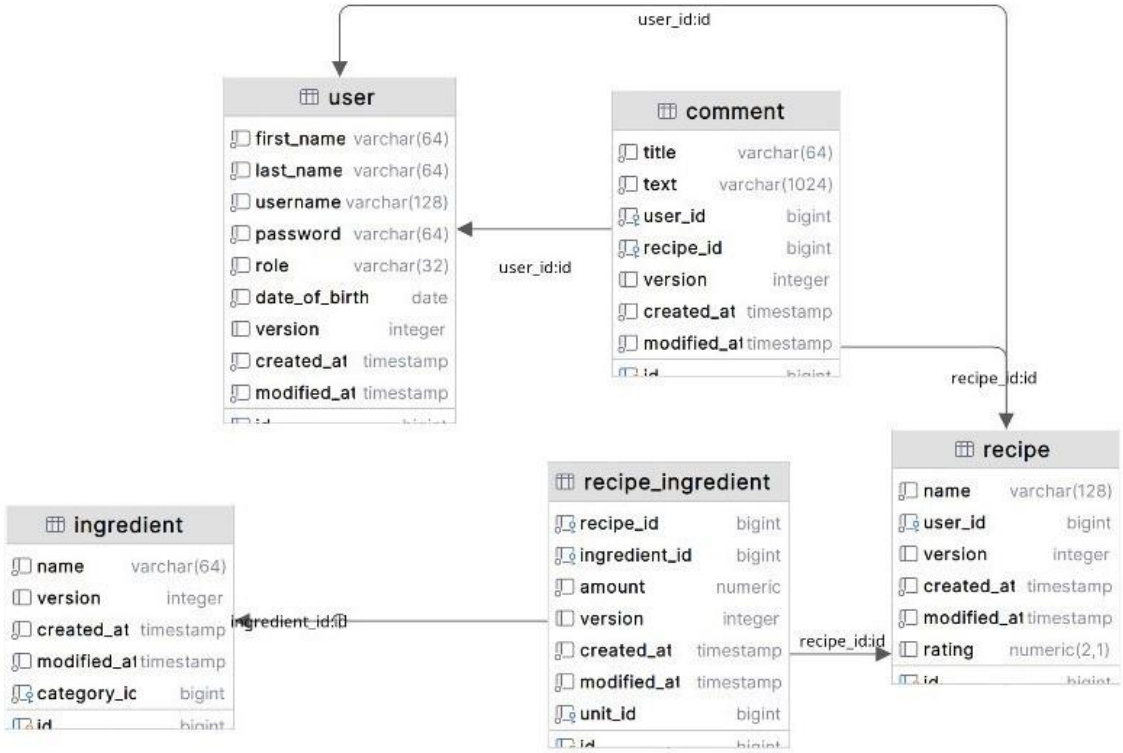
Treći korak je izrada pristupnih točaka aplikaciji. Kako bi korisnik mogao pristupiti logici sustava, potrebno je stvoriti URL-ove koji će služiti kao pristupne točke aplikacije. U ovom sloju se radi potpuna obrada podataka. Potrebno je odrediti gdje će se korisnik usmjeravati nakon koje radnje. Ovaj sloj koristi ugrađenu validaciju. Naposljetku je potrebno stvoriti korisničko sučelje. Najjednostavniji način te način koji je ugrađen u razvojno okruženje *IntelliJ* je *thymeleaf*.

4.1. Dizajn baze podataka

Na 4.1 se nalazi grafički dijagram baze podataka za web aplikaciju za recepte i za kuhanje. Na slici se vidi 5 tablica, svaka od tih tablica je predstavljena jednim entitetom. Svaka veza u tablici je veza 1 na više. Jedina veza koja nije veza 1 na više je veza između tablice *recipe* i tablice *ingredient*, njihova veza je više na više.

Pored svakog atributa u tablici se nalazi se tip podatka kojim je spremljen taj atribut u bazi. Tipovi podataka kojima se spremaju podaci u bazi podataka nemaju jednake nazive kao tipovi podataka u Java programskom jeziku. Primjer je podatak koji je spremljen u bazi s tipom *varchar*(X). *Varchar* je u Javi predstavljen kao *string*, a X u zagradi označava maksimalnu duljinu podataka koji se može pohraniti.

Veze između tablica su grafički prikazane strelicama. Smjer strelice označava koja tablica je „povezana“ na koju tablicu. Smjer strelice govori o relaciji između dvije tablice. Ako netko obriše nekoga korisnika (*user*-a) tada će se obrisati i svi recepti koji su povezani na tog korisnika. Ako se obriše recept tada se neće obrisati i korisnik na kojega je taj recept povezan. Na strelici je prikazano koji atribut je spojen s kojim. U relaciji *recipe – user* povezani su atributi *user_id* u tablici *recipe* i atribut *id* u tablici *user*.



Slika 4.1. Schema baze podataka

4.2. Razvoj poslužiteljskog dijela aplikacije

U poslužiteljskom djelu aplikacije, sustav se povezuje na bazu, pohranjuje podatke, sadrži logiku funkcionalnih i nefunkcionalnih zahtjeva, testira funkcionalnosti, dokumentira te validira i provjerava podataka.

4.2.1. Povezivanje klase s tablicama

Nakon dizajniranja baze podataka, potrebno je napisati klase koje će predstavljati sve entitete. Sve tablice imaju neke zajedničke atribute, poput:

- *id* – primarni ključ, tip podatka je *Long*
- *version* – inačica unosa, tip podatka je *Integer*
- *created_at* – datum i vrijeme stvaranja unosa, tip podatka je *Instant*, koje služi za prikaz vremena
- *modified_at* – datum i vrijeme posljednje promjene na unosu, tip podatka je kao i kod *created_at* atributa

Potrebno je stvoriti klasu koja će ujedini i brinuti se o svim ovim svojstvima svih tablica. Apstraktna klasa na 4.2 pod nazivom *BaseEntity* je najčešći izbor. Sadrži zajedničke atribute tablica s pravilnim anotacijama. Nasljeđivanjem ove klase entitet poprima sve ove atribute te ih može proširiti koliko je potrebno. U 4.1 je popis i objašnjenje svih anotacija korištenih za rad s osnovnim entitetom.

```
12 inheritors   luka.jurkic
@Data
@MappedSuperclass
@SuperBuilder
@NoArgsConstructor
@AllArgsConstructor
@EntityListeners(AuditingEntityListener.class)
public abstract class BaseEntity {

    @Access(AccessType.PROPERTY)
    @Id
    @Column(name = "id")
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    @Version
    @Column(name = "version")
    private Integer version;

    @CreatedDate
    @Column(name = "created_at")
    private Instant createdAt;

    @LastModifiedDate
    @Column(name = "modified_at")
    private Instant modifiedAt;
}
```

Slika 4.2. Bazni entitet

Tablica 4.2. Anotacije klasa entiteta

Ime anotacije	Opis anotacije
@Data	Objedinjuje nekoliko anotacija: @Getter, @Setter, @ToString, @RequiredArgsConstructor i @EqualsAndHashCode
@MappedSuperclass	Služi za entitete koji se nasljeđuju i nemaju svoju tablicu
@SuperBuilder	Stvara tzv. 'graditeljski' aspekt klasi, radi bolje s klasam koje se nasljeđuju
@EntityListeners	Definira klasu kako bi mogla biti entitet ili mapirana superklasa
@Access	Definira način na koji se gleda određeno polje, moguće opcije: PROPERTY i FIELD
@Id	Označava atribut kao primarni ključ
@GeneratedValue	Definira na koji se način primarni ključ mijenja
@Version	Automatski povećava atribut prilikom svake promjene unosa
@CreatedDate	Datum koji se postavlja automatski prilikom stvranaj unosa
@LastModifiedDate	Datum koji se automatski mijenja prilikom svake promjene unosa

Na 4.3 je klasa *UserEntity* koja služi za spajanje na tablicu *user* te nasljeđuje apstraktnu klasu *BaseEntity*. Kako bi klasa postala entitet potrebno je dodati dvije anotacije *„@Entity“* i *„@Table(name = „tableName“)“.* Pomoću prve anotacije *Spring* zna da se radi o klasi koja je entitet. Pomoću druge anotacije *Spring* zna na tablicu s kojim imenom je potrebno povezati tu klasu. Ostale četiri anotacije služe za smanjivanje ponavljajućeg koda, njihovi opisi se nalaze na 4.2.

Tablica 4.1. Opis anotacija user entiteta

Ime anotacije	Opis anotacije
@Getter	Stvara metoda dohvaćanja za sve atribute
@Setter	Stvara metode postavljanja za sve atribute
@NoArgsContructor	Stvara konstruktor koji ne prima argumente
@AllArgsConstructor	Stvara konstruktor koji prima sve argumente

Anotacije @Column se koristi kod jednostavnih atributa. Ta anotacija govori *Springu* kojem atributu u toj tablici treba povezati atribut iznad kojeg stoji anotacija. Kod atributa *role*, postoji anotaciju @Enumerated koja govori o kojem tipu podataka se radi te je potrebno odrediti način na koji se dohvaća *enum*. *Enum* se može gledati kao tekst ili kao brojčana vrijednost. *Enum* koji se koristi za korisničke privilegije se nalazi na 4.4. Tablica recepata sadrži poveznicu na korisnika, *spring* nudi opciju kako bi se entiteta *user* inicijalizira lista recepata, na taj način je ubrzana pretraga baze. Kada bi bilo potrebno dohvatiti sve recepte nekog korisnika, bilo bi potrebno pretražiti cijelu bazu recepata, na ovaj način je potrebno dohvatiti korisnika s tim imenom. Unutar anotacije @OneToMany se postavlja nekoliko vrijednosti. Prva vrijednost je ime atributa koji se nalazi u tablici recepata koja se vidi na 4.5. Duga vrijednost je način dohvaćanja tih vrijednosti. Najčešća vrijednost za taj parametar je LAZY, to označava dohvaćanje podataka tek onda kada su potrebni, te neće biti konstantno u memoriji aplikacije. Posljednji parametar je *orphanRemoval*, taj parametar se postavlja na *true* onda kada je potrebno obrisati sve entitete iz te liste u slučaju brisanja roditeljskog entiteta. Primjerice brisanje korisnika, ako je taj parametar postavljen na *true* tada će se obrisati i svi recepti povezani s tim korisnikom. Anotacija @OneToMany i lista u klasi tog entiteta nisu potrebni za rad, ali znatno olakšavaju i ubrzavaju rad aplikacije.

```
luka.jurkic *
19  @Entity
20  @Table(name = "recipe")
21  @Getter
22  @Setter
23  @NoArgsConstructor
24  @AllArgsConstructor
25  public class RecipeEntity extends BaseEntity {
26
27      @Column(name = "name")
28      private String name;
29
30      @ToString.Exclude
31      @ManyToOne
32      @JoinColumn(name = "user_id", foreignKey = @ForeignKey(name = "fk_recipe_user"))
33      private UserEntity user;
34
35      @OneToMany(mappedBy = "recipe", fetch = FetchType.LAZY, orphanRemoval = true)
36      private List<CommentEntity> comments;
37
38      @OneToMany(mappedBy = "recipe", fetch = FetchType.LAZY, orphanRemoval = true)
39      private List<RecipeIngredientEntity> ingredients;
40  }
```

Slika 4.3. *Recipe* entitet

```java
luka.jurkic *
@Entity
@Table(name = "user")
@Getter
@Setter
@NoArgsConstructor
@AllArgsConstructor
public class UserEntity extends BaseEntity {


    @Column(name = "first_name")
    private String firstName;

    @Column(name = "last_name")
    private String lastName;

    @Column(name = "username")
    private String username;

    @Column(name = "password")
    private String password;

    @Enumerated(EnumType.STRING)
    @Column(name = "role")
    private UserRole role;

    @Column(name = "date_of_birth")
    private LocalDate dateOfBirth;

    @OneToMany(mappedBy = "user", fetch = FetchType.LAZY, orphanRemoval = true)
    private List<RecipeEntity> recipes;

    @OneToMany(mappedBy = "user", fetch = FetchType.LAZY, orphanRemoval = true)
    private List<CommentEntity> comments;
}
```

Slika 4.4. *User* entitet

```java
luka.jurkic
public enum UserRole {
    ADMIN, USER, ADMINISTRATOR
}
```

Slika 4.5. *User enum*

Na 4.5 je klasa koja predstavlja tablicu recepata u bazi. Većina anotacija je ista, osim anotacija iznad atributa *UserEntity*. 3 nove anotacije koje se pojavljuju su:

- @ToString.Exclude – govori *Springu* ponašanje ako u slučaju brisanja objekta ove klase, da se ovaj atribut preskoči.

- @ManyToOne – anotacija koja služi za povezivanje dva atributa vezom više na jedan.
- @JoinColumn – povezuje tablicu korisnika s tablicom recepata, prima nekoliko parametara, od kojih su neki: *mappedBy*, koji govori koji je naziv atributa u tablici s kojim je tablica povezana na drugu tablicu i *foreignKey*, postavlja naziv stranog ključa s kojim je stvorena veza između dvije tablice.

