

AKADEMIA FINANSÓW I BIZNESU VISTULA

Wydział Informatyki, Grafiki i Architektury

Kierunek studiów: Informatyka

Jakub Kuźnicki

Numer albumu: 61961

***PROJEKT I IMPLEMENTACJA
SYSTEMU ZARZĄDZANIA
PRZEPISAMI KULINARNYMI***

Praca inżynierska

napisana pod kierunkiem

Promotor – dr hab. Paweł Gburzyński

Warszawa, 2026

Spis treści

1	Wprowadzenie	3
1.1	Cel i zakres pracy	3
1.2	Opis problemu i motywacja	3
1.3	Struktura pracy	4
2	Analiza wymagań	4
2.1	Wymagania funkcjonalne i нефункционалне	4
2.2	Analiza technologii i narzędzi	5
2.3	Projekt architektury systemu	5
3	Implementacja backendu z wykorzystaniem Spring Boot	7
3.1	Spring Framework vs Spring Boot	7
3.1.1	Uzasadnienie wyboru i kluczowe zależności	7
3.2	Struktura aplikacji Spring Boot	7
3.2.1	Konfiguracja głównej klasy aplikacji (@SpringBootApplication)	8
3.2.2	Struktura katalogów projektu	9
3.3	Architektura warstwowa aplikacji	10
3.3.1	Warstwa kontrolerów (@RestController)	11
3.3.2	Warstwa serwisów (@Service)	13
3.3.3	Warstwa dostępu do danych (@Repository)	14
3.3.4	Pełna lista endpointów	14
3.4	Projektowanie RESTful API	16
3.4.1	Obsługa parametrów oraz serializacja i deserializacja JSON	17
3.5	Modelowanie danych z JPA/Hibernate	17
3.6	Integracja z bazą danych PostgreSQL	19
3.7	Implementacja funkcjonalności biznesowych	20
3.7.1	Zarządzanie użytkownikami i autoryzacja	20
3.7.2	Zarządzanie przepisami	23
3.8	Obsługa błędów	24
3.9	Podsumowanie rozdziału	25
4	Implementacja aplikacji frontendowej	26
4.1	Przegląd architektury	26
4.1.1	Struktura projektu	27
4.1.2	Wzorce projektowe	28
4.2	Serwisy HTTP	29
4.2.1	Auth.service.ts	29
4.2.2	Recipe.service.ts	31
4.2.3	Filter.service.ts	32
4.2.4	Profile.service.ts	33
4.3	Kluczowe komponenty	35
4.3.1	Nawigacja i układ	36
4.3.2	Autoryzacja	37
4.3.3	Przepisy	39
4.3.4	System filtrowania	45
4.3.5	Profil użytkownika	46
4.3.6	Panel administratora	50

4.4	Systemy wsparcia	51
4.4.1	System powiadomień	51
4.4.2	System zarządzania stanem interfejsu	53
4.4.3	Zarządzanie językiem aplikacji	54
4.5	Podsumowanie	55
5	Weryfikacja, testowanie i konteneryzacja systemu	56
5.1	Środowisko testowe i wykorzystane narzędzia	56
5.2	Konteneryzacja środowiska	56
5.2.1	Budowa obrazów kontenerów	56
5.2.2	Orkiestracja usług	57
5.3	Testy warstwy serwerowej	58
5.3.1	Testy jednostkowe	58
5.3.2	Testy integracyjne	59
5.4	Testy warstwy klienckiej	61
5.4.1	Weryfikacja logiki i formularzy	61
5.4.2	Testy warstwy prezentacji	62
5.5	Testy wydajnościowe	63
5.5.1	Scenariusz testowy	64
5.5.2	Sposób uruchomienia	64
6	Podsumowanie i wnioski	65
6.1	Kierunki dalszego rozwoju aplikacji	66
6.2	Przepis na carbonarę	66
	Źródła	68

1 Wprowadzenie

W pracy podjęto próbę stworzenia systemu, który umożliwi w sposób intuicyjny użytkownikom przechowywanie, przeglądanie i zarządzanie przepisami. Praca została podzielona na sześć rozdziałów, z których każdy opisuje kolejny etap realizacji projektu. Zaczynając od analizy wymagań, przez implementację, aż po testowanie i wnioski końcowe.

1.1 Cel i zakres pracy

Celem niniejszej pracy dyplomowej jest zaprojektowanie i implementacja systemu informatycznego służącego do zarządzania przepisami kulinarnymi. System ma umożliwiać użytkownikom tworzenie, edytowanie, przeglądanie oraz usuwanie przepisów za pośrednictwem intuicyjnego interfejsu użytkownika.

Aplikacja frontendowa zostanie zbudowana przy użyciu frameworka Angular. Natomiast warstwa backendowa zostanie napisana przy użyciu technologii Spring Boot. Dane będą przechowywane w relacyjnej bazie danych PostgreSQL. Komunikacja między warstwami będzie zrealizowana za pomocą REST API (Representational State Transfer Application Programming Interface) – to styl architektury służący do projektowania interfejsów programistycznych (API), które komunikują się za pomocą standardowego protokołu HTTP.

Testy projektu zostaną podzielone na trzy grupy – testy jednostkowe i integracyjne (przy użyciu JUnit), testy frontendowe (przy użyciu Jasmine) oraz testy wydajnościowe (przy użyciu K6).

Praca obejmuje analizę wymagań, wybór odpowiednich technologii i narzędzi, projekt architektury systemu, implementację frontendu oraz backendu, integrację komponentów oraz testowanie funkcjonalne, wydajnościowe i konteneryzacje. Efektem końcowym pracy jest dostarczenie działającego prototypu systemu oraz analiza możliwości jego dalszego rozwoju.

1.2 Opis problemu i motywacja

Aktualnie, coraz więcej osób korzysta z aplikacji internetowych do organizowania codziennych aktywności, w tym również przygotowywania posiłków. Choć istnieje wiele platform i blogów przeznaczonych kulinariom, często zawierają one zbędny tekst taki jak: historie powstania dania, czy osobiste historie autorów przepisów. Niewiele aplikacji zezwala użytkownikom na dodawanie własnych przepisów, zazwyczaj ich publikacją zajmuje się administrator lub redakcja strony.

Podjęcie to prowadzi do marnowania czasu użytkownika oraz ogranicza możliwości personalizacji. Dodatkowo, wiele aplikacji nie posiada responsywnego UI (User Interface) – interfejsu użytkownika, lub posiadają zbyt dużą ilość animacji, które często stanowią przeszkodę dla osób z niepełnosprawnością. Często spotyka się również brak funkcjonalności filtrowania czy kategoryzacji przepisów, co nie wpływa pozytywnie na komfort użytkownika.

Motywacją do realizacji tego projektu jest chęć stworzenia elastycznego i nowoczesnego narzędzia, które umożliwi użytkownikom szybkie i bezpieczne zarządzanie przepisami kulinarnymi, bez zbędnego tekstu. Projekt stanowi również okazję do nauki technologii internetowych, takich jak Angular, Spring Boot i bibliotek z tym

związanych oraz PostgreSQL. Ponadto, pozwala rozwinąć się w dziedzinie testowania (jednostkowego, integracyjnego i wydajnościowego) i konteneryzacji przy użyciu Dockera i Docker Compose’a.

1.3 Struktura pracy

Niniejsza praca składa się z sześciu rozdziałów, z których każdy odpowiada kolejnemu etapowi realizacji projektu.

W rozdziale pierwszym przedstawiono cel pracy, zarysowano główny problem oraz wskazano motywację stojącą za wyborem tematu. Zaprezentowano również strukturę całej pracy.

Drugi rozdział poświęcony jest analizie wymagań funkcjonalnych i niefunkcjonalnych, przeglądowi dostępnych technologii oraz projektowi architektury systemu.

W rozdziale trzecim omówiona została implementacja backendu z wykorzystaniem Spring Boot. Przedstawiono strukturę aplikacji, sposób tworzenia RESTful API oraz integrację z relacyjną bazą danych PostgreSQL.

Rozdział czwarty opisuje implementację aplikacji frontendowej zbudowanej w frameworku Angular, w tym strukturę projektu, tworzenie komponentów i integrację z warstwą backendową.

Piąty rozdział dotyczy procesu testowania, optymalizacji oraz konteneryzacji systemu. Opisano testy jednostkowe, wydajnościowe, a także działania mające na celu poprawę wydajności aplikacji. Zaprezentowano również sposób konteneryzacji aplikacji przy użyciu Dockera oraz Docker Compose’a.

Ostatni, szósty rozdział zawiera podsumowanie zrealizowanych prac, wnioski wynikające z projektu oraz propozycje rozbudowy aplikacji.

2 Analiza wymagań

2.1 Wymagania funkcjonalne i niefunkcjonalne

Wymagania funkcjonalne określają, co system ma robić, to znaczy jego zachowanie z punktu widzenia użytkownika. W kontekście pracy, system zarządzania przepisami kulinarnymi powinien umożliwiać:

- Dodawanie nowych przepisów kulinarnych.
- Edytowanie istniejących przepisów.
- Usuwanie przepisów.
- Przeglądanie listy przepisów.
- Filtrowanie przepisów według kategorii, czasu przygotowania, ilości porcji lub nazwy.
- Rejestrację i logowanie użytkownika.
- Obsługę konta użytkownika (np. zmiana hasła lub nazwy użytkownika).

Wymagania niefunkcjonalne odnoszą się do jakości działania systemu. Dla projektu będą to:

- System powinien działać responsywnie i być dostępny zarówno na komputerach, jak i urządzeniach mobilnych.
- Interfejs użytkownika powinien być przejrzysty i intuicyjny.
- System powinien zapewniać bezpieczeństwo przesyłania danych (np. hasła powinny być hashowane).
- Backend powinien być skalowalny i odporny na błędy.
- Czas odpowiedzi interfejsu nie powinien przekraczać 500 ms w przypadku prostych zapytań.

2.2 Analiza technologii i narzędzi

Do realizacji projektu wybrano nowoczesne i szeroko stosowane technologie frontendowe i backendowe oraz biblioteki i API ułatwiające proces budowy oraz utrzymania systemu:

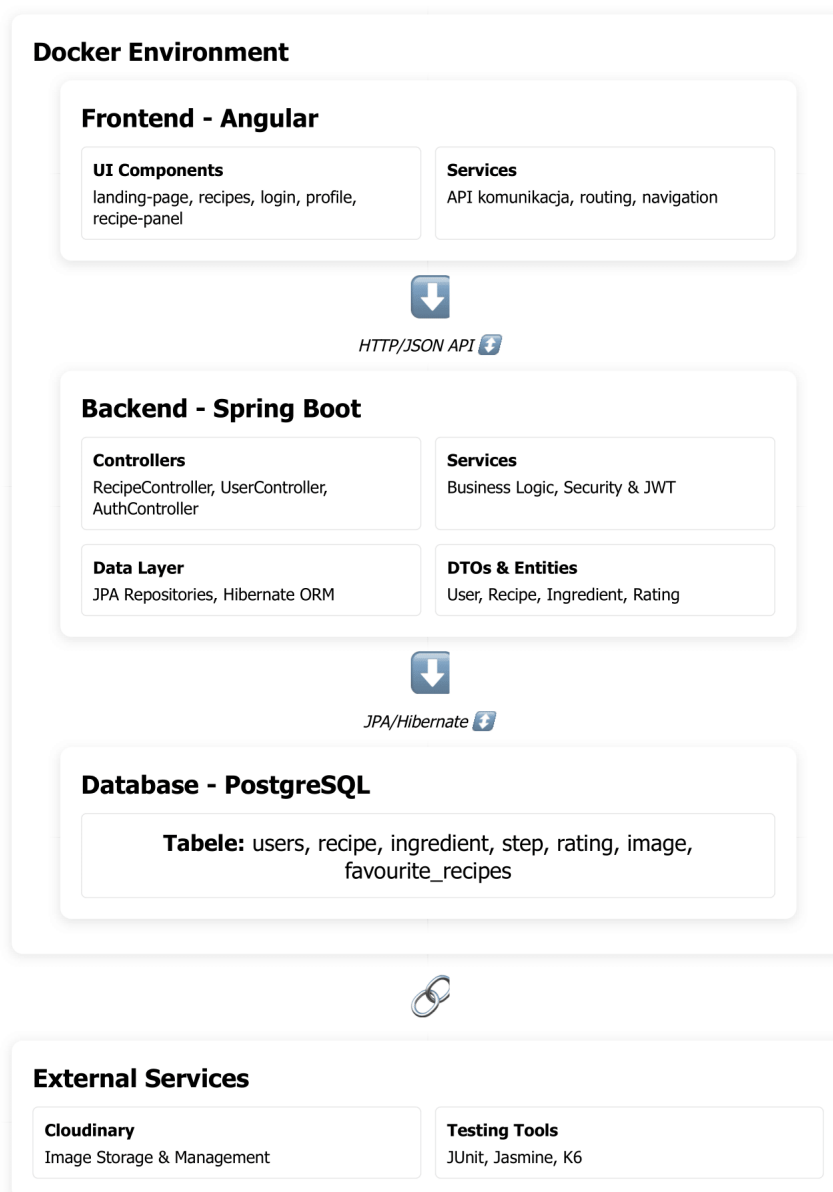
- **Angular** – framework frontendowy umożliwiający tworzenie SPA (Single Page Application) – dynamicznej jednostronicowej aplikacji. Pozwala to na czytelny podział na komponenty, serwisy oraz routing.
- **Spring Boot** – framework Java do budowy aplikacji backendowych w architekturze REST.
- **PostgreSQL** – relacyjna baza danych, w której przechowywane będą informacje o przepisach, dane użytkowników oraz linki do zdjęć.
- **REST API** – standard komunikacji pomiędzy frontendem a backendem, oparty o protokół HTTP.
- **Cloudinary** – zewnętrzne API, służące do przechowywania zdjęć.
- **Docker** – technologia konteneryzacji umożliwiająca uruchamianie aplikacji w spójnym środowisku niezależnie od systemu operacyjnego.
- **Docker Compose** – narzędzie do uruchamiania wielokontenerowych aplikacji Docker.
- **JUnit** – framework do pisania testów jednostkowych i integracyjnych w Javie.
- **Jasmine** – narzędzie do testowania aplikacji Angular.
- **K6** – narzędzie do testów wydajnościowych aplikacji webowych.

2.3 Projekt architektury systemu

System będzie składał się z trzech głównych warstw zamkniętych w kontenerach Docker'a uruchamianych za pomocą Docker Compose oraz jednej warstwy zewnętrznej:

- **Warstwa frontendowa (Angular)** – odpowiedzialna za interakcję z użytkownikiem, pobieranie i prezentację danych z backendu. Dane prezentowane będą w formie dynamicznych widoków (np. lista przepisów, formularz edycji).

- **Warstwa backendowa (Spring Boot)** – obsługująca logikę biznesową, komunikując się z bazą danych i udostępniającą informacje przez REST API.
- **Warstwa danych (PostgreSQL)** – baza danych przechowująca informacje o przepisach, kategoriach i użytkownikach.
- **Warstwa zewnętrzna (Clouinary oraz testy)** – obsługuje zagadnienia związane z przechowywaniem zdjęć oraz jest wsparciem funkcjonalnej części aplikacji. Działa poza bezpośrednim środowiskiem aplikacji.



Rysunek 1: Diagram przedstawiający główne warstwy systemu

Komunikacja między frontendem a backendem będzie odbywała się poprzez zapytania http (GET, POST, PUT, DELETE), natomiast dane będą przesyłane w formacie JSON.

3 Implementacja backendu z wykorzystaniem Spring Boot

3.1 Spring Framework vs Spring Boot

Spring Framework to rozbudowany ekosystem bibliotek i narzędzi Javy, który udostępnia Dependency Injection (wzorzec projektowy, w którym obiekty otrzymują zależności z zewnętrznego źródła), wspiera AOP (Aspect-Oriented Programming) oraz umożliwia integrację z bazami danych oraz różnorodnymi systemami. Jednak ma on swoje wady. Konfiguracja aplikacji w Springu, potrafi być czasochłonna i złożona. Zazwyczaj wymaga znajomości wielu komponentów i sposobu ich łączenia. Do uproszczenia tego procesu, stworzono Spring Boot, który automatyzuje konfigurację poprzez rozpoznawanie używanej biblioteki (na przykład JPA) i dobiera do nich odpowiednie ustawienia domyślne. Framework oferuje także skonfigurowane startery zależności pozwalające na uniknięcie tak zwanego „Dependency Hell”. Dzięki wbudowanym serwerom (Tomcat, Jetty lub Undertow) uruchamianie aplikacji nie wymaga dodatkowej konfiguracji. Spring Boot posiada integracje z modułami pozwalającymi na monitorowanie aplikacji, przez co jest atrakcyjnym wyborem w tworzeniu nowoczesnych systemów. Poprzez wsparcie JUnit oraz Mockito, framework ułatwia tworzenie testów jednostkowych i integracyjnych. Dodatkowo, dzięki prostej integracji ze Spring Cloud, stanowi popularny wybór przy budowie mikrousług.

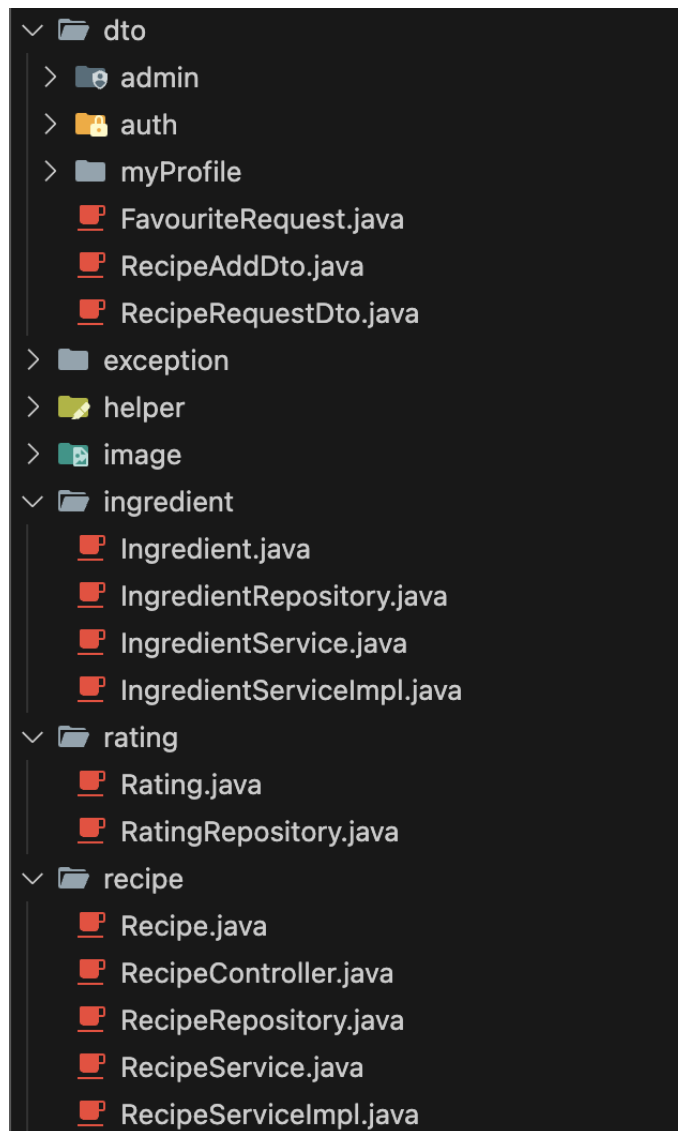
3.1.1 Uzasadnienie wyboru i kluczowe zależności

Do stworzenia backendu użyto Spring Boot, ponieważ pozwala na proste i szybkie stworzenie funkcjonalnej aplikacji, eliminując konieczność czasochłonnej konfiguracji. Dzięki starterom zależności i wbudowanemu serwerowi, można skupić się bezpośrednio na logice biznesowej. Dodatkowym atutem jest obszerna dokumentacja oraz szerokie wsparcie społeczności, to czyni ten framework atrakcyjnym wyborem do tego projektu.

Kluczowymi zaletami są zależności, które pomogły w tworzeniu projektu. Pierwsza z nich jest biblioteka „Lombok”. Eliminuje konieczność pisania zbędnego oraz powtarzalnego kodu. Przykładowo używając adnotacji `@Data`, powoduje automatyczne wygenerowanie getterów i setterów oraz metod jak `toString()`, `equals()`. Drugą zależnością jest `JsonWebToken`, pozwala na wysyłanie zaszyfrowanej informacji pomiędzy użytkownikiem a serwerem. Służy do uwierzytelniania użytkownika oraz zawiera w sobie informacje na jego temat (role, id, itp.). Trzecia i ostatnią kluczową zależnością jest `Cloudinary`, jest to usługa niezależna od Springa, jednak jest to usługa, którą można bezpośrednio zintegrować z projektem. Służy do zarządzania multimediami w chmurze. Umożliwia przesyłanie oraz przechowywanie obrazów, a także na ich automatyczna optymalizację. Daje to możliwość ominięcia tworzenia własnej infrastruktury do obsługi plików, co uprasza implementację i skalowalność aplikacji.

