

SVEUČILIŠTE U ZAGREBU
FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA

ZAVRŠNI RAD br. 448

Mobilna aplikacija za upravljanje receptima

Marko Tunjić

Zagreb, svibanj 2022.

SADRŽAJ

1. Uvod	1
2. Zahtjevi	3
2.1. Funkcionalni	3
2.2. Dodatne funkcionalnosti	4
3. Baza podataka	5
3.1. Entiteti	5
3.2. Relacije	5
4. Poslužiteljska strana	8
4.1. GraphQL	8
4.1.1. Povijest	8
4.1.2. Definicija	8
4.1.3. Implementacija	9
4.2. Autentifikacija i autorizacija	11
4.2.1. JWT	11
4.2.2. Autentifikacija	12
4.2.3. Autorizacija	13
4.3. Pristup podacima	14
4.3.1. Sloj za persistenciju	14
4.3.2. Sloj usluge	15
4.4. Upogonjavanje	16
5. Korisnička strana	17
5.1. Flutter	18
5.1.1. BLoC	18
5.2. Ekrani	18
5.2.1. Ekran dobrodošlice, prijave i registracije	19

5.2.2.	Ekran za pregled svih recepata	20
5.2.3.	Ekran za pregled pojedinog recepta	22
5.2.4.	Ekran za dodavanje novog recepta	22
5.2.5.	Ekran za upravljanje korisnicima	23
6.	Zaključak	24

1. Uvod

Internet i web stranice su već dugo vremena dio ljudske svakodnevice. Zbog današnje potrebe za brzinom i efikasnošću sve popularnije su mobilne aplikacije, jer su vrlo jednostavne za koristiti. Naime potrebno je samo izvaditi mobitel kliknuti na aplikaciju i sve je spremno. Upravo zato danas puno web aplikacija također ima i svoju mobilnu inačicu.

Tako se do danas pojavio velik broj jezika i radnih okvira za razvoj mobilnih aplikacija kao na primjer kotlin, swift, flutter, react native i slično. Problem kod mobilnih uređaja je što ovisno koji operacijski sustav uređaj koristi jedna aplikacija neće biti kompatibilna na svim ostalima. Neki od prethodno navedenih radnih okvira, kao na primjer: flutter i react native, rješavaju problem kompatibilnosti operacijskih sustava uređaja takozvanim "native" pristupom i jednom isprogramirana aplikacija u tim okvirima će se moći izvoditi na svim uređajima.

Unatoč tranziciji sa preglednika na mobilne uređaje nije nestala potreba za poslužiteljskim dijelom i bazom podataka, jer takva arhitektura omogućava komunikaciju i dijeljenje sadržaja među ljudima širom svijeta. Pa se tako na poslužiteljskoj strani i dalje koriste stvari kao Spring, ASP.NET i slični okviri, a za bazu upravitelji po izboru na primjer PostgreSQL, MSSQL, MySQL... Iz tog razloga je još uvijek popularan oblikovni obrazac MVC (Model-View-Controller).

Jedini problem je što mobilne aplikacije nažalost ne podržavaju HTTP i SOAP protokol na način kao i web preglednici pa se mobilne aplikacije koriste novijim načinom prijenosa podataka između upravljačkog dijela aplikacije i korisničkog dijela aplikacije. To je JSON format podataka s odgovarajućim JSON zahtjevima prema poslužitelju. Uz JSON se koristi neki od arhitekturnih stilova kao REST ili GraphQL koji su alternativa za SOAP protokol i s njime mogu i web i mobilne aplikacije uspješno uspostavljati komunikaciju.

Također zbog naglog razvoja interneta i pojave velikog broja poslužitelja i baza podataka pojavila se potreba za računalima na kojima će biti pokrenuti ti poslužitelji i baze podataka, tako se pojavio "oblak" i usluge u oblaku (eng. Cloud, Cloud Services). Uz pomoć tih usluga vrlo je lako pokrenuti instancu potrebne baze i poslužitelja u "oblaku" koji će biti globalno dostupan. Kao pružatelji takvih usluga se ističu: AWS, Heroku, Azure i tako dalje.

Cilj ovoga rada bio je izraditi mobilnu aplikaciju koja će korisnicima olakšati: pristup, pamćenje, dijeljenje i pretraživanje recepata. Sadržaj je podijeljen u 4 dijela u kojima će se opisati zahtjevi, način izrade svakog sloja aplikacije (baza podataka, poslužiteljski i korisnički dio), korištene tehnologije i konačni proizvod.

2. Zahtjevi

2.1. Funkcionalni

1. **Korisnici aplikacije:** Aplikaciji mogu pristupiti 3 vrste korisnika: anonimni, prijavljeni i administratori i ovisno o vrsti korisnika kojoj pripada svaki ima različite ovlasti i uloge.
2. **Mogućnosti anonimnih korisnika:** oni mogu samo pregledavati i pretraživati recepte. Recepti se mogu pretraživati po nazivu ili po sastojcima pri čemu se pretraga po sastojcima može obaviti po sastojcima koje recept ne smije sadržavati (što je bitno zbog alergija) ili po sastojcima koje korisnik ima pri ruci.
3. **Mogućnosti prijavljenih korisnika:** prijavljeni korisnici uz funkcionalnosti anonimnih korisnika mogu također i dodavati vlastite recepte, komentirati sve recepte i dodavati željene recepte u listu omiljenih recepata pri čemu korisnik na vlastiti e-mail dobije poruku sa listom potrebnih sastojaka koja može poslužiti kao lista za kupnju.
4. **Mogućnosti administratora:** administratori za razliku od prijavljenih korisnika ne mogu dodavati vlastite recepte, ali zato imaju ulogu odabrovanja recepata, jer naime prije negoli bilo koji recept postane javno vidljiv administratori ga moraju odobriti.
5. **Format recepta:** uz zahtjeve da recept ima svoje korake pripreme i sastojke potrebne za izradu jela, svaki recept također može imati priloženo više fotografija i/ili videozapisa koji pobliže opisuju recept i pomažu korisnicima pri kuhanju.
6. **Registracija:** korisnik će pri registraciji morati predati neku vlastitu postojeću e-mail adresu na koju će primati liste za kupnju od recepata koje dodaju u svoje favorite.

2.2. Dodatne funkcionalnosti

1. **Dodatne mogućnosti prijavljenih korisnika:** prijavljeni korisnici će dodatno moći brisati vlastite recepte, komentare na vlastitim receptima i vlastite komentare na tuđim receptima. Također će moći ocjenjivati recepte ocjenama od 0 do 5 i pretraživati recepte po trajanju pripreme.
2. **Dodatne mogućnosti administratora:** administratori osim što mogu odobriti recepte isto tako ih mogu poništiti, nadalje mogu brisati sve recepte i sve komentare sa svih recepata i za kraj mogu suspendirati korisnike aplikacije (eng. ban).
3. **Dodatni uvjeti:** duljina raznih podataka koje korisnik šalje će biti određena dizajnom baze podataka, a veličina fotografija i videozapisa će biti određena HTTP poslužiteljem.

3. Baza podataka

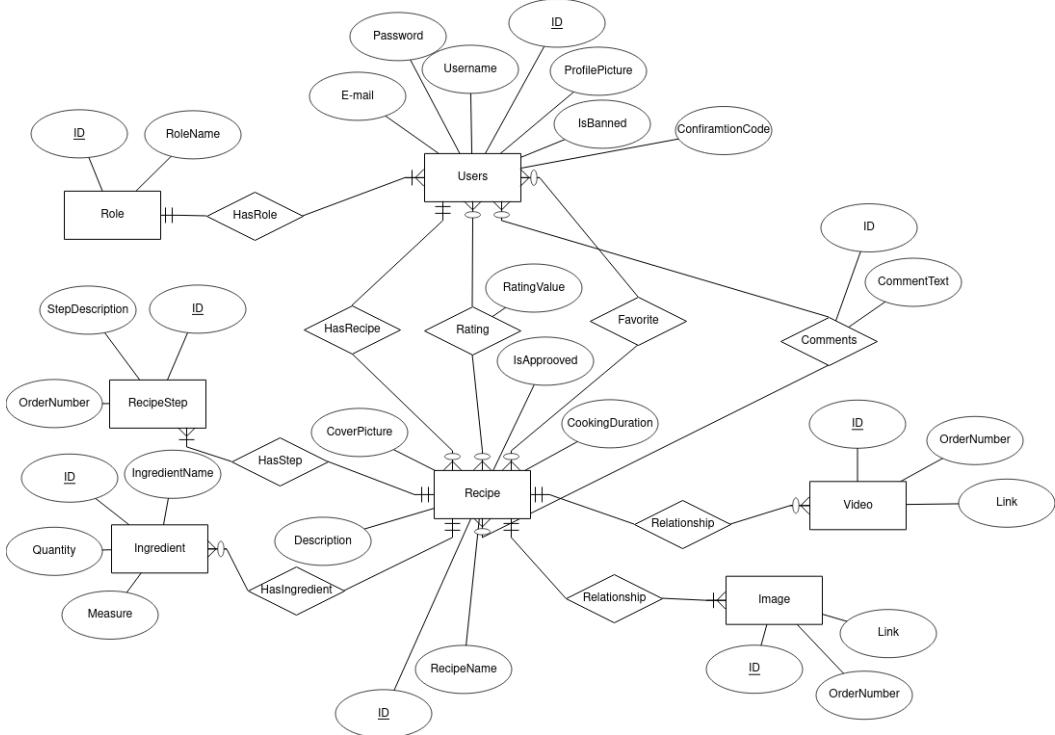
Kao upravitelj bazom podataka korišten je PostgreSQL i u svrhu lakšeg razvoja također je korištena aplikacija sa korisničkim sučeljem za izvođenje upita: PgAdmin. Sama baza je relacijskog tipa i modelirana je entitetima i vezama između njih. Za modeliranje je korišten web alat ERDPlus. Za upogonjavanje baze je korišten AWS (Amazon Web Services) točnije njihov RDS (Relational Database Service) servis koji se koristi upravo za upogonjavanje baza podataka u oblaku.