4.2.2. Migracijske skripte

Iako postoje napisane klase kojima se povezuju klase s tablicama, potrebno je napisati SQL kod koji će stvoriti tablice u bazi. Stvaranje tablica se radi preko migracijskih skripti, koje se pišu kako bi se mogla pratiti konzistentnost u bazi te zbog praćenja inačice baze. Na 4.6 se nalazi primjer koda migracijske skripte kojom se stvaraju tablice za korisnike i recepte. Prilikom kreiranja tablica naziv tablica se piše bez navodnika, osim u rijetkim slučajevima kao što je kreiranje tablice korisnika (*user*). *Spring* ima ugrađeni entitet *user* te kako bi ga bilo moguće ponovno stvoriti potrebno je staviti naziv tablice pod navodnike.

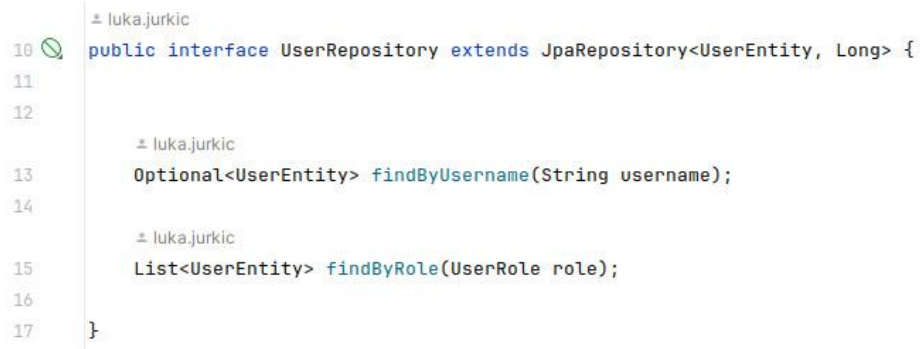
```
13  CREATE TABLE "user"
14  (
15      id            bigserial    NOT NULL PRIMARY KEY,
16      first_name    VARCHAR(64)  NOT NULL,
17      last_name     VARCHAR(64)  NOT NULL,
18      username      VARCHAR(128) NOT NULL,
19      password      VARCHAR(64)  NOT NULL,
20      role          VARCHAR(32)  NOT NULL,
21      date_of_birth DATE         NOT NULL,
22      version       integer,
23      created_at    timestamp    NOT NULL,
24      modified_at   timestamp    NOT NULL
25  );
26
27  CREATE TABLE recipe
28  (
29      id          bigserial    NOT NULL PRIMARY KEY,
30      name        varchar(128) NOT NULL,
31      user_id     bigserial    NOT NULL,
32      version     integer,
33      created_at  timestamp    NOT NULL,
34      modified_at timestamp    NOT NULL,
35      CONSTRAINT fk_recipe_user FOREIGN KEY (user_id) REFERENCES "user" (id)
36  );
```

Slika 4.6. SQL kod za tablice user i *recipe*

4.2.3. Repozitorij

Za promjenu podataka u tablicama u bazi se koriste napisani entiteti u obliku klasa. A te promjene se rade pomoću SQL upita prema tablici. Za većinu entiteta ti upiti izgledaju jednako, uz promjenu tipa entiteta. *Spring* nudi elegantno rješenje u obliku JPA repozitorija. Prema [2],

JPA repozitorij je sučelje koji se koristi bi se smanjila količina koda koja bi se inače trebala pisati kada bi se pristupalo bazi.



Slika 4.7. User JPA repozitorij

Na 4.7, je primjer jednog takvog sučelja. Sučelje pod nazivom *UserRepository* je prošireno sučeljem *JpaRepository* kojem su predana dva parametra koje pobliže određuju kako će se sučelje ponašati. Prvi parametar govori o kojem se entitetu radi, drugi parametar govori koji je tip podatka za primarni ključ koji je postavljen u tom entitetu. Korištenjem instance *UserRepository*-a moguće je izvršavati sve osnovne operacije nad tablicom *User*. Dvije metode koje se nalaze unutar sučelja su metode proširenja. Takve metode piše programer te služe kako bi se proširile funkcionalnosti repozitorija. Programeru je dovoljno napisati ime funkcije, povratni tip i parametre koje prima, a *Spring* će ju sam implementirati. Na ovaj način je omogućeno jednostavno pisanje funkcija koje rade s bazom bez pisanja SQL (engl. *Structured Query Language*) upita.

4.2.4. Klase za prijenos podataka

Sigurnost je nefunkcionalan zahtjev, ali se nalazi na vrlo visokoj razini potrebe. Sigurnost ne označava kvalitetne zaporke i dobro spremljene iste. Sigurnost također označava kolikoj količini podataka korisnik ima pristup. Prilikom kreiranja korisnika se unosi više podataka nego što korisnik treba vidjeti. Kako bi se ograničila razina podataka koje korisnik vidi potrebno je selektivno odabrati podaci koje se smiju prikazati korisniku. Podaci koji su nam potrebni od korisnika, podaci koji se spremaju te podaci koji se vraćaju nisu isti podaci. Razlika može biti u količini prikazanih podataka ili tipu podataka. Zato se dodaje podsloj koji služi za selektiranje podataka koji se smiju prikazati. Stvara se niz klasa koje sve trebaju biti osmišljene u skladu sa zahtjevima korisnika.

Klase za prijenos podataka se dijele na dvije skupine: DTO (eng. *Data Transfer Object*) i *Request* klase. DTO je klasa koja sadrži sve podatke koje korisnik može i smije vidjeti. *Request* klasa je klasa koja sadrži sve podatke koje korisnik šalje aplikaciji. Često te dvije skupine klasa imaju sličnih atributa pa se ti atributi izvlače u posebne klase pod nazivom *BaseEntity*, ali u ovom slučaju se umjesto riječi *Entity* piše ime entiteta o kojem se stvarno radi.

Na 4.8 je primjer *BaseUser* klase koja objedinjuje sve zajedničko *UserDTO* i *UserRequest* klasama. Prilikom rada s korisničkim podacima, uvijek je potrebno znati korisničko ime, te je ono postavljeno u osnovnu klasu za prijenos podataka.

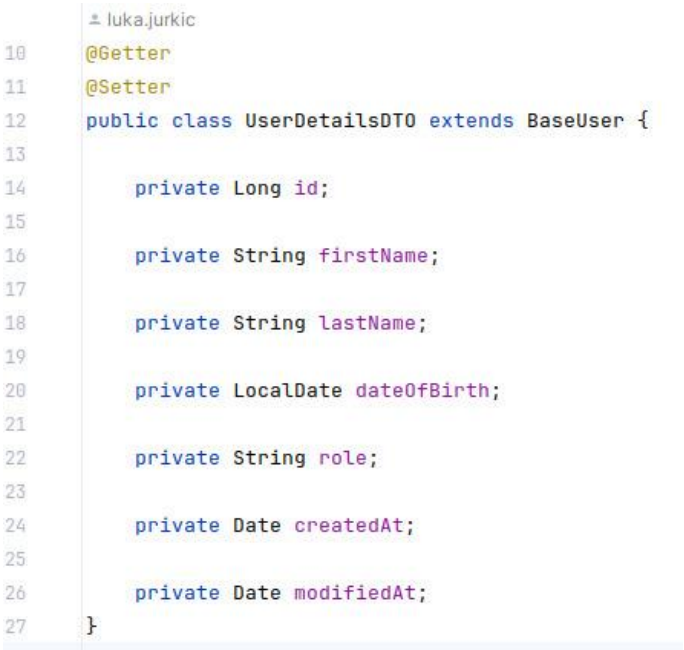
```
4 inheritors   luka.jurkic *
@Data
public class BaseUser {

    private String username;



}
```

Slika 4.8. Osnovna klasa korisnika

Ako se podaci o korisniku prikazuju korisnicima, tada se prikazuju osnovne podaci, njegov *id* i njegovu razinu prava. Ako je korisničke podatke potrebno prikazati vlasniku korisničkog računa tada se šalju svi podaci koji su spremljeni su bazi o tom korisniku. Potrebno je stvoriti dvije različite DTO klase, primjer jednostavnije je na 4.9, a detaljniju DTO klasu se nalazi na 4.10.

```
luka.jurkic
@Getter
@Setter
public class UserDTO extends BaseUser {

    private Long id;

    private String role;
}
```

Slika 4.9. DTO klasa korisnika



Slika 4.10 Detaljna DTO klasa korisnika

Prilikom rada s korisničkim računima potrebno je konstantno obraćati pozornost na sigurnost. Kada se dohvaćaju podaci od korisnika za kreiranje novog računa i ažuriranje podataka na postojećem računu neće se dohvaćati isti podaci. Razlika između toga dvoje je zaporka. Prilikom kreiranja računa korisnik unosi zaporku koju želi, prilikom ažuriranja podataka preporuka je odvojiti ažuriranje ostalih podataka od ažuriranja zaporke. Potrebno je stvoriti dvije *Request* klase koje se razlikuju u zaporki. Na 4.11 je klasa koja ne sadrži zaporku i ona će se koristiti za ažuriranje podataka korisnika. Dok na 4.12 *CreateRequest* klasa nasljeđuje prethodnu klasu i dodaje joj zaporku, te na taj način je ostvarena razdvojenost između klasa za kreiranje i ažuriranje. Kod *Request* klasa nigdje ne trebaju postojati atributi: *id*, *createdAt*, *modifiedAt* i *version*, te se vrijednosti automatski postavljaju prilikom spremanja objekta u bazu.

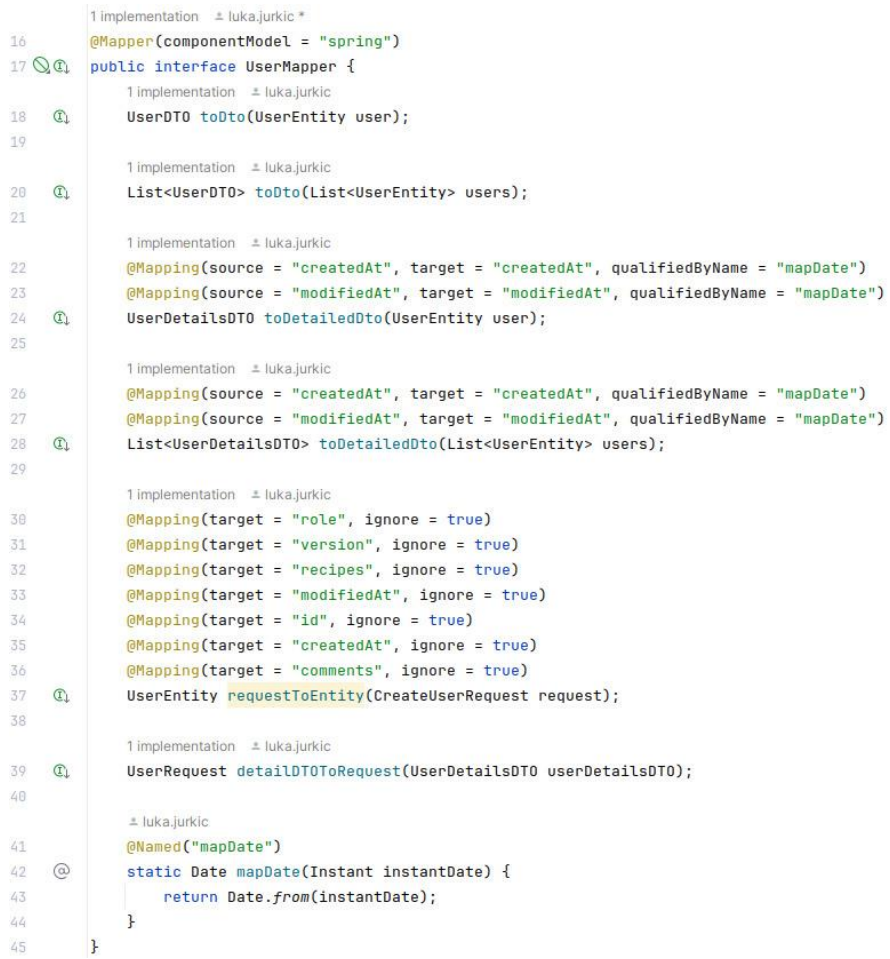
Kada su klase za prijenos podataka dobro postavljene tada je potrebno obratiti pažnju kako klase čiji se podaci šalju prema korisniku završavaju na DTO, a podaci klasa koji se dohvaćaju od korisnika završavaju na *Request*. Na taj način raste sigurnost aplikacije jer nepotrebni podaci neće biti prikazani.

Slika 4.11. *Request* klasa korisnika

Slika 4.12. *Request* klasa za stvaranje korisnika

4.2.5. Mapperi

Web aplikacije koriste jako puno klasa za prijenos podataka. Tokom rada s podacima dolazi do čestih konverzija podataka iz jednog tipa u drugi. Sučelja koja rade te pretvorbe zovu se *mapperi*. Prema [3], za stvaranje *mappera* od običnog sučelja potrebno je dodati anotaciju @Mapper te naznačiti *componentModel 'spring'*. Na 4.13 se nalazi *mapper* za korisničke klase za prijenos podataka. Slično kao i kod JPA repozitorija gdje je *spring* implementirao sve metode, to radi i ovdje. Unutar tijela sučelja je potrebno napisati ime *mapper* metode, povratni tip te parametar koji prima, te ako je moguće *spring* implementira dani *mapper*.