3.2 Struktura aplikacji Spring Boot

W projekcie zastosowano hybrydowe podejście organizacji pakietów. Upraszcza to czytelność, unika duplikacji konfiguracji oraz ułatwia utrzymanie wspólnych komponentów. To pozwala łączyć podejście warstwowe z podejściem funkcjonalnym.



Rysunek 2: Hybrydowa organizacja pakietów

Natomiast może prowadzić do duplikacji kodu. Przykładem duplikacji jest logika autoryzacji, która pojawia się w kilku miejscach.

```
1 boolean isOwner = recipeFound.getUser().getId() == currentUser.getId();
2 boolean isAdmin = jwtService.hasRole(token.substring(7), "ADMIN");
3
4 if (!isOwner && !isAdmin) {
5     throw new UnauthorizedException("You can edit only your own recipes!");
6 }
```

3.2.1 Konfiguracja głównej klasy aplikacji (@SpringBootApplication)

Konfiguracja używa meta-adnotacji `@SpringBootApplication`, która łączy w sobie trzy kluczowe adnotacje:

1. `@Configuration` – oznacza klasę jako źródło definicji bean'ów.

2. `@EnableAutoConfiguration` – uruchamia automatyczną konfigurację na podstawie zależności w classpath (zmienna mówiąca, gdzie spring musi szukać klas).
3. `@ComponentScan` – skanuje pakiety w poszukiwaniu komponentów Spring (klasy z `@Component`, `@Service`, `@Controller`, `@Repository`).

```
1 @SpringBootApplication
2 public class CibariaApplication {
3     public static void main(String[] args) {
4         SpringApplication.run(CibariaApplication.class, args);
5     }
6 }
```

Dodatkowo w aplikacji zawarta jest konfiguracja CORS (Cross-Origin Resource Sharing). Ta konfiguracja umożliwi komunikację między frontendem Angular działającym na porcie 4200, a backendem. Rozwiązuje to problemy z polityką same-origin przeglądarek. Wyjaśnienie konfiguracji:

1. `addMapping("/**")` – dotyczy wszystkich endpointów.
2. `allowedOrigins("http://localhost:4200")` – zezwala na żądania frontendu.
3. `allowedMethods("*")` – wszystkie metody http (GET, POST, PUT, DELETE).
4. `allowedHeaders("Authorization", "Content-Type", "*")` – dozwolone nagłówki, szczególnie „Authorization”, bardzo ważny dla JWT.

Alternatywą do tego podejścia jest użycie `@CrossOrigin` na poziomie kontrolerów, jednak produkuje to zbędne linijki kodu. Globalne podejście do konfiguracji jest bardziej praktyczne.

```
1 @Bean
2 public WebMvcConfigurer webMvcConfigurer() {
3     return new WebMvcConfigurer() {
4         @Override
5         public void addCorsMappings(CorsRegistry registry) {
6             registry.addMapping("/**")
7                 .allowedOrigins("http://localhost:4200")
8                 .allowedMethods("*")
9                 .allowedHeaders("Authorization", "Content-Type", "*");
10        }
11    };
12 }
```

3.2.2 Struktura katalogów projektu

Projekt został zainicjalizowany przy pomocy Spring Initializr, co oznacza, że otrzymał standardową strukturę folderów.

- `src/main/java` – kod źródłowy aplikacji
- `src/main/resources` – plik konfiguracyjny `application.yml`
- `src/test/java` – testy jednostkowe i integracyjne

Głównymi folderami są:

- `dto` – (Data Transfer Object) obiekty transferowe, używane do wymiany danych między warstwami.
- `admin`, `auth`, `image`, `ingredient`, `rating`, `recipe`, `step`, `user` – moduły domenowe zawierające:
 - Model (`Step.java`, `Recipe.java`).
 - Repozytorium.
 - Ewentualnie serwis (`Service` + `ServiceImpl`).
 - Ewentualnie kontroler (`Controller`).
- `exception` – obsługa wyjątków oraz błędów.
- `helper` – klasy pomocnicze.
- `cloudinary` – integracja z usługami Cloudinary.

3.3 Architektura warstwowa aplikacji

Aplikacja została zbudowana w oparciu o klasyczne podejście, jaką jest architektura warstwowa. Zapewnia wyraźne rozdzielenie funkcji między poszczególnymi komponentami systemu. To podejście prowadzi do łatwiejszego utrzymania kodu i upraszcza przyszły rozwój aplikacji. Spring Boot automatycznie zarządza cyklem życia komponentów i wstrzykiwaniem zależności, poprzez system adnotacji (np. `@RestController`).

Schemat architektury:

- Warstwa kontrolerów – (`@RestController`) Żądania/Odpowiedzi HTTP.
- Warstwa serwisów – (`@Service`) Logika biznesowa.
- Warstwa dostępu do danych – (`@Repository`) Komunikacja z bazą danych.

Każda warstwa komunikuje się tylko z warstwą bezpośrednio pod nią co ułatwia testowanie oraz wprowadzanie zmian, gdyż zmiana w jednej warstwie, nie wpłynie na pozostałe. Architektura jednak ma wady takie jak, trudność w rozbudowie (potrzeba wielu klas DTO), utrudnia elastyczność (momentami trudno ominąć koncepcje tej architektury). Zazwyczaj przy większych projektach może prowadzić do zbytniego rozrostu kodu i przechodzenia przez wszystkie warstwy nawet dla najprostszych operacji.

Przykład przepływu danych:

Klient wysyła żądanie http.

↓

Kontroler odbiera i waliduje parametry.

```

1 @GetMapping("/{id}")
2 public Recipe getById(@PathVariable int id) {
3     return recipeService.getById(id);
4 }

```



Serwis wykonuje logikę biznesową.

```

1 @Override
2 public Recipe getById(int id) {
3     return recipeRepository.findById(id).orElseThrow(
4         () -> new RecipeNotFoundException(String
5             .format("Recipe with id: %s does not exist in the database", id));
6 }

```



Repozytorium kontaktuje się z bazą danych.

```

1 public interface RecipeRepository extends JpaRepository<Recipe, Integer> {
2     @Query("SELECT u FROM Recipe u WHERE u.recipeName ILIKE %:query%")
3     List<Recipe> findByNameQuery(@Param("query") String query);
4
5     List<Recipe> findByUser(UserEntity user);
6 }

```



Wynik jest zwracany z powrotem do klienta i przechodzi przez wszystkie warstwy.

3.3.1 Warstwa kontrolerów (@RestController)

System został podzielony na pięć głównych modułów, z których każdy posiada minimum jeden kontroler:

1. Recipe,
2. User,
3. Image,
4. Auth (posiada dwa kontrolery, LoginController oraz RegisterController),
5. Admin.

Każdy z kontrolerów zawiera adnotację `@RequestMapping("**")`, która sprawia, że wszystkie endpointy znajdują się pod konkretną ścieżką `/api/*`. Większość kontrolerów obsługuje operacje CRUD (tworzenie, odczyt, aktualizacje, usuwanie), jednak Auth oraz Image są wyjątkiem, w nich znajduje się tylko operacja tworzenia.

Przykładowe endpointy:

1. Recipe - Wyświetlanie przepisów (GET `/recipes`)

```

1  @GetMapping
2  public RecipeRequestDto getRecipesByPage(
3      @RequestParam(defaultValue = "1", required = false) @Min(1) int page,
4      @RequestParam(defaultValue = "10", required = false) @Min(1) int size,
5      @RequestParam(required = false) List<String> category,
6      @RequestParam(required = false) Integer difficulty,
7      @RequestParam(required = false) String servings,
8      @RequestParam(required = false) String prepareTime,
9      @RequestParam(defaultValue = "true") Boolean isPublic,
10     @RequestParam(required = false) String language,
11     @RequestParam(required = false) List<String> ingredients
12 ){
13     return recipeService.getRecipeByPage(page,size,category,difficulty,
14     servings, prepareTime, isPublic, language, ingredients);
15 }

```

- Endpoint pozwala na wyświetlenie przepisów dla wszystkich użytkowników, nawet niezalogowanych.
- Posiada parametry odnoszące się do filtrowania przepisów (np. po składnikach).
- Automatycznie paginuje ilość wyświetlanych przepisów

2. Recipe - Dodawanie przepisu (POST /recipes)

```

1  @PostMapping(consumes = MediaType.MULTIPART_FORM_DATA_VALUE)
2  public Recipe save(@RequestParam("recipe") String json,
3      @RequestHeader("Authorization") String token,
4      @RequestParam(value = "images", required = false)
5      Optional<List<MultipartFile>> images) throws IOException {
6      ObjectMapper objectMapper = new ObjectMapper();
7      RecipeAddDto recipe = objectMapper.readValue(json,RecipeAddDto.class);
8
9      if(images.isPresent() && images.get() != null && !images.get().isEmpty()){
10         return recipeService.saveRecipeWithPhotos(recipe, images.get(), token);
11     } else {
12         return recipeService.saveRecipeWithoutPhoto(recipe, token);
13     }
14 }

```

- Endpoint pozwala na dodanie przepisu
- Wymaga autoryzacji
- Przyjmuje dane w formie JSON oraz opcjonalne zdjęcia w formacie multipart/form-data
- Po poprawnym zapisaniu, zwraca obiekt przepisu

3. Wyświetlenie danych użytkownika (GET /users/aboutme)

- Wymaga autoryzacji, na podstawie której wyświetla wszystkie dane użytkownika

3.3.2 Warstwa serwisów (@Service)

Warstwa serwisowa, pełni kluczową rolę w aplikacji, zawiera ona w sobie całą logikę biznesową systemu. Oznacza to, że wszystkie procesy, które otrzymuje od warstwy kontrolerów przetwarzane są tutaj np. weryfikacja danych, lub obsługa modyfikacji tła na profilu użytkownika. Klasy tej warstwy są oznaczane adnotacją @Service, która przekazuje Spring Boot'owi informację odnośnie automatycznego zarządzania nimi, umożliwia to wstrzykiwanie zależności. Framework automatycznie zarządza transakcjami dzięki adnotacji @Transactional, minimalizuje to utratę danych dzięki rollbackom przy niepowodzeniu zapisu oraz ułatwia spójność operacji (zapis kilku obiektów w ramach jednej transakcji).

Przykładem serwisu jest RecipeServiceImpl oraz metoda rating. Sprawdza, czy wszystkie warunki są spełnione, pobiera odpowiedniego użytkownika i przepis, a następnie aktualizuje lub tworzy nową ocenę w bazie.

```
1  @Transactional
2  @Override
3  public Recipe rating(int id, String token, int rating){
4      if (rating < 1 || rating > 5){
5          throw new IllegalArgumentException
6              ("Rating has to be from 1 to 5");
7      }
8      int userId = jwtService.extractId(token.substring(7));
9      UserEntity user = userRepository.findById(userId).orElseThrow(
10         () -> new UserNotFoundException(String
11             .format("User with id: %s does not exist in the database",
12                 userId)));
13     Recipe recipe = recipeRepository.findById(id).orElseThrow(
14         () -> new RecipeNotFoundException(String
15             .format("Recipe with id: %s does not exist in the database",
16                 id)));
17
18     Optional<Rating> existingRating = ratingRepository
19         .findByRecipeIdAndUserId(id, userId);
20
21     if (existingRating.isPresent()) {
22         Rating rating1 = existingRating.get();
23         rating1.setValue(rating);
24         ratingRepository.save(rating1);
25     } else {
26         Rating newRating = new Rating();
27         newRating.setRecipe(recipe);
28         newRating.setValue(rating);
29         newRating.setUser(user);
30         ratingRepository.save(newRating);
31     }
32     return recipe;
33 }
```

3.3.3 Warstwa dostępu do danych (@Repository)

Warstwa dostępu do danych odpowiada za komunikację aplikacji z bazą danych. W tym miejscu realizowane są operacje związane z przechowywaniem informacji. Są to operacje zapisu, odczytu, aktualizacji oraz usuwania rekordów. Klasy tej warstwy są oznaczone adnotacją `@Repository`. Informuje to Spring Boot o tym, że dana klasa pełni rolę komponentu dostępu do danych.

W Spring Boot zazwyczaj korzysta się z Spring Data JPA, co upraszcza zapytania, ponieważ są one generowane automatycznie na podstawie nazw metod (np. `findByEmail`, `findByUser`). Dodatkowo zezwala na bardziej złożone operacje, umożliwiając pisanie własnych zapytań SQL.

Przykładowo `UserRepository`, które rozszerza `JpaRepository<UserEntity, Integer>` i pozwala wyszukiwać użytkownika po nazwie lub emailu.

```
1 public interface UserRepository extends
2     JpaRepository<UserEntity, Integer> {
3     Optional<UserEntity> findByUsername(String username);
4     Optional<UserEntity> findByEmail(String email);
5 }
6
7 public interface RecipeRepository extends
8     JpaRepository<Recipe, Integer> {
9     @Query("SELECT u FROM Recipe u WHERE u.recipeName ILIKE %:query%")
10    List<Recipe> findByRecipeNameQuery(@Param("query") String query);
11
12    List<Recipe> findByUser(UserEntity user);
13 }
```

3.3.4 Pełna lista endpointów

Moduł autentykacji:

- `POST /api/authenticate` – Endpoint służący do logowania użytkownika. Przyjmuje dane uwierzytelniające (hasło, email), weryfikuje je i zwraca token JWT umożliwiający dostęp do chronionych zasobów.
- `POST /api/register` – Endpoint odpowiedzialny za rejestrację nowych użytkowników. Waliduje dane wejściowe, hashuje hasło algorytmem BCrypt i zapisuje użytkownika do bazy danych.

Moduł przepisów:

- `GET /api/recipes` – Pobiera listę z możliwością filtrowania oraz wspiera paginację.
- `GET /api/recipes/{id}` – Zwraca szczegóły wybranego przepisu, wraz z informacjami o krokach przygotowania, składnikach oraz ocenach.
- `POST /api/recipes` – Umożliwia dodanie nowego przepisu. Wymaga autoryzacji. Obsługuje przesyłanie obrazów w formacie `multipart/form-data`.

- PUT /api/recipes/{id} – Aktualizuje wybrany przepis. Dostępne tylko dla właściciela przepisu lub administratora.
- DELETE /api/recipes/{id} – Usuwa wybrany przepis. Wymaga uprawnień właściciela przepisu lub administratora.
- POST /api/recipes/{id} – Umożliwia ocenienie przepisu w skali od jeden do pięć. Wymaga zalogowanego użytkownika.
- GET /api/recipes/{id}/rating – Pobiera ocenę przepisu wystawioną przez zalogowanego użytkownika.
- GET /api/recipes/favourites/isFavourite – Sprawdza, czy użytkownika dodał przepis do ulubionych.
- GET /api/recipes/{id}/isOwner – Sprawdza, czy użytkownik jest właścicielem przepisu.
- POST /api/recipes/favourites/add – Dodaje przepis do ulubionych. Wymaga autoryzacji.
- POST /api/recipes/favourites/delete – Usuwa przepis z ulubionych. Wymaga autoryzacji.
- GET /api/recipes/search – Wyszukuje przepisy po wpisanej frazie.

Moduł użytkownika:

- GET /api/users – Wyświetla wszystkich użytkowników.
- GET /api/users/{id} – Wyświetla konkretnego użytkownika.
- PUT /api/users/{id} – Aktualizuje całego użytkownika. Wymaga autoryzacji.
- PUT /api/users/{id}/profile – Aktualizuje nazwę użytkownika oraz jego opis. Wymaga uprawnień właściciela profilu.
- PUT /api/users/{id}/email – Aktualizuje email użytkownika. Wymaga uprawnień właściciela oraz wpisania poprawnego hasła.
- PUT /api/users/{id}/password – Aktualizuje hasło użytkownika. Wymaga uprawnień właściciela oraz wpisania starego oraz nowego hasła.
- GET /api/users/aboutme – Zwraca szczegółowe informacje o profilu zalogowanego użytkownika.
- GET /api/users/recipes – Pobiera listę przepisów utworzonych przez zalogowanego użytkownika.
- GET /api/users/favourites – Zwraca listę przepisów dodanych do ulubionych przez zalogowanego użytkownika.
- PUT /api/users/{id}/profile-picture – Aktualizuje zdjęcie profilowe zalogowanego użytkownika.

- `PUT /api/users/{id}/background-picture` – Aktualizuje zdjęcie w tle zalogowanego użytkownika.
- `DELETE /api/users/{id}` – Usuwa konto wraz z wszystkimi powiązanymi danymi.

Panel administratora:

- `GET /api/admin/users` – Zwraca listę wszystkich użytkowników.
- `DELETE /api/admin/users/{id}` – Usuwa użytkownika z systemu.
- `PUT /api/admin/users/{id}/role` – Zmienia rolę użytkownika (USER/ADMIN).
- `GET /api/admin/recipes` – Pobiera wszystkie przepisy (prywatne i publiczne).
- `DELETE /api/admin/recipes/{id}` – Usuwa dowolny przepis z systemu.
- `PUT /api/admin/recipes/{id}` – Aktualizuje wybrany przepis (prywatny lub publiczny).
- `GET /api/admin/stats` – Zwraca statystyki systemu (liczba użytkowników, przepisów).

Moduł obrazu:

- `POST /api/image/addPhoto` – Przesyła obraz do serwisu Cloudinary i zwraca URL do obrazu.
- `POST /api/image/deletePhoto` – Usuwa obraz z Cloudinary na podstawie `public_id`.

3.4 Projektowanie RESTful API

Podczas projektowania RESTful API, kluczowe jest stosowanie się do sprawdzonych konwencji. Spring Boot jako popularny framework udostępnia programistom wiele konwencji.

Statusy HTTP stanowią podstawowy mechanizm komunikacji o wyniku operacji. Konwencja definiuje semantyczne odpowiedzi w postaci kodów:

- **2xx** – operacje zakończone sukcesem (np. 200 – OK, 201 – Created).
- **4xx** – operacje zakończone niepowodzeniem po stronie klienta (np. 400 – Bad Request, 401 – Unauthorized, 404 – Not Found).
- **5xx** – operacje zakończone błędem po stronie serwera (np. 500 Internal Server error).

Nazewnictwo zasobów (Resource Naming) opiera się na hierarchicznym nazewnictwie zasobów. URI (Uniform Resource Identifier – standard internetowy umożliwiający łatwą identyfikację zasobów w sieci) powinny reprezentować rzeczowniki, a nie czasowniki. Stosuje się zagnieżdżoną strukturę odzwierciedlającą relacje między zasobami (np. `users/{userId}/background-picture`). Dla kolekcji zasobów powinno się stosować liczbę mnogą.

Format danych w nowoczesnych API standardowo wykorzystuje JSON jako główny format wymiany informacji. Struktura pliku JSON powinna być spójna i posiadać jednolite nazewnictwo (camelCase lub snake_case). Dodatkowo zaleca się stosować standardowy format dat oraz wartości liczbowych.

Walidacja błędów w Spring Boot opiera się na zależności „Validation”. Framework udostępnia wiele adnotacji służących do walidowania, które można umieszczać bezpośrednio na polach encji lub parametrach metod:

- `@Valid` – aktywuje proces walidacji.
- `@NotNull` – sprawdza, czy wartość nie jest null.
- `@NotEmpty` – weryfikuje, czy string lub kolekcja nie są puste.
- `@Size(min, max)` – kontroluje rozmiar kolekcji lub długość tekstu.
- `@Email` – waliduje poprawność emaila.

Błędy walidacji automatycznie są przekształcane na błędy HTTP z kodem 400, a Spring Boot dostarcza mechanizmy przechwytywania dzięki `@ControllerAdvice`.

3.4.1 Obsługa parametrów oraz serializacja i deserializacja JSON

Obsługa parametrów żądań w Spring Boot odbywa się poprzez dedykowane adnotacje, które automatycznie mapują dane z żądań http na parametry metod kontrolera. `@RequestParam` służy do przechwytywania parametrów zapytania przekazywanych w URL po znaku zapytania, tym samym umożliwia filtrowanie, sortowanie czy paginację wyniku (np. `/recipes/search?query=chicken`). `@PathVariable` pozwala na wyodrębnienie zmiennych ze ścieżki URL (np. `/recipes/{id}`). Jest to kluczowe do identyfikacji konkretnych zasobów w hierarchicznej strukturze REST. Obie adnotacje obsługują automatyczną konwersję typów oraz walidację, tym samym zwiększając bezpieczeństwo API.