3.1. Entiteti

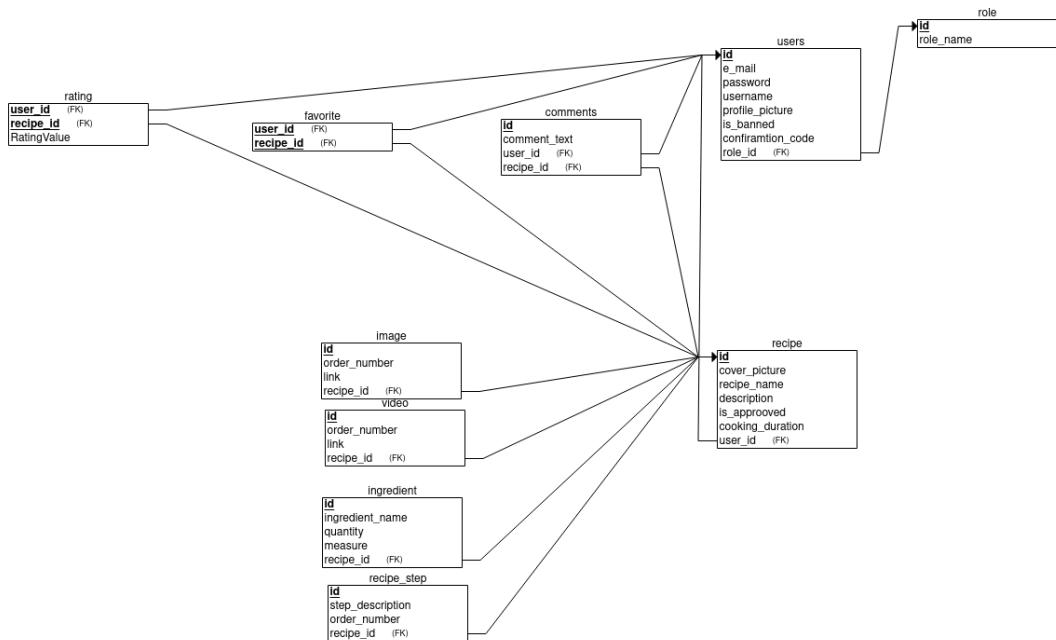
Entiteti su modelirani prema zahtjevima zadatka, a veze su modelirane prema stvarnim situacijama. Postojeći entiteti su: recept, sastojak, korak pripreme, korisnik, uloga, videozapis i fotografija. Dakle postoji glavni entitet koji je središte aplikacije, a to je recept njega opisuju njegov identifikacijski broj, ime, opis, trajanje pripreme te na kraju parametar koji pokazuje je li recept odobren od strane administratora ili nije. Nadalje recept ne bi imao smisla da nema svoje korake pripreme i potrebne sastojke. Također postoje fotografije i videozapisi jer svaki recept može imati više videozapisa ili fotografija koji će uređivati korisničko sučelje i pomoći korisnicima pri odabiru prikladnog recepta. Korisnici su također modelirani kao jedan od entiteta jer mogu interagirati sa receptima na razne načine, a te interakcije su opisane vezama kao što su komentiranje, dodavanje u favorite, ocjenjivanje... Slika modela (eng. Entity relationship diagram) je prikazana slikom 3.1

3.2. Relacije

Iz prethodno opisanog modela pretvorbom iz entiteta u relacijsku shemu dobiveno je 10 tablica. One odgovaraju entitetima i vezama između njih, tako se pojavljuju nove tablice koje predstavljaju: komentare, favorite i ocjene. Dobivene tablice su stvorene u oblaku na upogonjenoj bazi. Relacijska shema je prikaza slikom 3.2 i kodom 3.2



Slika 3.1: ERDiagram



Slika 3.2: Relacijska shema

h

```
CREATE TABLE role
(
    id SERIAL,
    role_name VARCHAR(20) NOT NULL,
    UNIQUE(role_name),
    PRIMARY KEY (id)
);

CREATE TABLE users
(
    e_mail VARCHAR(100) NOT NULL,
    password CHAR(60) NOT NULL,
    username VARCHAR(50) NOT NULL,
    id SERIAL,
    profile_picture VARCHAR(500) NOT NULL,
    is_banned BOOLEAN NOT NULL,
    role_id INT NOT NULL,
    is_confirmed BOOLEAN NOT NULL,
    confirmation_code char(30) NOT NULL,
    UNIQUE(username ,e_mail),
    PRIMARY KEY (id),
    FOREIGN KEY (role_id) REFERENCES Role(id)
);

CREATE TABLE recipe
(
    cover_picture VARCHAR(500) NOT NULL,
    recipe_name VARCHAR(50) NOT NULL,
    description VARCHAR(500) NOT NULL,
    id SERIAL,
    is_approved BOOLEAN NOT NULL,
    cooking_duration INT NOT NULL,
    user_id INT NOT NULL,
    PRIMARY KEY (id),
    FOREIGN KEY (user_id) REFERENCES Users(id)
);

CREATE TABLE recipe_step
(
    id SERIAL,
    step_description VARCHAR(500) NOT NULL,
    order_number INT NOT NULL,
    recipe_id INT NOT NULL,
    PRIMARY KEY (id),
    FOREIGN KEY (recipe_id) REFERENCES Recipe(id)
);

CREATE TABLE image
(
    id SERIAL,
    order_number INT NOT NULL,
    link VARCHAR(500) NOT NULL,
    recipe_id INT NOT NULL,
    PRIMARY KEY (id),
    FOREIGN KEY (recipe_id) REFERENCES Recipe(id)
);

CREATE TABLE ingredient
(
    id SERIAL,
    ingredient_name VARCHAR(50) NOT NULL,
    quantity INT NOT NULL,
    measure VARCHAR(50) NOT NULL,
    recipe_id INT NOT NULL,
    PRIMARY KEY (id),
    FOREIGN KEY (recipe_id) REFERENCES Recipe(id)
);

CREATE TABLE video
(
    id SERIAL,
    order_number INT NOT NULL,
    link VARCHAR(500) NOT NULL,
    recipe_id INT NOT NULL,
    PRIMARY KEY (id),
    FOREIGN KEY (recipe_id) REFERENCES Recipe(id)
);

CREATE TABLE favorite
(
    user_id INT NOT NULL,
    recipe_id INT NOT NULL,
    PRIMARY KEY (user_id , recipe_id),
    FOREIGN KEY (user_id) REFERENCES Users(id),
    FOREIGN KEY (recipe_id) REFERENCES Recipe(id)
);

CREATE TABLE rating
(
    rating_value INT NOT NULL,
    user_id INT NOT NULL,
    recipe_id INT NOT NULL,
    PRIMARY KEY (user_id , recipe_id),
    FOREIGN KEY (user_id) REFERENCES Users(id),
    FOREIGN KEY (recipe_id) REFERENCES Recipe(id)
);

CREATE TABLE comments
(
    id SERIAL,
    comment_text VARCHAR(200) NOT NULL,
    user_id INT NOT NULL,
    recipe_id INT NOT NULL,
    PRIMARY KEY (id),
    FOREIGN KEY (user_id) REFERENCES Users(id),
    FOREIGN KEY (recipe_id) REFERENCES Recipe(id)
);
```

Listing 3.1: SQL kod

4. Poslužiteljska strana

Poslužiteljska strana je pisana u programskom jeziku java verzije 11, pritom koristeći SpringBoot radni okvir. Kao arhitekturni stil korišten je GraphQL koji je alternativna REST-u. Projekt je modeliran kao maven projekt, a za upravljanje njime je korišten Git alat. Za upogonjavanje se također koristio AWS i njihov ElasticBeanstalk servis. U svrhe razvoja korišteno je okruženje VSCode.