Slika 4.13. User *mapper* klasa

Prilikom pretvaranja iz jednog tipa podatka u drugi može doći do nekompatibilnosti tipa podatka, do nedostajućih vrijednosti ili do različitih imena atributa. Tada je potrebno eksplicitno naglasiti *mapper* metodi način rješavanja toga problema. Nekada je to rješenje kraćeg tipa, a nekad je potrebno napisati dodatnu funkciju.

Na 4.13 kod metoda *toDetailedDto* dodana je anotacija @Mapping. Ta anotacija pobliže opisuje kako metoda treba raditi s nekim atributima. U svakom slučaju kod tih metoda postoji *source* atribut u zagradi anotacije. Ta vrijednost govori metodi koji je izvor za pretvorbu, taj naziv treba odgovarati nekom od atributa klase objekta koji se prima kao parametar u metodu. *Target* je atribut govori u koji atribut klase objekta povratnog tipa će se pretvorena vrijednost postaviti. Ovaj naziv treba odgovarati nekom od atributa iz klase koja je povratni tip. Posljednja vrijednost govori ime metode koja se koristi za pretvorbu iz jedne u drugu vrijednost, u ovom slučaju je naziv *mapDate*, ime metode koja se nalazi na dnu slike. Ako se radi o slučaju nedostajućih vrijednosti, tada se neki od atributa iz izvora mogu ignorirati, kao što je na primjeru anotacija

iznad metode *requestToEntity*. Razlog tomu je što se neke vrijednosti automatski generiraju, a neke se tek kasnije postavljaju ili je potrebna dodatna validacija prije dohvaćanja.

4.2.6. Prilagođene iznimke

Iznimke (engl. *Exceptions*) su ključni koncepti u programiranju i razvoj web aplikacija. Omogućuju rukovanje neočekivanim ili neželjenim situacijama koje se mogu pojaviti tokom izvršavanja programa. Iznimke omogućuju programerima izdvajanje i obradu greška na način koji ne prekida rad aplikacija ili bar daje korisniku informativnu povratnu poruku o problemu.

Prilagođene (engl. *Custom*) iznimke u *Spring Boot* Javi su korisnički definirane klase koje omogućuju preciznije opisivanje problema ili situacije koje se mogu pojaviti u aplikaciji. Prilagođene iznimke nasljeđuju postojeće klase iznimaka, te se mogu koristiti za rukovanje u slučajevima koji nisu pokriveni ugrađenim Java iznimkama.

Za korištenje prilagođenih iznimaka izdvajaju se 3 glavna razloga:

- Bolja čitljivost i razumljivost koda – prilikom čitanja i dokumentiranja koda jednostavnije se kod opisuje
- Specifična obrada grešaka – greške do koji dođe je moguće detaljnije opisati i otkriti
- Enkapsulacija logike grešaka – klase iznimaka zapakiraju detalje o greškama i isporučuju ih gdje je potrebno

Prilagođene iznimke su snažan alat u *Sping* Javi koji omogućuje programerima razvijanje robusne, intuitivne i lako održive aplikacije s preciznim rukovanjem pogrešaka. Za ispravnu implementaciju prilagođenih iznimaka potrebno je kreirati niz manjih klasa. Na taj način dodavanje novih iznimaka postaje jednostavno i brzo. Na slikama se nalaze klase koje je potrebno implementirati za ispravan rad prilagođenih iznimki. U 4.3 se nalazi popis slika s objašnjenjima.

Tablica 4.3. Objašnjenja slika prilagođenih iznimaka

Oznaka slike	Objašnjenje slike
4.14	Apstraktna klasa koju će naslijediti sve prilagođene iznimke, nasljeđuje glavnu iznimku u Javi pod nazivom *RuntimeException*, te implementira osnovne konstruktore
4.15	Klasa koja služi za enkapsulaciju podataka grešci koja se podiže
4.17	Klasa koja hvata sve iznimke koje se podignu prilikom izvođenja programa. Dvije metode razdvajaju prilagođene iznimke od ugrađenih. Dodaje mogućnost prilagođavanja ugrađenih iznimki.
4.16	Funkcionalno sučelje koje služi za držanje naziva grešaka
4.20	Nazivi grešaka se podižu u Json obliku te ih je potrebno deserializirati. Ova klasa prepisuje metode koje su potrebne za obavljanje tog zadatka.
4.19	Implementira sučelje za nazive grešaka

```
10 inheritors   luka.jurkic
@Getter
@Setter
public abstract class ServiceException extends RuntimeException {

    protected final ErrorKey errorKey;
    protected  final HttpStatus httpStatus;

    luka.jurkic
    protected ServiceException(ErrorKey errorKey, HttpStatus httpStatus) {
        this.errorKey = errorKey;
        this.httpStatus = httpStatus;
    }

    luka.jurkic
    protected ServiceException(ErrorKey errorKey, HttpStatus httpStatus, String message) {
        super(message);
        this.errorKey = errorKey;
        this.httpStatus = httpStatus;
    }
}
```

Slika 4.14. Prilagođena iznimka servisa

```
luka.jurkic
@Data
public class ErrorResponse {
    private final String errorKey;
    private final String message;
}
```

Slika 4.15. Klasa odgovara na grešku

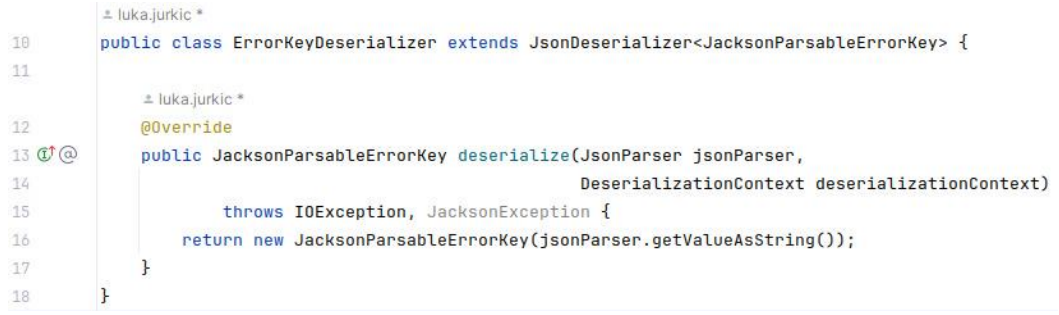
```
11 implementations  luka.jurkic
@FunctionalInterface
@JsonDeserialize(using = ErrorKeyDeserializer.class)
public interface ErrorKey {

    11 implementations  luka.jurkic
    String getKey();
}
```

Slika 4.16. Klasa ključa greške

```
luka.jurkic
@ControllerAdvice
public class RecipeWorldExceptionHandler {

    luka.jurkic
    @ExceptionHandler(ServiceException.class)
    public ResponseEntity<ErrorResponse> handle(ServiceException exception) {
        ErrorResponse errorResponse = new ErrorResponse(exception.getErrorKey().getKey(), exception.getMessage());
        return ResponseEntity.status(exception.getHttpStatus()).body(errorResponse);
    }

    luka.jurkic
    @ExceptionHandler(Exception.class)
    public ResponseEntity<ErrorResponse> handle(Exception exception) {

        if (exception instanceof MethodArgumentNotValidException || exception instanceof IllegalArgumentException) {
            ErrorResponse errorResponse = new ErrorResponse( errorKey: "Bad Request", exception.getMessage());
            return ResponseEntity.status(HttpStatus.BAD_REQUEST).body(errorResponse);
        } if (exception instanceof AccessDeniedException) {
            ErrorResponse errorResponse = new ErrorResponse( errorKey: "Access Denied", exception.getMessage());
            return ResponseEntity.status(HttpStatus.UNAUTHORIZED).body(errorResponse);
        } else {
            ErrorResponse errorResponse = new ErrorResponse( errorKey: "Server Error", exception.getMessage());
            return ResponseEntity.status(HttpStatus.INTERNAL_SERVER_ERROR).body(errorResponse);
        }
    }
}
```

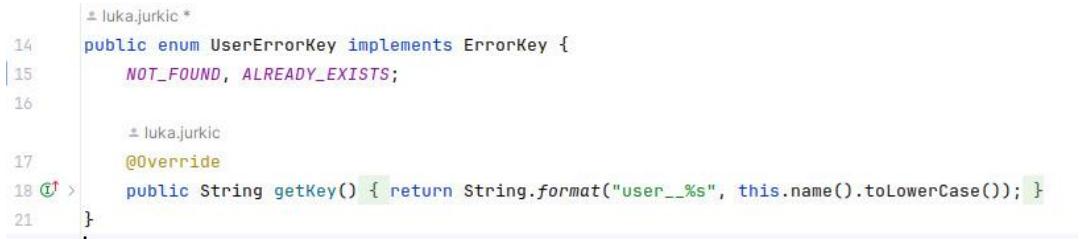
Slika 4.17. Klasa generalnog rukovatelja iznimkama

Slika 4.19. Klasa parsera



Slika 4.20. Klasa deserialajzera

Kada se stvaraju prilagođene greške tada se one stvaraju u obliku klasa. Prilagođene greške se mogu postaviti na način da jedna klasa sadrži sve prilagođene greške jednog entiteta. Za svaki entitet su potrebni *enum* i klasa, *enum* koji će sadržavati nazive greška, te klasa koja će sadržavati statične metode koje se pozivaju kada se podižu greške u kodu. Na 4.18 je *enum* koji sadrži nazive grešaka za slučajeve kada korisnik nije pronađen u bazi te kada korisnik već postoji u bazi podataka. *Enum* prepisuje metodu koja služi za dohvaćanje toga naziva greške u određenom formatu koji je jedinstven za prilagođene greške ovog entiteta.



Slika 4.18. *Enum* naziva korisničkih grešaka

Na 4.21 je implementirana prilagođena iznimka za korisnički entitet pod nazivom *UserServiceException* te implementira *ServiceException* i prepisuje potrebne konstruktore. Unutar klase postoje po dvije metode za svaku moguću grešku, jedna sadrži korisničku poruku, a druga sadrži naziv greške, dok se korisnička poruka automatski postavlja na vrijednost *null*. Prilikom podizanja greške dovoljno je pozvati neku od statičnih metoda na klasi *UserServiceExcpetion* te će metoda podići odgovarajuću iznimku s odgovarajućom porukom.

```
luka.jurkic *
8   public class UserServiceException extends ServiceException {
9
        luka.jurkic
10      protected UserServiceException(ErrorKey errorKey, HttpStatus httpStatus) {
11          super(errorKey, httpStatus);
12      }
13
        luka.jurkic
14      protected UserServiceException(ErrorKey errorKey, HttpStatus httpStatus, String message) {
15          super(errorKey, httpStatus, message);
16      }
17
        luka.jurkic
18      public static UserServiceException notFound() {
19          throw new UserServiceException(UserErrorKey.NOT_FOUND, HttpStatus.NOT_FOUND);
20      }
21
        luka.jurkic
22      public static UserServiceException notFound(String message) {
23          throw new UserServiceException(UserErrorKey.NOT_FOUND, HttpStatus.NOT_FOUND, message);
24      }
25
        luka.jurkic
26      public static UserServiceException alreadyExists() {
27          throw new UserServiceException(UserErrorKey.ALREADY_EXISTS, HttpStatus.BAD_REQUEST);
28      }
29
        luka.jurkic
30      public static UserServiceException alreadyExists(String message) {
31          throw new UserServiceException(UserErrorKey.ALREADY_EXISTS, HttpStatus.BAD_REQUEST, message);
32      }
33  }
```

Slika 4.21. Korisničke prilagođene iznimke

Za stvaranje novih prilagođenih iznimki dovoljno je napisati *enum* koji će sadržavati popis mogućih grešaka za taj entitet te treba implementirati sučelje *ErrorKey* te klasu koja će sadržavati metode koje će podizati iznimke koristeći greške iz *enuma*.

4.2.7. Servisi i implementacija

Sloj modela se bazira na servisima. Prema [4], servisi su najčešće sučelja koja definiraju metode koje se bave logikom aplikacije. Ta sučelja programer treba implementirati. Prilikom implementacije koristi se instanca repozitorija koji je potreban kako bi se ispravno rukovalo s podacima. Servisi koriste instance *mappera*, jer se u servisima obavlja pretvorba podataka.

Servisi često koriste klase u kojima je izdvojena logika specifičnih zadataka, takve se klase nazivaju „*utility*“ klase.