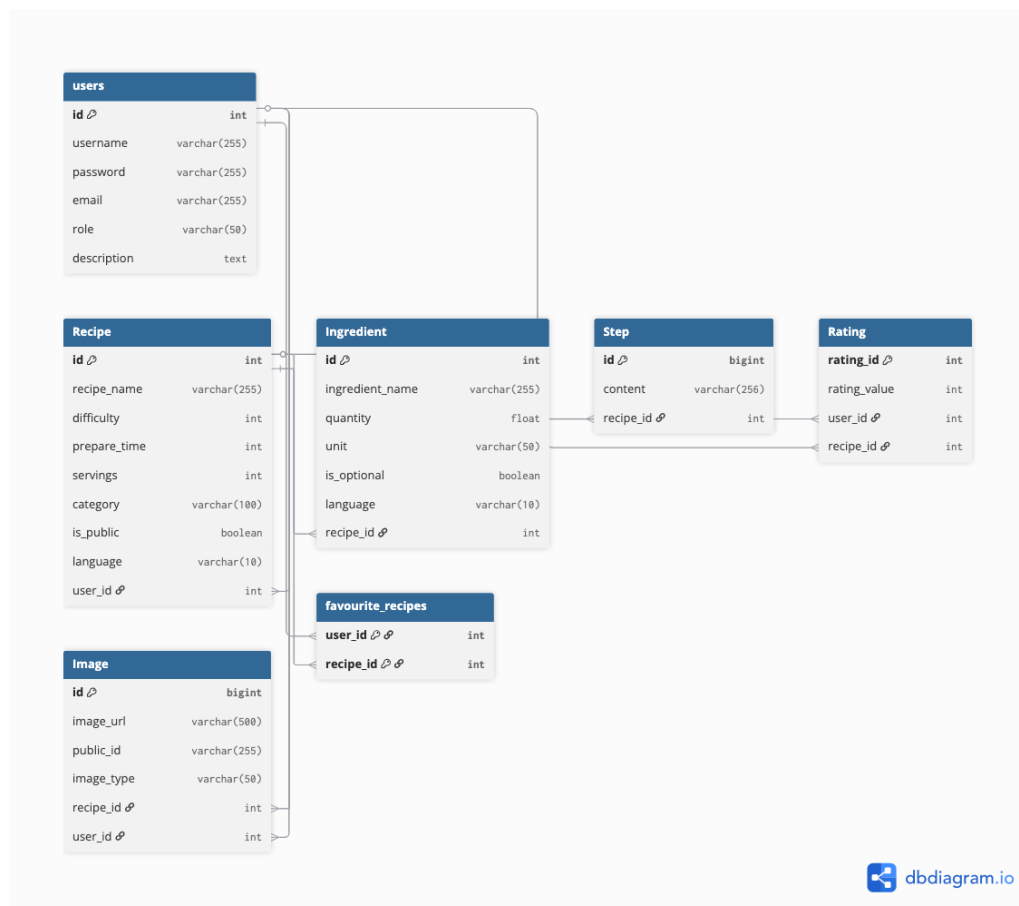
Serializacja i deserializacja JSON stanowią fundament mechanizmu wymiany danych w RESTful API. `@RequestBody` automatycznie deserializuje przychodzące dane w formie JSON na obiekty Javy. Wykorzystuje do tego bibliotekę Jackson jako domyślny konwerter. `@ResponseBody` wykonuje proces odwrotny, serializuje obiekty Javy na format JSON w odpowiedzi http.

3.5 Modelowanie danych z JPA/Hibernate

Spring Data JPA jest częścią większego Spring Data, która ułatwia implementacje repozytoriów bazujących na JPA (Java Persistent API). Jest to zestaw reguł i adnotacji, mówiący jak mapować klasy Java na tabele w bazie danych. Natomiast Hibernate to inaczej implementacja JPA. Jest to konkretny silnik odpowiedzialny za tłumaczenie operacji na obiektach na zapytania SQL i odwrotnie.

W Spring Boot encje definiuje się za pomocą adnotacji `@Entity`, która oznacza klasę jako trwałą encję JPA. Każda encja reprezentuje tabelę w bazie danych, a jej pola odpowiadają kolumnom tabeli. Adnotacja `@Table` także pozwala na określenie nazwy tabeli oraz dodatkowych właściwości, takich jak indeksy czy ograniczenia integralności.

Mapowanie relacji między tabelami w bazie danych realizowane jest poprzez specjalnie przeznaczone do tej czynności adnotacje. `@OneToMany` definiuje relacje jedno-do-wielu, gdzie jedna encja może być powiązana z kolekcją innych encji. `@ManyToOne` reprezentuje relacje wiele-do-jednego, zazwyczaj używa się jej do definiowania kluczy obcych. Przy nich zazwyczaj występuje adnotacja `@JoinColumn`, stosuje się ją najczęściej, aby wskazać klucz obcy. `@ManyToMany` jest relacją wiele-do-wielu, automatycznie generuje tabelę łączącą. Każda z tych adnotacji obsługuje parametry takie jak `cascade` czy `mappedBy`. W sytuacji, gdy relacja wymaga samodzielnego określenia nazwy oraz struktury tabeli pośredniczącej, stosuje się adnotację `@JoinTable`.



Rysunek 3: Relacje między encjami w bazie danych

Adnotacje JPA zapewniają precyzyjną kontrolę nad mapowaniem danych. Główną adnotacją jest `@Id` oznaczającą klucz główny encji, podczas gdy `@GeneratedValue` definiuje strategię generowania wartości klucza (np. AUTO, IDENTITY). `@Column` umożliwia szczegółową konfigurację kolumn, włączając nazwę, typ, ograniczenia długości.

3.6 Integracja z bazą danych PostgreSQL

Konfiguracja połączenia z bazą danych opiera się na mechanizmie automatycznej konfiguracji, która upraszcza proces łączenia z bazą danych w porównaniu do tradycyjnych rozwiązań w Javie.

Podstawowa konfiguracja odbywa się poprzez zdefiniowanie właściwości połączenia w plikach konfiguracyjnych aplikacji. Spring Boot automatycznie wykrywa zależności JDBC w classpath i na tej podstawie konfiguruje DataSource – główny komponent odpowiedzialny za połączenie.

Właściwości połączenia definiowane są w pliku `application.properties` lub `application.yml`, jednak w projekcie został użyty `application.yml`. Upraszcza to czytelność poprzez brak konieczności powtarzania prefixów, indentacje pokazujące relacje między właściwościami. Yaml także wspiera i automatycznie rozpoznaje typy danych. Pliki konfiguracyjne obejmują URL bazy danych, dane uwierzytelniające oraz sterownik JDBC. Framework automatycznie ładuje te właściwości przy uruchamianiu aplikacji, tworząc skonfigurowany DataSource. Jest on następnie wstrzykiwany do komponentów wymagających dostępu do bazy danych.

Zarządzanie schematem bazy danych realizowane jest przez parametr `ddl-auto`, który kontroluje sposób synchronizacji struktury bazy danych z definicjami encji JPA. Wartość `update` zapewnia automatyczne dostosowanie schematu do zmian w encjach bez utraty istniejących danych. Inne dostępne opcje to `create` (odtworzenie schematu), `create-drop` (usunięcie po wyłączeniu aplikacji), `validate` (tylko weryfikacja zgodności) oraz `none` (brak automatycznej modyfikacji).

Pliki konfiguracyjne zezwalają także na tworzenie profili, które zaś umożliwiają definiowanie różnych ustawień dla różnych środowisk. Spring Boot automatycznie ładuje odpowiedni profil na podstawie aktywnych profili aplikacji. Pozwala to na elastyczne zarządzanie konfiguracją bez potrzeby zmiany kodu źródłowego.

```
1  spring:
2    application:
3      name: cibaria
4    datasource:
5      url: ${DB_URL}
6      username: ${POSTGRES_USER}
7      password: ${POSTGRES_PASSWORD}
8      driver-class-name: org.postgresql.Driver
9      hikari:
10       maximum-pool-size: 25
11       minimum-idle: 5
12       connection-timeout: 30000
13       idle-timeout: 600000
14       max-lifetime: 1800000
15    jpa:
16      database-platform: org.hibernate.dialect.PostgreSQLDialect
17      hibernate:
18        ddl-auto: update
19    servlet:
20      multipart:
21        max-file-size: 5MB
22
```

```
23 server:
24     servlet:
25         context-path: /api
```

Konfiguracja aplikacji:

- Nazwa aplikacji - cibaria (wymyślona nazwa aplikacji).

Konfiguracja bazy danych:

- Połączenie – używa zmiennej środowiskowej dla URL, użytkownika oraz hasła.
- Hikari connection pool (użyte pod testy):
 - Maksymalnie 25 połączeń w puli.
 - Minimum 5 bezczynnych połączeń.
 - Timeout połączenia: 30 sekund.
 - Timeout bezczynności: 5 minut.
 - Maksymalny czas życia połączenia: 30 minut.

Konfiguracja JPA/Hibernate:

- Dialekt: PostgreSQL.
- `ddl-auto: update` – automatyczne aktualizowanie schematu bazy danych przy starcie.

Konfiguracja serwletu:

- Multipart – maksymalny rozmiar przesyłanego pliku to 5MB.
- `context-path: /api` – wszystkie endpointy będą dostępne pod prefixem `/api`.

3.7 Implementacja funkcjonalności biznesowych

W niniejszym podrozdziale przedstawiono implementację kluczowych funkcjonalności systemu, przedstawiając praktyczne zastosowanie opisanych wcześniej rozwiązań technicznych w kontekście logiki biznesowej aplikacji.

3.7.1 Zarządzanie użytkownikami i autoryzacja

Rejestracja użytkowników realizowana jest poprzez endpoint `POST /register`, który przyjmuje dane w formacie JSON. System automatycznie waliduje poprawność adresu email, wymagania dotyczące hasła czy nazwy użytkownika dzięki adnotacji `@Valid`. Po pomyślnej weryfikacji, token JWT zostaje przyznany, a hasło użytkownika jest hashowane przy użyciu algorytmu BCrypt, zapewniając wysoki poziom ochrony przed nieautoryzowanym dostępem.

```

1  @Override
2  public TokenResponseDto save(RegisterDto dto) {
3      Optional<UserEntity> isUser = userRepository
4          .findByEmail(dto.getEmail());
5
6      if(isUser.isPresent()){
7          throw new UserEmailAlreadyExistException(
8              "User with given email: " + dto.getEmail() +
9              " already exist in database");
10     }
11
12     UserEntity newUser = new UserEntity();
13     newUser.setEmail(dto.getEmail());
14     newUser.setPassword(passwordEncoder.encode(dto.getPassword()));
15     newUser.setUsername(dto.getUsername());
16     UserEntity userDb = userRepository.save(newUser);
17     TokenResponseDto token = new TokenResponseDto();
18     token.setToken(jwtService.generateToken(userDetailsService
19         .loadUserByUsername(userDb.getEmail())));
20
21     return token;
22 }

```

Logowanie użytkowników odbywa się przez endpoint POST /authenticate, który przyjmuje dane w formacie JSON, zawierające email oraz hasło. System wymaga od użytkowników podania obu parametrów, następnie automatycznie waliduje wprowadzone dane poprzez AuthenticationManager z Spring Security. Przy pomyślnym zalogowaniu, użytkownik otrzymuje ważny token JWT.

```

1  @PostMapping("/authenticate")
2  public TokenResponseDto authenticate(@Valid @RequestBody
3      LoginFormDto loginFormDto) {
4      try{
5          authenticationManager.authenticate(
6              new UsernamePasswordAuthenticationToken(loginFormDto.email(),
7              loginFormDto.password()));
8          TokenResponseDto token = new TokenResponseDto();
9          token.setToken(jwtService.generateToken(
10              userDetailsService.loadUserByUsername(loginFormDto.email())));
11          return token;
12      } catch (BadCredentialsException ex) {
13          throw new UserNotFoundException("Invalid credentials");
14      }
15 }

```

Mechanizm autoryzacji w systemie opiera się na tokenie JWT. Podejście to eliminuje konieczność przechowywania stanu sesji po stronie serwera, przekłada się to na lepszą skalowalność i prostsze zarządzanie autoryzacją.

W momencie logowania lub rejestracji użytkownik przekazuje swoje dane, które są weryfikowane. W przypadku poprawnej autoryzacji serwer generuje token JWT, składający się z trzech części:

- **Header** – informacje o algorytmie podpisu.
- **Payload** – dane użytkownika (oprócz standardowych claims, system wykorzystuje też własne pola):
 - **provider** – identyfikator źródła tokenu.
 - **id** – unikalny identyfikator użytkownika.
 - **roles** – rola użytkownika.
 - **Subject** – nazwa użytkownika.
 - **issuedAt** – data wydania tokenu.
 - **expiration** – data wygaśnięcia (siedemdziesiąt dwie godziny).
- **Signature** – podpis cyfrowy utworzony na podstawie tajnego klucza serwera, zapewniający autentyczność tokenu.

```
1 public String generateToken(UserDetails userDetails) {  
2     UserEntity user = userRepository  
3         .findByEmail(userDetails.getUsername())  
4         .orElseThrow(() -> new UsernameNotFoundException  
5             ("User not found"));  
6  
7     Map<String, Object> claims = new HashMap<>();  
8     claims.put("provider", "kkBackend");  
9     claims.put("id", String.valueOf(user.getId()));  
10    claims.put("roles", user.getRole().split(","));  
11  
12    return Jwts.builder()  
13        .claims(claims)  
14        .subject(userDetails.getUsername())  
15        .issuedAt(Date.from(Instant.now()))  
16        .expiration(Date.from(Instant.now().plusMillis(EXPIRATIONTIME)))  
17        .signWith(generateKey())  
18        .compact();  
19 }
```

Token posiada datę wygaśnięcia, co ogranicza czas jego ważności i minimalizuje ryzyko nadużyć w przypadku przechwycenia.

Weryfikacja tokenu odbywa się poprzez nagłówek **Authorization** w formacie **Bearer <token>**. Po stronie serwera działa filtr **JwtAuthenticationFilter**, który przechwytuje każde żądanie HTTP i wykonuje ekstrakcję tokenu z nagłówka. Następnie weryfikuje jego integralność poprzez ponowne obliczenie podpisu z wykorzystaniem klucza. Dalej przechodzi do sprawdzenia daty ważności; jeśli użytkownik posiada wygaśnięty token, odrzuca go. Ostatecznie wyciąga dane z tokenu i tworzy obiekt autoryzacji, który jest umieszczany w kontekście Spring Security.

```

1 @GetMapping("/aboutme")
2 public MyProfileDto getMyProfile(@RequestHeader("Authorization")
3     String token){
4     return userService.getMyProfile(token);
5 }

```

Przykład komunikacji:

```

{
  "username": "test",
  "password": "password",
  "email": "test@test.com"
}

```

Rysunek 4: Żądanie POST /api/register

Odpowiedź:

```

1 {
2   "token": "eyJhbGciOiJIUzUxMiJ9.eyJwcm92aWRLciI6ImtrQmFja2VuZFRlYW0iLCJyb2xleSI6WyJVU0VSIl0sImkIjoIMTY2Iiwic3ViIjoiaGVzZEB0ZXN0LmNvbSI6Im1hdCI6MTc2MDcxMjE3NCwiZXhwIjoxNzYwOTcxMzc0fQ.UDDCjr_pbFy0fvW0sS5AA0hV8rkFV2iXgfiaTSAv4LK637v1qrhm3Qdb1_xkYPNWTaoiR_VNUj2HL53GdCcHng",
3   "type": "Bearer"
4 }

```

Rysunek 5: Odpowiedź na żądanie POST /api/register

Rozwiązanie to pozwala w prosty sposób uzyskać informacje o aktualnie zalogowanym użytkowniku oraz jego uprawnieniach, co umożliwia precyzyjną kontrolę dostępu.

Po pomyślnym zalogowaniu użytkownik zyskuje dostęp do panelu zarządzania swoim profilem, który umożliwia kompleksową personalizację danych konta. System umożliwia zmianę nazwy użytkownika, adresu e-mail, hasła oraz opisu profilu. Dzięki integracji z usługą Cloudinary jest w stanie zmienić swoje zdjęcie profilowe lub zdjęcie tła profilu. W przypadku rezygnacji z korzystania z aplikacji, użytkownik ma możliwość trwałego usunięcia swojego konta. Skutkuje to całkowitym wymazaniem jego danych z systemu.

3.7.2 Zarządzanie przepisami

Moduł zarządzania przepisami stanowi centralną część aplikacji i odpowiada za prezentację i modyfikację dań. Niezalogowany użytkownik posiada możliwość przeglądania wszystkich publicznych przepisów, jak również filtrowania ich według kategorii, czasu przygotowania, ilości porcji, języka, składników, nazwy oraz trudności. Dzięki temu system jest użyteczny dla osób nieposiadających konta, zapewniając im dostęp do bazy przepisów.

Aby móc w pełni skorzystać z funkcjonalności aplikacji, takich jak dodawanie przepisów, edycja własnych przepisów, ocenianie potraw czy dodawanie ich do ulubionych, wymagane jest zalogowanie się do systemu. Token JWT gwarantuje, że wyłącznie właściciel przepisu lub osoba posiadająca odpowiednie uprawnienia (Admin) może wprowadzać w nim zmiany.

Operacje na przepisach są realizowane poprzez zestaw endpointów z prefixem `/recipes`. Dzięki zastosowaniu mechanizmów sortowania oraz paginacji, możliwe jest sprawne zarządzanie dużą ilością przepisów przy zachowaniu wysokiej wydajności aplikacji.

```
1  @GetMapping
2  public RecipeRequestDto getRecipesByPage(
3      @RequestParam(defaultValue = "1", required = false) @Min(1) int page,
4      @RequestParam(defaultValue = "10", required = false) @Min(1) int size,
5      @RequestParam(required = false) List<String> category,
6      @RequestParam(required = false) Integer difficulty,
7      @RequestParam(required = false) String servings,
8      @RequestParam(required = false) String prepareTime,
9      @RequestParam(defaultValue = "true") Boolean isPublic,
10     @RequestParam(required = false) String language,
11     @RequestParam(required = false) List<String> ingredients
12 ){
13     return recipeService.getRecipeByPage(page, size, category, difficulty,
14         servings, prepareTime, isPublic, language, ingredients);
15 }
```

Na powyższym przykładzie widnieje sposób pobierania listy przepisów. Operacja ta została zaimplementowana jako metoda kontrolera. Parametry zapytania są przekazywane jako `@RequestParam` i obejmują między innymi numer strony, rozmiar strony i wcześniej wymienione filtry. Dzięki takiemu podejściu aplikacja może w efektywny sposób obsługiwać dużą liczbę rekordów, zapewniając użytkownikowi szybki dostęp do interesujących go treści.

3.8 Obsługa błędów

Aby zapewnić spójność i przewidywalność działania aplikacji w przypadku wystąpienia błędu, zaimplementowano mechanizm globalnej obsługi wyjątków. Pozwala to w jednolity sposób reagować na różnego rodzaju wyjątki, eliminując konieczność powtarzania logiki obsługi w poszczególnych komponentach. Ponadto w systemie zastosowano niestandardowe klasy wyjątków, ułatwiające identyfikację źródła problemu.

`@ControllerAdvice` to mechanizm Spring Boot umożliwiający globalne przechwytywanie i obsługę wyjątków oraz błędów w całej aplikacji. Działa jako centralny punkt zarządzania błędami wszystkich kontrolerów. Framework automatycznie skanuje klasy oznaczone adnotacją `@ControllerAdvice` i rejestruje je jako globalne handlersy wyjątków. Gdy w którymś kontrolerze wystąpi wyjątek, Spring sprawdza, czy istnieje odpowiedni handler w `@ControllerAdvice`. Jeśli tak, przekazuje mu kontrolę zamiast zwracać domyślną wartość błędu. Gdy w trakcie obsługi żądania

pojawi się wyjątek, Spring przekazuje go do odpowiedniej metody oznaczonej adnotacją `@ExceptionHandler`, która określa, które typy wyjątków mają być obsługiwane przez daną metodę.

Metoda obsługująca wyjątek przygotowuje obiekt odpowiedzi zawierający kod statusu HTTP, opis błędu oraz datę jego wystąpienia.

```
1  @ControllerAdvice
2  public class GlobalExceptionHandler {
3
4      @ExceptionHandler(RecipeNotFoundException.class)
5      public ResponseEntity<ErrorObject> handleRecipeNotFoundException
6      (RecipeNotFoundException ex) {
7          ErrorObject errorObject = new ErrorObject();
8          errorObject.setStatusCode(HttpStatus.NOT_FOUND.value());
9          errorObject.setMessage(ex.getMessage());
10         return new ResponseEntity<ErrorObject>
11             (errorObject, HttpStatus.NOT_FOUND);
12     }
13 }
```

W powyższym przykładzie w przypadku wystąpienia wyjątku `RecipeNotFoundException` zwracany jest obiekt `ErrorObject` zawierający status błędu 404, komunikat o treści wyjątku oraz datę jego wystąpienia.