4.1. GraphQL

4.1.1. Povijest

Jedan od većih problema dizajniranja web poslužitelja pa tako i samih web aplikacija u REST arhitekturnom stilu je problem prekomjernog dohvata podataka odnosno nedovoljnog dohvata podataka (eng. overfetching and underfetching). Ako bi se taj problem pokušao riješiti sa stvaranjem više krajinjih točaka (eng. endpoint), a uz to i više prijenosnih objekata takozvanih DTO-ova (Data Transfer Object), došlo bi do nagomilavanja izvornog koda. Tako je tvrtka Facebook došla do ideje GraphQL-a. Naime razvojni programeri su tada radili sa velikim brojem podataka koji su bili međusobno ugnježđeni i povezani na razne načine i da bi aplikacija radila dovoljno brzo i efikasno bilo je nužno dohvatiti samo one podatke koji se uistinu i koriste, a da se dohvaćaju sa što manjeg broja krajinjih točaka.

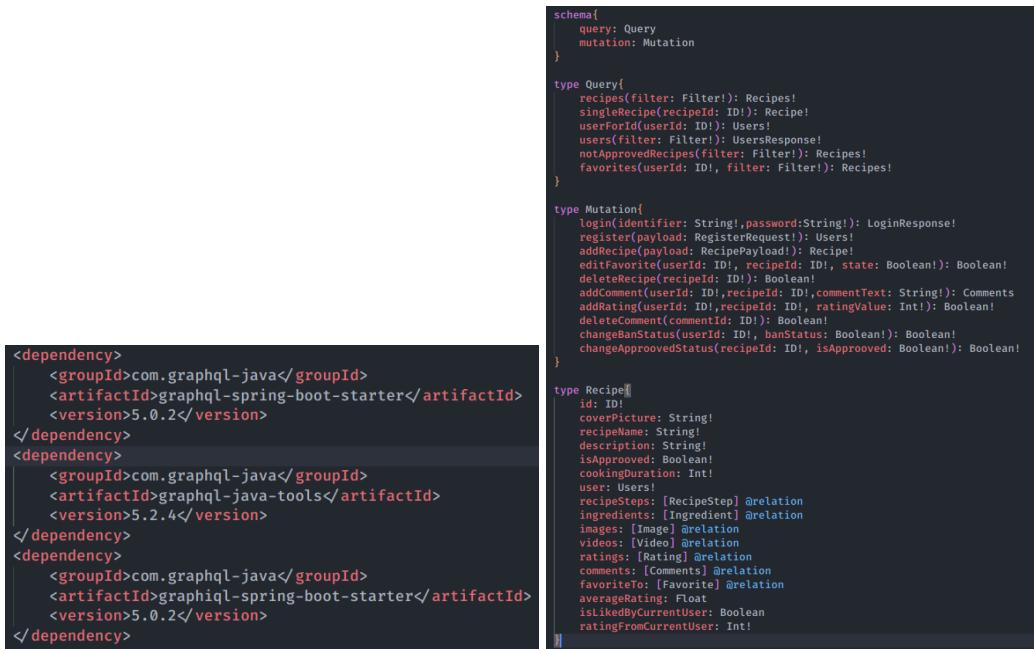
4.1.2. Definicija

GraphQL je upitni jezik specifično dizajniran za korisničku stranu aplikacije da bi mogla od poslužiteljske strane zatražiti i dobiti samo one podatke koji su potrebni. Također dopušta i definiranje formata i oblika podataka koji će se zahtijevati od poslužiteljske strane, a da se ujedno upiti šalju na samo jednu krajinju točku. U GraphQL-u postoje 2 vrste upita: upiti i mutacije (eng. query and mutation), razlika je u tome što

upiti služe za dohvaćanje podataka, a mutacije za mijenjanje podataka.

4.1.3. Implementacija

Unatoč tome GraphQL nudi puno izmjena i lakši pristup podacima u usporedi sa REST-om implementacija nije otežana. U okviru ovoga rada u pom.xml datoteku u kojoj se dodaju ovisnosti (eng. dependencies) bilo je potrebno dodati 2 nove ovisnosti: graphql-spring-boot-starter, graphql-java-tools i po potrebi treću ovisnost koja nudi mogućnost testiranja sučelja, a ta je graphiql-spring-boot-starter (Slika 4.1). Nakon toga ove ovisnosti zahtijevaju izradu jedne datoteke u kojoj će pisati sve moguće vrste podataka koje se mogu tražiti, i sve moguće upite koji se mogu primiti i takva datoteka se najčešće imenuje schema.graphqls (). U slučaju ovog rada podatci su bili ujedno i relacijski modeli iz baze podataka, a upiti postoje razni i bili su dodavani jedan po jedan ovisno o potrebi aplikacije (Slika 4.1). Iz priložene slike vidimo da postoji velik broj mutacija i upita, ali to i dalje ne mijenja činjenicu da postoji samo jedna krajnja točka koja je u ovom slučaju /graphql, također je priložen primjer jednog tipa podatka: Recipe i iz njegovog opisa možemo primjetiti da je to upravo jedan od relacijskih modela iz baze podataka u koji su dodani još neki dodatni parametri. Treba samo zapamtiti da se imena i tipovi polja iz javine klase i tipa podatka iz graphql sheme moraju podudarati jer se koristi refleksija za dohvati tih polja.



```

<dependency>
    <groupId>com.graphql-java</groupId>
    <artifactId>graphql-spring-boot-starter</artifactId>
    <version>5.0.2</version>
</dependency>
<dependency>
    <groupId>com.graphql-java</groupId>
    <artifactId>graphql-java-tools</artifactId>
    <version>5.2.4</version>
</dependency>
<dependency>
    <groupId>com.graphql-java</groupId>
    <artifactId>graphiql-spring-boot-starter</artifactId>
    <version>5.0.2</version>
</dependency>

```

```

schema{
    query: Query
    mutation: Mutation
}

type Query{
    recipes(filter: Filter!): Recipes!
    singleRecipe(recipeId: ID!): Recipe!
    userForId(userId: ID!): Users!
    users(filter: Filter!): UsersResponse!
    notApprovedRecipes(filter: Filter!): Recipes!
    favorites(userId: ID!, filter: Filter!): Recipes!
}

type Mutation{
    login(identifier: String!, password:String!): LoginResponse!
    register(payload: RegisterRequest!): Users!
    addRecipe(payload: RecipePayload!): Recipe!
    editFavorite(userId: ID!, recipeId: ID!, state: Boolean!): Boolean!
    deleteRecipe(recipeId: ID!): Boolean!
    addComment(userId: ID!, recipeId: ID!, commentText: String!): Comments!
    addRating(userId: ID!, recipeId: ID!, ratingValue: Int!): Boolean!
    deleteComment(commentId: ID!): Boolean!
    changeBanStatus(userId: ID!, banStatus: Boolean!): Boolean!
    changeApprovedStatus(recipeId: ID!, isApproved: Boolean!): Boolean!
}

type Recipe{
    id: ID!
    coverPicture: String!
    recipeName: String!
    description: String!
    isApproved: Boolean!
    cookingDuration: Int!
    user: Users!
    recipeSteps: [RecipeStep] @relation
    ingredients: [Ingredient] @relation
    images: [Image] @relation
    videos: [Video] @relation
    ratings: [Rating] @relation
    comments: [Comments] @relation
    favoriteTo: [Favorite] @relation
    averageRating: Float!
    isLikedByCurrentUser: Boolean!
    ratingFromCurrentUser: Int!
}

```

(a) GraphQL ovisnosti

(b) Primjer GraphQL sheme

Slika 4.1: GraphQL postavke

Nakon dodavanja sheme i ovisnosti potrebno je dodati takozvane Resolvere. U ovome projektu postoje 3 vrste resolvera: QueryResolver, MutationResolver i obični Resolver, oni su u prethodno navedenim ovisnostima definirani kao sučelja: GraphQLQueryResolver, GraphQLMutationResolver, GraphQLResolver. Prvi služi za prihvatanje upita, drugi služi za prihvatanje mutacija, a treći služi za rješavanje problema spomenutih dodatnih parametara koji su umetnuti u GraphQL tip, na način da se definira način dohvata tih podataka. Također se može implementirati sučelje GraphQLErrorHandler koji će presretati greške sa poslužiteljske strane i vraćati odgovarajuću poruku korisničkoj strani, jer naime GraphQL ima jednu manu, a ta mana je da se na korisničku stranu uvijek vraća HTTP status kod 200 OK, pa se rukovanje greškama obavlja na drukčiji način. Taj način je da se u odgovor u slučaju greške doda parametar errors koji će sadržavati sve greške koje su se dogodile (slika 4.2). Ukoliko se ne implementira sučelje GraphQLErrorHandler poslužitelj će uvijek vraćati predefiniranu pogrešku (slika 4.3).

```

query Recipes($filter: Filter!){
  recipes(filter: $filter){
    user{
      id,
      username,
      profilePicture
    }
    id,
    ingredients{
      ingredientName
    }
    averageRating,
    cookingDuration,
    coverPicture,
    recipeName,
    isLikedByCurrentUser
  }
  numberOfPages,
  currentIndex
}

```

Slika 4.2: Primjer greške ukoliko postoji GraphQLErrorHandler

```

query Recipes($filter: Filter!){
  recipes(filter: $filter){
    user{
      id,
      username,
      profilePicture
    }
    id,
    ingredients{
      ingredientName
    }
    averageRating,
    cookingDuration,
    coverPicture,
    recipeName,
    isLikedByCurrentUser
  }
  numberOfPages,
  currentIndex
}

```

Slika 4.3: Primjer predefinirane pogreške

Za kraj nakon implementacije svih navedenih sučelja (slika 4.4) sada se uspješno može izvršiti prvi upit koji je demonstriran kroz sučelje GraphQL koji prikazuje strukturu upita i odgovora (slika 4.5).