4.22 prikazuje primjer korisničkog servisa, ovaj servis je zadužen za rad s podacima korisnika. Servis definira niz metoda koje su prikazane u 4.4 s pripadnim objašnjenjima. Kako bi se ovakav servis mogao koristiti potrebno ga je implementirati. Klase koje implementiraju sučelja se često nazivaju istim imenom kao i sučelje uz sufiks *Impl*.

```
1 implementation  ± luka.jurkic *
137 public interface UserService {
        1 implementation  ± luka.jurkic
138     UserDTO getUser(String username);
        1 implementation  ± luka.jurkic
139     UserDetailsDTO getUserDetails(String username);
        1 implementation  ± luka.jurkic
140     List<UserDTO> getUsers();
        1 implementation  ± luka.jurkic
141     List<UserDetailsDTO> getUsersDetails();
        1 implementation  ± luka.jurkic
142     UserDetailsDTO createUser(CreateUserRequest userRequest);
        1 implementation  ± luka.jurkic
143     void deleteUser(String username);
        1 implementation  ± luka.jurkic
144     UserDetailsDTO updateUser(String username, String password, UserRequest userRequest);
        1 implementation  ± luka.jurkic
145     boolean isValidPassword(String password);
146 }
```

Slika 4.22. Sučelje korisničkog servisa

Tablica 4.4. Opis metoda korisničkog servisa

Naziv metode	Parametri	Opis metode
getUser	Username	Dohvaća osnovne podatke korisnika s predanim korisničkim imenom
getUserDetails	Username	Dohvaća detaljne podatke korisnika s predanim korisničkim imenom
getUsers	-	Dohvaća osnovne podatke svih korisnika u bazi
getUsersDetails	-	Dohvaća detaljne podatke svih korisnika u bazi
createUser	createUserRequest	Stvara novog korisnika s podacima iz instance klase request
deleteUser	Username	Briše korisnika s predanim korisničkim imenom
updateUser	Username, password, userRequest	Mijenja podatke korisnika s korisničkim imenom prema podacima u instanci klase request
isValidPassword	password	Provjerava je li zaporke ispravna

Prilikom implementacije servisa koriste se *mapperi*, repozitorij, prilagođene iznimke te *utility* klase. 4.23 prikazuje instance potrebne za implementaciju sučelja korisničkog servisa. Instance koje se koriste u servisima se često označavaju s ključnom riječi *final*. Varijable se tako označavaju kako ne bi došlo do promjena nad istima. Varijabli koja je postavljena kao *final*, nakon što se jednom postavi, vrijednost se ne može mijenjati.

```
luka.jurkic *
@Service
@Transactional
public class UserServiceImpl implements UserService {

    private final UserRepository userRepository;
    private final UserMapper userMapper;
    private final PasswordEncoder passwordEncoder;
    private final RatingMapper ratingMapper;

    luka.jurkic *
    public UserServiceImpl(UserRepository userRepository, UserMapper userMapper, PasswordEncoder passwordEncoder, RatingMapper ratingMapper) {
        this.userRepository = userRepository;
        this.userMapper = userMapper;
        this.passwordEncoder = passwordEncoder;
        this.ratingMapper = ratingMapper;
    }

```

Slika 4.23. Implementacija servisa

Kako bi *Spring* prepoznao klasu kao servis potrebno je iznad implementacije klase dodati anotaciju @Service. Uz anotaciju servisa potrebna je anotacija @Transactional. Neke metode servisa znaju biti veće, zahtijevaju više vremena za izvođenje te koriste razne resurse. U toku izvođenja programa u bilo kojem trenutku može doći do podizanja iznimke i do neočekivanog rada sustava. Kako bi se zadržala konzistentnost baze podataka dodaje se navedena anotacija. U slučaju prekida rada servisa ona poništava sve nedovršene promjene na bazi, drugim riječima promjene na bazi neće biti vidljive dok one nisu potpune.

4.24 prikazuje implementirane metode sučelja korisničkog servisa čija se objašnjenja nalaze u 4.4. Iznad svake metode se nalazi anotacija @Override koja označava sučeljem definiranu metodu i prepisanu metodu novom implementacijom. Implementacija koristi ugrađene metode repozitorija, kao što su: *save, delete* i *findAll*. To su metode koje JPA repozitorij nudi bez potrebe proširenja. *delete* i *findAll* metode su intuitivne, dok metoda *save* objedinjuje dvije mogućnosti. *Save* metoda objedinjuje mogućnosti stvaranja i ažuriranja unosa. Metoda provjerava postoji li korisnik s *idjem* onoga koji se sprema te ako postoji ažurira podatke, ako ne postoji stvara novi unos i dodjeljuje novi *id*.

```java
luka.jurkic *
@Service
@Transactional
public class UserServiceImpl implements UserService {
    //code omitted for brevity
    luka.jurkic
    @Override
    public UserDTO getUser(String username) { return userMapper.toDto(fetchUser(username)); }
    luka.jurkic
    @Override
    public UserDetailsDTO getUserDetails(String username) { return userMapper.toDetailedDto(fetchUser(username)); }
    luka.jurkic
    @Override
    public List<UserDTO> getUsers() { return userMapper.toDto(userRepository.findAll()); }
    luka.jurkic
    @Override
    public List<UserDetailsDTO> getUsersDetails() { return userMapper.toDetailedDto(userRepository.findAll()); }
    luka.jurkic
    @Override
    public UserDetailsDTO createUser(CreateUserRequest userRequest) {
        checkUniqueness(userRequest);
        UserEntity user = userMapper.requestToEntity(userRequest);
        user.setPassword(passwordEncoder.encode(userRequest.getPassword()));
        user.setRole(UserRole.USER);
        user = userRepository.save(user);
        return userMapper.toDetailedDto(user);
    }
    luka.jurkic
    @Override
    public void deleteUser(String username) { userRepository.delete(fetchUser(username)); }
    luka.jurkic *
    @Override
    public UserDetailsDTO updateUser(String username, String password, UserRequest userRequest) {
        if (isNotValidUser(username, password)) { throw UserServiceException.badCredentials(); }
        UserEntity user = fetchUser(username);
        if (!Objects.equals(username, userRequest.getUsername())) { checkUniqueness(userRequest); }
        user.setUsername(userRequest.getUsername());
        user.setLastName(userRequest.getLastName());
        user.setFirstName(userRequest.getFirstName());
        user.setDateOfBirth(convertToLocalDate(userRequest.getDateOfBirth()));
        return userMapper.toDetailedDto(userRepository.save(user));
    }
    new *
    @Override
    public boolean isValidPassword(String password) {
        return password.matches( regex: "^(?=.*[a-zA-Z])(?=.*[0-9])[a-zA-Z0-9]+$") && password.length() >= 8 && password.length() <= 20;
    }
```

Slika 4.24. Implementacija korisničkog sučelja

Implementacija koristi *utility* klasu *PasswordEncoder*. Instanca te klase prima niz znakova te vraća kodiranu vrijednost tih znakova. Način na koji se kodira je definiran od strane programera. Prije spremanja podataka u bazu bitno je kodirati zaporke kako netko ne bi vidio zaporke koje su spremljene u bazi. Konfiguraciju ove *utility* klase se može vidjeti na 4.25.

Osim *utility* klasa, implementacija koristi dodatne privatne metode. Kod koji se ponavlja se izvlači u privatne metode kako bi se povećala čitljivost koda i olakšalo otkrivanje grešaka. Privatne metode koje koristi implementacija se mogu vidjeti na 4.26.

```java
luka.jurkic
@Configuration
public class PasswordEncoderConfig {

    luka.jurkic
    @Bean
    public PasswordEncoder passwordEncoder() {
        return new BCryptPasswordEncoder(BCryptPasswordEncoder.BCryptVersion.$2A, strength: 8);
    }
}
```

Slika 4.25. Klasa za kodiranje zaporki

```java
luka.jurkic *
@Service
@Transactional
public class UserServiceImpl implements UserService {
    //code omitted for brevity
    new *
    private UserEntity fetchUser(String username) {
        return userRepository.findByUsername(username).orElseThrow(UserServiceException::notFound);
    }
    new *
    private void checkUniqueness(UserRequest request) {
        userRepository.findByUsername(request.getUsername()).ifPresent(user -> {
            throw UserServiceException.alreadyExists("User with username " + user.getUsername() + " already exists.");
        });
    }
    new *
    private LocalDate convertToLocalDate(Date dateToConvert) {
        return new java.sql.Date(dateToConvert.getTime()).toLocalDate();
    }

```

Slika 4.26. Privatne metode sučelja

4.2.8. Validacija atributa

Prema [5], validacija je proces provjere jesu li uneseni podaci od strane korisnika ispravni u skladu s određenim pravilima ili uvjetima. Validacija je ključna prilikom izrade aplikacija kako bi podaci ostali konzistentni te kako bi se smanjio broj nepredviđenih slučajeva u radu sustava.

Validacija može biti jednostavna, poput provjere je li neko polje popunjeno te može biti složenija gdje se provjerava oblik nekog niza znakova, na primjer provjera formata e-pošte. Postoji nekoliko vrsta validacije:

- Klijentska validacija – izvodi se na klijentskoj strani, kod web aplikacija najčešće u web preglednicima, većinom je brža te pruža bržu povratnu informaciju, ali se može zaobići
- Serverska validacija – izvodi se na serveru nakon što se podaci pošalju, sporija je od klijentske validacije, ali je konzistentnija i većinom nezaobilazna

Spring Boot ima visoko razvijen stupanj validacije. Omogućuje niz raznih anotacija koje obavljaju validaciju umjesto programera. Programer dobije povratnu informaciju o uspješnoj ili neuspjeloj validaciji. Ulazna točka podataka na server su *request* klase. Kako bi se osigurala podatke potrebno ih je validirati prije nego što se krenu obrađivati, zato se anotacije za validaciju postavljaju iznad atributa klasa *request*. Na 4.27 je *user request* s dodanom potrebnom validacijom. Validacija se najčešće dodaje prema pravilima koji se definiraju prilikom kreiranja tablica. Ako se za polje prilikom kreiranja baze stavi maksimalna duljina 64 znaka, tada se i za validaciju na atributu klase postavlja maksimalna duljina od 64 znaka. U 4.5, uz anotacije s 4.27, je popis i objašnjenja čestih anotacija koje se koriste za osnovnu validaciju. Uz anotacije navedene u tablici, moguće je kreirati personalizirane anotacije s posebnim pravilima.

```
1 inheritor  luka.jurkic
@Getter
@Setter
public class UserRequest extends BaseUser {

    @NotBlank(message = "First name is required")
    @Size(min = 1, max = 64, message = "Size must be between 1 and 64")
    private String firstName;

    @NotBlank(message = "Last name is required")
    @Size(min = 1, max = 64, message = "Size must be between 1 and 64")
    private String lastName;

    @NotNull(message = "Date of birth is required")
    @Past(message = "Date must be in past")
    private Date dateOfBirth;
}
```

Slika 4.27. Validirana *requeest* klasa

Tablica 4.5. Anotacija za validaciju

Anotacija	Objašnjenje
@NotNull	Vrijednost atributa ne smije biti *null*
@NotBlank	Vrijednost atributa ne smije biti *null* ili prazan tekst (razmaci)
@NotEmpty	Koristi se iznad liste, oznaka kako lista ne bi bila prazna
@Size	Postavlja ograničenje na veličinu, najčešće niza znakova, postavlja minimum i maksimum.
@Past, @Future, @PastAndPresent, @FutureAndPresent	Postavlja se iznad atributa koji pohranjuju vrijeme i datum, kako bi se odredila granica vrijednosti atributa
@Positive, @Negative	Postavlja se iznad brojčanih atribute te određuje raspon u kojem brojevi smiju biti

4.2.9. Sigurnost

Prema [6], sigurnost u web aplikacijama je ključno koje se bavi zaštitom podataka i resursa od neovlaštenog pristupa, zlouporabe i napada. *Spring* nudi razne klase koje olakšavaju implementaciju i uključivanje sigurnosnih aspekata u web aplikaciju. Osnovna sigurnost u web aplikacijama se sastoji od dva dijela: autorizacija i autentikacija.

Autentikacija je proces provjere identiteta korisnika. U web aplikaciji, to obično podrazumijeva provjeru korisničkog imena i zaporke. Ako su korisničko ime i zaporka ispravni tada se korisnik smatra autenticiranim. Postoje 3 osnovne vrste autentikacije koje Spring Boot nudi:

- Osnovna autentikacija – koristi korisničko ime i zaporku koji se šalju kao dio HTTP (engl. *The Hypertext Transfer Protocol*) zaglavlja, najjednostavnija je forma autentikacije, najčešće korištena ako nije zadano zahtjevima drugačije.
- Autentikacija bazirana na formama – korisnik se prijavljuje pomoću HTML formi, unosi svoje korisničke podatke, te podatke server provjerava i ako uspješno prođe provjeru kreira se sesija koji se šalje nazad te se korisnik smatra autenticiranim.
- Autentikacija bazirana na tokenima – često se koristi kod RESTful servisa, nakon uspješne prijave korisniku se dodjeljuje jedinstveni token (na primjer *JSON Web Token*) koji se koristi za autentikaciju prilikom svakog sljedećeg zahtjeva korisnika

Autorizacija je proces koji se događa nakon autentikacije i određuje koji resursi su dostupni kojem korisniku, dijeli se na dva tipa:

- Uloge – uloge predstavljaju prava koja su dodijeljena korisnicima, primjer uloga su: USER, ADMIN, MODERATOR i slično
- Ovlasti – ovlasti su specifične dozvole unutar aplikacije, one se često dodjeljuju dinamično tokom rada aplikacije, primjer su prava čitanja, zapisivanja ili mijenjanja dokumenata

Spring Boot nudi razne mogućnosti autenticiranja korisnika, neki od kojih su: *BasicAuth, OAuth2, JWT Authentication, Digest Authentication, OpenId* i drugi. Kako u nefunkcionalnim zahtjevima aplikacije nije navedeno koja je minimalna razine zaštite, zbog jednostavnosti implementacije korištena je *BasicAuth* autentikacija.

Kako bi bila jednostavno proširiva, potrebno je izgraditi sigurnost u web aplikaciju preko niza klasa. Klase koje se koriste za implementiranje sigurnosti u web aplikacije su označene s

@Configuration. Ta anotacija govori *Spring*-u o kojoj se konfiguraciji radi te korištenje nove umjesto već napisane konfiguracije.

