# Department of Informatics

# University of Leicester

# CO7201 Individual Project

**Final Report**

# Online Collaborative Recipe Book

**Brian Steve Pinto**

[bsp7@student.le.ac.uk](mailto:bsp7@student.le.ac.uk)

**229047308**

**Project Supervisor: Prof. Shigang Yue**

**Second Marker: Prof. Paul Holmes.**

Word Count: 10,007

08/09/2023

## DECLARATION

All sentences or passages quoted in this report, or computer code of any form whatsoever used and/or submitted at any stages, which are taken from other people's work have been specifically acknowledged by clear citation of the source, specifying author, work, date and page(s). Any part of my own written work, or software coding, which is substantially based upon other people's work, is duly accompanied by clear citation of the source, specifying author, work, date and page(s). I understand that failure to do this amounts to plagiarism and will be considered grounds for failure in this module and the degree examination as a whole.

Name: Brian Steve Pinto

Date: 08/09/2023

## Abstract

Imagine being far from home, craving for the dishes that bring comfort and happiness. The 'Online Collaborative Recipe Book' helps in bridging such a gap. This application allows individuals to explore the variety of recipe collections that are offered by the online global community. Individuals can add their own recipes showcasing their culinary expertise. This helps in building a community among people who have similar interests. This application provides a platform where everyone may discover a taste of their interest by encouraging culinary exploration, creativity, and cultural exchange. This includes a few unique features such as a shopping list feature, where the individual exploring a recipe finds that the ingredient is not available, this can be added to the shopping list. It has review and rating features that help individuals to rate the recipes, helping other users to discover popular recipes This application intends to be the go-to place for those looking for the comfort of familiar flavors, providing more than simply recipes.

This report provides detailed information on the application's requirements, implementation, background research and, explaining the research papers referred to gain knowledge on this application development.

## Table of Contents

# 1. Introduction

This report provides a walkthrough of the entire process of creating a web application that provides an online platform for individuals to exchange their unique and favorite recipes. It discusses all the steps taken and methods followed in researching, designing, developing, and evaluating the innovative culinary platform, offering insights into the challenges encountered, solutions devised, and the impact it aims to make on the culinary community.

## 1.1. Aim

The main aim of this project is to create a dynamic web application that provides a platform to bring together individuals who share a deep passion for cooking allowing them to share their signature recipes and showcase their special delicacies. This platform aims to guide people with zero to little knowledge of cooking to prepare delicious meals. It not only aims to facilitate the exchange of traditional and contemporary recipes but also encourages culinary experimentation, innovation, and cultural exchange.

The application will include functionalities like advanced filtering providing users with refined search results and leaving them satisfied by giving them exactly what they are looking for. It also will have special features like an integrated shopping list for planning their next recipe and a recipe progress tracker to give user a better experience while cooking.

## 1.2. Objective

To attain the ultimate goal of this project, certain objectives have to be fulfilled. The overall objectives of the web application are discussed in short below.

The application should enable users to register with it in order to maintain a profile to give them an identity and presence within the application. This becomes essential to interact with the various features that this app has to offer. The features like posting recipes and writing reviews will require a user to have an account in the app.

The ingredients data used in posting a recipe must be accurate. The database should be populated with real food ingredient data so that the recipes posted are legitimate. Accurate ingredient information will result in precise and truthful recipes thereby contributing to a better quality and enhanced user experience.

To achieve fine-searching functionality, the recipes posted must have various distinguishable traits. When posting a new recipe, a user must have the option to classify it into various categories. A recipe should be distinguishable by cuisines, courses, and diets. This will play a major role in the advanced filtering feature to give refined search results.

The application must have distinctive and helpful features that will draw users to choose this application over various other similar ones. An integrated shopping list is a useful feature that allows users to list down ingredients that they will need to purchase to cook a recipe.

## 1.3. Challenges

Various challenges are anticipated throughout the course of this project's development. These challenges may contribute to the complexities and become an obstacle and may impact the project's success and delivery.