3.9 Podsumowanie rozdziału

Podsumowując, implementacja systemu została oparta na architekturze warstwowej, dzięki czemu aplikacja posiada wyraźne rozdzielenie logiki na kontrolery, serwisy i warstwę dostępu do danych, co ułatwia utrzymanie i testowanie kodu. Backend został zbudowany w technologii Spring Boot, zapewniając automatyczną konfigurację i integrację z bazą danych PostgreSQL. Umożliwiło to stworzenie spójnego RESTful API. Mechanizm autoryzacji oparto na tokenach JWT, co pozwoliło na wyeliminowanie sesji po stronie serwera i uzyskanie architektury *stateless*, istotnej z punktu widzenia skalowalności. Przedstawiono logikę biznesową aplikacji między innymi: zarządzanie profilem użytkownika, wyświetlanie, filtrowanie, dodawanie oraz aktualizacje przepisów. W systemie zastosowano globalną obsługę wyjątków z wykorzystaniem adnotacji `@ControllerAdvice`, zapewniającą jednolity i czytelny sposób wyświetlania błędów.

Przyjęte podejście niesie za sobą zalety, takie jak bezpieczeństwo danych (hasła hashowane algorytmem BCrypt), wysoka czytelność i modularność kodu.

Możliwość dalszego rozwoju systemu obejmują wprowadzenie mechanizmu odświeżania tokenów oraz wieloskładnikowe uwierzytelnianie (2FA), rozbudowę logiki biznesowej o funkcje społecznościowe, takie jak komentarze czy dodawanie innych użytkowników do listy znajomych.

4 Implementacja aplikacji frontendowej

4.1 Przegląd architektury

Do stworzenia strony internetowej aplikacji użyto frameworka Angular. Jest to rozbudowane środowisko, które oferuje wiele wbudowanych funkcjonalności, dzięki czemu nie ma potrzeby korzystania z zewnętrznych bibliotek. Pozwala to uniknąć czasochłonnej integracji i zapewnia spójność projektu. Angular posiada między innymi wbudowany routing, obsługę formularzy, internacjonalizację (i18n) oraz wiele innych narzędzi. Dzięki temu projekt jest spójny, ustrukturyzowany, a jego architektura opiera się na modułach, komponentach i serwisach. Umożliwia to łatwą skalowalność oraz zachowuje czytelność i separację odpowiedzialności. Framework jest napisany w TypeScript i ściśle z nim zintegrowany. Zalety takiego rozwiązania to statyczne typowanie, które zmniejsza ilość błędów oraz przejrzystość kodu. Dodatkowo, Angular oferuje wbudowany mechanizm wstrzykiwania zależności (Dependency Injection), co ułatwia testowanie komponentów, dynamiczne podmienianie serwisów oraz modularność kodu.

Aplikacja została zbudowana na wersji Angular 18.2.13, który uruchamiany jest w środowisku Node.js 18 (alpine). Do budowy oraz testowania wykorzystano narzędzie Angular CLI 18.2.20, które automatyzuje proces kompilacji, generowania komponentów, uruchamianie testów oraz optymalizację kodu.

Kluczowe zależności projektu:

- `@angular/core 18.2.13` – główny moduł frameworka odpowiedzialny za logikę komponentów i serwisów.
- `@angular/router 18.2.13` – moduł nawigacji pomiędzy widokami aplikacji.
- `@angular/forms 18.2.13` – moduł do obsługi formularzy.
- `RxJS 7.8.2` – biblioteka programowania reaktywnego.
- `Zone.js 0.14.10` – mechanizm śledzenia operacji asynchronicznych.
- `TypeScript 5.5.4` – język bazujący na JavaScript z obsługą statycznego typowania.
- `@angular-devkit/build-angular 18.2.20` – zestaw narzędzi do kompilacji i optymalizacji kodu.

Dzięki takiemu zestawieniu aplikacja jest testowalna, skalowalna oraz łatwa do wdrożenia, przy jednoczesnym zachowaniu spójności wersji Angulara i jego zależności.

Aplikacja została zbudowana w oparciu o architekturę komponentową typu *standalone*. Rozwiązanie to pozwala tworzyć aplikację bez konieczności definiowania tradycyjnych modułów (`NgModule`). Upraszcza to strukturę projektu oraz konfigurację. Każdy komponent lub serwis może zostać oznaczony jako *standalone*.

```
1 @Component({  
2   selector: 'app-admin-panel',  
3   standalone: true,
```

```

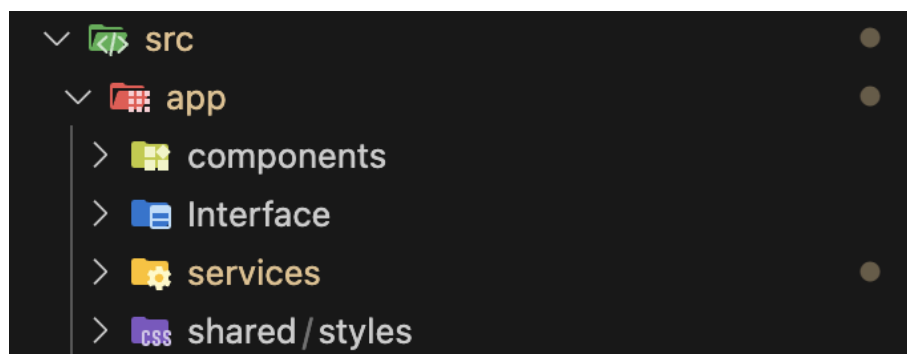
4   imports: [
5       CommonModule,
6       FormsModule,
7       TranslateModule,
8       ToastNotificationComponent,
9   ],
10  templateUrl: './admin-panel.component.html',
11  styleUrls: ['./admin-panel.component.css'],
12  })

```

Umożliwia to bezpośrednie wykorzystanie w innych częściach aplikacji bez konieczności deklarowania w osobnych modułach. Zastosowanie architektury standalone pozwala na redukcję złożoności projektu, szybszy rozwój, lepszą testowalność, mniejszy narzut konfiguracji oraz ułatwia skalowanie. W projekcie każdy kluczowy komponent został napisany jako samodzielny komponent standalone.

4.1.1 Struktura projektu

Struktura aplikacji została podzielona między trzy główne katalogi i jeden pomniejszy. Pozwala to na czytelną nawigację po projekcie.



Rysunek 6: Struktura projektu

Katalogi projektu:

- **Components/** - zawiera wszystkie komponenty interfejsu użytkownika. Każdy stanowi samodzielny komponent (standalone) posiadający własne pliki:
 - Plik TypeScript (**.ts**) z logiką.
 - Plik szablonu (**.html**) odpowiedzialny za szkielet widoku.
 - Plik stylów (**.css**) definiujący wygląd komponentu.
 - Plik testów (**.spec.ts**) zawierający logikę testów.
- **Services/** - zawiera serwisy odpowiedzialne za logikę biznesową oraz komunikację z backendem.
- **Interface/** - katalog przechowujący definicje interfejsów, które opisują struktury danych.

- **Shared/styles** - zawiera plik (.css) odpowiedzialny za wspólne dla całej aplikacji style do przycisków.

Taka organizacja folderów zapewnia modularność oraz przejrzystość. Umożliwia także niezależne rozwijanie poszczególnych części aplikacji oraz testowanie komponentów.

4.1.2 Wzorce projektowe

W projekcie zostało wykorzystane programowanie reaktywne oparte na bibliotece RxJS do zarządzania asynchronicznym strumieniem danych. Pierwszym przykładem jest zarządzanie stanem autoryzacji w `auth.service.ts`. Ekspozuje observable (strumień danych tylko do odczytu) informujący o stanie logowania.

```
1 private isLoggedInSubject = new BehaviorSubject<boolean>(false);
2 public isLoggedIn$ = this.isLoggedInSubject.asObservable();
```

Użycie `BehaviorSubject` gwarantuje, że nowi subskrybenci natychmiast otrzymają ostatnią wartość. Jest to kluczowe podczas inicjalizacji komponentów.

Następnym przykładem jest mechanizm **Debounce**. Zastosowano go w celu optymalizacji komunikacji między aplikacją frontendową a serwerem API. Wykorzystuje operator `debounceTime()` z biblioteki RxJS. Rozwiązanie to zostało zaimplementowane we wszystkich komponentach wysyłających zapytania do API.

```
1 constructor(
2   private recipeService: RecipeService,
3   private router: Router,
4   private notificationService: NotificationService,
5   private authService: AuthService
6 ) {
7   // Setup debounced submit attempts
8   this.submitAttempts$
9     .pipe(debounceTime(1000), takeUntil(this.destroy$))
10    .subscribe(() => {
11      this.executeSubmit();
12    });
13 }
```

Zastosowanie operatora `debounceTime(1000)` powoduje opóźnienie wysłania zapytania o 1000 milisekund. W praktyce oznacza to, że przy wielokrotnym wywołaniu akcji w krótkim odstępie czasu, tylko ostatnie zostanie przetworzone. Debounce przyczynia się do znaczącej redukcji zapytań HTTP kierowanych do serwera, znacząco zmniejszając obciążenie. Poprawia także responsywność interfejsu użytkownika.

W aplikacji zaimplementowano wzorec zapobiegający wyciekowi pamięci, który systematycznie anuluje aktywne subskrypcje w momencie niszczenia komponentu. W tym celu wykorzystano metodę `takeUntil()` oraz Lifecycle hook `ngOnDestroy()`. Każdy komponent wykorzystujący subskrypcje deklaruje prywatny obiekt `Subject`, który służy jako sygnalizator zakończenia.

```

1 private destroy$ = new Subject<void>();
2
3 ngOnDestroy(): void {
4     this.destroy$.next();
5     this.destroy$.complete();
6 }

```

Mechanizm działa według następującego schematu:

Tworzony jest obiekt `Subject` jako sygnalizator zakończenia.

↓

Dla każdej subskrypcji dodawany jest operator `takeUntil(this.destroy$)`, który nasłuchuje zakończenia.

↓

W momencie niszczenia komponentu, Angular automatycznie wywołuje metodę `ngOnDestroy()`, która emituje wartość przez `next()`, co powoduje anulowanie wszystkich powiązanych subskrypcji.

↓

Ostatecznie, wywołanie `complete()` zwalnia zasoby obiektu `Subject`.

Zastosowanie takiego rozwiązania skutecznie zapobiega wyciekom pamięci. Eliminuje ryzyko wykonywania operacji na nieistniejących komponentach, co mogłoby prowadzić do błędów.

4.2 Serwisy HTTP

4.2.1 Auth.service.ts

W aplikacji zaimplementowano autentykację opartą na tokenie JWT. Podczas procesu logowania, po pomyślnej weryfikacji danych, serwer zwraca token w odpowiedzi HTTP. Aplikacja zapisuje go w Web Storage API.

```

1 login(email: string, password: string, rememberMe: boolean = false):
2     Observable<any>
3 {
4     const body = { email: email, password: password };
5     return this.http.post(this.apiUrl + '/authenticate', body).pipe(
6         tap((response: any) => {
7             if (response && response.token) {
8                 if (rememberMe) {
9                     localStorage.setItem('token', response.token);
10                } else {
11                    sessionStorage.setItem('token', response.token);
12                }
13                this.isLoggedInSubject.next(true);
14            }
15        })
16    );
17 }

```

W przypadku, jeśli użytkownik zaznaczy opcję „Zapamiętaj mnie”, token jest zapisywany w `localStorage`, co zapewnia trwałość mimo zamknięcia przeglądarki.

Alternatywnie, token jest przechowywany wewnątrz `sessionStorage`, który jest czyszczony automatycznie po wyłączeniu sesji przeglądarki lub zamknięciu karty z aplikacją.

Dekodowanie i walidowanie tokenu JWT odbywa się za pomocą trzech metod: `isAdmin()`, `getUserRoles()`, `hasRole()`, każda z nich działa w podobny sposób.

```
1  getUserRoles(): string[] {
2      const token = this.getToken();
3      if (!token) return [];
4      try {
5          const payload = JSON.parse(atob(token.split('.')[1]));
6          return payload.roles || [];
7      } catch (error) {
8          return [];
9      }
10 }
```

Wstępnie token jest dzielony za pomocą metody `split('.')`, a następnie wybierany jest drugi segment zawierający payload. Segment jest zakodowany, dlatego wykorzystywana jest funkcja `atob()` do dekodowania. Wynik następnie jest parsowany do zmiennej `payload` i ostatecznie zwracane są role użytkownika. Metoda jest zabezpieczona blokiem `try-catch`, który obsługuje potencjalne błędy wynikające z nieprawidłowego formatu tokenu lub uszkodzenia. W przypadku wystąpienia błędu, metoda zwróci pustą tablicę.

Serwis wykorzystuje wzorzec `Observable` do reaktywnego zarządzania stanem autentykacji wewnątrz całej aplikacji.

```
1  private isLoggedInSubject = new BehaviorSubject<boolean>(false);
2  public isLoggedIn$ = this.isLoggedInSubject.asObservable();
3
4  constructor(private http: HttpClient, private router: Router) {
5      this.checkInitialAuthState();
6  }
7
8  private checkInitialAuthState(): void {
9      const token = this.getToken();
10     if (token) {
11         this.isLoggedInSubject.next(true);
12     } else {
13         this.isLoggedInSubject.next(false);
14     }
15 }
```

Podczas inicjalizacji serwisu, w konstruktorze wywoływana jest metoda `checkInitialAuthState()`, która sprawdza czy token znajduje się w `localStorage` bądź w `sessionStorage` przy użyciu metody pomocniczej `getToken()`. Następnie bazując na zwróconej odpowiedzi aktualizuje stan autentykacji. Eliminuje to potrzebę ponownego logowania po odświeżeniu strony.

Proces wylogowywania przebiega przy użyciu metody `logout()`.

```

1 logout(): void {
2   localStorage.removeItem('token');
3   sessionStorage.removeItem('token');
4   this.isLoggedInSubject.next(false);
5   this.router.navigate(['/login']);
6 }

```

Usuwa ona token z `localStorage` oraz `sessionStorage`, aktualizuje stan autentykacji na `false`, co powoduje aktualizację wszystkich komponentów subskrybujących strumień `isLoggedIn$`. Ostatecznie użytkownik przekierowywany jest na stronę logowania przy użyciu serwisu `Router`.

Dekodowanie tokenu JWT po stronie klienta nie stanowi zagrożenia, ponieważ odczytane metadane nie zawierają wrażliwych informacji o użytkowniku takich jak hasło a jedynie dane identyfikujące. Faktyczna walidacja tokenu odbywa się po stronie serwera przy odkodowaniu podpisu cyfrowego.

4.2.2 Recipe.service.ts

Serwis odpowiada za komunikację z API serwera, w kontekście zarządzania przepisami. W celu bezpieczeństwa, przed wysłaniem zapytania wymagającego autoryzacji wykorzystywana jest prywatna metoda `getAuthHeaders()`. Wydobywa token z serwisu autoryzacji i zapisuje go do zmiennej, po czym, jeśli go znajdzie zwraca obiekt nagłówków zawierający pole `Authorization` z tokenem. Jeśli token nie jest dostępny, zwracany jest pusty obiekt nagłówków. Skutkuje to żądaniem bez danych uwierzytelniających i zostaje odrzucone przez serwer z kodem błędu 401 Unauthorized.

```

1 private getAuthHeaders(): HttpHeaders {
2   const token = this.authService.getToken();
3   if (token) {
4     return new HttpHeaders().set('Authorization', `Bearer ${token}`);
5   }
6   return new HttpHeaders();
7 }

```

Przykładem zabezpieczonego endpointu jest dodawanie przepisu, które wykorzystuje token w celach autoryzacji. Przed wysłaniem żądania, dane z formularza są zbierane do obiektu `FormData`. Następnie do zmiennej `headers` przypisywany jest nagłówek wydobyty z metody `getAuthHeaders()`. Ostatecznie, wysyłane jest żądanie HTTP POST przekazując do klienta `HttpClient` zawierające, URL na który ma zostać wysłane żądanie, przygotowany obiekt `FormData` oraz nagłówek autoryzacyjny.

```

1 postRecipe(recipeData: FormData): Observable<any> {
2   const headers = this.getAuthHeaders();
3   return this.http.post<any>(this.url, recipeData, { headers });
4 }

```

4.2.3 Filter.service.ts

Serwis `FilterService` odpowiedzialny jest za logikę filtrowania, paginacji oraz pobierania danych dotyczących przepisów z serwera. Zaprojektowany został zgodnie z wzorcem reaktywnego programowania. Serwis jest odpowiedzialny za zarządzanie stanem filtrów, wykonywaniem zapytań do API na podstawie filtrów oraz dynamicznym generowaniu opcji do wykorzystania w interfejsie użytkownika.

Sercem serwisu jest `BehaviorSubject`, pełniące funkcję centralnego źródła prawdy dla aktualnych filtrów.

```
1 private filterState$ = new BehaviorSubject<FilterState>({
2   currentPage: 1,
3   pageSize: 12,
4 });
```

Każdy komponent subskrybujący do `filterState$` natychmiast otrzyma bieżący stan. Modyfikacja stanu odbywa się poprzez metodę `updateFilters()`, co gwarantuje przewidywalny przepływ danych.

```
1 updateFilters(filters: Partial<FilterState>): void {
2   const currentState = this.filterState$.value;
3   const newState = { ...currentState, ...filters };
4   this.filterState$.next(newState);
5 }
```

Prywatna metoda `buildParams()` odpowiada za tłumaczenie stanu `filterState` na zapytania http. Zawiera ona logikę biznesową np. konwersje zakresów liczbowych (czas przygotowania czy liczba porcji) na format tekstowy oczekiwany przez API serwera.

```
1 private buildParams(filters?: Partial<FilterState>): any {
2   const currentFilters = { ...this.currentFilters, ...filters };
3   const params: any = {
4     page: currentFilters.currentPage,
5     size: currentFilters.pageSize,
6   };
7
8   if (currentFilters.difficulty) {
9     params.difficulty = currentFilters.difficulty;
10  }
11  if (currentFilters.prepTimeFrom !== undefined ||
12      currentFilters.prepTimeTo !== undefined) {
13    const prepTimeFrom = currentFilters.prepTimeFrom ?? 0;
14    const prepTimeTo = currentFilters.prepTimeTo ?? 99999;
15    params.prepareTime = `${prepTimeFrom}-${prepTimeTo}`;
16  }
17  if (currentFilters.servingsFrom !== undefined ||
18      currentFilters.servingsTo !== undefined) {
19    const servingsFrom = currentFilters.servingsFrom ?? 0;
20    const servingsTo = currentFilters.servingsTo ?? 99999;
```

```

21     params.servings = `${servingsFrom}-${servingsTo}`;
22   }
23   if (currentFilters.category) {
24     params.category = currentFilters.category;
25   }
26   if (currentFilters.recipeLanguage) {
27     params.language = currentFilters.recipeLanguage;
28   }
29   if (currentFilters.query) {
30     params.query = currentFilters.query;
31   }
32   if (currentFilters.ingredients &&
33       currentFilters.ingredients.length > 0) {
34     params.ingredients = currentFilters.ingredients;
35   }
36   return params;
37 }

```

Metody takie jak `loadCategories()`, `loadLanguages()` oraz `loadIngredients()` dynamicznie tworzą opcje dla kontrolerek filtrujących w interfejsie użytkownika. Zamiast wysłać zapytanie o listy wartości, sprawdzają pełną listę przepisów, z których wydobywają wymagane dane. Przy użyciu obiektu `Set` wyciągane są unikalne, posortowane wartości. Rozwiązanie to gwarantuje, że dostępne filtry zawsze odpowiadają tym dostępnym w systemie.

```

1  loadCategories(): Observable<string[]> {
2    return new Observable((observer) => {
3      this.http.get<RecipesResponse>(this.url).subscribe({
4        next: (response) => {
5          if (response && Array.isArray(response.content)) {
6            const categories = Array.from(
7              new Set(response.content.map((recipe) => recipe.category))
8            ).sort();
9            observer.next(categories);
10           } else {
11             observer.next([]);
12           }
13           observer.complete();
14         },
15         error: (err) => observer.error(err),
16       });
17     });
18 }

```

4.2.4 Profile.service.ts

`ProfileService` jest odpowiedzialny za logikę biznesową oraz stan interfejsu użytkownika związanego z profilem użytkownika. Jego głównymi zadaniami są zarządzanie trybem pracy widoku profili oraz realizacja zapytań do API serwera.