SecurityConfig klasa je zadužena za stvaranje autentikacije korisnika. Ta klasa stvara novu autentikaciju koristeći korisničke podatke, te kodirati zaporku kako ne bi došlo do curenja podataka. Primjer te klase se nalazi na 4.28. Metoda *authenticationProvider* je glavna metoda koja vraća novostvorenu autentikaciju za korisnika. *PasswordEncoder* je ista klasa kao i s 4.25.

```
luka.jurkic
@Configuration
public class SecurityConfig {

    private final UserDetailsService userDetailsService;
    private final PasswordEncoder passwordEncoder;

    luka.jurkic
    @Autowired
    public SecurityConfig(UserDetailsService userDetailsService, PasswordEncoder passwordEncoder) {
        this.userDetailsService = userDetailsService;
        this.passwordEncoder = passwordEncoder;
    }

    luka.jurkic
    @Bean
    public AuthenticationProvider authenticationProvider() {
        DaoAuthenticationProvider authProvider = new DaoAuthenticationProvider();
        authProvider.setUserDetailsService(userDetailsService);
        authProvider.setPasswordEncoder(passwordEncoder);
        return authProvider;
    }
}
```

Slika 4.28. Konfiguracija sigurnosti

Klasa s 4.28 koristi instancu klase *UserDetailService*. Ta klasa je ugrađena u Spring Boot, s postojećom implementacijom. Ako se radi s korisnicima koji se spremaju u tablicu koju je programer stvorio tada je potrebno prepisati postojeću konfiguraciju. Primjer kako izgleda dohvaćanje korisnika, stvaranje instance *UserDetailService*, popunjavanje korisničkim podacima te vraćanje istih je dano na primjeru 4.30 . @Bean anotacija govori *Spring*-u u slučaju pojave instance *UserDetailsService*, vrati novu instancu koristeći danu metodu.

Slika 4.30. Konfiguracija korisničkih detalja

Spring u pozadini prilikom rada s korisničkim računima koristi *UserDetailManager*. Prilikom prepisivanja *UserDetailService* implementacije potrebno je prepisati i *UserDetailManager* implementaciju. Primjer se može vidjeti na 4.29.

Slika 4.29. Konfiguracija menadžera korisničkih detalja

Osim samoga postavljanja sigurnosti, potrebno je sigurnost web aplikacije konfigurirati. Konfiguracija sigurnosti se radi kroz metodu koja se naziva *securityFilterChain.* Pomoću ove metode se mogu postaviti i razine pristupa raznim korisnicima. Može se postaviti da korisnici s nekim pravima imaju pristup nekim resursima, dok korisnicima s drugim pravima nemaju. U ovoj metodi se postavljaju prava za statične resurse. Statični resursi su resursi koji se koriste za prikaz, oni uključuju: slike, ikone, CSS dokumente i slično. Potrebno je postaviti stranice koje se koriste za prijavu i odjavu korisnika, te se postavljaju HTML dokumenti na koje web aplikacija vodi prilikom prijave i odjave. Zbog jednostavnosti se popis dopuštenih resursa postavlja u liste na način da se grupiraju po nekoj sličnosti, primjerice razini prava ili prema tipu *request*-a, to jest HTTP metoda. Na 4.31 se nalaze liste resursa koji su otvoreni, to znači da za njih nije potrebna autentikacija. Na 4.32 se nalazi implementacija metode *securityFilterChain*.

```
luka.jurkic
@Configuration
@EnableWebSecurity
@EnableMethodSecurity(securedEnabled = true)
public class BasicAuthWebSecurityConfiguration {

    private final String[] API_GET_URL_WHITELIST = {
            "/api/users/{username}",
            "/api/users/roles",
            "/api/users/isUnique",
            "/api/users/isValidPassword",
            "/api/ingredients/**",
            "/api/recipes/**",
            "/api/recipes/{recipeName}/ingredients",
            "/api/comments/recipe/{recipeName}",
            "/api/comments/details/{recipeName}",
    };

    private final String[] API_POST_URL_WHITELIST = {
            "/api/users"
    };
    private final String[] MVC_URLS_WHITELIST = {
            "/users/new",
            "/users/create",
            "/users",
            "/index",
            "/",
            "/login",
            "/recipes",
            "recipes/exportToPdf/{recipeName}"
    };

```

Slika 4.31. URL *whitelist*

```
luka.jurkic *
@Configuration
@EnableWebSecurity
@EnableMethodSecurity(securedEnabled = true)
public class BasicAuthWebSecurityConfiguration {

    //code omitted for brevity

    new *
    @Bean
    public SecurityFilterChain securityFilterChain(HttpSecurity httpSecurity) throws Exception {
        httpSecurity.csrf(AbstractHttpConfigurer::disable)
                .authorizeHttpRequests((authorize) -> authorize
                        .requestMatchers("/error").permitAll()
                        .requestMatchers(HttpMethod.GET, API_GET_URL_WHITELIST).permitAll()
                        .requestMatchers(HttpMethod.POST, API_POST_URL_WHITELIST).permitAll()
                        .requestMatchers(MVC_URLS_WHITELIST).permitAll()
                        .requestMatchers(PathRequest.toStaticResources().atCommonLocations()).permitAll()
                        .anyRequest().authenticated())
                .httpBasic(Customizer.withDefaults())
                .formLogin(form ->
                        form.loginPage("/login")
                                .loginProcessingUrl("/authenticate")
                                .permitAll())
                .logout(logout -> logout
                        .logoutSuccessUrl("/login")
                        .logoutSuccessHandler(new CustomLogoutSuccessHandler()));
        return httpSecurity.build();
    }
```

Slika 4.32. Konfiguracija lanca sigurnosti

4.2.10. Testiranje

Testiranje i pokrivenost testovima su još jedan ključni aspekt u razvoju kvalitetnog i održivog softvera. Ovi proces i osiguravaju da aplikacija ispravno funkcionira, da su sve funkcionalnosti testirane te da se greške otkrivaju i ispravljaju u ranim fazama razvoja. Testovi se dijele na tri glavne skupine:

- Jedinični testovi – testiraju zasebne metode ili klase u izolaciji od ostatka sistema
- Integracijski testovi – testovi koji testiraju nekoliko komponenti te funkcionalnosti aplikacije
- Testovi prihvaćanja – testovi s pogleda krajnjeg korisnika

Prema [7], pokrivenost testovima odnosi se na postotak koda aplikacije koji je pokriven testovima. Visoka pokrivenost testovima ne garantira nedostatak grešaka, ali značajno povećava sigurnost da aplikacija radi kako je predviđeno. Pokrivenost testovima se mjeri na nekoliko načina

- Pokrivenost linija koda – procjenjuje koliko je linija koda izvršeno tokom testiranja

- Pokrivenost grana – procjenjuje koliko su grane (*if-else, switch case*) u kodu testirane
- Pokrivenost metoda – procjenjuje koliko metoda u kodu je testirano

Prilikom pisanja testova u *Spring Boot*-u, preporuča se nekoliko pravila. Pisanje testova paralelno s razvojem koda, testovi bi trebali biti pisani zajedno s kodom kako bi se osigurala veća pokrivenost testovima. Potrebno je osigurati da su svi slojevi aplikacije adekvatno testirani. Prilikom viših razina razvoja aplikacije, testove je potrebno integrirati u CI/CD *pipeline* kako bi se testovi pokretali prilikom svake izgradnje aplikacije. Redovito analizirati pokrivenost testovima kako bi se identificirala područja koja trebaju biti dodatno testirana. Testiranje i pokrivenost testovima osiguravaju kvalitetu, pouzdanost i održivost web aplikacija, što ih čini spremnim za puštanje u rad te dugoročno održavanje.

Testovi se dijele na obične testove i parametrizirane testove. Obični testovi (u daljnjem tekst: testovi) su testovi koji testiraju jednu funkcionalnost s jednom setom parametara. Takvi testovi se pišu kada se za jedan set parametara očekuje jedinstveni rezultat. Paramtrizirani testovi se pišu onda kada niz različitih vrijednosti parametara očekuju isti rezultat.

Spring Boot nudi visoku razinu podrške za testiranje putem anotacije @SprignBootTest, koja omogućuje pokretanje cijele aplikacije ili njenih dijelova unutar testa. Testovi se konstantno mijenjaju, nadopunjuju ili smanjuju. Dobra struktura testova je nužna za njihovu jednostavnu modifikaciju.

Prilikom generiranja projekta i razvojnom okruženju *IntelliJ*, ono postavlja mapu unutar koje se postavljaju testovi. Ta mapa sadrži dvije pod mape pod nazivima *java* i *resources*. Unutar *maper resources* je mapa *migrations* u kojoj se nalaze migracije koje se pokreću prilikom testiranja. Baza podataka se konstantno mijenja, kako bi se zadržala konzistentnost testova dodaju se migracijske SQL skripte. Te skripte dodaju podatke u potpuno novu bazu koja je jednaka pravoj bazi, ali se podaci dviju baza ne miješaju. Na taj način prilikom svakog pokretanja testova baza je jednaka. Unutar mape *java* nalazi se niz pod mapa:

- *clients* – mapa u kojoj se nalaze klase koje simuliraju korisnika i rade pozive prema aplikaciji
- *constants* – mapa u kojoj se nalaze klase unutar kojih su podaci koji se koriste za testiranje
- *factory* - mapa s klasama koje stvaraju korisničke zahtjeve prema aplikaciji
- *frontend* - testovi

- *parameters* – mapa s klasama koje sadrže parametre koji se koriste za paremetrizirane testove

4.33 prikazuje migracijsku skriptu za tablicu sastojaka pod nazivom V99.0.10__ingredient_migration.sql. Prvi broj u nazivu označava da se radi o testnoj migraciji, a ne regularnoj, drugi broj je ekstenzija, treći broj označava redoslijed izvođenja. Ako se pokrene nekoliko migracijskih skripti izvode se prvo one s manjim brojem. Nakon toga se nalazi opis migracijske skripte. Prilikom pokretanja testa, dovoljno je odraditi ovu skriptu i tada će se u bazi nalaziti svi podaci koji se nalaze u ovoj skripti. Kada test završi svi podaci se brišu te se baza vraća na početno stanje prije pokretanja testova.

```
INSERT INTO ingredient (id, name, category_id, version,
                        created_at, modified_at)
VALUES
    (10, 'beef', 10, 1, now(), now()),
    (11, 'onion', 12, 1, now(), now()),
    (12, 'carrot', 12, 1, now(), now()),
    (13, 'garlic', 13, 1, now(), now()),
    (14, 'chilli', 12, 1, now(), now()),
    (15, 'curry', 13, 1, now(), now()),
    (16, 'tomato', 15, 1, now(), now()),
    (18, 'garlic', 12, 1, now(), now()),
    (19, 'pork', 10, 1, now(), now()),
    (20, 'chicken', 10, 1, now(), now());
```

Slika 4.33. Testna migracija

Testni klijent je klasa koja služi za spajanje na aplikaciju pomoću jedinstvenih URL-a. Popis tih URL-a se nalazi na početku klase *IngredientTestClient*. Unutar te klase nalazi se jednaki broj metoda kao i kod servisa kojeg se testira.

Unutar klasa za konstante se nalazi niz varijabli koje predstavljaju podatke koji se koriste za testiranje. Te varijable najčešće odgovaraju vrijednostima koje se nalaze unutar testnih migracijskih skripti.

Factory klase služe za stvaranje *request* objekata koji se šalju kao oblik korisničkog unosa. Svaki entitet ima svoju personaliziranu klasu tvornica. Najčešće broj metoda tvornica odgovara duplom broj klasa *request* za pojedinačni entitet. Jedna metoda predstavlja pravilno generirani *request* objekt, druga metoda predstavlja parametriziranu tvornicu koja stvara *request* objekt prema predanim parametrima.

Klasa *IngredientParameters,* sadrži niz metoda koje vraćaju tok argumenata koji predstavljaju parametre. Svaka metoda jedinstvenog imena opisuje za koji slučaj i za koje testove se ti parametri odnose. Tok sadrži više argumenata unutar kojih se nalaze parametri. Parametri unutar argumenata trebaju biti istog tipa na pojedinačnim mjestima, inače može doći do pada testove zbog pogreške testova.

Kao i kod klasa za prijenos podataka, te kod entiteta je bilo potrebno izvući stvari koje se ponavljaju u klase koje su sve klase nasljeđivale, tako je i kod testova. Testovi zahtijevaju konfiguraciju, kako se ta konfiguracija ne bi pisala na početku svake klase testova ona se izvlači u posebnu apstraktnu klasu pod nazivom *AbstractIT*, gdje IT označava integracijske testove. Web aplikacija za recepte i kuhanje koristi *PostgreSQL* za bazu podataka, koja je smještena u *docker* te je zato potrebno podići novu instancu *PostreSQL* baze prije pokretanja testova. Testovi koriste anotaciju @FlywayTest koja označava da se prilikom početka testa pokrene potpuno nova instance baze. Cijela konfiguracija testova se nalazi na 4.34.