- Finding a good dataset with ingredient information to be used in populating the database of the application is challenging.

- Designing the database is another challenging task for this project. The database needs to be designed efficiently to reduce redundancy and increase performance. The indexing of the database also should be done correctly to make the search functionality easier.

- File upload and hosting (Images of recipes) can also be a challenge. The file size and format should be taken into consideration as it might consume significant bandwidth to upload a large file and can pose a security threat to our application if someone were to upload a virus to our server. Storage should also be taken into consideration when managing and organizing the files.

## 1.4. Risks

It is crucial to assess and acknowledge various risks involved during the development phase. Some of the major risks are listed below:

- The developer is not experienced in developing frontend code using React framework. It is important to gain thorough knowledge and invest time in reading the documentation and articles. Learning and implementing side-by-side is challenge.
- Sticking to a tight time plan mentioned is hard to achieve because of the numerous features to be implemented by a single developer. It is crucial to follow the time plan mentioned and stick to it. Any delay in the delivery will be a liability to the project

completion and might need a re-evaluation of the implementation approach.

- There is always a risk of incompatibility between various dependencies and libraries that can lead to the malfunction of the project and may not provide desirable results. There might also be version conflicts between dependencies encountered during development. And there will always be difficulties while integrating third-party APIs when working with it for the first time.

- Building and using a complex backend API with a lot of endpoints is difficult to manage and use. To cope with this difficulty the author decided to use Swagger to document the backend API. Learning to write the documentation and specifications for this is time-consuming and hard due to the developer's inexperience.

- Writing complex database queries for various interactions is hard. It is difficult to write all the SQL queries manually for complex operations expecting no bugs. The database transactions are done by an ORM which is a new technology for the developer. Once again learning and using it is slightly harder when done hand-in-hand.

The choice of technologies and frameworks that are new to the developer requires learning prior to implementation which consumes more time than already planned. Deciding to use all new frameworks and integrating them comes with a risk of having bugs and code breakage.

# 2. Background Research

The available recipe-sharing platforms share similar features with this application. The search option in other apps has to offer limited filters to refine recipe search results. One of the vital filtering options necessary yet neglected by other applications is to exclude recipes containing certain food allergens. Food allergies need to be carefully taken into consideration while cooking any recipes. Another feature that sets this application apart from the rest is the integrated shopping list. Creating a shopping list is necessary to remind the user to buy ingredients that may be listed in the recipe that he/she is planning to cook.

Another useful feature that makes this project unique is the recipe progress tracker. This feature might come in handy while cooking together with a friend, or if a user has to ask another person to take over mid-cooking. The website will let the user know which step is he on. The user just has to click on the recipe steps to highlight the current step the user is on. Having a feature like this on the recipe application is useful if the user forgets the step that he should follow next.

The following provides a synopsis of reading lists and prior work facilitating a thorough understanding of technologies utilized in the development of the project.

i. Programming Language and HTTP server:
Although JavaScript is the number one trending programming language according to the 2023 survey results [2], the developer chose TypeScript which is a “Superset of JavaScript” [3] to catch bugs and errors early in the development phase. To build a robust HTTP server in TypeScript, Express, “a minimalist web frame” [4] is preferred to create a clean, modular and maintainable API. This framework is straight-forward in defining all the HTTP methods and sending appropriate responses with accurate status codes [5].

ii. Database and Prisma ORM:
“Databases are essential components for many modern applications and tools.” [6] Prisma is a growing ORM (Object Relational Mapping) tool that is type-safe and easily integrated with TypeScript and is used to interact with the database. The main advantage of using a tool like this is to skip writing complex database queries manually. Prisma in particular generates a client library tailored to the provided schema. This client has an optimized query building engine for efficient and performant database operations. This project uses the MySQL database to store the application data. However, if there is a need to change the database to a different one, Prisma offers “hassle-free migration” [7] to different databases.

iii. Swagger API Documentation:

To enhance the usability of the Application Programming Interface, Swagger/OpenAPI is used to document all the API endpoints. This tool provides a webpage containing all the information necessary to interact with the API. The documentation is written in YAML containing details like endpoint, method, JSON schema of the body, and possible responses. This specification is represented on a webpage on the API [8].

iv. React and Vite tool:

React is a popular front-end library preferred by a lot of developers. The documentation for this library is well written and covers most of the concepts that you need to know to use it in order to build a good website [9]. In order to provide wide support across various web browsers, the modern JavaScript code must be transpiled to an older version (CommonJS). This is done by a tool called module bundler. The developer has used Vite (bundling tool) to create a React project instead of the traditional Create React App tool. Create React App uses Babel as the transpiler (JSX compiler) and Webpack to bundle the application code. This process takes a lot of time in comparison to the newly introduced bundler Vite. Vite uses Hot Module Replacement (HMR) to replace only the code that has been altered rather than recompile the whole application, unlike Webpack [10].

v. Redux Toolkit:

The support for various plugins is what makes React a powerful front-end library. One of the plugins that has been used in this project is Redux Toolkit. Redux is used for state management. The main advantage of using Redux Toolkit is the Redux Toolkit Query (RTK Query) plugin which is helpful in data fetching. A custom dedicated API can be created for the client application to interact with the backend server API [11].

vi. Ingredients dataset:

To populate the database with ingredient information, various datasets were explored. The ingredient data that is used in this application to populate the database has been scraped from Nutritionix website [12]. This website provides nutritional data for each ingredient which might be helpful in the development of the application. Puppeteer, a web-scraping tool is used to extract all the ingredient data available on this site [13].

## 3. Requirements

In this section the requirements of the project have been discussed. The requirements can be classified into essential, recommended and optional requirements. Essential requirements outline the minimum requirements and core features that the project must possess for proper functioning of the app and to make sure the app serves its main purpose. Recommended features give a gist of requirements that are recommended for the smooth functioning of the system. And optional requirements are some of the additional features that are not essential but increase the value and user experience of the system.

### 3.1. Essential Requirements

- The Users of the application can create an account using their email. This feature enables the users to maintain an account. Users can log in, use the application and safely log out of the application.
- Logged-in Users can publish their recipes to the web application and manage their recipes. While uploading the recipe, the user will have to provide a title for their recipe, description, ingredients required and mention the steps to prepare the dish.
- Users of the app can search for recipes that they want to try out. Prepare the recipes using the steps mentioned. Only the logged-in users will be able to rate and leave a review. This will help the owner of the recipe as well as others who would like to try the recipe to know how many of them liked and disliked the recipe. Some reviews may be about suggesting changes in the recipe which will help the users improvise it.
- Users should categorize the recipe into cuisines (Italian, Chinese, French, Arabic), diets (Vegan, Pescetarian, Keto, Non-vegetarian, Gluten-free), and courses (appetizer, soup, lunch, dinner). This will help the users filter the recipes based on the user requirements while searching for a recipe.
- Logged-in users can save the recipes in their collections such as favorites. This enables the users to access their favorite recipes quickly.

### 3.2. Recommended Requirements

- The shopping list feature is available to the registered users which enables users to add ingredients to the list which they do not have but are required to cook the meal shown in the recipe. The shopping list can be accessed when the user goes out to shop for groceries.
- The following feature will let the users of the application follow each other which

establishes connections between different users. Logged-in users will be able to view the recipes of the users whom they are following in their feed on the go.

- While publishing a recipe, users can mention the time taken to cook the recipe. This will be helpful to filter out the recipes by the time taken to cook.
- After a user registers to the app a verification mail is sent to the registered email address asking to verify if the correct email was used while registering. The user should open the email and click on the verification link to activate the account. If a user tries to use someone else's email to register, there is also a link that says "Not me" to delete the account. This feature is essential to confirm users' identities.