Przy użyciu `BehaviorSubject` zdefiniowano trzy niezależne strumienie danych:

1. `editMode$` - zarządza stanem trybu edycji podstawowych danych profilu.
2. `settingsMode$` - kontroluje widoczność sekcji zmiany hasła oraz emaila.
3. `deleteMode$` i `showDeleteModal$` - odpowiadają za procesie usuwania konta, w tym aktywacja trybu oraz wyświetlanie okna potwierdzającego.

Wszystkie metody komunikujące się z autoryzowanymi endpointami wykorzystują znaną wcześniej metodę pomocniczą `getAuthHeaders()`. Logika serwisu obejmuje pobieranie danych o profilu, przepisach i ulubionych przepisach użytkownika przy użyciu metod `getUserProfile()`, `getUserRecipes()` oraz `getUserFavourites()`.

```
1 getUserProfile(): Observable<any> {  
2   const headers = this.getAuthHeaders();  
3   return this.http.get<any>(`${environment.apiUrl}/users/aboutme`, {  
4     headers,  
5   });  
6 }
```

Za aktualizację profilu użytkownika odpowiedzialne są metody:

- `updateUserProfile()`,
- `updateUserEmail`,
- `updateUserPassword()`.

Proces usuwania konta obsługiwany jest przez metodę `deleteUser()`.

W serwisie zaimplementowano logikę odpowiedzialną za zmianę zdjęcia profilowego oraz zdjęcia tła profilu użytkownika. Metody `editProfilePicture()` oraz `editBackgroundPicture()` wchodzi w interakcję z modelem DOM. Tworzą one element `<input type="file">` i symulują jego kliknięcie, otwierając okno wyboru pliku. Po wybraniu pliku, metoda sprawdza rozmiar pliku, a następnie w żądaniu wysyła obiekt `FormData` do serwera. Po pomyślnym przesłaniu, serwer zwraca adres URL do nowego obrazu, który jest ustawiany jako nowy obraz w odpowiednim elemencie ``.

```
1 editProfilePicture(userId: number, event: Event): void {  
2   const fileInput = document.createElement('input');  
3   fileInput.type = 'file';  
4   fileInput.accept = 'image/*';  
5   fileInput.onChange = (e: Event) => {  
6     const input = e.target as HTMLInputElement;  
7     if (input.files && input.files.length > 0) {  
8       const file = input.files[0];  
9       if (file.size > 5 * 1024 * 1024) {  
10        alert('File size exceeds 5 MB limit.');11        return;  
12      }  
13      this.uploadProfilePicture(userId, file).subscribe({  
14        next: (imageUrl) => {
```

```

15         this.setImageSafely('.profile-picture', imageUrl);
16     },
17     error: (error) => {
18         alert('Failed to upload profile picture.');
```

4.3 Kluczowe komponenty

Aplikacja opiera się na głównym komponencie `AppComponent`, który pełni rolę nadrzędnego komponentu definiującego układ strony. Jego szablon zawarty jest w `app.component.html`. Pierwszym komponentem jest nawigacja, której wyświetlanie jest warunkowe. Opiera się o flagę `isMobile`, która jest dynamicznie aktualizowana w zależności od rozdzielczości ekranu. Do wyświetlania nawigacji służą dwa komponenty: `app-navbar` oraz `app-mobile-nav` dla urządzeń mobilnych.

Drugim elementem jest dyrektywa `router-outlet`. Działa jako dynamiczny kontener, w którym Angular renderuje komponent odpowiadający ścieżce URL.

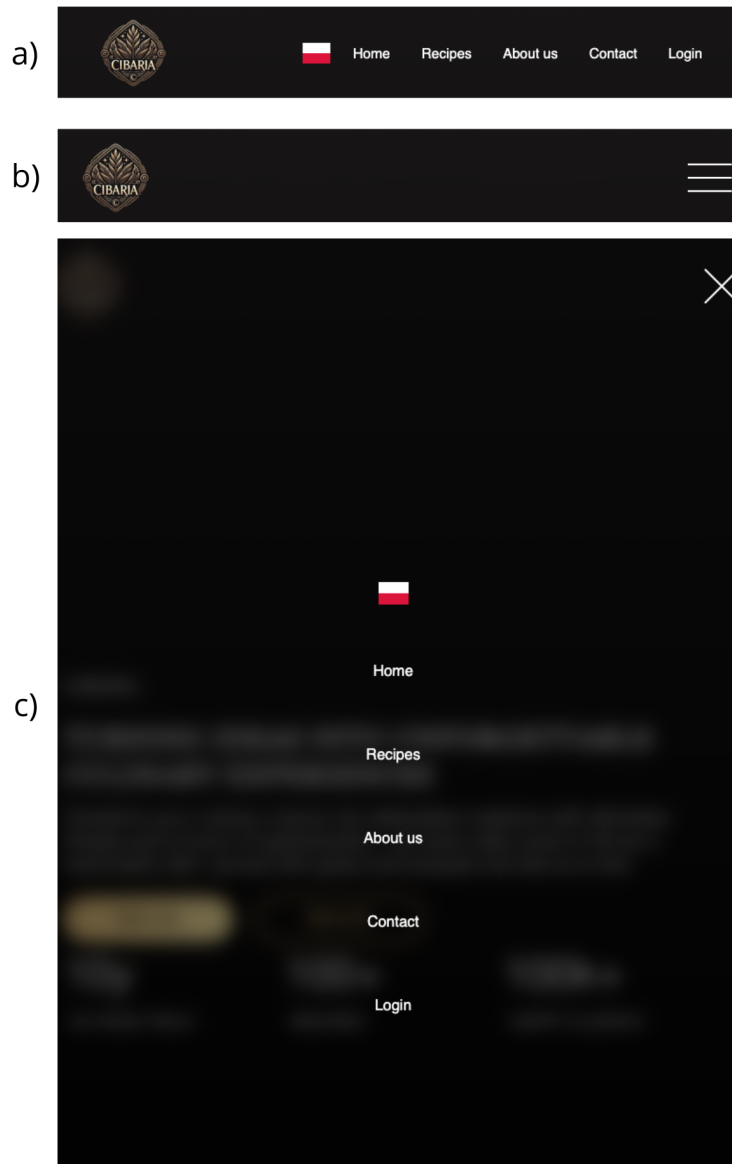
Ostatnim elementem jest `app-footer-section`, czyli komponent stopki, zamyka on strukturę każdej strony.

```

1  @if (!isMobile) {
2      <app-navbar></app-navbar>
3  } @else {
4      <app-mobile-nav></app-mobile-nav>
5  }
6
7  <router-outlet></router-outlet>
8  <app-footer-section></app-footer-section>
```

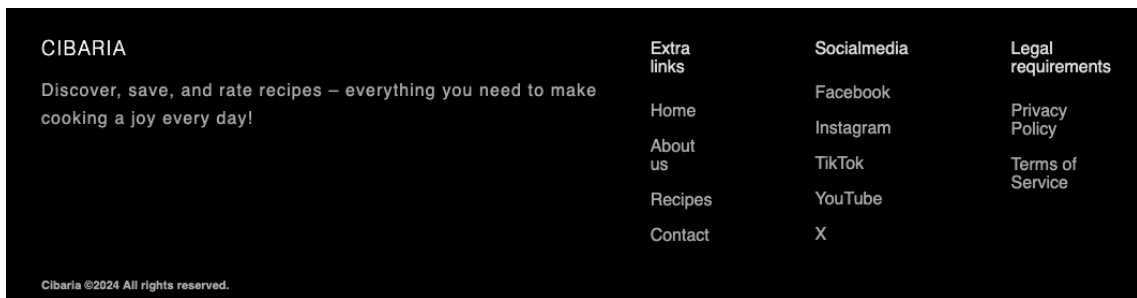
4.3.1 Nawigacja i układ

Nawigacja składa się z dwóch komponentów. Pierwszym z nich jest *navbar* dla widoków desktopowych, a drugim *mobile-nav* dla urządzeń mobilnych. Struktura obu komponentów, to lista elementów zawierających linki do poszczególnych części aplikacji. Wyświetlanie elementów jest dynamiczne i bazuje na flagach `isLoggedIn` oraz `isAdmin`. Przykładowo, jeśli użytkownik nie jest zalogowany, opcja wylogowania oraz dodania przepisu nie będzie widoczna. Z kolei dostęp do panelu administratora jest ograniczony tylko dla zalogowanego użytkownika posiadającego rolę `ADMIN`.



Rysunek 7: Responsywny komponent nawigacji: a) widok desktopowy, b) widok mobilny zwinięty, c) widok mobilny rozwinięty

Układ każdej strony zamykany jest poprzez komponent stopki (`app-footer-section`). Zawiera on podstawowe informacje, odnośniki do podstron, sekcję z przykładowymi linkami do mediów społecznościowych oraz odnośnik do dokumentów prawnych mających charakter demonstracyjny.



Rysunek 8: Komponent stopki

4.3.2 Autoryzacja

LoginComponent służy do rejestracji oraz logowania użytkowników. Wykorzystuje on w tym celu serwis autoryzacji (`auth.service.ts`) oraz strumienie typu `Subject<void>` sygnalizujące akcję użytkownika, takie jak próby logowania (`loginAttempts$`) oraz rejestracja (`registerAttempts$`). Metoda wywoływana przez interfejs użytkownika (`onLogin()`, `onRegister()`) nie uruchamia bezpośrednio logiki biznesowej, tylko emituje sygnał do strumienia `loginAttempts$`. Następnie sygnał jest przetwarzany przez *pipe* (potok operatorów), w którym kluczowy jest `debounceTime(500)`. Operator blokuje wielokrotne, szybkie wywołania przepuszczając sygnał dopiero po pięciuset milisekundach od przerwania akcji użytkownika. Po przejściu przez filtr, wykonywana jest metoda `executeLogin()`, odpowiedzialna za właściwy proces uwierzytelniania oraz przekierowania zalogowanego użytkownika na jego profil. Podejście to zmniejsza obciążenie serwera i poprawia responsywność interfejsu.

```

1 private loginAttempts$ = new Subject<void>();
2 private registerAttempts$ = new Subject<void>();
3
4 constructor(
5     private authService: AuthService,
6     // ...
7 ) {
8     this.loginAttempts$
9         .pipe(debounceTime(500), takeUntil(this.destroy$))
10        .subscribe(() => {
11            this.executeLogin();
12        });
13
14    this.registerAttempts$
15        .pipe(debounceTime(500), takeUntil(this.destroy$))
16        .subscribe(() => {
17            this.executeRegister();
18        });
19 }
20
21 private executeLogin(): void {
22     if (this.isLoginLoading) {
23         return;

```

```

24   }
25   this.isLoginLoading = true;
26   this.lastLoginAttempt = Date.now();
27
28   this.authService
29     .login(this.formEmail, this.formPassword, this.rememberMe)
30     .subscribe({
31       next: (response) => {
32         this.router.navigate(['/profile']);
33         this.isLoginLoading = false;
34       },
35       error: (err) => {
36         this.notificationService.error('Email or password is incorrect');
37         this.isLoginLoading = false;
38       },
39     });
40 }

```

Komponent zawiera także dodatkową warstwę ochrony poprzez ograniczenie czasowe `minTimeBetweenAttempts` (dwie sekundy). System sprawdza odstęp czasowy między próbami zalogowania lub rejestracji i blokuje zbyt częste żądania wyświetlając komunikat ostrzegawczy.

```

1   const now = Date.now();
2   if (now - this.lastRegisterAttempt < this.minTimeBetweenAttempts) {
3     this.notificationService.warning('Please wait before trying again');
4     return;
5   }

```

Dodatkowo dzięki flagom `isLoginLoading` oraz `isRegisterLoading`, komponent kontroluje stan interfejsu podczas akcji asynchronicznych, wprowadzając dodatkową warstwę ochronną przeciwko wielokrotnym wywołaniom tej samej akcji.

a)

Log in to explore the world of Cibaria!

E-mail

Password

Remember me ☐

Login

Don't have an account?

Register

b)

Sign up and start your journey with Cibaria!

Username 0/16

Email

Password

Confirm Password

Register

Already have an account?

Login

Rysunek 9: Komponent logowania a) ekran logowania b) ekran rejestracji

4.3.3 Przepisy

RecipeComponent `RecipesComponent` odpowiada za wyświetlanie, filtrowanie oraz paginację przepisów. Filtracja przepisów odbywa się poprzez subskrypcję do strumienia `filters$` z `FilterService` przechowującego stan filtrów. Komponent nasłuchuje zmian w strumieniu i automatycznie synchronizuje interfejs użytkownika.

Metoda `loadRecipes()` pobiera przepisy na podstawie aktualnie wybranych filtrów. Odpowiedź z serwera przechodzi dodatkową walidację i zostają wyświetlone wyłącznie przepisy oznaczone jako publiczne (`isPublic === true`).

```

1 loadRecipes(): void {
2   const currentFilters = this.filterService.currentFilters;
3   this.filterService
4     .loadRecipes(currentFilters)
5     .pipe(takeUntil(this.destroy$))
6     .subscribe({
7       next: (response) => {
8         if (response && Array.isArray(response.content)) {
9           this.recipesArray = response.content.filter(
10             (recipe) => recipe.isPublic === true
11           );
12           this.totalPages = response.totalPages;
13         } else {
14           this.notificationService.error(
15             'Failed to load recipes. Please try again later.',
16             5000
17           );

```

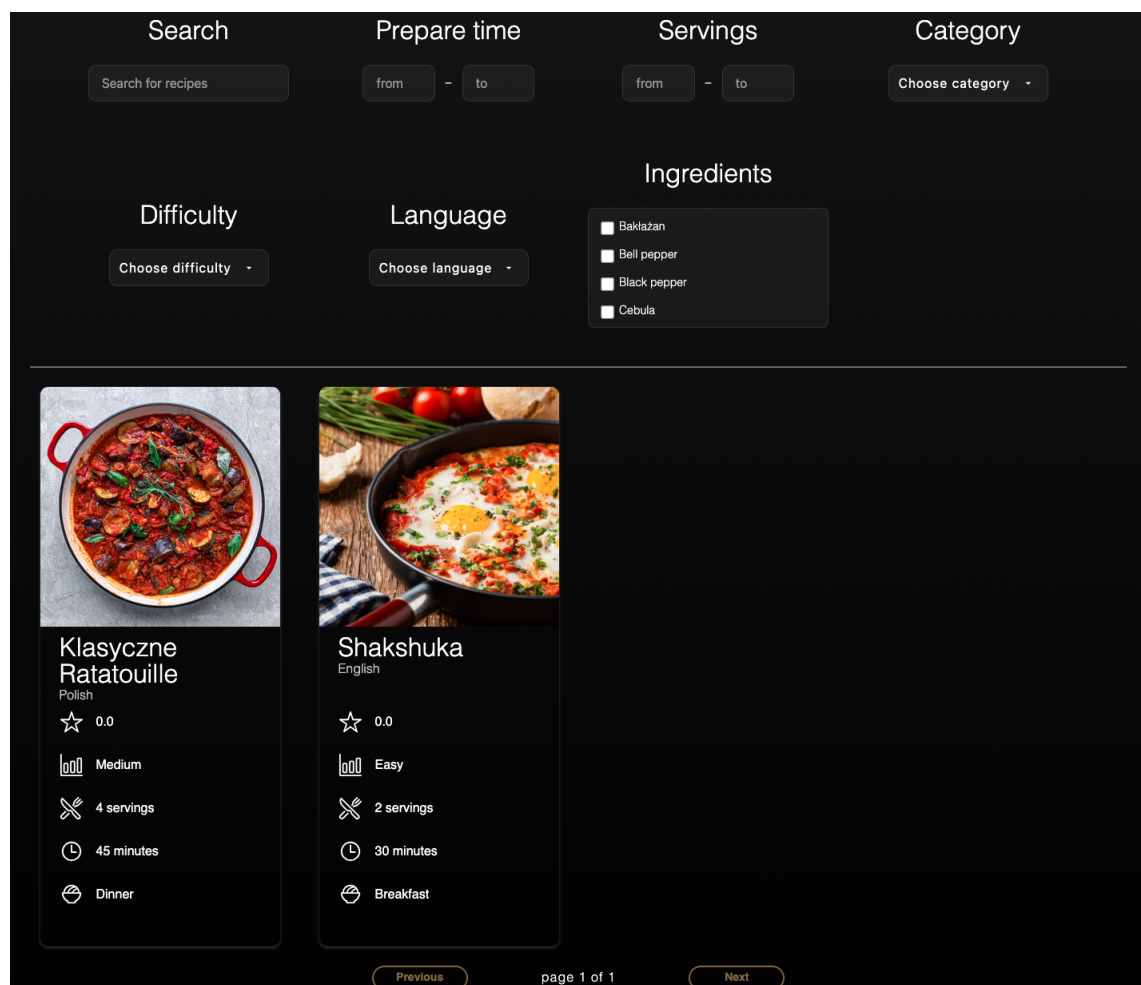
```

18         this.recipesArray = [];
19     }
20 },
21     error: () => {
22         this.notificationService.error(
23             'Failed to load recipes. Please try again later.',
24             5000
25         );
26         this.recipesArray = [];
27     },
28 });
29 }

```

Paginację zarządza metoda `onPageChange()`, która nie tylko zmienia stronę wyników, ale także automatycznie przewija widok do góry.

Komponent dostosowuje interfejs użytkownika do rozmiaru ekranu. Odpowiada za to dekorator `@HostListener`, który monitoruje zmiany szerokości okna przeglądarki i przy przekroczeniu 1350 pikseli przełącza tryb wyświetlania filtrów.



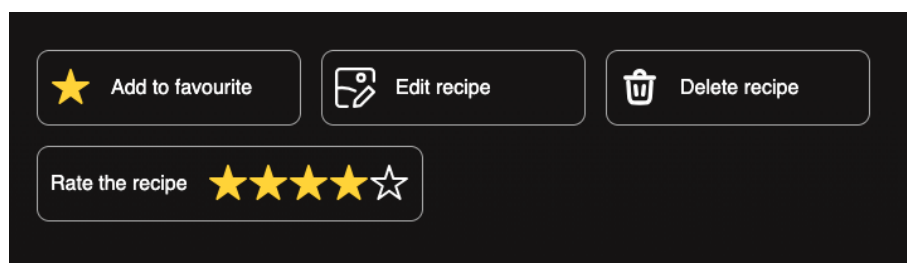
Rysunek 10: Komponent wyświetlania wszystkich publicznych przepisów

RecipeDetailedComponent `RecipeDetailedComponent` odpowiedzialny jest za wyświetlanie szczegółów pojedynczego przepisu wraz z interaktywnymi funkcjonalnościami. Dzieli się na cztery obszary: wyświetlanie danych, system oceniania, zarządzanie ulubionymi oraz kontrolę uprawnień właściciela.

Ładowanie oraz weryfikacja danych rozpoczynają się podczas inicjalizacji komponentu, z którego wydobywane jest ID przepisu. Następnie metoda `loadRecipeDetails()` pobiera dane przepisu i przekształca składniki do odpowiedniego formatu. W przypadku braku przepisu (error 404 Not Found), użytkownik automatycznie zostanie przekierowany na stronę błędu. Równolegle sprawdzane są uprawnienia użytkownika oraz ocena przepisu, czy użytkownik ma przepis w ulubionych.

```
1 private loadRecipeDetails(): void {
2   this.recipeService.loadRecipeDetails(this.recipeId).subscribe({
3     next: (response) => {
4       this.recipeDetails = response;
5       this.ingredients = this.recipeDetails.ingredients
6         .map((ingredient) => ({
7           name: ingredient.ingredientName,
8           quantity: Number(ingredient.quantity),
9           unit: ingredient.unit,
10        }));
11      this.checkOwnership();
12    },
13    error: (err) => {
14      // Check if it's a 404 error (recipe not found)
15      if (err.status === 404) {
16        this.router.navigate(['/not-found']);
17      } else {
18        this.notificationService.error('Error loading recipe details');
19      }
20    },
21  });
22 }
```

System oceniania wykorzystuje pięciogwiazdkową skalę. Użytkownik może ocenić przepis zaznaczając ilość gwiazdek. Ocenianie zabezpieczone jest poprzez `debounceTime(500)`, które zapobiega przeciążeniom serwera.



Rysunek 11: Panel interakcji użytkownika

Dodawanie przepisu do ulubionych jest obsługiwane przy pomocy metody `toggleFavourite()`, która sprawdza stan autoryzacji i kontroluje częstotliwość żądań.

Opcja usunięcia oraz edycji przepisu jest dostępna tylko dla właściciela przepisu. Usuwanie przepisu posiada dwuetapową weryfikację. Po wywołaniu `delete()` wyświetlane jest okno potwierdzenia, które użytkownik może anulować `cancelDelete()` lub potwierdzić `confirmDelete()`. Po pomyślnym usunięciu, użytkownik zostaje poinformowany o konieczności odświeżenia strony w celu zauważenia zmian. Jeśli użytkownik chce edytować przepis, zostanie przekierowany do podstrony `update-recipe`, gdzie ma możliwość aktualizacji przepisu.