Slika 4.34. Konfiguracija testova

Pisanje testova je dugačak i kompleksan posao. Klase testova mogu doseći i do nekoliko stotina, čak i tisuću linija koda. Kako bi se jednostavnije snalazili među svim testovima, potrebno je unaprijed dogovoriti pravila sortiranja testova. Testovi se često sortiraju po tipu HTTP metode. Unutar toga se sortiraju tako da testovi koji prolaze idu prvi, nakon toga idu slučajevi gdje testovi padaju. Napomena: pad testa može biti očekivani slučaj.

Na 4.35 su prikazane metode koje testiraju dohvaćanje sastojka prema njegovoj jedinstvenoj oznaci. Jedan test testira slučaj kada je sastojak uspješno dohvaćen, a drugi testira kada sastojak nije pronađen. Ako je sastojak uspješno dohvaćen tada se provjerava povratnu vrijednost te vidjeti je li dohvaćena sastojak koji je predviđen. U slučaju kada se testira dohvaćanje nepostojećeg sastojka, provjerava se povratna greška od aplikacije.

Iznad svakog testa potrebno je postaviti nekoliko anotacija. Anotacija @Test govori *Spring*-u da se radit o metodi koja se koristi kao test. Prilikom pokretanja svih testova od jednom, oni se ispisuju u konzolu. Imena testova su dugačka kako bi se detaljno opisalo što test radi. Kako bi se uredio ispis u konzoli koristi se anotacija @DisplayName koja postavlja niz znakova koji će se

prikazivati za taj test kod ispisa u konzolu. @FlywayTest označava potrebnu za pokretanjem nove instance baze prilikom pokretanja toga testa, ako je potrebno odraditi testne migracije tada se one postavljaju u zagradu pod atributom *locationsForMigrate* odvojene zarezom.

Naziv metode koja se pokreće kao test je proizvoljan. Postoji konvencija za pisanje imena testova. Iz naziva testa je potrebno saznati informacije o testu kao što su: koji se parametri predaju testu, što se testira i što se očekuje od testa. Očekivati se može HTTP status kod ili neka povratna vrijednost.

```
luka.jurkic *
38  public class IngredientApiIT extends AbstractIT {
39
40      @Inject IngredientRepository ingredientRepository;
41
        luka.jurkic *
42      @Test
43      @DisplayName("GET Ingredient - OK")
44      @FlywayTest(locationsForMigrate = {"migrations/ingredient"})
45      public void givenValidId_whenGetIngredient_thenExpect200() {
46          IngredientDTO ingredient = IngredientTestClient.getIngredient(INGREDIENT_ID) Response
47                  .then().statusCode( expectedStatusCode: 200) ValidatableResponse
48                  .and().extract() ExtractableResponse<Response>
49                  .body().as(IngredientDTO.class);
50          assertThat(ingredient.getName()).isEqualTo(INGREDIENT_NAME);
51          assertThat(ingredient.getCategory().getName()).isEqualTo(INGREDIENT_CATEGORY);
52      }
53
        luka.jurkic *
54      @Test
55      @DisplayName("GET Ingredient - Not found")
56      @FlywayTest(locationsForMigrate = {"migrations/ingredient"})
57      public void givenInvalidId_whenGetIngredient_thenExpect404() {
58          IngredientTestClient.getIngredient(NON_EXISTING_INGREDIENT_ID) Response
59                  .then().statusCode( expectedStatusCode: 404) ValidatableResponse
60                  .body( path: "errorKey", equalTo(INGREDIENT_NOT_FOUND));
61      }
```

Slika 4.35. Testovi

Neki testovi daju jednak rezultat za različite vrijednosti ulaznih parametara, zbog duljine testiranja oni se pišu u posebne klase te se uključuju u testove. Uključivanje parametara iziskuje promjenu anotacije @Test u anotaciju @ParameterizedTest te dodavanje anotacije @MethodSource(). Unutar zagrade se piše puna putanja do metode koja vraća parametre za taj test. *Spring* to radi na način da parametre iz metode predaje testu u obliku parametara metode, zato je potrebno u zagradu metode testa napisati pravilnim tipom i redoslijedom parametre koji se primaju. Ako metoda testna vraća listu kao povratnu vrijednost tada je potrebno promijeniti tip u koji se prebacuje nakon izvođenja, na 4.36 to se može vidjeti na liniji 96.

```
86      //code omitted for brevity
87
        luka.jurkic *
88      @ParameterizedTest
89      @MethodSource("hr.dice.luka_jurkic.parameters.IngredientParameters#okForGetIngredientsByCategory")
90      @DisplayName("GET Ingredients by Category - OK")
91      @FlywayTest(locationsForMigrate = {"migrations/ingredient"})
92      public void givenValidCategory_whenGetIngredientsByCategory_thenExpect200(String category, Integer count) {
93          List<IngredientDTO> ingredients = IngredientTestClient.getIngredientsByCategory(category) Response
94                  .then().statusCode( expectedStatusCode: 200) ValidatableResponse
95                  .and().extract() ExtractableResponse<Response>
                    new *
96                  .body().as(new TypeRef<>() {});
97          assertThat(ingredients).isNotNull();
98          assertThat(ingredients.size()).isEqualTo(count);
99      }
100
        luka.jurkic
101     @ParameterizedTest
102     @MethodSource("hr.dice.luka_jurkic.parameters.IngredientParameters#notFoundForGetIngredientsByCategory")
103     @DisplayName("GET Ingredients by Category - Bad Request")
104     @FlywayTest(locationsForMigrate = {"migrations/ingredient"})
105     public void givenInvalidCategory_whenGetIngredientsByCategory_thenExpect404(String category) {
106         IngredientTestClient.getIngredientsByCategory(category) Response
107                 .then() ValidatableResponse
108                 .statusCode( expectedStatusCode: 404)
109                 .body( path: "errorKey", equalTo(INGREDIENT_CATEGORY_NOT_FOUND));
110     }
111
112     //code omitted for brevity
```

Slika 4.36. Parametrizirani testovi

4.3. Izrada poslužiteljsko-klijentskog dijela aplikacije

Prema [8], upravljač je komponenta u MVC arhitekturi koja služi kao posrednik između korisnika i poslovne logike aplikacije. U *Spring Boot*-u, upravljači se implementiraju kao Java klase koje označavaju anotacijom @Controller. Pod riječi upravljač se također odnose i REST upravljači čija je anotacija @RestController. Uloga REST upravljača je jednaka ulozi MVC upravljača, razlika je u povratnom tipu.

@Controller je anotacija koja označava klasu kao upravljač koja obrađuje web zahtjeve i vraća naziv pogleda koji treba prikazati korisniku. @RestController je anotacija koja objedinjuje anotacije @Controller i @ResponseBody, što znači da upravljač ne vraća naziv pogleda, već vraća podatke kao JSON ili XML.

Funkcije sloja upravljača uključuju sljedeće zadaće: primanje HTTP zahtjeva, interakcija sa slojem servisa, vraćanje odgovora korisniku te validaciju podataka. Upravljači obrađuju HTTP zahtjeve (GET, POST, PUT, DELET i ostali) koje aplikacija prima. Koriste anotaciju @RequestMapping ili preciznije anotacije poput @PostMapping ili @GetMapping za mapiranje URL-ova na metode u upravljaču. Upravljači komuniciraju sa slojem servisa kako bi dobili potrebne podatke ili izvršili određene akcije na osnovu zahtjeva korisnika. Vraćanje odgovornosti korisniku se odvija nakon obrade zahtjeva. Korisniku se podaci mogu vratiti na više načina

- HTML pogled: u slučaju web aplikacija koje koriste tehnologije poput *Thymeleaf*-a
- JSON/XML: u slučaju *RESTful* servisa.

Upravljači često uključuju validaciju ulaznih podataka putem anotacija kao što su @Valid ili @Validated, osiguravajući da su podaci koje aplikacija prima ispravni prije nego što se proslijede dalje.

4.3.1. REST upravljači

Na 4.37 je prikazan dio REST upravljača. Upravljači su klase s posebnim anotacijama. Anotacije koje se nalaze na Slici X su @RestController, @RequestMapping te u zagradi te anotacije je dodana putanja te anotacija @Validated. Putanja koja se postavlja kod anotacije @RequestMapping se koristi kako bi se svaki upravljač dodati odvojio od ostalih. Koristeći ovaj način pristup ovom upravljaču je mogući jedino ako URL počinje s putanjom koja je zadana u zagradi kod anotacije.

Upravljač sa slike koristi servis koji se bavi podacima vezanima za taj entitet. Referenca na servis treba biti *final* tipa kako se ne bi mijenjala tokom izvođenja. Ovisnost o servisu se ubacuje preko konstruktora.

Iznad prve metode na 4.37 se nalazi anotacija @GetMapping s putanjom unutar zagrade. Ta putanja je URL kojim će se moći pristupiti toj metodi. Anotacija također govori kako je potrebno uputiti GET zahtjev kako bi se moglo pristupiti toj metodi. Kako je iznad klase anotacija @RequestMapping, ova anotacija se nadovezuje na putanju koja je definirana kod glavne anotacije klase. Metoda vraća objekt koji je u pozadini pretvoren u JSON dokument te se vraća kao odgovor upravljača. Metoda koristi servis, te neku njegovu metodu. Metode unutar upravljača se često nazivaju istim imenom kao i one unutar servisa kako bi se jednostavnije moglo povezati funkcije metoda unutar upravljača s funkcijama metoda unutar servisa.

Iznad druge metode uz anotaciju @GetMapping se nalazi i anotacija @Secured s listom *string*-ova unutar zagrada. Ova anotacija govori da kako bi se pristupilo ovoj metodi uz samu putanju potrebno je biti autenticiran te je potrebno imati ovlasti neke od onih koji se nalaze unutar liste *stringova* kod anotacije. Kako bi se precizno definiralo koje ovlasti mogu pristupiti metodi potrebno je ispred naziva ovlasti dodati „ROLE_“.

Treće metoda je označena anotacijom @PostMapping. Putanja nije dodana, to ukazuje da se

```
luka.jurkic *
@RestController
@RequestMapping(path = "/api/users")
@Validated
public class UserController {

    private final UserService userService;

    luka.jurkic
    public UserController(UserService userService) { this.userService = userService; }

    luka.jurkic
    @GetMapping("/{username}")
    public UserDTO getUser(@PathVariable String username) {
        return userService.getUser(username);
    }

    luka.jurkic
    @GetMapping("/user/details")
    @Secured({"ROLE_ADMINISTRATOR", "ROLE_ADMIN", "ROLE_USER"})
    public UserDetailsDTO getUserDetails(@NotNull Authentication authentication) {
        return userService.getUserDetails(authentication.getName());
    }

    new *
    @PostMapping
    public UserDetailsDTO createUser(@Valid @RequestBody CreateUserRequest createUserRequest) {
        return userService.createUser(createUserRequest);
    }

    //code omitted for brevity
```

Slika 4.37. User REST kontroler

koristi putanja koja se nalazi iznad klase upravljača. Ovoj metodi se pristupa isključivo POST HTTP metodom.

Dohvaćanje podataka od strane korisnika se može provest na nekoliko načina. Varijabla to jest podataka se može nalaziti u putanji kojom se dolazi do metode upravljača. U prvoj metodi na

4.37 je primjer kako se dohvaća korisničko ime (*username*) koje se nalazi u putanji. Kako bi se označilo da je nešto varijabla u putanji, a ne dio putanje, ta se riječ postavlja u vitičaste zagrade. Da bi se odredio tip varijable i dohvatila varijabla iz putanje koristi se anotacija @PathVariable ispred tipa i imena varijable koje treba odgovarati onom u putanji. Na taj način *Spring Boot* uzima varijablu koja se nalazi u putanji dodaje prebacuje ju u odgovarajući tip podatke te pohranjuje u varijablu koja se potom može koristiti unutar metode.

Anotacija @NotNull ispred varijable označava da objekt koji se prima ne smije biti *null* vrijednosti. Takav primjer se nalazi u drugoj metodi na 4.37. *Authentication* je tip podatka u kojem su spremljeni podaci o autenticiranom korisniku koju generira *Spring Boot*. U slučaju da korisnik nije autenticiran tada je ta varijabla postavljena na unutarnji tip korisnika *anonymus.*

Ako korisnik šalje niz podataka, tada se oni pakiraju objekt koji se najčešće nazive *request*. Treća metoda na 4.37 daje primjer kako se dohvaća objekt koji nije generiran od *Spring Boot*-a. Potrebna je anotacija @RequestBody, ona zatraži tijelo HTTP zahtjeva koji se šalje prilikom pozivanja upravljač metoda. Tada se svi podaci koji se nalaze unutar tijela HTTP zahtjeva, najčešće u JSON ili XML formatu pretvaraju i popunjava se *request* objekt. Anotacija @Valid pokreće proces validacije te se sva pravila koja su dodana prilikom pisanja *request* klase provjerava. Pravila se mogu vidjeti na 4.27. Ako neko od pravila nije ispunjeno tada se podiže ugrađena iznimka, najčešće *IllegalArgumentException*.

Uz anotacije prikazane na 4.37, jedna od češćih anotacija je @RequestParam. Ako je potrebno prihvatiti nekoliko vrijednosti od korisnika, a ne nije ih potrebno zapakirati u objekt ili se logički ne mogu zapakirati u objekt, tada se koriti navedena anotacija.