## 3.3. Optional Requirements

- When the users publish their recipes to the platform the nutritional facts of the recipe like proteins, carbohydrates, amino acids, etc. so that the people can know the composition of their meals.
- Users can chat with each other whom they follow and share the recipes. Users can collaborate on recipes as well as exchange cooking tips.

# 4. Technical Specification

This section various technologies that have been used to develop the applications are discussed. Each component of the entire system uses different technologies. The tools and dependencies used in backend, frontend and the development environment are mentioned. This section also includes a table (Table 4.1: Technical Specifications) that contains a summary of all the tools and technologies used.

## 4.1. Backend Technology

The programming language used in the backend is TypeScript (v5.15) which is compiled to ES6 to run on NodeJS (v18.2) runtime. To implement the HTTP server in the backend Express (v4.18) library is used. The database choice of this application is MySQL (v8.0) to store the application data and is coupled with Prisma (v 4.16) to perform all the database operation which serves as an Object Relational Mapping (ORM) tool. To test and document the REST API, the backend server is integrated with Swagger (v 3.0) which generates a clean and organized documentation for all the routes. Additionally, Postman (v10.11) is used to perform tests.

## 4.2. Frontend Technology

The TypeScript (v5.15) language is used to develop the frontend for this project. ReactJs (v18.2) is the main library used to develop the user interface. To perform all the routing and navigation in the frontend this library is integrated with React Router (v6.14). Storing the images uploaded in the application is performed using a cloud service provided by Filestack. React Bootstrap (v2.8) is used as the main component library to provide responsiveness to the application components. Addition to this the custom components are built using Styled Components (v6.0). The tests are run using React Testing Library (v14.0) to test the frontend application.

## 4.3. Environment

The project is developed on an Apple MacBook air (apple M2 chipset) with MacOS (v13.5) operating system. Visual studio code (v1.7.9) is the text-editor/IDE (Integrated Development Environment) used to maintain the codebase of the project. The packages that are used in the application are managed by Yarn (v1.22.19) package manager. To view the state of the database MySQL Workbench (v8.0.33) is used.

*Table 4.1: Technical Specifications*

| Type | Name | Version |
|---|---|---|
| **Programming Languages** | TypeScript | 5.15 |
| | JavaScript | ES6 |
| | SQL | N/A |
| | YAML | 3 |
| **Data Management Tools** | MySQL | 8.0 |
| | Prisma | 4.16.2 |
| | MySQL Workbench | 8.0.33 |
| | Filestack | N/A |
| **Package Manager** | Yarn | 1.22.19 |
| **Operating system** | MacOS | 13.5 |
| **Libraries** | React | 18.2 |
| | Express | 4.18 |
| | Redux Toolkit | 1.9.5 |
| | React Bootstrap | 2.8 |
| **Testing** | Swagger | 3.0 |
| | Postman | 10.11 |
| | React TestingLibrary | 14.0 |
| **Other Tools and Technologies** | HTML | 5 |
| | CSS | 3 |
| | Firefox | 116.0.3 |
| | Visual Studio Code | 1.79 |

# 5. Methodology

This section discusses the software development lifecycle. It includes a brief explanation of the methodology used and mentions all the roles and terms of the methodology followed in brief. The milestones of the project is jotted down and put together in a table (Table 5.1: Project Milestones).

## 5.1. Development Approach

The software development approach followed in the development of this web application is Agile methodology. This methodology is implemented with Scrum framework which helps in project management and development of the software application. There are three roles in this framework which include Product Owner, Scrum Master and the development team.

The Product Owner is the project head who owns the project, the role Scrum Master is played by Prof. Shigang Yue, the author's project supervisor, who conducts meetings every two weeks to monitor the progress of the application and provides feedback, and the Development team is the author who is the developer and tester of this application. The Agile Artifacts consist of the Product backlog, which is a Trello dashboard in this application development, that consists of all the features that need to be implemented and the timeline allocated for each feature, sprint backlog which is the set of features that are taken from the Trello dashboard to implement during that sprint and increments which is the new improvements made in the application after each sprint. During each sprint set of features is implemented and presented to the Supervisor to receive feedback, and the application is modified accordingly to meet the project expectations. This cycle is carried out throughout the project development process for successful application development.