Slika 4.4: Izgled graphql implementacije

```

query Recipes($filter: Filter!){
  recipes(filter: $filter) {
    recipes {
      id,
      user {
        id,
        username,
        profilePicture
      },
      ingredients {
        ingredientName
      },
      averageRating,
      cookingDuration,
      coverPicture,
      recipeName,
      isLikedByCurrentUser
    }
    numberOfPages,
    currentIndex
  }
}

QUERY VARIABLES
{
  "filter": {
    "index": 1
  }
}

```

```

{
  "data": {
    "recipes": [
      {
        "user": {
          "id": "51",
          "username": "marko",
          "profilePicture": "https://firabasestorage.googleapis.com/v0/b/finalbsthesis.appspot.com/o/marko_profilePicture.png?alt=media"
        },
        "id": "52",
        "ingredients": [
          {
            "ingredientName": "eggs"
          },
          {
            "ingredientName": "sugar"
          },
          {
            "ingredientName": "milk"
          },
          {
            "ingredientName": "butter"
          },
          {
            "ingredientName": "wheat flour"
          },
          {
            "ingredientName": "baking powder"
          },
          {
            "ingredientName": "vanilla sugar"
          },
          {
            "ingredientName": "salt"
          }
        ],
        "averageRating": 5,
        "cookingDuration": 30,
        "coverPicture": "https://firabasestorage.googleapis.com/v0/b/finalbsthesis.appspot.com/o/recipeImage_1815604081941518692.png?alt=media",
        "recipeName": "classic american pancakes",
        "isLikedByCurrentUser": false
      }
    ]
  }
}

```

Slika 4.5: Primjer ispravnog upita

4.2. Autentifikacija i autorizacija

4.2.1. JWT

U području autentifikacije i autorizacije se vidi jedna velika razlika između mobilne aplikacije i aplikacije koja se pokreće na web pregledniku, a to je odsutnost sesije. Zato se mobilne aplikacije koriste drugim mehanizmom, a to su tokeni. U ovoj aplikaciji se koriste tokeni pod imenom JWT (JSON Web Token), takvi tokeni koriste kriptografske algoritme (kao na primjer HMACSHA256) da bi se podatci na siguran način mogli prenositi između dvije strane. Pa će se u takvom tokenu nalaziti korisničko ime, istek tokena i slični podatci koji će omogućiti autentifikaciju i autorizaciju korisnika aplikacije.

4.2.2. Autentifikacija

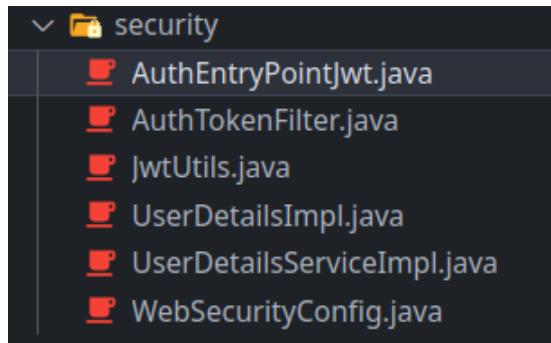
Da bi se ovo ostvarilo prvo je bilo potrebno dodati ovisnost prema SpringSecurity-u (slika 4.6) i prema jjwt. Nakon toga da bi aplikacija mogla doći do podataka trebalo je implementirati dva sučelja: UserDetails i UserDetailsService. Prvo sučelje služi za omotavanje osnovnih podataka o korisniku, kao na primjer korisničko ime, lozinka, je li korisnik suspendiran, je li korisnik potvrdio svoj identitet, koje su korisnikove uloge (administrator ili korisnik) i slično, a drugo sučelje služi za dohvatih podataka na osnovu korisničkog imena. U ovoj aplikaciji sučelje UserDetails omotava podatke iz baze podataka, a drugo sučelje dohvaća podatke iz baze i sprema ih u prvo sučelje.

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-security</artifactId>
</dependency>
<dependency>
    <groupId>io.jsonwebtoken</groupId>
    <artifactId>jjwt</artifactId>
    <version>0.9.1</version>
</dependency>
```

Slika 4.6: Ovisnost za autentifikaciju i autorizaciju

Nadalje da bi se ti podatci mogli dohvatiti u bilo kojem trenutku potrebno ih je spremiiti negdje u nekom trenutku. Zato je bilo potrebno napraviti jedan Filter. Filter su posebni objekti u web aplikaciji koji presreću bilo koji zahtjev koji stigne na poslužitelj, obrade taj zahtjev i onda ga proslijede dalje prema web aplikaciji. Filter za autentifikaciju će provjeriti postoji li HTTP zaglavljje pod imenom: Authorization i ako postoji prepostaviti će da se unutra nalazi token kojeg će pokušati isparsirati i iz njega dohvatiti korisničko ime. Na osnovu tog korisničkog imena i uz pomoć sučelja UserDetailsService vrlo jednostavno se dohvati sučelje UserDetails. Na kraju će filter spremiti bitne podatke u takozvani SecurityContextHolder koji će pružiti mogućnost dohvata trenutnog korisnika u bilo kojem trenutku.

Jedan od mogućih scenarija je da će podatci poslani u tokenu biti ili nevažeći ili isteknuti zato je potrebno napraviti takozvani EntryPoint koji će u slučaju pogrešne autorizacije korisniku vratiti odgovarajuću poruku. Taj EntryPoint se naziva AuthenticationEntryPoint koji će preko HTTP odgovora poslati pogrešku u svojoj metodi commence. Struktura svih implementiranih sučelja u projektu prikazana je na slici 4.7



Slika 4.7: Struktura sigurnosnog dijela

4.2.3. Autorizacija

Na kraju je potrebno definirati autorizaciju i spojiti s autentifikacijom, a to se obavlja kroz takozvani WebSecurityConfig. U kojem je definirano koji će se EntryPoint koristiti u slučaju pogreške, koji algoritam se koristi za šifriranje lozinki, kada se poziva filter za autentifikaciju i kada se obavlja autorizacija. U ovoj aplikaciji se koristi prethodno definirani EntryPoint, za šifriranje lozinke se koristi BCrypt algoritam, filter se poziva prije svakog zahtjeva, a autorizacija je omogućena da se definira za svaku metodu zasebno uz pomoć anotacije @PreAuthorize (slika 4.8). Koristi se na način da se u tijelo anotacije stave sve uloge koje smiju pristupati anotiranoj metodi (slika 4.9). Te anotacije se najčešće koriste ili u QueryResolver-u ili u MutationResolver-u i tako će se ograničiti pristup neautoriziranim korisnicima.

```

@Bean
public PasswordEncoder passwordEncoder() {
    return new BCryptPasswordEncoder();
}

@Override
protected void configure(HttpSecurity http) throws Exception {
    http.cors().and().csrf().disable()
        .exceptionHandling().authenticationEntryPoint(unauthorizedHandler).and()
        .sessionManagement().sessionCreationPolicy(SessionCreationPolicy.STATELESS).and()
        .authorizeRequests()
        .anyRequest().permitAll();
    http.addFilterBefore(authenticationJwtTokenFilter(), beforeFilter: UsernamePasswordAuthenticationFilter.class);
}

```

Slika 4.8: Izgled sigurnosne konfiguracije