AddRecipePanelComponent Komponent jest odpowiedzialny za dodawanie przepisów przy pomocy formularza. Wstawiać nowy przepis mogą wyłącznie zalogowani użytkownicy. Formularz `FormGroup`, posiada kontrolki odnośnie do struktury przepisu oraz walidatory.

```
1 recipeForm = new FormGroup({
2   title: new FormControl('', [Validators.required]),
3   description: new FormControl('', [Validators.required]),
4   category: new FormControl('', [Validators.required]),
5   servings: new FormControl(null, [Validators.required,
6                                     Validators.min(1)]),
7   prepareTime: new FormControl(null, [
8     Validators.required,
9     Validators.min(1),
10  ]),
11  quantity: new FormControl(this.ingredients, [Validators.required]),
12  unit: new FormControl(this.ingredients, [Validators.required]),
13  difficulty: new FormControl(null, [Validators.required]),
14  images: new FormControl<File | null>(null, [Validators.required]),
15  steps: new FormControl(this.steps, [Validators.required]),
16  ingredients: new FormControl(this.ingredients, [Validators.required]),
17  isPublic: new FormControl(this.isPublic, [Validators.required]),
18  language: new FormControl(this.recipeLanguage, [Validators.required]),
19 });
```

Dodawanie oraz usuwanie elementów zostało zaimplementowane metodami `addIngredient()`, `addStep()`, `removeIngredient()`, `removeStep()`, reszta kontrolek zarządzana jest poprzez `<select>` w szkielecie komponentu. System weryfikuje czy dany składnik już istnieje i powiadamia o tym użytkownika. Dodatkowo jeśli użytkownik chce, aby składnik był opcjonalny, może to wykonać metodą `toggleOptional()`.

System wysyłania obrazów został zaimplementowany na dwa sposoby: użytkownik może wybrać plik poprzez okno dialogowe lub przeciągnąć je z pulpitu. Następnie sprawdzany jest rozmiar pliku, jeśli przekroczy pięć megabajtów, użytkownik zostanie o tym poinformowany. W momencie wprowadzenia pliku, wyświetlany jest podgląd zdjęcia przy użyciu `FileReader` API.

```
1 const reader = new FileReader();
2 reader.onload = (e) => {
3   const result = e.target?.result as string;
4   this.imagePreview = result;
```

```

5   this.recipeForm.patchValue({
6     // update the form with the selected file
7     images: file,
8   });
9 };

```

Ostatecznie, jak użytkownik wprowadzi wszystkie wymagane dane, może wykonać wysłanie przepisu. W tym celu ponownie wykorzystano bibliotekę RxJS oraz strumień typu `Subject` `submitAttempts$` z operatorem `debounceTime(1000)`, który opóźnia wykonanie o sekundę. Metoda `executeSubmit()` wprowadza dodatkową weryfikację przed wysłaniem danych:

- Sprawdza flagę `isSubmitting`, blokując równoległe wysłanie kilku żądań.
- Weryfikuje autoryzację użytkownika.
- Kontroluje, czy formularz został wypełniony (*pristine* lub *untouched*).
- Wymaga minimalnie jednego składnika i kroku przygotowania.

Po pomyślnej weryfikacji, metoda tworzy obiekt `FormData`, umożliwiając przesłanie danych JSON oraz obrazu w jednym żądaniu. Dane przepisu są serializowane do formatu JSON i dołączane jako pole `recipe`. Obraz dodawany jest osobno jako pole `images`. Flaga `isSubmitting` pozostaje aktywna przez cały czas trwania żądania.

W przypadku pomyślnego wysłania, formularz jest resetowany metodą `resetForm()`, która czyści wszystkie pola oraz usuwa podgląd obrazu. Użytkownik dostaje powiadomienie o pomyślnym utworzeniu przepisu. Jeśli wystąpi błąd, zostaje wyświetlony komunikat o niepowodzeniu, a flaga `isSubmitting` jest resetowana, umożliwiając ponowną próbę.

```

1   this.submitAttempts$
2     .pipe(debounceTime(1000), takeUntil(this.destroy$))
3     .subscribe(() => {
4       this.executeSubmit();
5     });
6
7   private executeSubmit() {
8     if (this.isSubmitting) {
9       return;
10    }
11    if (!this.authService.isAuthenticated()) {
12      this.notificationService.error('User is not logged in!', 5000);
13      return;
14    }
15    if (this.recipeForm.pristine || this.recipeForm.untouched) {
16      this.notificationService.error('Please fill in the form!', 5000);
17      return;
18    }
19    if (this.ingredients.length === 0) {
20      this.notificationService.error(

```

```

21     'Please add at least one ingredient!',
22     5000
23   );
24   return;
25 }
26 if (this.steps.length === 0) {
27   this.notificationService.error('Please add at least one step!', 5000);
28   return;
29 }
30
31 this.isSubmitting = true;
32 this.lastSubmitAttempt = Date.now();
33 const formData = new FormData();
34
35 formData.append(
36   'recipe',
37   JSON.stringify({
38     recipeName: this.recipeForm.value.title!,
39     category: this.recipeForm.value.category!,
40     difficulty: this.recipeForm.value.difficulty,
41     servings: this.recipeForm.value.servings,
42     prepareTime: this.recipeForm.value.prepareTime,
43     ingredients: this.ingredients,
44     steps: this.steps,
45     isPublic: this.recipeForm.value.isPublic,
46     language: this.recipeForm.value.language,
47   })
48 );
49
50 if (
51   this.fileInput.nativeElement.files &&
52   this.fileInput.nativeElement.files.length > 0
53 ) {
54   const file = this.fileInput.nativeElement.files[0];
55   formData.append('images', file);
56 }
57
58 this.recipeService.postRecipe(formData).subscribe({
59   next: () => {
60     this.notificationService.success('Recipe has been created', 5000);
61     this.resetForm();
62     this.isSubmitting = false;
63   },
64   error: () => {
65     this.notificationService.error('Failed to add the recipe!', 5000);
66     this.isSubmitting = false;
67   },
68 });
69 }

```

EditRecipeComponent EditRecipeComponent dzieli większość funkcjonalności z wcześniej opisanym komponentem dodawania przepisów. Wykorzystuje identyczny system dodawania obrazów, sposoby walidacji danych oraz zabezpieczenia przed wielokrotnym wysłaniem formularza poprzez strumień *updateAttempts\$*. Główną różnicą jest wykorzystanie metody *loadRecipeDetails()* podczas wstępnego ładowania komponentu. Pobiera ona dane na podstawie ID z parametrów routingu i mapuje je do formularza. Obsługa obrazów również różni się od poprzedniego komponentu. Komponent przechowuje URL obecnego obrazu w *currentImageUrl* oraz flagę *hasExistingImage*. Użytkownik może zachować istniejący obraz lub usunąć go i zastąpić nowym. W metodzie *executeUpdate()*, jeśli nie wybrano nowego obrazu, system wysła informację o zachowaniu obecnego obrazu poprzez *keepExistingImage*: „true”. Po pomyślnej aktualizacji przepisu, użytkownik powiadamiany jest o sukcesie oraz odświeżany jest widok z nowymi danymi, jest to istotne zwłaszcza dla obrazu.

4.3.4 System filtrowania

RecipeFiltersComponent to uniwersalny komponent służący do filtrowania przepisów z dwoma trybami pracy. Przy użyciu flagi *useCustomData* określa czy korzysta z serwisu filtrów (*FilterService*), czy przyjmuje dane poprzez *@Input*. Komponent używa system właściwości *@Input* (*customCategories*, *customLanguages*, *customIngredients*, *showTabs*, *showIngredients*) oraz emituje zdarzenia poprzez *@Output* (*filtersChanged*, *tabChanged*). Pozwala to na wykorzystanie go na stronie głównej oraz w profilach użytkowników.

```
1 if (this.useCustomData) {
2   // For custom usage, load custom categories and languages
3   this.categoriesArray = this.customCategories.map((cat) => ({
4     key: cat.toUpperCase(),
5     value: cat,
6   }));
7   this.languagesArray = this.customLanguages;
8   this.ingredientsArray = this.customIngredients;
9   this.loadCurrentFiltersFromCustom();
10 } else {
11   this.loadLanguages();
12   this.loadIngredients();
13   this.loadCurrentFilters();
14 }
```

Wszystkie zmiany filtrów przechodzą przez strumień *filterChange\$* z operatorem *debounceTime(300)*. Metoda *executeFilterUpdate()* tworzy obiekt *FilterState* z wszystkimi aktywnymi filtrami i przekazuje go do serwisu lub emituje zdarzenie.

```
1 private executeFilterUpdate(): void {
2   if (this.useCustomData) {
3     // For custom usage, just emit the change
4     this.filtersChanged.emit();
5   } else {
6     // For FilterService usage
```

```

7      const filters: Partial<FilterState> = {
8          query: this.searchQuery,
9          prepTimeFrom: this.prepTimeFrom,
10         prepTimeTo: this.prepTimeTo,
11         servingsFrom: this.servingsFrom,
12         servingsTo: this.servingsTo,
13         category: this.selectedCategory || undefined,
14         difficulty:
15             this.selectedDifficulty === '' ?
16             undefined : +this.selectedDifficulty,
17         recipeLanguage: this.selectedLanguage || undefined,
18         ingredients:
19             this.selectedIngredients.length > 0
20             ? this.selectedIngredients
21             : undefined,
22         currentPage: 1,
23     };
24     this.filterService.updateFilters(filters);
25     this.filtersChanged.emit();
26 }
27 }

```

Komponent obsługuje osiem typów filtrów: wyszukiwanie tekstowe, czas przygotowania, liczba porcji, kategoria, trudność, język oraz składniki. Lista składników jest automatycznie aktualizowana po zmianie języka przy użyciu metody `onLanguageChange()`. Wejściowo wyświetlane są wszystkie dostępne składniki.

4.3.5 Profil użytkownika

System zarządzania profilem użytkownika składa się z czterech komponentów: `ProfileComponent` (główny komponent), `EditProfileComponent` (edycja profilu), `SettingsProfileComponent` (ustawienia konta) oraz `DeleteProfileComponent` (usuwanie konta). Ich komunikacja odbywa się poprzez `ProfileService`, który zarządza ich stanem i synchronizuje dane między nimi.

ProfileComponent Jest to obszerny komponent obsługujący profil użytkownika, jego przepisy oraz ulubione. Implementuje lokalne filtrowanie danych bez wysyłania zapytań do serwera. Komponent zarządza trzema głównymi obszarami danych: informacjami o użytkowniku (`username`, `userPhotoUrl`, `backgroundImageUrl`), przepisami użytkownika (`userRecipes`) oraz ulubionymi (`favouriteRecipes`). Podczas inicjalizacji profilu, wszystkie obszary są ładowane jednocześnie.

```

1  private initializeData(): void {
2      this.loadUserData();
3      this.loadUserRecipes();
4      this.loadUserFavourites();
5  }

```

Interfejs podzielony jest na dwie zakładki: własne przepisy oraz ulubione. Przełączanie między nimi odbywa się przy użyciu metody `setActiveTab()`, która auto-

matycznie przeładowuje opcje filtrowania i resetuje paginację. Dodatkowo metoda dynamicznie generuje filtry na podstawie aktualnie wyświetlanych danych.

```
1 private loadCategories(): void {
2     const source = this.getActiveRecipeSource();
3     if (source.length > 0) {
4         const uniqueCategories = Array.from(
5             new Set(source.map((recipe) => recipe.category))
6         ).sort();
7         this.categoriesArray = uniqueCategories;
8     } else {
9         this.categoriesArray = [];
10    }
11 }
```

Komponent wykonuje filtrowanie w pamięci przeglądarki. Metoda `applyAllFilters()` sekwencyjnie aplikuje wszystkie aktywne filtry. Filtrowanie po składnikach wykorzystuje logikę „wszystkie z wybranych”.

```
1 if (this.filters.ingredients && this.filters.ingredients.length > 0) {
2     filtered = filtered.filter((recipe) =>
3         this.filters.ingredients!.every((filterIngredient) =>
4             recipe.ingredients?.some((recipeIngredient) =>
5                 recipeIngredient.ingredientName
6                     .toLowerCase()
7                     .includes(filterIngredient.toLowerCase())
8             )
9         )
10    );
11 }
```

Do komunikacji z `RecipeFiltersComponent` wykorzystany został `@ViewChild`. Komponent filtrów działa wtedy w trybie niestandardowym (`useCustomData = true`) i przyjmuje dynamicznie generowane listy kategorii, języków i składników. Zmieniając język, lista składników jest automatycznie przeładowywana.

Implementacja paginacji opiera się na metodzie `getPaginatedItems()`, która wyświetla odpowiednią ilość elementów tablicy na podstawie `currentPage` i `pageSize`. Całkowita liczba stron jest obliczana przy każdej zmianie filtrów.

EditProfileComponent Komponent umożliwia zmianę nazwy użytkownika oraz opisu profilu.

Implementuje niezależne strumienie aktualizacji dla wymienionych pól. Pozwala to na zapisywanie zmian bez konieczności edycji wszystkich danych jednocześnie. Każde pole posiada strumień `Subject` z operatorem `debounceTime(800)`. Dodatkowo zastosowano zabezpieczenie zapobiegające zbyt częstym żądaniom poprzez `minTimeBetweenUpdates`.

Komponent przechowuje oryginalne wartości (`originalUsername`, `originalDescription`) i przed wysłaniem porównuje je z nowo wprowadzonymi danymi. Jeśli wartości są identyczne, użytkownikowi wyświetli się komunikat bez wykonania zbędnego zapytania do serwera.

```

1 if (newUsername === this.originalUsername) {
2   this.notificationService.info('Username is already up to date', 5000);
3   return;
4 }

```

Flagi `isUpdatingUsername` i `isUpdatingDescription` blokują możliwość ponownego wysłania podczas trwającej operacji.

Metoda `loadUserData()` jest wywoływana przy inicjalizacji oraz po każdej udanej aktualizacji. Zapewnia to synchronizację danych z `ProfileService`. Przycisk „Cancel” przywraca oryginalne wartości i zamyka tryb edycji. Metoda `save()` aktualizuje jednocześnie oba pola, delegując walidacje do odpowiednich metod.

Rysunek 12: EditProfileComponent

SettingsProfileComponent Komponent odpowiada za zmianę danych uwierzytelniających użytkownika, konkretnie adresu email oraz hasła. Wymaga potwierdzenia tożsamości poprzez podanie aktualnego hasła przy każdej operacji, stanowi to dodatkową warstwę bezpieczeństwa.

Zmiana hasła wymaga spełnienia kryteriów: długość od ośmiu do sześćdziesięciu czterech znaków, minimum jedna cyfra i wielka litera. System dodatkowo sprawdza, czy hasło różni się od obecnego. Obie funkcje walidują format wprowadzonych danych przy użyciu wyrażeń regularnych. Dodatkowo zmiana adresu email wymaga podania hasła jako potwierdzenia tożsamości.

```

1 if (!/.*\d.*/.test(newPassword)) {
2   this.notificationService.error(
3     'Password must contain at least one digit'
4   );
5   return;
6 }

```

```

7
8 if (!/.*[A-Z].*/.test(newPassword)) {
9   this.notificationService.error(
10     'Password must contain at least one uppercase letter'
11   );
12   return;
13 }

```

Podobnie jak `EditProfileComponent`, wykorzystuje strumienie z `debounceTime(1000)` oraz zabezpieczenia przed zbyt częstym wysyłaniem zapytań `minTimeBetweenUpdates`. Po udanej zmianie danych, użytkownik otrzymuje informacje o konieczności ponownego zalogowania. Metoda `handleApiErrors()` sprawdza kody odpowiedzi HTTP i wyświetla zrozumiałe komunikaty.

Rysunek 13: SettingsProfileComponent

DeleteProfileComponent Komponent implementuje mechanizm bezpiecznego usunięcia konta z dwustopniową weryfikacją. Nasłuchując zmian `showDeleteModal$` z `ProfileService`, komponent kontroluje wyświetlanie okna potwierdzenia.

Po wywołaniu `confirmDeleteUser()`, wykonywane jest żądanie do `ProfileService`. Jeśli żądanie zakończy się sukcesem, użytkownik zostanie automatycznie wylogowany przy użyciu `authService.logout()`, a okno zostaje zamknięte.

```

1 confirmDeleteUser(): void {
2   if (!this.userId) return;
3
4   this.profileService
5     .deleteUser(this.userId)
6     .pipe(takeUntil(this.destroy$))
7     .subscribe({
8       next: () => {
9         this.notificationService.success(
10           'Profile deleted successfully',

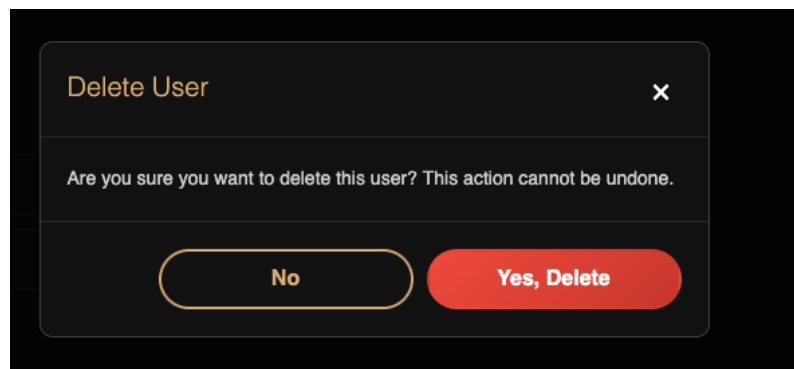
```

```

11         3000
12     );
13     this.notificationService.info('Refresh to update changes', 3000);
14     this.authService.logout();
15     this.cancelDeleteUser();
16 },
17 error: () => {
18     this.notificationService.error('Failed to delete user', 3000);
19     this.cancelDeleteUser();
20 },
21 });
22 }

```

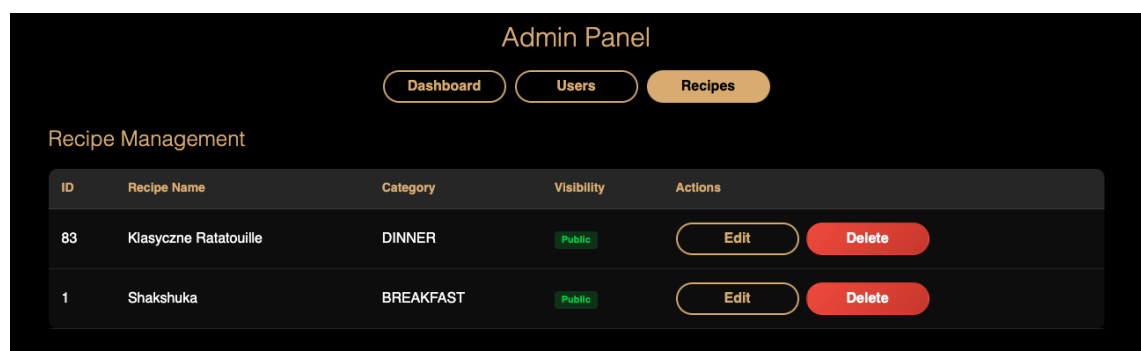
System zabezpiecza przed przypadkowym usunięciem. Użytkownik może anulować operację dzięki metodzie `cancelDeleteUser()`, która resetuje stan modalnego okna oraz flagę trybu usuwania w `ProfileService`.



Rysunek 14: DeleteProfileComponent

4.3.6 Panel administratora

`AdminPanelComponent` to centrum zarządzania systemem dostępne wyłącznie dla użytkowników posiadających rolę administratora. Komponent składa się z trzech głównych obszarów: panelu ze statystykami, zarządzania użytkownikami oraz zarządzania przepisami.



Rysunek 15: AdminPanelComponent

Zaimplementowano weryfikację uprawnień administratora podczas inicjalizacji poprzez `authService.isAdmin()`. W przypadku braku autoryzacji, użytkownik otrzymuje informację o odmowie dostępu. Interfejs podzielony jest na trzy zakładki kontrolowane przez metodę `setActiveTab()`, która dynamicznie ładuje odpowiednie dane.

```
1 setActiveTab(tab: 'dashboard' | 'users' | 'recipes'): void {
2   this.activeTab = tab;
3   switch (tab) {
4     case 'dashboard':
5       this.loadDashboardData();
6       break;
7     case 'users':
8       this.loadUsers();
9       break;
10    case 'recipes':
11      this.loadRecipes();
12      break;
13  }
14 }
```

Sekcja użytkowników umożliwia modyfikację ról, adresów email i nazw użytkowników poprzez formularz edycji. Metoda `saveUser()` wysyła zaktualizowane dane do `AdminService` i odświeża listę lokalnie bez przeładowywania. Usuwanie użytkowników wymaga dwustopniowej weryfikacji. Metoda `deleteUser()` otwiera okno potwierdzenia, a `confirmDeleteUser()` wykonuje operację.