## 5.2. Milestones

The following table is a detailed time-plan that explains the tasks that was needed to be performed during the whole development phase. It contains the milestones that need to be met, the timeline planned to perform the tasks and the status of each milestone.

*Table 5.1: Project Milestones*

| Task | Timeline | Status |
|---|---|---|
| All the necessary requirements are gathered for the development of the software application. | 19 June-21 June | Completed |
| Background research is carried out to gain more knowledge on the application development. | 22 June-25 June | Completed |
| **Project Setup**<br><br>The repository is cloned to local machine to create a local repository.<br>Basic HTTP server is created with all the necessary data.<br>Front-end is setup to build the application. | 26 June- 2 July | Completed |
| **User Authentication Backend:**<br>Database is setup and connection are made to the server.<br>Register route is developed in the backend for user to register.<br>Token generation is done using JWT Token.<br>Login route is created for user to login to the account.<br>Swagger is integrated to document the API routes.<br>**Front-end:**<br>Registration page and Login page is built and connected to the server.<br>API routes and front-end UI are tested. | 3 July- 9 July | Completed |
| **Populate database with Ingredients.**<br><br>Scripts are written to scrape the data from online website resource.<br>Data that are scraped from the websites are scraped for obtaining ingredients data.<br>The ingredients data obtained from the websites are seeded into the database. | 10 July-16 July | Completed |
| **Recipe Feature**<br>Create Recipe Feature that performs CRUD operations to manage recipes within the application. | 17 July-31 July | Completed |

| | | |
|---|---|---|
| Recipe feature routes to add, view, edit and delete are created.<br>Recipe feature frontend is implemented for all CRUD operations.<br>Setup global state management and API to interact with the backend server using Redux Toolkit. | | |
| Review Feature<br>Create Review and Rating feature in the backend.<br>Create Review and Rating feature in the frontend.<br>Search functionality to search the ingredients. | 1 Aug-18 Aug | Completed |
| Favorites Feature.<br>Create function to save recipe to favorites. | 29 July - 18 Aug | Completed |
| Shopping list Feature<br>Create shopping list feature to add ingredients to shopping list and to view the list.<br>Recipe Progress Tracker to keep track of steps followed. | 7 Aug – 25 Aug | Completed |
| End-to-End Testing is performed which includes API testing, User-Interface testing, and Manual testing. | 14 Aug – 31 Aug | Completed |
| Report Writing | 26 June – 5 Sep | Completed |

# 6. Design and Architecture

This section offers a deeper understanding of the design and overall architecture of the web application.

## 6.1. System Architecture

The technology stack used in this application is MERN-TS. There is a change in the acronym, MERN typically stands for MongoDB Express React NodeJS. In this case, MySQL is used instead of MongoDB as the database. Express.js is the backend framework used to develop the REST API. NodeJS is the runtime used for this application along with TypeScript language to provide static types. Express.js is a popular, lightweight and flexible framework for creating backend APIs. React on the other hand is a powerful open-source UI framework that is component-based and has lots of useful plugins necessary to create a good website. Filestack is a cloud-based file-handling service that can easily be integrated with React. This is used to store and host recipe images in this case. When it comes to database choice, MySQL is an open-source, easy-to-use, reliable and efficient database. This when coupled with Prisma (ORM) is a good combination for a scalable application. It increases the efficiency of database transactions with the help of its query engine which highly optimizes the SQL query to ensure efficient and performant database interactions.