```

@PreAuthorize("hasAuthority('USER') or hasAuthority('MODERATOR') or isAnonymous()")
public Recipes getRecipes(Filter filter) {
    return recipeService.getRecipesForFilter(filter);
}

```

Slika 4.9: Izgled metode sa anotacijom @PreAuthorize

4.3. Pristup podacima

Za pristup podacima se koriste dva sloja prvi sloj je sloj za perzistenciju a drugi sloj je sloj usluge. Sloj za perzistenciju modeliran je uz pomoć takozvanog ORM-a (Object Relational Mapper) čija je zadaća preslikavanje relacija u javne objekte. Postoji više vrsta različitih ORM-ova, a ovdje se koristi Hibernate. Za sve to se morala dodati ovisnost prema odgovarajućim JPA i JDBC maven ovisnostima (slika 4.10 (a)).

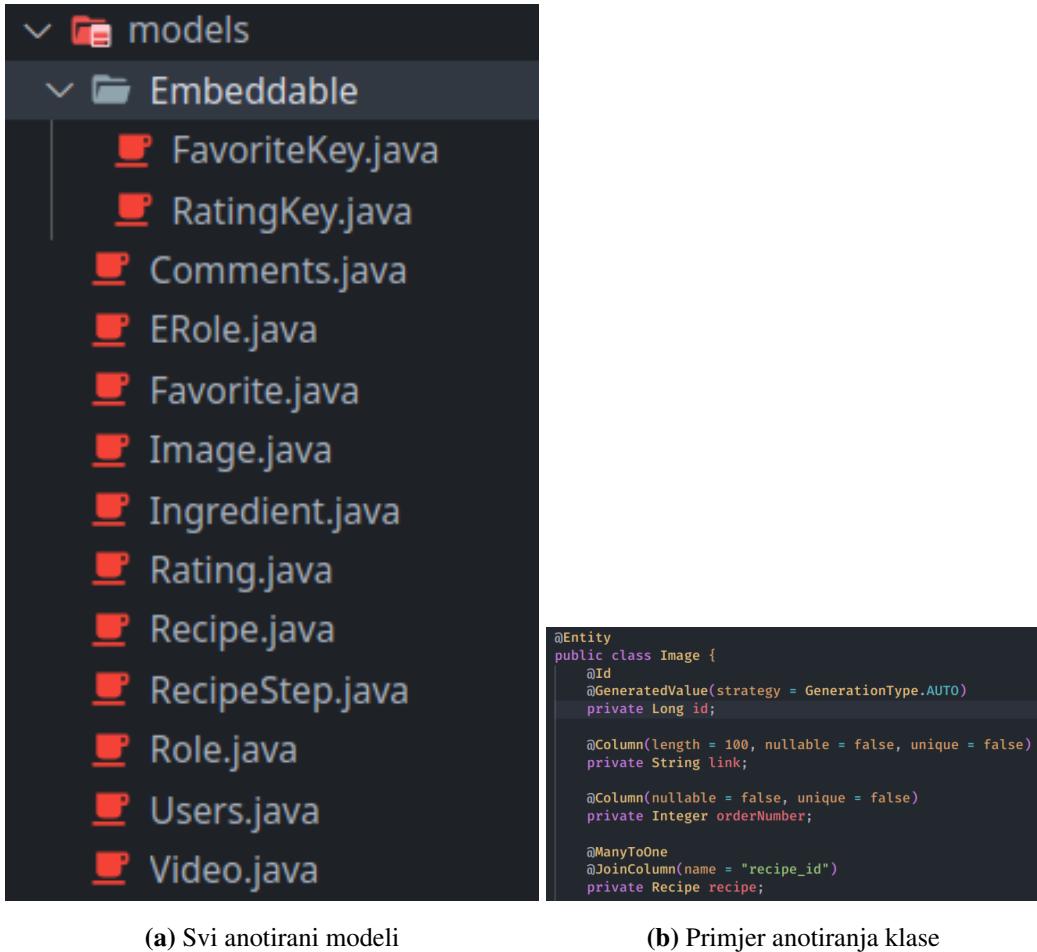


Slika 4.10: Ovisnosti za pristup podacima i sloj usluge

4.3.1. Sloj za perzistenciju

U programskom jeziku java hibernate se koristi kroz JPA (Java Persistence API) programsko sučelje. JPA zahtijeva da se način preslikavanja iz relacije u objekt definira ili anotacijama ili u posebnoj konfiguracijskoj datoteci, u ovome projektu je to učinjeno preko anotacija, a sve takve klase prikazane su na slici 4.11 (a). Anotacije i način anotiranja koji se mora poštivati prikazan je na slici 4.11 (b). Može se primjetiti da u modelima ovoga projekta nisu anotirane sve veze koje su određene samom bazom podataka jer je to riješeno uz pomoć Resolvera iz GraphQL-a. Nakon kreiranja svih modela potrebno je napraviti sučelja koja nasleđuju sučelje JPARepository preko kojeg se pristupa samim podacima, a u tom sučelju su definirane metode kao na primjer metoda za dohvaćanje svih relacija, metoda za dohvaćanje relacija po identifikacijskom atributu, ili nekom drugom atributu i slične metode.

Također iz slike 4.11 (b) može se primjetiti da se slika u bazu spremi kao hiperveza na samo sliku, a to je moguće jer se za spremanje slika koristi firebase. Za pristup



(a) Svi anotirani modeli

(b) Primjer anotiranja klase

Slika 4.11: JPA konfiguracija

firebase-u je bilo potrebno dodati ovisnost (slika 4.10) i jednu datoteku u kojoj su zapisani podaci za pristup bazi koja iz sigurnosnih razloga nije priložena. Nakon je toga je vrlo lako uz pomoć metode *create* primljenu sliku dignuti na firebase oblak.

4.3.2. Sloj usluge

Jedno od SOLID načela programiranja ("*Single responsibility*") nalaže da jedna klasa ima samo jednu odgovornost zato je sloj za perzistenciju od *resolver-a* odvojen slojem usluge. U sloju usloge se nalazi sva logika potrebna za pristup i obradi podatka da bi se razdvojila odgovornost obrade podataka od odgovornosti obrade zahtjeva.

Također osim same obrade podataka u sloju usluge se nalazi i klasa koja služi za slanje električne pošte. Ta usluga se koristi za slanje poruke za potvrdu identiteta i za slanje liste za kupnju koja se šalje ukoliko neki korisnik doda recept u svoje favorite. Tijelo poruke se gradi kao HTML dokument, a šalje se uz pomoć Java Mail API-a za

koji je bilo potrebno dodati ovisnost prikazanu na slici 4.10. I konačno sve usluge su prikazane na slici 4.10 (b).

4.4. Upogonjavanje

Upogonjavanje se obavlja uz pomoć github actions gdje je definiran takozvani *CI/CD pipeline* (slika 4.12) uz pomoć kojeg se kod prevodi u .jar arhivu i podiže na AWS *ElasticBeanstalk* servis. Jedini problem je bio što nakon upogonjavanja poslužitelj može primati relativno male zahtjeve , a za ovu aplikaciju su potrebni poprilično veliki zahtjevi jer se šalju i fotografije i videozapisi zato se uz .jar arhivu mora dodati i konfiguracijska datoteka za nginx poslužitelj koja će povećati pretpostavljenu maksimalnu veličinu zahtjeva. To je postignuto uz pomoć maven-ovog antrun *plugin-a* (slika 4.13) koji će .jar arhivu i konfiguracijsku datoteku spremiti u .zip arhivu i proslijediti to dalje.

```
build_and_deploy_backend:
  name: Build jar and deploy
  runs-on: ubuntu-latest
  steps:
    - name: Checkout source code
      uses: actions/checkout@v2
    - name: Generate deployment package
      working-directory: ./backend/zavrnsi_rad
      run: mvn clean package
    - name: Deploy to EB
      uses: einaregilsson/beanstalk-deploy@v20
      with:
        wait_for_environment_recovery: 60
        aws_access_key: ${{ secrets.AWS_ACCESS_KEY_ID }}
        aws_secret_key: ${{ secrets.AWS_SECRET_ACCESS_KEY }}
        application_name: zavrnsi_rad
        environment_name: Zavrnsirad-env
        version_label: ${{ github.run_number }}
        region: us-east-1
        deployment_package: backend/zavrnsi_rad/target/zavrnsi_rad-0.0.1-SNAPSHOT.zip
```

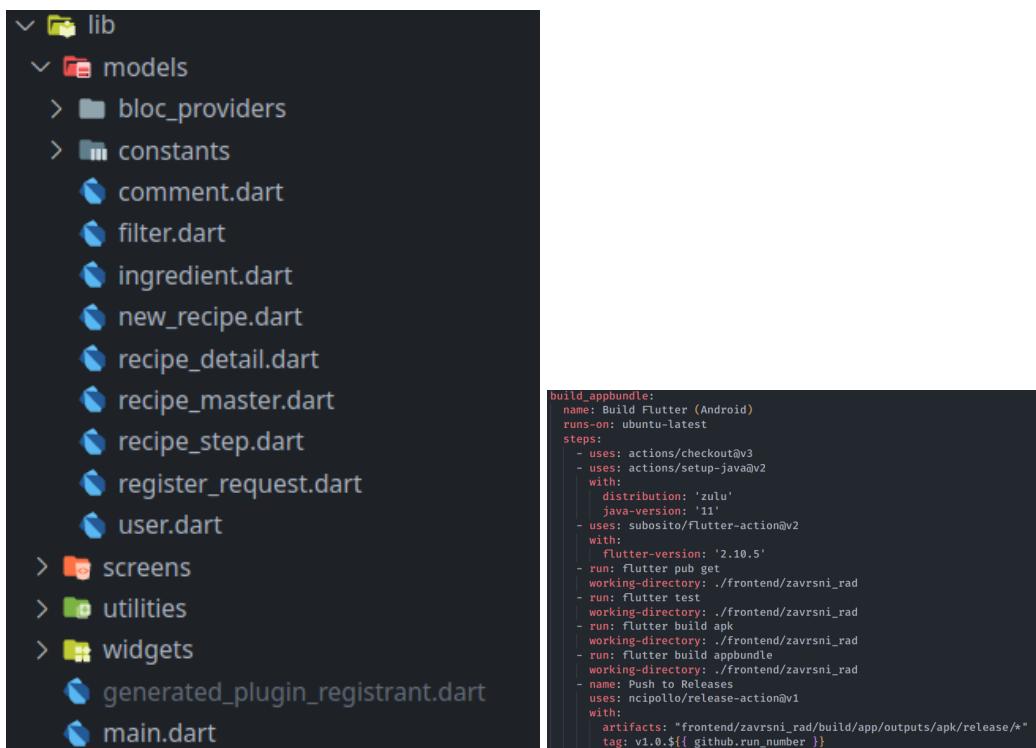
Slika 4.12: Github actions zadatak za podizanje web poslužitelja