Administrator ma możliwość usuwać i edytować wszystkie przepisy w systemie. Edycja przekierowuje do komponentu `EditRecipeComponent` przy użyciu `this.router.navigate(['/update-recipe', recipeId])`. Edycja przepisu także wymaga potwierdzenia i aktualizuje listę lokalnie po pomyślnej operacji.

Zakładka `dashboard` wyświetla aktualne statystyki systemu poprzez `getStatus()`. Metoda wyświetla liczbę użytkowników, przepisów oraz inne metryki. Wszystkie operacje asynchroniczne wykorzystują flagę `loading`, aby zasygnalizować stan ładowania.

4.4 Systemy wsparcia

Systemy wsparcia stanowią nierozłączną część infrastruktury aplikacji. W skład tej warstwy wchodzi serwis odpowiedzialny za powiadamianie użytkownika (`NotificationService`), zarządzanie stanem interfejsu (`ScrollLockService`) oraz synchronizację języka (`LanguageService`). Serwisy te zostały zaprojektowane z zasadą pojedynczej odpowiedzialności.

4.4.1 System powiadomień

`NotificationService` implementuje system powiadomień typu toast, umożliwiający komunikację z użytkownikiem z dowolnego miejsca aplikacji. System wykorzystuje wzorzec `Observable` z `BehaviorSubject` do przechowywania i wysyłania aktywnych powiadomień.

System obsługuje cztery typy komunikatów: *error*, *success*, *warning* oraz *info*, każdy z nich jest konfigurowalny. Struktura komunikatu zawiera unikalny identyfikator generowany metodą `generateId()`, typ, wiadomość oraz opcjonalny czas trwania (domyślnie pięć sekund).

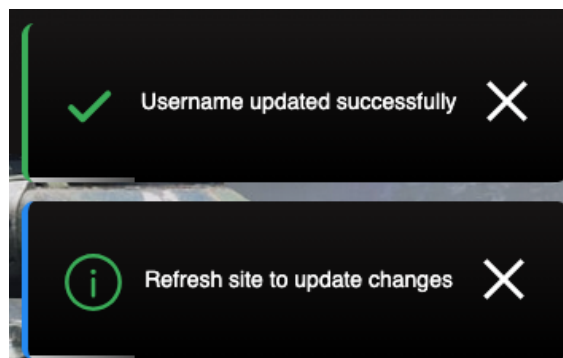
```
1 export interface Notification {
2   id: string;
3   type: 'success' | 'error' | 'warning' | 'info';
4   message: string;
5   duration?: number; // Duration in milliseconds, optional
6 }
```

Kluczową funkcjonalnością jest automatyczne usuwanie powiadomienia po określonym czasie. Metoda `AddNotification()` dodaje powiadomienie do strumienia i jednocześnie planuje jego usunięcia przy użyciu `setTimeout()`.

```
1 private AddNotification(notification: Notification) {
2   const currentNotifitcations = this.notificationSubject.value;
3   this.notificationSubject.next([...currentNotifitcations, notification]);
4
5   // Automatically remove the notification after its duration
6   setTimeout(() => {
7     this.removeNotification(notification.id);
8   }, notification.duration || 3000);
9 }
```

`BehaviorSubject` przechowuje aktywne komunikaty w tablicy, umożliwiając komponentom subskrypcję zmian poprzez `notifications$`. Metoda `remove()` i `clear()` pozwalają na ręczne usunięcie pojedynczych powiadomień lub wyczyszczenie całej listy.

`ToastNotificationComponent` wyświetla powiadomienia otrzymane z `NotificationService`.



Rysunek 16: Dwa typy powiadomień: `success` oraz `info`

Podczas inicjalizacji komponentu, subskrybowany jest strumień `notifications$` i przechowywana jest referencja do subskrypcji.

```

1 ngOnInit() {
2   this.subscription = this.notificationService.notifications$.subscribe(
3     (notification) => (this.notifications = notification)
4   );
5 }

```

Umożliwia to ręczne usunięcie powiadomienia metodą `removeNotification()`, która deleguje operacje do serwisu.

Wewnątrz szablonu wykorzystano dyrektywę `@switch` do dynamicznego wyboru ikony w zależności od typu powiadomienia.

```

1 @switch (notification.type) {
2   @case ('success') {
3     
5   }
6   @case ('error') {
7     
9   }
10  @case ('warning') {
11    
13  }
14  @case ('info') {
15    
17  }
18 }

```

Każdy typ komunikatu ma dedykowaną ikonę SVG, co zapewnia prostą identyfikację powiadomienia.

4.4.2 System zarządzania stanem interfejsu

`ScrollLockService` zarządza blokowaniem przewijania strony podczas wyświetlania okien modalnych lub mobilnych widoków menu. Serwis wykorzystuje `Renderer2` jako zalecany przez Angular sposób manipulacji DOM.

Serwis operuje poprzez dodawanie i usuwanie klasy CSS `overflow-h` do elementu `<body>`.

```

1 lockScroll() {
2   this.renderer.addClass(this.document.body, 'overflow-h');
3 }
4
5 unlockScroll() {
6   this.renderer.removeClass(this.document.body, 'overflow-h');
7 }

```

Klasa `overflow-h` zdefiniowana jest w globalnym CSS. Takie ustawienie powoduje zablokowanie możliwości przewijania.

```
1 .overflow-h {  
2   overflow: hidden !important;  
3   position: fixed !important;  
4   width: 100% !important;  
5   height: 100% !important;  
6 }
```

Serwis jest wstrzykiwany do komponentów wykorzystujących modalne interfejsy (menu filtrów, okna usuwania profilu, panele potwierdzenia), zapobiegając niepożądanemu przewijaniu podczas interakcji z nakładkami.

4.4.3 Zarządzanie językiem aplikacji

`LanguageService` odpowiada za globalną synchronizację języka interfejsu w całym systemie. Wykorzystuje `BehaviorSubject` do przechowywania aktualnego języka oraz `localStorage` do zachowania preferencji między sesjami.

Podczas inicjalizacji, serwis odczytuje zapisany język z `localStorage`, lub domyślnie ustawia język angielski ('en').

```
1 private getSavedLanguage(): string {  
2   return localStorage.getItem('language') || 'en';  
3 }
```

Metoda `setLanguage()` aktualizuje język w dwóch miejscach. Emituje nową wartość przez `BehaviorSubject` (powiadamiając wszystkich subskrybentów) oraz zapisuje wybór w `localStorage`. Komponenty mogą subskrybować zmiany poprzez strumień `language$` lub pobrać wartość przy użyciu metody `getLanguage()`, która zwraca wybrany język.

```
1 setLanguage(language: string): void {  
2   this.languageSubject.next(language);  
3   localStorage.setItem('language', language);  
4 }  
5  
6 getLanguage(): string {  
7   return this.languageSubject.value;  
8 }
```

Internacjonalizacja (i18n) Aplikacja wykorzystuje bibliotekę `@ngx-translate` do obsługi zmiany języka interfejsu. System opiera się na plikach JSON zawierających klucze translacji dla każdego języka. Struktura tych plików jest hierarchiczna i zawiera strukturę kluczy.

```
1 "REGISTER": {  
2   "CATCH_PHRASE": "Zarejestruj się i rozpocznij swoją podróż z Cibarią!",  
3   "USERNAME_PLACEHOLDER": "Nazwa użytkownika",
```

```

4   "PASSWORD_PLACEHOLDER": "Hasło",
5   "CONFIRM_PASSWORD_PLACEHOLDER": "Potwierdź hasło",
6   "SUBMIT_BUTTON": "Zarejestruj się"
7 },

```

W szablonach komponentów wykorzystywany jest *pipe* translate.

```

1 <label for="name">{{ "CTA.NAME" | translate }}</label>

```

Klucz CTA.NAME jest mapowany na odpowiednią wartość z aktywnego pliku (na przykład en.json, pl.json).

Komponenty importują TranslateModule i ustawiają domyślny język w konstruktorze.

```

1 constructor(private translate: TranslateService) {
2   this.translate.setDefaultLang('en');
3 }

```

Zmiana języka odbywa się przy użyciu metody `changeLanguage()`, wywoływanej w odpowiedzi na akcję użytkownika.

4.5 Podsumowanie

W niniejszym rozdziale przedstawiono implementację aplikacji frontendowej wykorzystującą Angular 18.2.13 z architekturą opartą na elementach *standalone*. Uprościło to strukturę projektu poprzez wyeliminowanie tradycyjnych modułów `NgModule`.

Kluczowym aspektem implementacji było reaktywne programowanie z biblioteką RxJS. Wzorce `Observable`, `BehaviorSubject` oraz operatory `debounceTime` czy `takeUntil` zapewniły efektywne zarządzanie strumieniami informacji. Optymalizowały komunikację z serwerem oraz zapobiegały wyciekowi pamięci.

Warstwa komunikacji z serwerem została zrealizowana przy użyciu dedykowanych serwisów zawierających logikę biznesową. System autoryzacji JWT implementuje automatyczne dołączanie tokenów oraz zarządzanie stanem uwierzytelnienia.

Zaimplementowano pełen zestaw komponentów interfejsu, gdzie każdy posiada walidację danych, obsługę błędów oraz mechanizmy powiadomień.

Systemy wsparcia działają w tle wspierając główną funkcjonalność aplikacji. System filtrowania został zrealizowany dwutorowo, przez `FilterService` dla strony głównej oraz lokalne filtrowanie dla profilu użytkownika. Zapewnia to responsywność przy dużych zbiorach danych.

Została zastosowana architektura oparta na separacji odpowiedzialności i wzorcach. Zapewnia modularność oraz łatwość przyszłej rozbudowy. Potencjalne kierunki rozwoju obejmują implementację *route guards*, *lazy loading*, rozszerzenie internacjonalizacji oraz dalszą optymalizację dla urządzeń mobilnych.

5 Weryfikacja, testowanie i konteneryzacja systemu

Celem procesu testowania było zapewnienie wytrzymałości systemu oraz potwierdzenie zgodności z założonymi wymaganiami. W aplikacji przyjęto strategię testowania obejmującą różne poziomy abstrakcji. Zaczynając od testów jednostkowych sprawdzających pojedyncze metody, przez testy integracyjne upewniające się, że komponenty ze sobą współgrają, aż po testy warstwy prezentacji oraz testy wydajnościowe. Niniejszy rozdział opisuje środowisko testowe, użyte narzędzia oraz przedstawia wyniki a także sposób konteneryzacji i orkiestracji środowiska wielokontenerowego.

5.1 Środowisko testowe i wykorzystane narzędzia

Proces testowania oparto na zewnętrznym środowisku testowym wykorzystującym konteneryzację oraz biblioteki dla każdej warstwy systemu. Strategia ta pozwoliła na przeprowadzenie testów jednostkowych, integracyjnych oraz wydajnościowych bez naruszania danych produkcyjnych.

W warstwie serwerowej fundamentem testów jest framework JUnit 5, współpracujący z biblioteką Mockito, umożliwiającą odizolowanie testowanej logiki poprzez symulacje zależności. W celu przyspieszenia wykonywania testów integracyjnych, zastąpiono bazę danych PostgreSQL na lekką bazę danych in-memory H2 Database. Cały proces budowy i testowania aplikacji backendowej został zautomatyzowany przy użyciu skryptu oraz platformy Docker, gwarantując powtarzalność wyników niezależnie od systemu operacyjnego hosta.

Weryfikacja warstwy prezentacji bazuje na frameworku Jasmine, służącym do definiowania specyficznych zachowań aplikacji. Jako środowisko rozruchowe (test runner) wykorzystano narzędzie Karma. Narzędzie to skonfigurowane jest do pracy z przeglądarką Chrome w trybie headless (bez interfejsu graficznego), dodatkowo testy zostały wykonane także w przeglądarce Firefox. Uzupełnieniem są testy wydajnościowe wykonane przy użyciu narzędzia k6, symulującego ruch użytkowników w celu weryfikacji czasu odpowiedzi API pod obciążeniem.

5.2 Konteneryzacja środowiska

W celu zapewnienia przenośności aplikacji oraz spójności środowiska rozruchowego (eliminacja problemu różnic konfiguracyjnych między systemami), proces wdrażania systemu oparto na platformie Docker. Konfigurację zdefiniowano osobno dla każdej warstwy aplikacji.

5.2.1 Budowa obrazów kontenerów

W celu optymalizacji rozmiaru obrazów oraz zwiększenia bezpieczeństwa, dla obu warstw (backend i frontend) zastosowano technikę wieloetapowego budowania (multi-stage build). Pozwala na oddzielenie środowiska kompilacji, zawierającego narzędzia deweloperskie od lekkiego środowiska produkcyjnego.

W przypadku backendu proces zdefiniowany w pliku Dockerfile przebiega w dwóch etapach:

1. **Etap budowania:** Wykorzystuje obraz `maven:3.9-eclipse-temurin-21-alpine` do skompilowania kodu źródłowego i zbudowania pliku `.jar`.
2. **Etap uruchomienia:** Gotowa aplikacja jest kopiowana do lekkiego obrazu `eclipse-temurin:21-jre-alpine`, zawierającego jedynie środowisko uruchomieniowe Java (JRE) co minimalizuje rozmiar kontenera.

Analogiczne podejście zastosowano dla aplikacji frontendowej:

1. **Etap budowania:** Obraz `Node:18-alpine` służy do instalacji zależności i przepisania kodu Angulara do statycznych plików HTML/JS/CSS.
2. **Etap serwowania:** Wynikowe pliki są przenoszone do wydajnego serwera `nginx:alpine`, który został skonfigurowany do obsługi aplikacji typu SPA (Single Page Application).

5.2.2 Orkiestracja usług

Do zarządzania wielokontenerową architekturą wykorzystano Docker Compose. Plik `docker-compose.yml` definiuje trzy współpracujące ze sobą usługi:

1. `cibaria_database`: Kontener bazy danych PostgreSQL (wersja 15-alpine), skonfigurowany ze zmiennymi środowiskowymi oraz wolumenem zapewniającym trwałość danych.
2. `cibaria_backend`: Usługa backendowa zależna od dostępności do bazy danych, komunikująca się z nią wewnątrz prywatnej sieci wirtualnej.
3. `cibaria_frontend`: Usługa frontendowa, mapująca zewnętrzny port 80 kontenera na port 4200 hosta, umożliwiając dostęp do aplikacji z poziomu przeglądarki.

```
1 services:
2   cibaria_database:
3     container_name: cibaria_database
4     image: postgres:15-alpine
5     ports:
6       - 5433:5432
7     environment:
8       POSTGRES_USER: ${POSTGRES_USER}
9       POSTGRES_PASSWORD: ${POSTGRES_PASSWORD}
10      POSTGRES_DB: ${POSTGRES_DB}
11     volumes:
12       - pgdata:/var/lib/postgresql/data
13     env_file:
14       - .env
15
16   cibaria_backend:
17     container_name: cibaria_backend
18     build: ./backend
19     ports:
```

```

20     - 8080:8080
21     environment:
22     - POSTGRES_USER=${POSTGRES_USER}
23     - POSTGRES_PASSWORD=${POSTGRES_PASSWORD}
24     - DB_URL=${DB_URL}
25     - CLOUD_NAME=${CLOUD_NAME}
26     - API_SECRET_KEY=${API_SECRET_KEY}
27     - API_KEY=${API_KEY}
28     - SECRET_KEY=${SECRET_KEY}
29     env_file:
30     - .env
31     depends_on:
32     - cibaria_database
33
34     cibaria_frontend:
35     container_name: cibaria_frontend
36     build:
37     context: ./frontend
38     dockerfile: Dockerfile
39     ports:
40     - 4200:80
41     depends_on:
42     - cibaria_backend
43
44     volumes:
45     pgdata: {}

```

Konfiguracja pozwala na uruchomienie całego środowiska jedną komendą, co znacząco usprawnia procesy testowe.

5.3 Testy warstwy serwerowej

Weryfikacja logiki biznesowej aplikacji serwerowej została podzielona na dwa etapy, co pozwoliło na szybkie wykrywanie błędów na poziomie szczególnych klas oraz weryfikację poprawności procesów.

5.3.1 Testy jednostkowe

Testy jednostkowe skupiają się na izolowanej weryfikacji metod bez uruchamiania pełnego kontekstu aplikacji. W tym celu wykorzystano bibliotekę Mockito.

Przykładem takiego podejścia są testy klasy `RecipeServiceImpl`. Weryfikacja metody `testSaveRecipeWithPhotos_Success` sprawdza, czy system poprawnie reaguje na udaną próbę dodania przepisu ze zdjęciem. Zamiast łączyć się z rzeczywistą bazą danych, repozytoria `UserRepository`, `RecipeRepository` oraz serwisy `JwtService`, `ImageService` są „zaślepiane” (mockowane), aby zwrócić poprawne wartości dla metod z wcześniej wymienionych serwisów oraz repozytoriów. Dzięki temu test wykonuje się bardzo szybko, weryfikując, że serwisy nie rzucają wyjątku i tworzą przepis.

```

1  @Test

```

```

2 void testSaveRecipeWithPhotos_Success() throws IOException {
3     MultipartFile mockFile = mock(MultipartFile.class);
4     List<MultipartFile> images = List.of(mockFile);
5     Image mockImage = new Image();
6
7     when(jwtService.extractId("testtoken123")).thenReturn(1);
8     when(userRepository.findById(1)).thenReturn(Optional.of(testUser));
9     when(imageService.createPhoto(mockFile, ImageType.RECIPE))
10         .thenReturn(mockImage);
11     when(recipeRepository.save(any(Recipe.class))).thenReturn(testRecipe);
12
13     Recipe result = recipeService
14         .saveRecipeWithPhotos(testRecipeAddDto, images, testToken);
15
16     assertNotNull(result);
17     verify(imageService).createPhoto(mockFile, ImageType.RECIPE);
18     verify(recipeRepository).save(any(Recipe.class));
19 }

```

Analogicznie przetestowano pozostałe klasy w tym oraz innych serwisach, gdzie symulowano odpowiedzi z API. W `ImageService` symulowano odpowiedzi z zewnętrznego API Cloudinary, co pozwoliło na przetestowanie logiki bez konieczności wysyłania plików.

5.3.2 Testy integracyjne

Testy integracyjne weryfikują współpracę wszystkich warstw (kontroler, serwis, baza danych, repozytorium) aplikacji w ramach pełnego kontekstu Spring Boot. W klasie `AdminControllerIntegrationTest` wykorzystano adnotację `@SpringBootTest` wraz z `@AutoConfigureMockMvc`, co pozwala na wysyłanie żądań do wirtualnego serwera aplikacji bez konieczności uruchamiania go na porcie sieciowym.