Na korisničkom dijelu aplikacije je moguće stvoriti varijablu u koju se pohranjuju podaci koje korisnik unese, tada se ta varijabla šalje na poslužiteljski dio aplikacije. @RequestParam prima zagrade jedan argument *name*. To je naziv varijable koja se prima s korisničkog dijela aplikacije. Potrebno je da nazivi varijabli odgovaraju, tada se poslije anotacije postavlja tip varijable i naziv varijable u koju će se lokalno pohraniti vrijednost, taj naziv ne treba odgovarati nazivu na korisničkom dijelu.

4.3.2. MVC upravljači

4.39 prikazuje upravljač koji se koristi za generiranje HTML dokumenata na zahtjev korisnika. Korisnik pristupa pojedinim metodama ovoga upravljača pomoću URL putanja koje se nalaze iznad svake metode. Upravljaču je potreba niz referenci na servise i *mappere* kako bi ispravno

radio. Na 4.38 je prikazan upravljač koji služi za rad s korisničkim podacima. *String*-ovi koji se često koriste se postavljaju u statične varijable na početku klase.

```
luka.jurkic *
@Controller
@RequestMapping("/users")
public class UserMVController {

    private static final String LOGIN_PAGE = "/login";
    private static final String COMMENTS_ATTRIBUTE = "comments";
    private static final String REQUEST_ERROR_ERROR_CODE = "request.error";
    private static final String USER_DETAILS_PAGE = "user/show_userDetails";

    private final UserService userService;
    private final UserMapper userMapper;
    private final CommentService commentService;
    private final RecipeService recipeService;

    luka.jurkic
    public UserMVController(UserService userService, UserMapper userMapper, CommentService commentService, RecipeService recipeService) {
        this.userService = userService;
        this.userMapper = userMapper;
        this.commentService = commentService;
        this.recipeService = recipeService;
    }

```

Slika 4.38. Instance user MVC kontrolera

Iznad klase koja se ponaša kao upravljač je potrebno staviti anotaciju @Controller. Ta anotacija čini klasu upravljačom te govori *Spring Boot*-u traženje URL-ove te prosljeđuje korisničke zahtjeve prema metodama. Uz anotaciju @Controller koristi se već spomenuta anotacija @RequestMapping čija je funkcija jednaka kao i kod REST upravljača.

Metode ovog tipa upravljač vraćaju *String* kao povratni tip, za razliku od REST upravljača koji su vraćali objekte ili liste objekata. *String* koji ovaj upravljač vraća je naziv predloška koji se nalazi u resursima koji se treba generirati i prikazati korisniku. Većina predložaka koristi različite podatke koje je potrebno prikazati. Metode upravljača popunjavaju objekt klase *Model* podacima koji se zatim šalju na predložak. Klasa *Model* je klasa koja služi za prijenos podataka između predložaka i upravljača. Instanca ove klase se ne stvara u metodi upravljača nego se prima kao argument.

Upravljač priprema podatke te ih obrađuje, pozivanjem odgovarajuće metode odgovarajućeg servisa. Podaci koji se vrate od servisa se zatim postavljaju u model te se vraća ime predloška. Na model se može poslati varijabla primitivnog tipa ili se može poslati instanca objekta neke klase.

```java
luka.jurkic *
@Controller
@RequestMapping("/users")
public class UserMVController {

    //code omitted for brevity

    luka.jurkic
    @GetMapping("/details")
    public String getUserDetails(Model model) {
        Authentication authentication = SecurityContextHolder.getContext().getAuthentication();
        UserDetailsDTO user = userService.getUserDetails(authentication.getName());
        model.addAttribute( attributeName: "user", userMapper.detailDTOToRequest(user));
        model.addAttribute( attributeName: "userRole", user.getRole());
        model.addAttribute(COMMENTS_ATTRIBUTE, commentService.getCommentsContentByUser(authentication.getName()));
        model.addAttribute( attributeName: "createdAt", user.getCreatedAt());
        model.addAttribute( attributeName: "modifiedAt", user.getModifiedAt());
        return USER_DETAILS_PAGE;
    }

    luka.jurkic
    @GetMapping
    public String getUser(@RequestParam(name = "username") String username, Model model) {
        UserDTO user = userService.getUser(username);
        model.addAttribute( attributeName: "user", user);
        model.addAttribute( attributeName: "recipes", recipeService.getRecipesByUser(username));
        model.addAttribute(COMMENTS_ATTRIBUTE, commentService.getCommentsByUser(username));
        return "user/show_user";
    }

    luka.jurkic
    @GetMapping("/new")
    public String createUserForm(Model model) {
        CreateUserRequest request = new CreateUserRequest();
        model.addAttribute( attributeName: "request", request);
        return "user/form_createUser";
    }
    luka.jurkic *
    @PostMapping("/create")
    public String createUser(@Valid @ModelAttribute(name = "request") CreateUserRequest request, BindingResult bindingResult, Model model) {
        if(!userService.isUnique(request.getUsername())) { bindingResult.rejectValue( field: "username", REQUEST_ERROR_ERROR_CODE, defaultMessage: "Username taken"); }
        if(!userService.isValidPassword(request.getPassword())) { bindingResult.rejectValue( field: "password", REQUEST_ERROR_ERROR_CODE, defaultMessage: "Password is invalid"); }
        if (bindingResult.hasErrors()) { return "user/form_createUser"; }
        userService.createUser(request);
        model.addAttribute( attributeName: "users", userService.getUsers());
        return LOGIN_PAGE;
    }
```

Slika 4.39. User MVC kontroler

U četvrtoj metodi na 4.39 se pripremaju podaci za stvaranje novog korisnika. Stvara se instanca klase *request* koja se šalje na predložak. Predložak tu instancu prikazuje praznu i korisnik popunjava podatke. U metodi ispod se ta instanca vraća u upravljač te se ti podaci temeljito provjeravaju. Upravljači su zaduženi za validaciju i provjeru podataka. Slika prikazuje korištenje ugrađene validacije anotacijom @Valid te validacije koja je napravljena specifično za taj *request* objekt.

Ako se pronađu greške nakon validacije, korisniku se prikazuje isti predložak s podacima koje je već upisao te mu se prikazuju greške koje su dovele do pada validacije. Ako je validacija uspješna, podaci se prosljeđuju na servis koji ih obrađuje te upravljač vraća novi predložak.

4.4. Dizajniranje korisničkog sučelja aplikacije

4.4.1. Thymeleaf

Prema [9], *Thymeleaf* je podrška za HTML predloške koji je baziran na javi. On omogućava jednostavno generiranje HTML, XML i drugih vrsta dokumenata- Prilikom izrade web

aplikacija najčešće je korišten za dinamično generiranje HTML stranica na serveru, podacima koji se primaju iz upravljača.

Ključne karakteristike *thymeleafa*-a:

- Prirodni predlošci: predlošci generirani *thymeleaf*-om su validni HTML dokumenti te ih se može pregledati u većini web preglednika bez pokretanja aplikacije.
- Integracija sa *Spring Boot*-om: *thymeleaf* je duboko integriran sa *Spring Boot*-om, to omogućuje jednostavno povezivanje s podacima i modelima koji dolaze iz upravljača.
- Podrška za *fragment*-e: pruža podršku za *fragmente*, omogućavajući modularizaciju te ponovno korištenje koda.
- Uvjetni prikazi i iteracija: *thymealeaf* omogućuje uvjetni prikazivanje elemenata te omogućuje iteraciju to jest prolaz kroz liste unutar samog HTML dokumenta.
- Internacionalizacija: podržava internacionalizaciju putem *„th:text“* atributa koji omogućuje prikazivanje tekstova na različitim jezicima koristeći *„.properties“* datoteke.

4.40 prikazuje jedan HTML predložak koji se korisi kako bi se prikazali osnovni podaci o kornisniku. Struktura HTML dokumenta, početak i njegov kraj, su jednaki kao i kod predložaka koji ne koriste *thymeleaf.* Zbog tog razloga se ovaj predložak može prikazati u web pregledniku i dok aplikacija nije pokrenuta.

```
<!DOCTYPE html>
<html lang="en" xmlns:th="http://www.thymeleaf.org">
<div><th:block th:replace="fragments/head :: genericHead"></th:block></div>

<body class="d-flex flex-column min-vh-100 col-md-8 mx-auto">
<div th:replace="fragments/header :: header"></div>
<h1 th:text="${user.username} + '\'s profile'" class="text-center"></h1>
<p th:text="'Role: ' + ${user.role}" class="text-center"/>
<h3 class="mt-3">Recipes: </h3>
<div class="row">
    <div class="mx-auto">
        <div class="d-flex align-items-center justify-content-evenly">
            <div th:each="recipe: ${recipes}">
                <form th:action="@{/recipes}" method="get">
                    <input type="submit" name="recipeName" th:value="${recipe.name}"
                           class="link-info link-offset-2 link-underline-opacity-25 link-underline-opacity-100-hover"/>
                </form>
            </div>
        </div>
    </div>
</div>
<h3 th:text="'Comments: ' + ${comments.size()}" class="mt-3"></h3>
<div th:each="comment: ${comments}">
    <h5 th:text="${comment.title} + ' on ' + ${comment.recipeName}"/>
</div>
<div th:replace="fragments/footer :: footer"></div>
</body>
</html>
```

Slika 4.40. HTML predložak

U početnoj <html> oznaci je potrebno uključiti *thymeleaf*. Prilikom uključivanja se određuje kratica koja će se koristiti za pristupanje *thymeleaf* oznakama. Unutar oznake <head> se nalaze osnovi podaci o dokumentu, ti se podaci rijetko mijenjaju te su izvučeni u *fragment*. Unutar <body> oznake se nalazi sadržaj stranice. U 4.6 se nalazi popis *thymeleaf* oznaka i njihova objašnjenja.

Tablica 4.6. *Thymeleaf* oznaka

Thymeleaf oznaka	Opis
th:replace	Mijenja blok koda koji je dan kao parametar s blokom u kojem se oznaka nalazi
th:block	Označava blok koda
th:text	Služi za prikaz dinamično generiranog teksta
th:each	Služi kao petlja, najsličnija je *foreach* petlji
th:value	Postavlja vrijednost *input* oznake na neku vrijednost
th:action	Postavlja URL na koji će *form*-a biti poslana nakon potvrđivanja

4.4.2. HTML dokumenti i statične datoteke

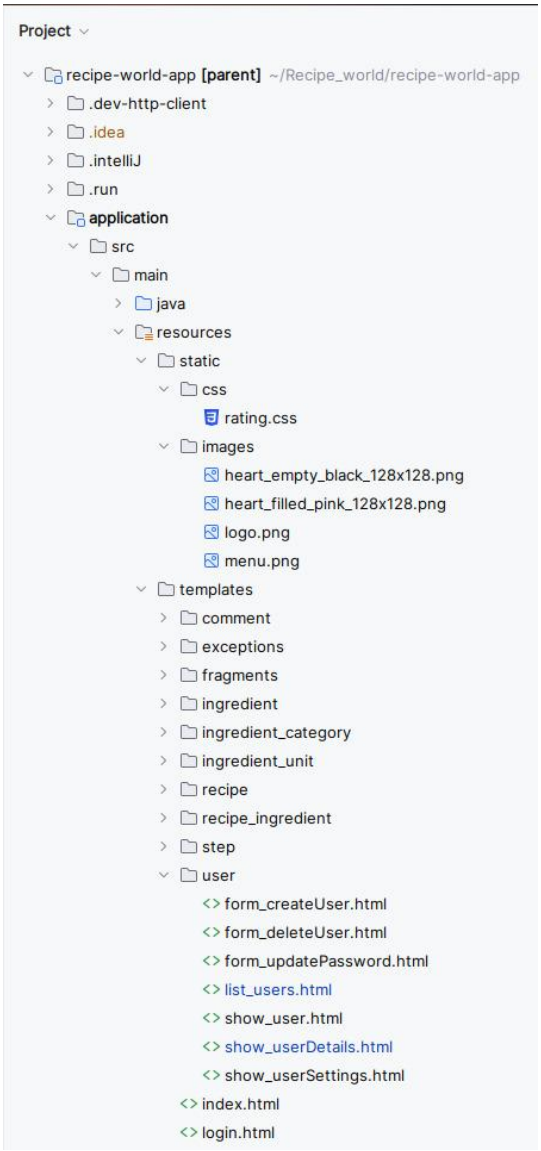
HTML dokumenti u *Spring Boot* aplikacijama služe kao predlošci (eng. *Templates*) za prikazivanje podataka korisnicima. Ti dokumenti su najčešće smješteni unutar *resource* mape i *templates* podmape unutar aplikacije jer je ta putanja zadana kao putanja unutar koje se spremaju predlošci koje koristi *thymeleaf*. Kada korisnik pristupi određenoj URL putanji, *Spring Boot* upravljač vraća ime HTML datoteke koja se prikazuje, te šalje potrebne podatke na taj predložak.

Statične datoteke su datoteke koje se ne mijenjaju tokom izvršavanja aplikacije i direktno se poslužuju klijentu. Najčešće statične datoteke su slike, ikone, CSS dokumenti, *JavaScript*

dokumenti, fontovi i ostali. *Spring Boot* automatski poslužuje datoteke iz ovih direktorija putem URL putanje koja odgovara strukturi direktorija.