```
<plugin>
  <groupId>org.apache.maven.plugins</groupId>
  <artifactId>maven-antrun-plugin</artifactId>
  <executions>
    <execution>
      <id>prepare</id>
      <phase>package</phase>
      <configuration>
        <target>
          <copy todir="${{project.build.directory}}/${{project.build.finalName}}/" overwrite="false">
            <fileset dir="/" includes=".ebextenstions/**" />
            <fileset dir="${{project.build.directory}}" includes="*.jar" />
          </copy>
          <copy todir="${{project.build.directory}}/${{project.build.finalName}}/" overwrite="false">
            <fileset dir="/" includes=".platform/**" />
            <fileset dir="${{project.build.directory}}" includes="*.jar" />
          </copy>
          <zip destfile="${{project.build.directory}}/${{project.build.finalName}}.zip" basedir="${{project.build.directory}}/${{project.build.finalName}}" />
        </target>
      </configuration>
      <goals>
        <goal>run</goal>
      </goals>
    </execution>
  </executions>
</plugin>
```

Slika 4.13: Plugin za stvaranje zip arhive za upogonjavanje

5. Korisnička strana

Korisnička strana je pisana u programskom jeziku dart, pritom koristeći Flutter radni okvir. Struktura projekta je napravljena potpuno slobodno (slika 5.1(a)). Za komunikaciju sa poslužiteljskom stranom korišten je paket flutter-graphql. Za upogonjavanje aplikacije korištena je github-ova mogućnost stvaranja izdanja. Upogonjavanje je obavljeno uz pomoć github actions (slika 5.1)(b). U svrhe razvoja korišteno je okruženje VSCode.



(a) Izgled strukture projekta

(b) GitHub actions za upogonjavanje aplikacije

Slika 5.1: Osnovne karakteristike korisničke strane

5.1. Flutter

Flutter je google-ov javno dostupni (*eng. Open Source*) projekt koji je namijenjen za stvaranje lijepih i brzih mobilnih, web, računalnih ili ugrađenih aplikacija. Karakteristika *flutter-a* je da se aplikacija sa samo jednim programskim kodom može pokrenuti na svim navedenim uređajima. Također jedna od prednosti tog ovog radnog okvira je da je vrlo lako napraviti dinamične aplikacije jer se koristi upravljanje stanjem (*eng. state management*). Postoji više načina kako pristupiti ovoj funkcionalnosti, a u ovome radu korištem je *BLoC* pristup koji je ujedno i google-ov preporučeni pristup.

Također se sadržaj dinamički mijenja u ovisnosti o upitima na poslužiteljsku stranu i ta dinamičnost se postiže uz pomoć *widgeta* namijenjenih za slanje graphql upita koji se zovu *Query* i *Mutation*. Navedeni *widgeti* imaju jednu funkciju koja se poziva svaki put kada se promjeni stanje upita i zove se *builder*. Toj funkciji se kao ulazni parametar šalje rezultat i stanje izvođenja upita to jest mutacije i dužnost joj je vratiti *widget* koji će se prikazati na ekranu.

5.1.1. BLoC

BLoC (Business Logic Components) je obrazac za upravljanjem stanja aplikacije. Karakteristika ovog obrasca je da međusobno razdvaja prikaz komponente, upravljanje njenim stanjem i logiku iza upravljanja stanjem. Srž obrasca je da bi sve u aplikaciji trebalo biti modelirano kao tok događaja na način da jedna komponenta na osnovu interakcije s okolinom kreira događaje i šalje ih u ulazni tok događaja, a svaka komponenta koja ovisi o tom toku događaja će kroz izlazni tok događaja primiti te podatke i na osnovu njih ažurirati svoje stanje. U implementaciji se to postiže uz pomoć 3 stvari: klase koje predstavljaju događaje (u njima se mogu prenositi potrebni podaci) (slika 5.2 (a)), *BlocProvider* (slika 5.2 (a)) i *BlocBuilder* (slika 5.2 (b)). *BlocProvider* predstavlja dio *BLoC-a* koji se bavi presretanjem svakog ulaznog događaja, obrađivanjem i slanjem u izlazni tok, a *BlocBuilder* služi za izgradnju komponente na osnovu trenutnog događaja u izlaznom toku.

5.2. Ekrani

U radnom okviru flutter ne postoji razlika između ekrana (*eng screen*) i na primjer teksta jer je sve definirano kao takozvani *widget*, ali iz semantičkih razloga svaku skupinu

```

import 'package:bloc/bloc.dart';

abstract class ImageEvent {}

class AddImage extends ImageEvent {
    final File image;
    AddImage({required this.image}) : super();
}

class RemoveImage extends ImageEvent {
    final int index;
    RemoveImage({required this.index}) : super();
}

class BlocImages extends Bloc<ImageEvent, List<File>> {
    BlocImages() : super([]) {
        on<AddImage>((event, emit) {
            state.add(event.image);
            emit([ ... state]);
        });
        on<RemoveImage>((event, emit) {
            state.removeAt(event.index);
            emit([ ... state]);
        });
    }
}

```