Konfiguracja testowa zdefiniowana przy użyciu `@TestPropertySource`, wymusza użycie lekkiej bazy danych H2. Kluczowym elementem testów jest metoda przygotowująca setup poprzedzona adnotacją `@BeforeEach`, która przed każdym testem zasila bazę danych testowymi informacjami oraz generuje tokeny JWT.

```

1 @BeforeEach
2 void setup() {
3     // Create test admin user with proper role
4     adminUser = new UserEntity();
5     adminUser.setUsername("testAdmin");
6     adminUser.setEmail("admin@integration.test");
7     adminUser.setPassword("hashedPassword");
8     adminUser.setRole("ADMIN");
9     adminUser = userRepository.save(adminUser);
10
11     // Create test regular user
12     regularUser = new UserEntity();
13     regularUser.setUsername("testUser");
14     regularUser.setEmail("user@integration.test");

```

```

15     regularUser.setPassword("hashedPassword");
16     regularUser.setRole("USER");
17     regularUser = userRepository.save(regularUser);
18     userRepository.flush();
19
20     // Create test recipe owned by regular user
21     testRecipe = new Recipe();
22     testRecipe.setRecipeName("Test Recipe");
23     testRecipe.setDifficulty(2);
24     testRecipe.setIsPublic(true);
25     testRecipe.setLanguage("en");
26     testRecipe.setUser(regularUser);
27     testRecipe = recipeRepository.save(testRecipe);
28     recipeRepository.flush();
29
30     // Generate JWT tokens for testing using UserDetails
31     User adminUserDetails = new User(adminUser.getEmail(),
32         adminUser.getPassword(),
33         java.util.Collections.singletonList
34             (new SimpleGrantedAuthority("ROLE_ADMIN")));
35
36     User regularUserDetails = new User(regularUser.getEmail(),
37         regularUser.getPassword(),
38         java.util.Collections.singletonList(new SimpleGrantedAuthority
39             ("ROLE_USER")));
40
41     adminToken = "Bearer " + jwtService.generateToken(adminUserDetails);
42     userToken = "Bearer " + jwtService.generateToken(regularUserDetails);
43 }

```

Testy sprawdzają poprawność logiki biznesowej oraz spójność danych. Przykładowo test `getStats_ShouldReturnRealDatabaseCounts_WhenAdmin` weryfikuje, czy endpoint poprawnie łączy dane z bazy, zwracając oczekiwaną liczbę przepisów oraz użytkowników w formacie JSON.

```

1  @Test
2  void getStats_ShouldReturnRealDatabaseCounts_WhenAdmin() throws Exception {
3      mockMvc.perform(get("/admin/stats")
4          .header("Authorization", adminToken))
5          .andExpect(status().isOk())
6          .andExpect(jsonPath("$.totalUsers").value(2))
7          .andExpect(jsonPath("$.adminUsers").value(1))
8          .andExpect(jsonPath("$.regularUsers").value(1))
9          .andExpect(jsonPath("$.totalRecipes").value(1))
10         .andExpect(jsonPath("$.publicRecipes").value(1))
11         .andExpect(jsonPath("$.privateRecipes").value(0));
12 }

```

Natomiast test `deleteUser_ShouldRemoveUserAndRecipes_WhenAdmin` potwierdza działanie kaskadowego usuwania danych. Po wywołaniu endpointu DELETE

sprawdza bezpośrednio w repozytorium czy encja użytkownika oraz powiązane z nią przepisy zostały usunięte.

```
1 @Test
2 void deleteUser_ShouldRemoveUserAndRecipes_WhenAdmin() throws Exception {
3     recipeRepository.deleteAll(recipeRepository.findByUser(regularUser));
4     recipeRepository.flush();
5
6     mockMvc.perform(delete("/admin/users/" + regularUser.getId())
7         .header("Authorization", adminToken))
8         .andExpect(status().isOk());
9
10    assertFalse(userRepository.existsById(regularUser.getId()));
11    assertEquals(0, recipeRepository.findByUser(regularUser).size());
12 }
```

Dodatkowo w celu zapewnienia izolacji testów zastosowano adnotację `@Transactional`. Gwarantuje, że wszelkie operacje na bazie danych są wykonywane w ramach pojedynczego testu są wycofywane po jego zakończeniu. Zapobiega to wzajemnemu wpływaniu na siebie poszczególnych przypadków testowych.

5.4 Testy warstwy klienckiej

Testy warstwy frontendowej skupiają się na weryfikacji logiki komponentów. Wykorzystano wbudowane narzędzie Angulara `TestBed` do konfiguracji modułów testowych oraz `fakeAsync` aby obsługiwać metody asynchroniczne.

5.4.1 Weryfikacja logiki i formularzy

Testy komponentów zawierających formularz, takich jak `LoginComponent`, sprawdzają poprawność wprowadzanych danych oraz reakcje systemu na działania użytkownika. W środowisku testowym użyto zależności takich jak `AuthService` czy `Router`, które zostały zastąpione przez obiekty typu `Spy` z narzędzia `Jasmine` (`jasmine.createSpyObj`). Pozwala to na izolację komponentu od warstwy sieciowej oraz nawigacyjnej.

Testy obejmują sprawdzenie mechanizmu ograniczającego częstotliwość prób logowania (*rate limiting*). Wpływając na czas systemowy przy użyciu `spyOn(Date, 'now')`, test potwierdza, że zbyt częste wywołanie metody `onLogin` zakończy się wyświetleniem ostrzeżenia i nie wysłaniem żądania do serwera.

```
1 it('should prevent rapid login attempts with rate limiting', () => {
2     // needed because rapid firing onLogin is ~2ms
3     spyOn(Date, 'now').and.returnValue(0, 1500);
4     component.formPassword = 'abc123';
5     component.formEmail = 'abc123@gmail.com';
6
7     component.onLogin();
8     component.onLogin();
9
10    expect(notificationServiceMock.warning).toHaveBeenCalledWith(
```

```

11         'Please wait before trying again'
12     );
13     // 2 beacuse executeLogin & onLogin
14     expect(notificationServiceMock.warning).toHaveBeenCalledTimes(2);
15 });

```

Dodatkowo mechanizm `fakeAsync` i funkcja `tick()`, pozwoliły na przetestowanie asynchronicznego procesu logowania. Upewnialiśmy, że po pozytywnej odpowiedzi serwera, użytkownik zostanie przekierowany, a w przypadku błędu wyświetli się komunikat.

```

1  it('should handle successful login and navigate to profile',
2      fakeAsync(() => {
3      authServiceMock.login.and.returnValue(of({ success: true }));
4      component.formPassword = 'abc123';
5      component.formEmail = 'abc123@gmail.com';
6
7      component.onLogin();
8      tick(600);
9
10     expect(routerMock.navigate).toHaveBeenCalled();
11     expect(component.isLoginLoading).toBe(false);
12 }));
13
14 it('should handle login failure and show error message',
15     fakeAsync(() => {
16     routerMock.navigate.calls.reset();
17     authServiceMock.login.and.returnValue(throwError('Login failed'));
18     component.formPassword = 'abc123';
19     component.formEmail = 'abc123@gmail.co';
20
21     component.onLogin();
22     tick(600);
23
24     expect(notificationServiceMock.error).toHaveBeenCalledWith(
25         'Email or password is incorrect'
26     );
27     expect(routerMock.navigate).not.toHaveBeenCalled();
28     expect(component.isLoginLoading).toBe(false);
29 }));

```

5.4.2 Testy warstwy prezentacji

W przypadku komponentów wizualnych, takich jak `RecipeCardComponent`, kluczowe było przetestowanie logiki transformacji surowych danych na formę prezentowaną użytkownikowi. Wewnątrz procedury `beforeEach` przygotowywano zestaw obiektów imitujących konkretne przepisy (`mockRecipe`, `mockProfileRecipe`, `mockProfileRecipeWithoutImage` oraz `mockProfileRecipeWithImage`). Obiekty te reprezentują różne warianty modelu danych, co pozwoliło na pokrycie wszystkich możliwości bez konieczności łączenia się z backendem.

Testy metody `getRecipeImage` weryfikują elastyczność komponentu w obsłudze danych przychodzących z API. Potwierdzają również, że system posiada dodatkowe zabezpieczenia w przypadku braku grafiki.

```
1 it('should return Recipe image when recipe has images', () => {
2   component.recipe = mockRecipe;
3   const imageUrl = component.getRecipeImage();
4   expect(imageUrl).toBe('https://images.com/test-image.jpg');
5 });
```

Szczególną uwagę poświęcono metodom pomocniczym, odpowiedzialnym za przekształcanie danych numerycznych. Testy metody `getAverageRating` sprawdzają poprawność obliczania średniej arytmetycznej oraz formatowania do jednego miejsca po przecinku. Dodatkowo zweryfikowano także odporność na błędy, upewniając się, że dla pustych wartości lub tablic, zostanie zwrócona wartość „0.0”. Analogicznie przetestowano metodę `getDifficulty`, odpowiedzialną za mapowanie wartości (1-3) liczbowych na tekst („Easy”, „Medium”, „Hard”) oraz obsługę wartości spoza zakresu jako „Unknown”.

W przypadku komponentów niezawierających logiki, pełniących rolę wyłącznie reprezentacyjną, przetestowano, czy komponent zostaje utworzony.

```
1 it('should create', () => {
2   expect(component).toBeTruthy();
3 });
```

5.5 Testy wydajnościowe

W celu weryfikacji zachowania aplikacji pod obciążeniem oraz wykrycia wąskich gardeł, przygotowano testy w oparciu o narzędzie k6. Skrypt `basic-test.js` symuluje ruch generowany przez wielu użytkowników jednocześnie. Każdy z użytkowników wykonuje złożone operacje w systemie.

Zamiast stałej liczby użytkowników zastosowano model zmiennego ruchu zdefiniowanego w obiekcie `option.stages`. Pozwala to na sprawdzenie aplikacji w wielu fazach.

```
1 stages: [
2   { duration: "30s", target: 20 },
3   { duration: "1m", target: 50 },
4   { duration: "2m", target: 40 },
5   { duration: "30s", target: 90 },
6 ],
```

Pierwsza część testu trwa trzydzieści sekund i posiada dwudziestu wirtualnych użytkowników wykonujących operacje. Następnie przez jedną minutę wykonywane jest obciążenie w postaci pięćdziesięciu wirtualnych użytkowników przez jedną minutę. W kolejnej fazie testowana jest stabilizacja, gdzie przez dwie minuty czterdziestu użytkowników wykonuje operacje. Ostatecznie następuje weryfikacja systemu pod największym obciążeniem, ilość użytkowników nagle skacze do dziewięćdziesięciu. Etap trwa przez trzydzieści sekund.

Taka konfiguracja pozwala na bezpieczny rozruch puli połączeń do bazy danych oraz pamięci podręcznej aplikacji.

5.5.1 Scenariusz testowy

Logika testu została zaprojektowana tak, aby odwzorować pełen cykl życia użytkownika w systemie. Główna metoda realizuje następujące kroki:

1. **Rejestracja oraz autentykacja:** Generuje unikalne dane, rejestruje użytkownika oraz loguje go w celu uzyskania tokenu JWT.
2. **Operacja publiczne:** Weryfikuje wydajność publicznych endpointów, w tym pobieranie listy przepisów, filtrowanie oraz wyszukiwanie po nazwie przepisu. Zastosowano tutaj asercje (*check*) sprawdzające, czy czas odpowiedzi serwera nie przekracza tysiąca milisekund (1000ms).
3. **Zarządzanie treścią:** Symuluje dodawanie, edycje oraz usunięcie przepisu. Ze względu na specyfikację backendu wymagającą Multipart/form-data (dla obsługi zdjęć), w skrypcie zaimplementowano metodę pomocniczą konstruującą nagłówki i ciało żądania.

```
1 function createMultipartRequest(data, token) {
2   const boundary = "----WebKitFormBoundary" +
3     Math.random().toString(36).substr(2);
4   const body =
5     `-----${boundary}\r\n` +
6     `Content-Disposition: form-data; name="recipe"\r\n\r\n` +
7     `${JSON.stringify(data)}\r\n` +
8     `-----${boundary}--\r\n`;
9
10  return {
11    body: body,
12    headers: {
13      Authorization: `Bearer ${token}`,
14      "Content-Type": `multipart/form-data; boundary=----${boundary}`,
15    },
16  };
17 }
```

4. **Interakcje profilu:** Testuje dodawanie przepisu do ulubionych, ocenianie oraz aktualizacje danych profilu (zmiana emaila oraz hasła).
5. **Czyszczenie:** Na końcu każdego cyklu testowy użytkownik usuwany jest z bazy danych, zapobiegając nadmiernemu przyrostowi danych testowych.

W celu odwzorowania rzeczywistego zachowania użytkowników, pomiędzy operacjami zastosowano metodę `sleep()`, wprowadzającą losowe opóźnienie. Zapobiega to nienaturalnemu zalewaniu serwera żądaniami w falach.

5.5.2 Sposób uruchomienia

Testy wydajnościowe uruchamiane są przeciwko działającej instancji aplikacji. To znaczy, że najpierw należy uruchomić aplikację przy użyciu docker'a

(`docker compose up -d`). Następnie, aby uruchomić testy wydajnościowe należy mieć zainstalowane narzędzie `k6` i uruchomić testy komendą (`k6 run performance-tests/basic-test.js`).

Wynikiem działania testu jest raport wyświetlany w konsoli deweloperskiej, zawierający kluczowe metryki.

```
■ TOTAL RESULTS

checks_total.....: 7875    29.247698/s
checks_succeeded...: 100.00% 7875 out of 7875
checks_failed.....: 0.00%   0 out of 7875

✓ GET /recipes = status 200
✓ GET /recipes = has content
✓ GET /recipes = response time < 1000ms
✓ GET /recipes/1 = status 200 or 404
✓ GET /recipes with filters = status 200
✓ Search recipes = status 200
✓ GET /users/aboutme = status 200
✓ POST /recipes/ = status 200
✓ PUT /recipes/{id} = status 200
✓ GET /users/recipes = status 200
✓ POST /recipes/favourites/add = status 200
✓ GET /recipes/favourites/isFavourite = status 200
✓ GET /users/favourites = status 200
✓ POST /recipes/favourites/delete = status 200
✓ PUT /users/{id}/profile = status 200
✓ PUT /users/{id}/email = status 200
✓ PUT /users/{id}/password = status 200
✓ GET /recipes/{id}/rating = status 200
✓ POST /recipes/{id} = status 200
✓ GET /recipes/{id}/isOwner = status 200
✓ DELETE /recipes/{id} = status 200

HTTP
http_req_duration.....: avg=30ms    min=762µs    med=4.48ms    max=294.86ms    p(90)=82.94ms    p(95)=103.34ms
{ expected_response:true }...: avg=30ms    min=762µs    med=4.48ms    max=294.86ms    p(90)=82.94ms    p(95)=103.34ms
http_req_failed.....: 0.00%   0 out of 9376
http_reqs.....: 9376    34.822403/s

EXECUTION
iteration_duration.....: avg=29.78s min=29.71s med=29.77s max=30.17s    p(90)=29.83s    p(95)=29.86s
iterations.....: 375    1.392748/s
vus.....: 1    min=1    max=89
vus_max.....: 90    min=90    max=90

NETWORK
data_received.....: 10 MB    38 kB/s
data_sent.....: 3.9 MB    15 kB/s
```

Rysunek 17: Wyniki testu wydajnościowego

6 Podsumowanie i wnioski

Głównym celem niniejszej pracy inżynierskiej było zaprojektowanie i zaimplementowanie nowoczesnej aplikacji do zarządzania przepisami kulinarnymi. Realizacja projektu objęła stworzenie kompletnego rozwiązania *full-stack*, składającego się z warstwy serwerowej opartej na frameworku Spring Boot, warstwy klienckiej wykonanej w Angular oraz relacyjnej bazy danych PostgreSQL. Całość została osadzona w środowisku kontenerowym Docker, zapewniając aplikacji przenośność, łatwość skalowania i wdrażania aplikacji.

Przeprowadzone testy funkcjonalne oraz wydajnościowe potwierdziły, że aplikacja spełnia założone wymagania. System umożliwia bezpieczną rejestrację i autoryzację użytkowników przy użyciu standardu JWT, zarządzanie treścią (CRUD). Dodatkowo umożliwia interakcję z multimediami poprzez integrację z chmurą Cloudinary. Testy wydajnościowe wykazały, że architektura serwerowa zachowuje stabil-

ność, nawet przy nagłym wzroście liczby użytkowników. Świadczy to o poprawnym zarządzaniu zasobami i połączeniami do bazy danych.

6.1 Kierunki dalszego rozwoju aplikacji

Mimo osiągnięcia założonych celów, aplikacja posiada potencjał na dalszy rozwój. Do kluczowych obszarów, które mogłyby zostać w przyszłości rozbudowane należą:

1. **Zabezpieczenie tras po stronie klienta:** Mimo, że implementacja backendu poprawnie weryfikuje tokeny i blokuje nieuprawnione żądania, frontend opiera się na statycznej definicji tras wewnątrz pliku `app.routes.ts`. Przez to interfejs użytkownika pozwala na nawigację do chronionych widoków, nawet dla niezalogowanych użytkowników, co kończy się błędem dopiero błędem ładowania danych. Wprowadzenie mechanizmu Angular Route Guards pozwoliłoby na weryfikację uprawnień przed załadowaniem komponentu automatycznie przekierowując niezalogowanych użytkowników na stronę logowania.
2. **Rozbudowa funkcji społecznościowych:** Aktualnie system koncentruje się na relacji użytkownik-przepis. Naturalnym kierunkiem rozwoju jest wprowadzenie interakcji między użytkownikami.
 - **Publiczne profile:** Obecny kontroler `UserController` udostępnia głównie metody do zarządzania własnym kontem. Rozszerzenie API o endpointy publiczne (np. `GetUserByUsername` – wyszukiwanie użytkownika po jego nazwie) pozwoliłoby na przeglądanie innych użytkowników i ich kolekcji przepisów.
 - **System znajomości:** Implementacja relacji wiele-do-wielu między użytkownikami pozwoliłaby na stworzenie spersonalizowanej podstrony wyświetlającej przepisy dodane przez znajomych.
3. **Generator list zakupów:** Praktycznym usprawnieniem byłoby stworzenie automatycznej listy zakupów, która na podstawie wybranych przepisów dodawałaby składniki do listy. System mógłby sumować składniki z kilku przepisów i umożliwiać eksport listy do formatu PDF.

6.2 Przepis na carbonarę

Rzymski klasyk w najlepszym wydaniu. Kremowy sos na bazie żółtej i sera Pecorino, chrupiące guanciale oraz duża ilość czarnego pieprzu.

Składniki

- **Makaron:** 200g (tradycyjnie Spaghetti lub Rigatoni, ale każdy kształt się sprawdzi)
- **Mięso:** 75g Guanciale (włoskie podgardle) lub boczku wędzonego
- **Ser:** 100g sera Pecorino Romano (opcjonalnie Grana Padano lub Parmezan)
- **Jajka:** 3 żółtka
- **Przyprawy:** Świeżo mielony czarny pieprz

- **Sól:** Do gotowania makaronu

Przygotowanie

1. **Gotowanie makaronu:** W dużym garnku zagotuj wodę i mocno ją posól. Wrzuć makaron i gotuj go *al dente* (zgodnie z czasem na opakowaniu minus 1 minuta).
2. **Przygotowanie bazy mięsnej:** Guanciale (lub boczek) pokrój w drobną kostkę albo w paski, w zależności od preferencji. Wrzuć mięso na zimną, suchą patelnię i smaż na średnim ogniu, aż tłuszcz się wytopi, a kawałki staną się złociste i chrupiące.
3. **Masa jajeczna:** W międzyczasie zetrzyj ser na drobnych oczkach. W misce wymieszaj żółtka ze startym serem oraz dużą ilością świeżo mielonego pieprzu, aż powstanie gęsta pasta. Aby ją nieco rozluźnić, dodaj 1-2 łyżki gorącej wody z gotującego się makaronu.
4. **Hartowanie sosu:** Gdy mięso będzie gotowe, ostrożnie wlej wytopiony, gorący tłuszcz z patelni do miski z masą jajeczną, cały czas energicznie mieszając. Dzięki temu sos nabierze głębokiego smaku, a żółtka się zahartują.
5. **Łączenie smaków:** Ugotowany makaron odcedź (nie przelewaj zimną wodą!) i wrzuć bezpośrednio na patelnię, na której smażyło się mięso. Podsmaż go przez chwilę, aby zebrał resztki smaku z dna patelni.
6. **Finał:** Przełóż gorący makaron z patelni bezpośrednio do miski z przygotowaną wcześniej masą serowo-jajeczną. Mieszaj energicznie, aż ciepło makaronu sprawi, że sos zgęstnieje, stając się kremowy i aksamitny.
7. **Podanie:** Serwuj natychmiast, posypując dodatkową porcją pieprzu i sera.

Literatura

- [1] GeeksforGeeks, *Spring Dependency Injection with Example*, [online]. Dostępne pod adresem: <https://www.geeksforgeeks.org/advance-java/spring-dependency-injection-with-example/>
- [2] GeeksforGeeks, *Spring Boot Features for Java Development*, [online]. Dostępne pod adresem: <https://www.geeksforgeeks.org/blogs/spring-boot-features-for-java-development/>
- [3] Spring.io, *Spring Boot Project Overview*, [online]. Dostępne pod adresem: <https://spring.io/projects/spring-boot>
- [4] Project Lombok, *@Data Feature Documentation*, [online]. Dostępne pod adresem: <https://projectlombok.org/features/Data>
- [5] Spring.io, *Guides: Building an Application with Spring Boot*, [online]. Dostępne pod adresem: <https://spring.io/guides/gs/spring-boot>
- [6] Spring Framework Documentation, *Java Bean Validation*, [online]. Dostępne pod adresem: <https://docs.spring.io/spring-framework/reference/core/validation/beanvalidation.html>
- [7] Spring.io, *Spring Data JPA Overview*, [online]. Dostępne pod adresem: <https://spring.io/projects/spring-data-jpa#overview>
- [8] Angular.dev, *Angular v18 Overview*, [online]. Dostępne pod adresem: <https://v18.angular.dev/overview>
- [9] Angular.dev, *Testing Guide*, [online]. Dostępne pod adresem: <https://angular.dev/guide/testing>
- [10] Grafana k6, *k6 Documentation*, [online]. Dostępne pod adresem: <https://grafana.com/docs/k6/latest/>
- [11] JUnit 5, *JUnit 5 User Guide*, [online]. Dostępne pod adresem: <https://docs.junit.org/current/overview.html>
- [12] Docker Docs, *Docker Compose Overview*, [online]. Dostępne pod adresem: <https://docs.docker.com/compose/>