4.41 prikazuje strukturu statičnih datoteka i HTML dokumenata unutar projekta. Statične datoteke zajendno s predlošcima se nalaze u mapi *resources*. Nakon toga se dijele na niz podmapa. Slike koje se spremaju pohranom kao slika unutar mape projekta ne bi trebale zauzimati puno prostora. Slike koje se spremaju na taj način su najčešće ikone koje se koriste za prikaz na stranici. Ikone su male sličice, malih dimenzija, s malo boje i nisu kompleksne.

Unutar mape s predlošcima preporuka je napraviti podmape te dodatno organizirati strukturu. Na 4.41 su predlošci podijeljeni prema entitetu koji je glavni na tom predlošku te iz kojeg upravljača se poziva taj predložak. Još jedan način za podijelu predložaka bio bi prema tipu akcije koje



Slika 4.41. Struktura statičnih dokumenata

izvode. Tada bi podmape mogle biti: *form*, *list*, *show* i tako dalje.

4.4.3. Fragmenti

Fragmenti u *Spring Boot*-u koriste za ponovnu upotrebnu dijelova HTML koda unutar nekoliko predložaka. Ovo je posebno korisni za dijelove stranice koji se ponavljaju, poput zaglavlja, podnožja, navigacijskih traka i slično. Podrška za fragmente unutar *thymeleaf*-a se provodi kroz atribut *„th:fragment“*.

4.42 i 4.43 prikazuju dva *fragment*-a. Na prvoj slici se nalazi fragment za podnožje stranice. Podnožje stranice je isto na svakoj stranici pa se taj kod izvlači u posebnu datoteku. Iako kod nije dugačak problem nastaje kada bi bilo potrebno promijeniti podnožje, tada bi bilo potrebno promijeniti podnožje na svakoj stranici zasebno. Na ovaj se način izgled mijenja na više mjesta istovremeno. *Fragment*-i trebaju biti unutar jedne oznake, na prvoj slici to je oznaka <footer>, a na drugoj .

```
1
2   <!-- generic footer fragment -->
3   <footer class="page-footer font-small blue pt-4 mt-auto text-white bg-dark">
4       <div class="container">
5           <p class="text-center text-muted">© 2024 FERIT</p>
6       </div>
7   </footer>
```

Slika 4.42. Fragment podnožja

4.43 prikazuje *fragment* unutar kojeg je kod za izbornik stranice. Izbornici su isti na svakoj stranici te je on izvučen u poseban dokument. 4.43 uključuje i sliku koja se nalazi kao statična datoteka. Slika je uključena na liniji 4 koristeći *thymeleaf* oznaku *th:src*.

```
<!-- generic menu fragment -->
<div  th:fragment="menuFragment" class="btn-group dropend">
    <button type="button" class="btn btn-secondary dropdown-toggle" data-bs-toggle="dropdown" aria-expanded="false" style="max-width: 60px">
        <img  th:src="@{/images/menu.png}" class="mw-100 h-auto" alt="Menu"/>
    </button>
    <ul class="dropdown-menu" th:if="${#authorization.expression('isAuthenticated()')}">
        <li><a th:href="@{/}" class="dropdown-item">Home</a></li>
        <li><hr class="dropdown-divider"></li>
        <li><a th:href="@{/ingredients/new}" class="dropdown-item">New Ingredient</a></li>
        <li><a th:href="@{/ingredients/all}" class="dropdown-item">List Ingredients</a></li>
        <li><hr class="dropdown-divider"></li>
        <li><a th:href="@{/recipes/new}" class="dropdown-item">Create Recipe</a></li>
        <li><a th:href="@{/recipes/import}" class="dropdown-item">Import Recipe</a></li>
        <li><a th:href="@{/recipes/all}" class="dropdown-item">My Recipes</a></li>
        <li><a th:href="@{/recipes/favorites}" class="dropdown-item">Favorites</a></li>
        <li><a th:href="@{/users/rated}" class="dropdown-item">Rated</a></li>
        <li><hr class="dropdown-divider"></li>
        <li th:if="${!#authorization.expression('hasRole(''ROLE_USER'')')}">
            <a th:href="@{/users/all}" class="dropdown-item">See Users</a>
        </li>
        <li th:if="${#authorization.expression('hasRole(''ROLE_ADMINISTRATOR'')')}">
            <a th:href="@{/categories/all}" class="dropdown-item">See Categories</a>
        </li>
        <li th:if="${#authorization.expression('hasRole(''ROLE_ADMINISTRATOR'')')}">
            <a th:href="@{/categories/new}" class="dropdown-item">New Category</a>
        </li>
        <li th:if="${#authorization.expression('hasRole(''ROLE_ADMINISTRATOR'')')}">
            <a th:href="@{/units/all}" class="dropdown-item">See Units</a>
        </li>
        <li th:if="${#authorization.expression('hasRole(''ROLE_ADMINISTRATOR'')')}">
            <a th:href="@{/units/new}" class="dropdown-item">New Unit</a>
        </li>
        <li th:if="${!#authorization.expression('hasRole(''ROLE_USER'')')}"><hr class="dropdown-divider"></li>
        <li><a th:href="@{/users/details}" class="dropdown-item">Profile</a></li>
        <li><a th:href="@{/users/settings}" class="dropdown-item">Settings</a></li>
        <li>
            <form action="#" th:action="@{/logout}" method="post"><button type="submit" class="dropdown-item">Logout</button></form>
        </li>
    </ul>
    <ul class="dropdown-menu" th:if="${#authorization.expression('!isAuthenticated()')}">
        <li><a th:href="@{/}" class="dropdown-item">Home</a></li>
        <li><hr class="dropdown-divider"></li>
        <li><a th:href="@{/login}" class="dropdown-item">Sign in</a></li>
        <li><a th:href="@{/users/new}" class="dropdown-item">Sign up</a></li>
    </ul>
</div>
```

Slika 4.43. Fragment izbornika

Zbog pojednostavljivanja rada s korisničkim razinama na razini cijele aplikacije, *thymeleaf* omogućuje provjeru razine prava korisnika prilikom generiranja dokumenta. Uz provjeru prava moguće je i provjera autentikacija korisnika, to jest može se provjeriti je li korisnik uspješno autenticiran. S ove dvije razine provjere može se postići razni različiti prikazi dokumenata u ovisnosti o tome je li korisnik autenticiran ili nije, te ako je s kojom razinom prava.

5. ZAKLJUČAK

Web aplikacija izrađena je u Java programskom jeziku u okviru MVC arhitekture. Rješavanje problema sigurnosti , komunikacije korisnika te pohranu velike količine podataka zahtijevalo je dobro osmišljeno i izgrađeno rješenje. Kompleksnost ili nedostatak osnovnih funkcionalnosti su osnovni problemi s kojima su se suočavala postojeća rješenja. Coolinarika te Minimalist baker su kompleksna rješenja, sadrže sve potrebne elemente, no često su previše kompleksna za korisnike početnike. Recepttura i Uvijekgladna, jednostavnije stranice za pronalazak recepata, korisnički pristupačnije, ali njihove opcije nisu dovoljne za korisnike koji bi htjeli veću manipulaciju osobnim receptima te interakciju s receptima ostalih korisnika.

Zbog potreba jednostavnosti web aplikacije te minimalnih zahtjeva koje korisnicima pružaju sve osnove mogućnosti za radi s receptima i komunikacijom s ostalim korisnicima, ova web aplikacija predstavlja sve osnovne mogućnosti bez kompleksnosti većih web aplikacija.

Web aplikacija za recepte i kuhanje izrađena u *Spring Boot Java* tehnologiji predstavlja funkcionalno i skalabilno rješenje koje omogućava korisnicima kreiranje, pretragu i upravljanje receptima i sastojcima. Kroz dodavanje različitih razina korisničkih prava (USER, ADMIN i ADMINISTRATOR) aplikacija osigurava sigurnost i odgovarajuću kontrolu pristupa.

Aplikacija nudi intuitivno korisničko sučelje kroz jasno definirane HTML predloške koji se dinamički popunjavaju. Također koristi snagu *Spring Boot* okvira za jednostavno upravljanje poslovnom logikom, validaciju podataka te sigurnost. Korištenjem MVC arhitekture rezultat je dobro organiziran kod i olakšano održavanje. Integracija s *thymeleaf*-om omogućuje dinamičko generiranje HTML stranica koje korisnicima daje brz i jednostavan pristup i rad s podacima.

Ovakva aplikacija obogaćuje korisničko iskustvo ljubitelja kuhanja, već također pruža snažnu osnovu za daljnji razvoj i dodavanje novih funkcionalnosti. Kroz korištenje ustaljenih praksi u razvoju i sigurnosti web aplikacija, osigurana je pouzdanost i efikasnost, čineći ovu aplikaciju korisnim alatom za sve koji žele istraživati i dijeliti svijet recepata i kulinarstva.

LITERATURA

[1] L. Fadnes, C. Celis-Morales, J.-M. Økland, S. Parra-Soto, K. Livingstone, F. Ho, J. Pell, R. Balakrishna, E. J. Arjmand, K. A. Johansson, Ø. Haaland and J. Mathers, "National Library of Medicine," PubMed Central, 27 6 2024. [Online]. Available: https://www.ncbi.nlm.nih.gov/pmc/articles/PMC10661734/. [Accessed 31 8 2024].

[2] O. Gierke, T. Darimont, C. Strobl, M. Paluch, M. Paluch and G. Turnquist, "Spring Documentation - JPA," VMware, Inc., 2024. [Online]. Available: https://docs.spring.io/spring-data/jpa/reference/index.html. [Accessed 8 31 2024].

[3] M. Yaduvanshi, "Simplify Model Mapping in Spring Boot with MapStruct and Lombok," Medium, 2023. [Online]. Available: https://mayankposts.medium.com/simplify-model-mapping-in-spring-boot-with-mapstruct-and-lombok-65d93b56a76b. [Accessed 8 31 2024].

[4] A. Boyko and C. Strobl, "Building REST services with Spring," VMware Tanzu, 2024. [Online]. Available: https://spring.io/guides/tutorials/rest. [Accessed 8 31 2024].

[5] A. Ugarte and K. Gilmore, "Spring Validation," Baeldung, 2024. [Online]. Available: https://www.baeldung.com/spring-boot-bean-validation. [Accessed 8 31 2024].

[6] A. Wilkinson and B. Hale, "Spring Security," VMware Tanzu, 2024. [Online]. Available: https://spring.io/projects/spring-security. [Accessed 8 31 2024].

[7] T. Bui, "JaCoCo Code Coverage with Spring Boot," Medium, 2023. [Online]. Available: https://medium.com/@truongbui95/jacoco-code-coverage-with-spring-boot-835af8debc68. [Accessed 8 31 2024].

[8] P. Dutta and G. Piwowarek, "Spring Controller," Baeldung, 2024. [Online]. Available: https://www.baeldung.com/spring-controllers. [Accessed 8 31 2024].

[9] The Thymeleaf Team, "Thymeleaf," The Thymeleaf Team, 2024. [Online]. Available: https://www.thymeleaf.org/documentation.html. [Accessed 31 8 2024].

[10] P. Eisentraut and A. Freund, "PostgreSQL," PostgreSQL Global Development Group, 2024. [Online]. Available: https://www.postgresql.org/. [Accessed 31 8 2024].

SAŽETAK

U uvodnom dijelu rada dani su primjeri rješenja koja ova aplikacija pokriva. Uzimajući u obzir sve potrebne aspekte problema, ovoj aplikaciji cilj je pokriti što više mogućih značajki. Prema uzoru na druge dane primjere, aplikacija uklanja neke nepotrebne značajke te objedinjuje one najpotrebnije. Cijela web aplikacija se bazira na Java *Spring Boot* tehnologiji te su omogućene sve funkcionalnosti koje su potrebne za osnovan rad aplikacije. Osim Jave kao programskog jezika kojim je u cijelosti izrađena logika aplikacije, korišteni su HTML i CSS za stvaranje grafičkog korisničkog sučelja. Korištenjem PostgreSQL baze podataka te JPA repozitorija omogućena je jednostavna i brza komunikacija s bazom podataka. Naposljetku, aplikacija je funkcionalna s velikim popisom jednostavni proširenja koji bi poboljšali aplikaciju.

Ključne riječi: Java Spring Boot, kuhanje, PostgreSQL, recepti, Thymeleaf, web aplikacija

ABSTRACT

Web application for recipes and cooking

In the introductory section of the paper, examples of solutions that this application covers are provided. Taking into account all the necessary aspects of the problem, the goal of this application is to cover as many possible features as possible. Based on the other provided examples, the application removes some unnecessary features and combines the most essential ones. The entire web application is based on Java Spring Boot technology, and all the functionalities necessary for the basic operation of the application are enabled. In addition to Java as the programming language used to implement the application's logic, HTML and CSS were used to create the graphical user interface. By using a PostgreSQL database and JPA repository, simple and fast communication with the database is enabled. Finally, the application is functional with a large list of simple extensions that would improve the application.

Keywords: cooking, Java Spring Boot, PostgreSQL, recipes, Thymeleaf, web application

ŽIVOTOPIS

Autor ovog završnog rada, Luka Jurkić je rođen u Pakracu 22.2.2003. Završio je srednjoškolsko obrazovanje u Tehničkoj školi Kutina, smjer tehničar za računarstvo. Luka Jurkić je trenutno student prijediplomskog sveučilišnog studija, smjer računarstvo, pod smjer programsko inženjerstvo na Fakultetu elektrotehnike računarstva i informacijskih tehnologija Osijek.

_____

Potpis autora