```

BlocBuilder<BlocImages, List<File>>(
    builder: ((context, state) {
        newRecipe.images = state
            .map((e) => base64Encode(e.readAsBytesSync()))
            .toList();
        return ImagesInputWidget(images: state);
    }),
), // BlocBuilder

```

(a) BLoCProvider i modeli događaja

(b) Primjer BLoCBuilder-a

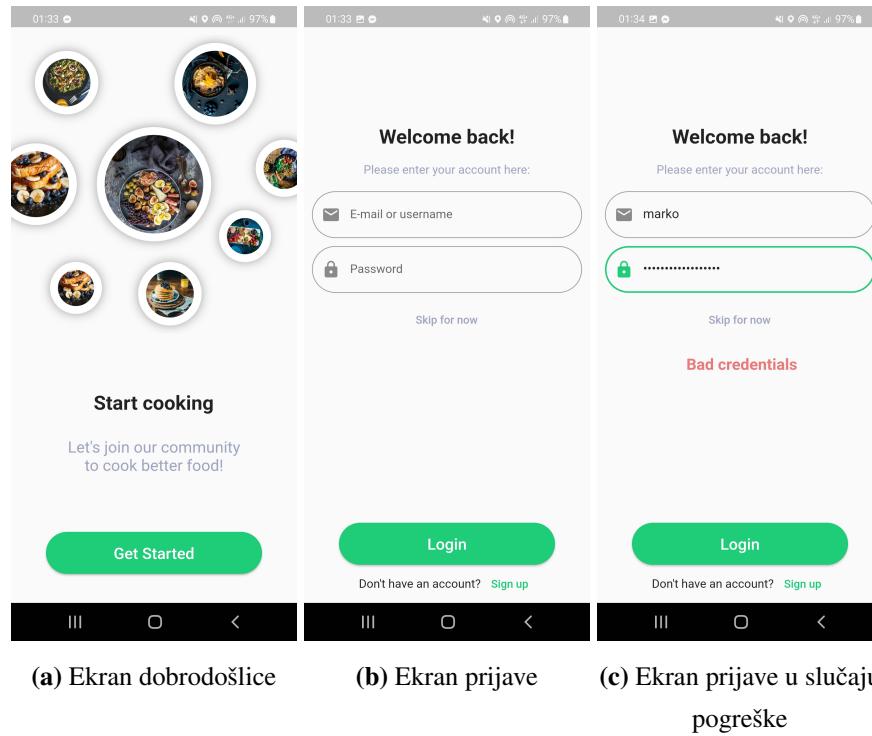
Slika 5.2: Implementacija BLoC-a

widgeta koja se prikazuje u istome trenutku nazivamo ekran, također se broj ekrana često uzima kao stupanj složenosti aplikacije. Tako ovaj rad ima 9 ekrana koji nisu dostupni svim korisnicima nego su podijeljeni po ulogama.

5.2.1. Ekran dobrodošlice, prijave i registracije

Ekran dobrodošlice (slika 5.3 (a)) nema nekakvu posebnu namijenu osim estetske to jest da bi korisniku ostavila dobar dojam. Također karakteristika ekrana dobrodošlice je da se prikazuje samo jednom i to kada se aplikacija prvi puta pokreće na trenutnom uređaju. To svojstvo se postiže podizanjem zastavice u memoriji uređaja uz pomoć sučelja "SharedPreferences". Ekran za prijavu sadrži 2 polja za unos i 3 gumba (slika 5.3 (b)). Prvo polje je polje za unos korisničkog imena ili e-pošte, drugo polje je polje za unos pripadajuće lozinke. Ukoliko je uneseno krivo korisničko ime ili lozinka prikazuje se odgovarajuća poruka (slika 5.3 (c)). Nadalje prvi gumb je gumb za preskakanje prijave i prelazak na stranicu sa svim receptima, drugi gumb je gumb za potvrdu prijave, a treći gumb je gumb za prelazak na ekran za registraciju.

Ekran za registraciju sadrži 5 polja za unos: unos računa elektroničke pošte, unos korisničkog imena, unos lozinke, potvrda lozinke i odabir profilne slike koje je, za razliku od ostalih opcionalno polje te 2 gumba (slika 5.4 (a)). Karakteristika polja za lozinku je da ima posebne validacijske kriterije, koji zahtijevaju da lozinka mora sadržavati barem 8 znakova, jedno veliko slovo i jedan specijalni znak. Nakon uspješne registracije



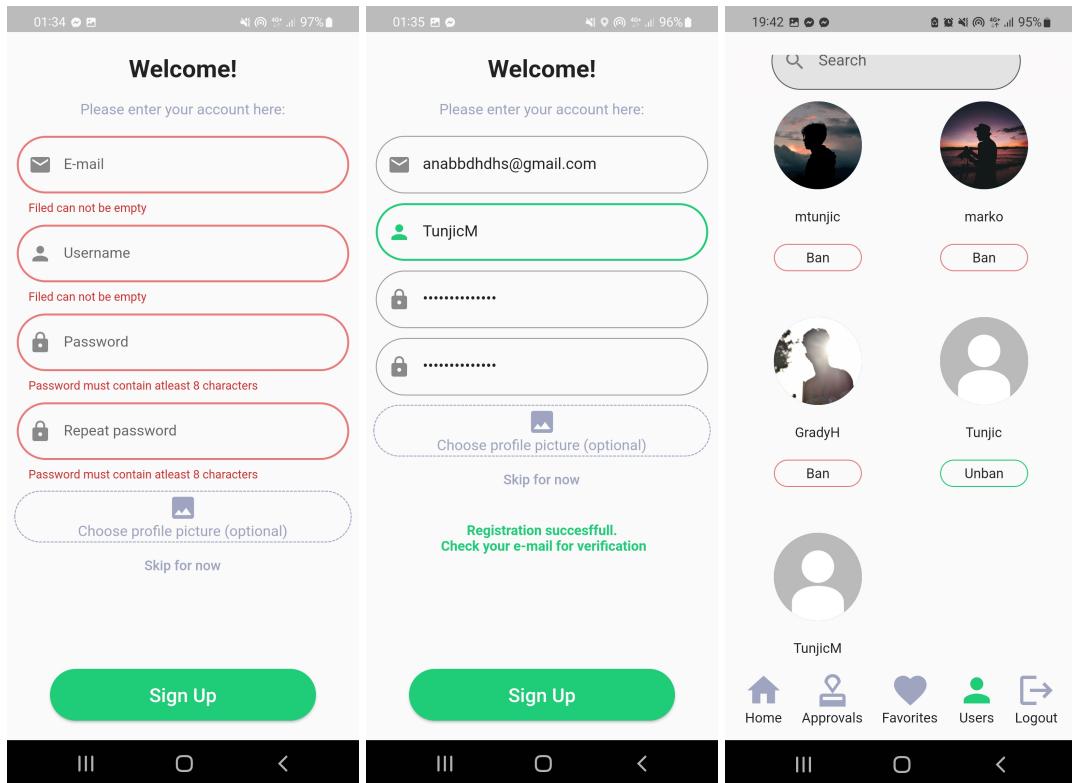
Slika 5.3: Ekran dobrodošlice i prijave

prikaže se poruka da je registracija uspješno izvršena (slika 5.4 (b)) i da je potrebno izvršiti potvrdu identiteta putem e-mail račun-a koji je bio unesen.

5.2.2. Ekran za pregled svih recepata

Ekran za pregled svih recepata će uvijek biti prvi ekran koji će se prikazati ukoliko postoji prijavljeni korisnik, a to se također postiže uz pomoć dijeljenih tajni (eng. SharedPreferences). Estetski gledano ekran će biti jednak i za korisnike (prijavljene ili neprijavljene) i za administratore jedine dvije razlike su u prikazu gumba za dodavanje u favorite i u navigacijskoj traci koja sadržava samo one gumbove koji vode do ekrana koji je dozvoljen trenutno prijavljenom to jest neprijavljenom korisniku (slika 5.5).

U dijeljenje tajne se spremaju podatci o trenutnom korisniku kao na primjer: korisničko ime, identifikacijski broj i JWT token za autentifikaciju i autorizaciju. To je bitno jer će se prije svakog upita, koji se obavlja dalnjom interakcijom korisnika i aplikacije, u HTTP zaglavljje staviti spomenuti token uz pomoć kojeg će poslužiteljska strana obaviti autentifikaciju i autorizaciju. Ukoliko korisnik ne bude uspješno autoriziran biti će odjavljen i preusmjeren na ekran za prijavu sa odgovarajućom porukom.

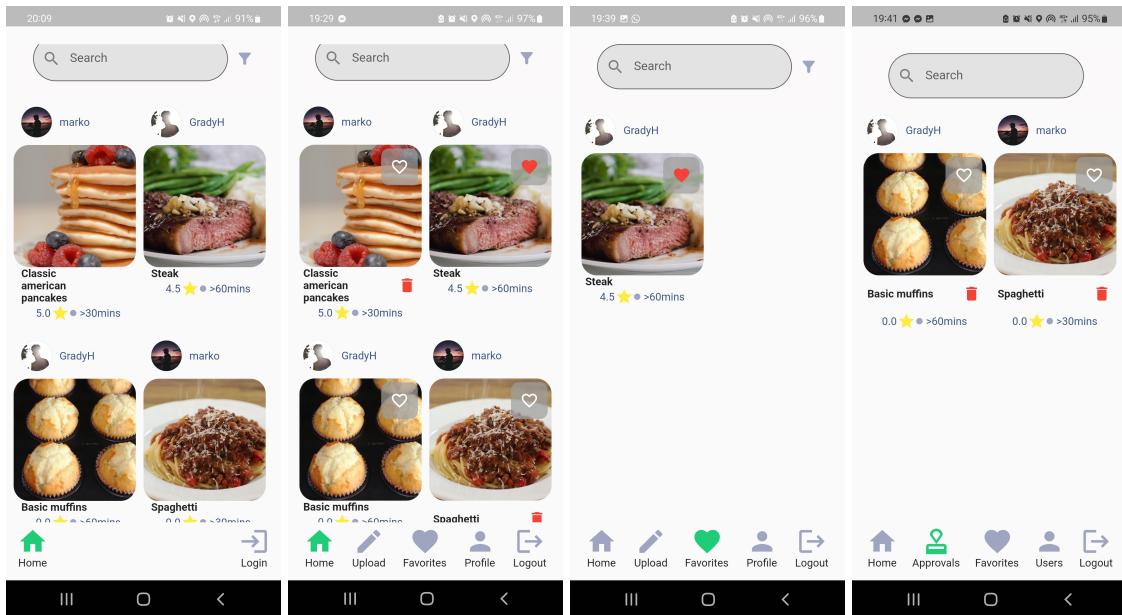


(a) Ekran neuspješne registracije (b) Ekran uspješne registracije (c) Ekran za upravljanje korisnicima

Slika 5.4: Ekrani za registraciju i upravljanje korisnicima

Na ovom ekranu možemo vidjeti sve recepte koji sa nalaze u bazi podataka. Recepti su podijeljeni u stranice, a svaka stranica sadrži 10 recepata, ukoliko postoji više od 10 recepata na dnu ekrana će se pojaviti jedan poseban *widget* koji služi za prelazak na sljedeću, prethodnu ili neku određenu stranicu. Svaki recept je opisan autorom i proflinom slikom autora, pozadinskom slikom recepta, nazivom, prosječnom ocjenom i vremenom potrebnim za pripremu. Klikom na bilo koji opisnik recepta se otvara novi ekran, a to je ekran za pregled pojedinog recepta. Na kraju postoji i gumb koji služi za prisanje recepta i koji je prikazan samo u slučaju da je recept pripada trenutno prijavljenom korisniku ili je trenutno prijavljeni korisnik zapravo administrator.

Također na samom vrhu ekrana postoji *widget* koji služi za filtriranje recepta po nazivu, a pored njega se nalazi jedan gumb koji prilikom dodira otvara dijalog kojim se mogu filtrirati recepti po sastojcima i vremenu pripreme.



Slika 5.5: Ekrani sa receptima

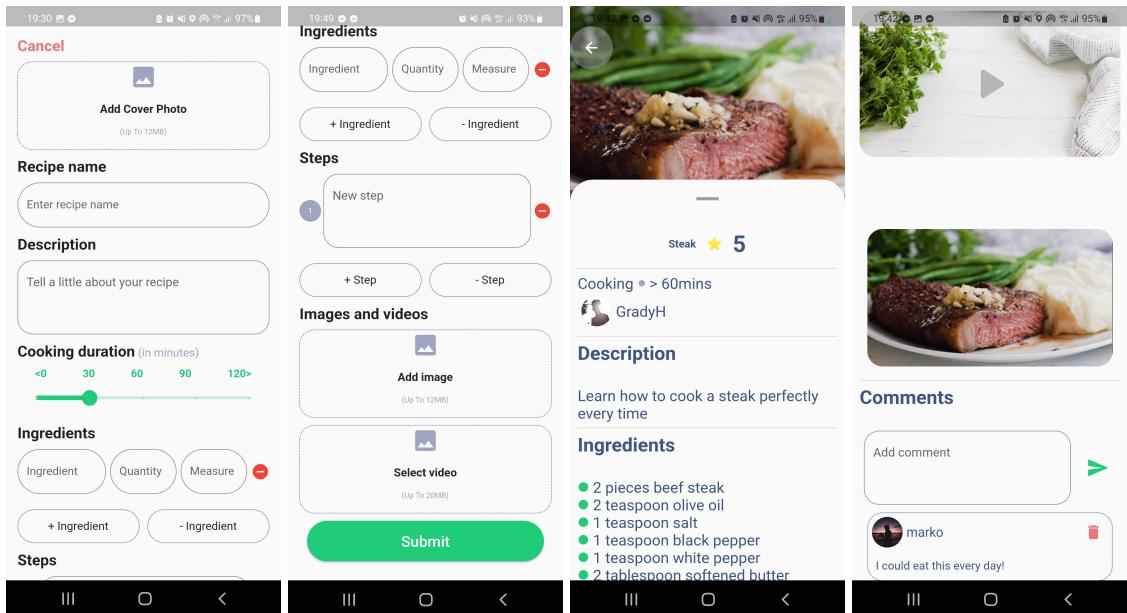
5.2.3. Ekran za pregled pojedinog recepta

Ekran za pregled pojedinog recepta će prikazivati sve podatke o receptu, a to su: sastojci, koraci pripreme, svi videozapisi i fotografije koje se nalaze u scrolabilnoj listi s kojom se interagira pokretima lijevo i desno, te dodatno ukoliko trenutno prijavljeni korisnik ima dovoljne ovlasti moći će interagirati sa gumbom za ocijenjivanje i poljem za unos komentara

Ovaj ekran je također poprilično jednak i za obične korisnike i za administratore, sa razlikom da administratori imaju pristup gumbu za odobravanje ili onemogućavanje recepta i ne prijavljeni korisnici nemaju pristup gumbu za ocijenjivanje i komentiranje (slika 5.6 (c) i (d)).

5.2.4. Ekran za dodavanje novog recepta

Ekran za dodavanje novog recepta je dostupan samo prijavljenim korisnicima koji nisu administratori. Ovaj ekran se sastoji od velikog broja polja za unos i odabir videozapisa i fotografija te gumba za potvrdu. Karakteristika svakog polja je da ne smije biti prazno i ne smije sadržavati više znakova od broja koji je određen bazom podataka.



(a) Prvi dio ekrana za dodavanje recepta

(b) Drugi dio ekrana za dodavanje recepta

(c) Prvi dio ekrana za detalje o receptu

(d) Drugi dio ekrana za detalje o receptu

Slika 5.6: Ekrani za pojedini recept

Posebne sekcije su sekcije za unos sastojaka, koraka pripreme, videozapisa i fotografija jer su sve te sekcije modelirane BLoC obrascom jer ne možemo unaprijed znati koliko takvih objekata korisnik želi dodati pa je bilo potrebno dodati opciju za dinamičko dodavanje polja za unos (slika 5.6 (a) i (b)).

5.2.5. Ekran za upravljanje korisnicima

Posljedni ekran je ekran za upravljanje korisnicima je dostupan samo administratorima. Na ovoj stranici se nalazi popis svih korisnika koji su također podijeljeni u stranice gdje svaka stranica sadrži 10 korisnika i straničenje funkcioniра na potpuno jednak način kao i straničenje za recepte. Jedan korisnik je opisan svojim korisničkom imenom profilnom slikom i gumbom za suspendiranje to jest odsuspendiranje korisnika. Nema opcije za prisanje korisnika jer bi to značilo da bi pojedini recepti ostali bez autora ili bi bili obrisani iz baze što nije poželjna funkcionalnost. Pa je korisniku samo onemogućena prijava u sustav (slika 5.4 (c)).

6. Zaključak

Mobilne aplikacije su postale važan dio modernog društva. Iz tog razloga ih je potrebno napraviti što je bolje moguće. Za takve svrhe postoje razni obrasci, načela programiranja, stilovi i tehnike: neke od korištenijih su MVC obrasac, podjela na poslužiteljsku stranu, korisničku stranu i bazu podataka, native pristup razvoju aplikacija, oblak, GraphQL stil i slično.

Najbitnija stvar je da krajnji korisnik bude zadovoljan, a to se postiže uz ergonomski dizajn aplikacije, brzinu i sigurnost. Zato postoje alati kao: flutter uz pomoć kojeg se lako napravi aplikacija koja je laka za koristit, SpringBoot s kojim se lako implementira sigurnost i podjela uloga, GraphQL koji nudi brzinu i optimalnost aplikacije te oblak uz pomoć kojeg sve postaje javno dostupno.

Mobilna aplikacija za upravljanje receptima

Sažetak

Mobilne aplikacije su danas vrlo popularna stvar i u posljednje vrijeme popularnost sve više raste. Uzrok tomu je brzina i lakoća korištenja u usporedbi za klasičnim web aplikacijama. Iako su mobilne i web aplikacije u puno stvari slične kao na primjer MVC obrazac ili razdvajanje na poslužiteljsku i korisničku stranu postoji i puno razlika a neke su nemogućnost korištenja SOAP protokola u mobilnim aplikacijama, odsutnost sjednica, različiti jezici i radni okviri na korisničkoj strani i slično.

Kao zamjena za SOAP protokol se vrlo često koristi REST arhitekturni stil, ali postoje i različiti stilovi kao na primjer GraphQL. GraphQL predstavlja stil u kojem se korisničkoj strani dopušta potupuna sloboda u biranju podataka, njihovom formatu i obliku. Njegova karakteristika je što rješava problem dohvata previše podataka, problem dohvata premalo podataka, problem velikog broja završnih točki na poslužitelju... U spring boot radnom okviru se GraphQL vrlo lako implementira uz pomoć ovisnosti prema graphql-spring-boot-starter i graphql-java-tools. Naime potrebno je samo definirati shemu i napraviti takozvane resolvere.

Sada zbog GraphQL-a korisnička strana vrlo lako dohvati podatke koji su joj potrebni. Na primjer u flutter radnom okviru uz pomoć paketa flutter-graphql koji sadrži *widget-e* koji su zaduženi za komunikaciju sa graphql poslužiteljem. Također GraphQL dopušta komunikaciju uz pomoć tokena pa je tako uz na primjer *JWT token* vrlo lako obaviti autorizaciju i autentifikaciju te spremiti taj token u dijeljene tajne i na uz pomoć njega obaviti filtriranje ekrana koje trenutni korisnik smije vidjeti.

Ključne riječi: Mobilna aplikacija, baza podataka, poslužiteljska strana, korisnička strana, oblak, PostgreSQL, SpringBoot, GraphQL, Flutter, Native, AWS .

Mobile application for recipe management

Abstract

Mobile applications are a very popular thing nowadays and since the past few years the popularity has grown by quite a lot. That is because they are very easy to use and very fast in comparison with web applications. Web and mobile applications have a lot in common like: MVC pattern or splitting the application on the database, backend, frontend, but they also have some differences like the absence of SOAP protocol in mobile applications, no sessions in mobile applications, different coding languages and frameworks on the frontend and so on.

A commonly used substitute for the SOAP protocol is REST architectural style, but there are some other options like GraphQL. GraphQL is a style that allows the frontend to define which data to fetch and in which shape and form. Its characteristic is that it solves the problem of overfetching, underfetching, then the problem of lot of endpoints... In the SpringBoot framework the GraphQL functionality is very easily implemented with the graphql-spring-boot-starter and graphql-java-tools dependency. The only thing that is necessary is a schema and some resolvers.

Now because of GraphQL the frontend can fetch the necessary data without any problems. For example in the flutter framework with the help of flutter-graphql package that contains widgets that are used for communication with a GraphQL server. Also GraphQL allows communications with tokens so it is possible to use JWT tokens for authorization and authentication, which will be saved in Shared Preferences and used for filtering the screens that the current user can see.

Keywords: Mobile application, database, backend, frontend, cloud, PostgreSQL, SpringBoot, GraphQL, Flutter, Native, AWS.