The architecture used for the server-side code is the Monolith architecture with the MRCS (Model-Route-Controller-Service) structure. This architecture (shown in Figure 6.1) is simple to develop, test and deploy. For the application of a smaller scale, it makes sense to use monolith architecture and not microservice architecture. This architecture is considered mainly due to its simplicity in the development process and deployment process. The code base uses the MRCS directory structure to make the code more modular and maintainable. A typical MRCS code structure has a models directory that contains schemas for database entities. Here since Prisma is used the schema definitions are in the prisma directory under schema.prisma file. The routes directory contains the definitions for the endpoints of the backend API. The controllers directory has the controllers which map to the API endpoints. The controllers perform the duty of handling the logic behind an HTTP request to their respective API routes. In other words, they handle the request parameters, request headers, request body and request query run the logic and send a response with appropriate status codes. The services directory has files for each database entity which contains functions for database operations. These functions return results of database operations or throw errors if any. This way the codebase is easy to maintain, and the code is modular and separated.

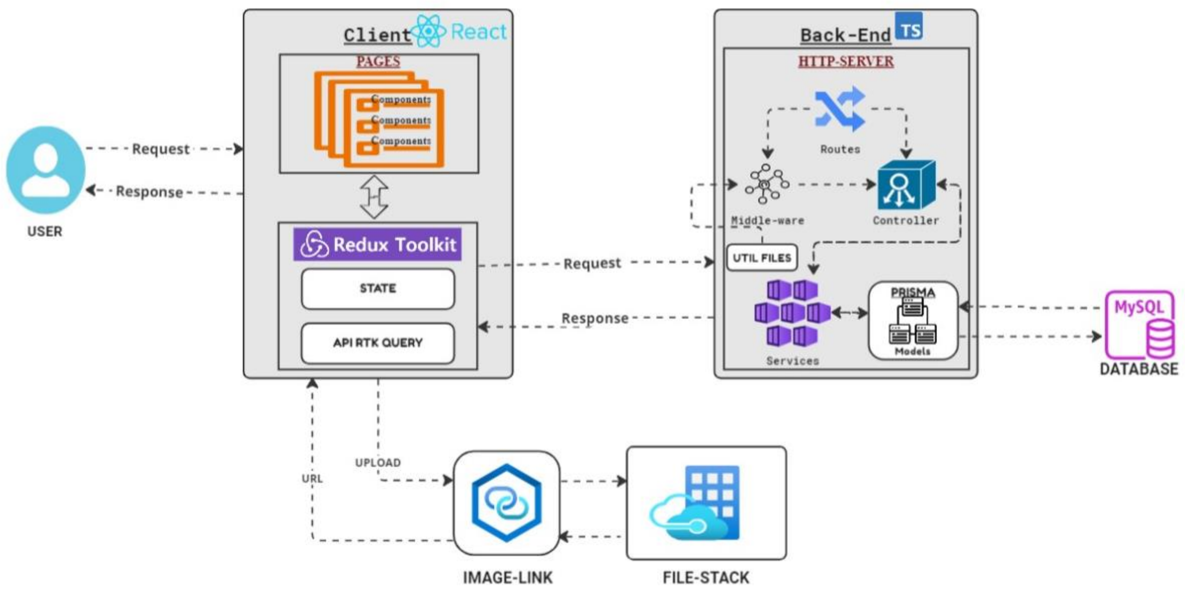


*Figure 6.1: Architecture Diagram*

This API is integrated with Swagger which is an open-source tool to document the API. The documentation for each endpoint is written above each route definition in a multi-line comment and is written in YAML. The schema definitions for swagger requests, responses, or any request body for an endpoint are defined in interfaces directory above each interface. The error responses are defined in the HTTPError.ts file. All the swagger documentation is written in YAML and broken down into chunks and written above respective modules for more clarity and maintainability.

The client-side code is divided into various components for reusability and modularity. React lets us write loosely coupled components with the help of an extension syntax for JavaScript called JSX (JavaScript as XML) to write both the logic and markup combined rather than writing them separately. The development server for the front-end is set up using vite which uses SWC (Speedy Web Compiler, open-source transpiler built using Rust) instead of Babel to transpile in development for faster development server startup, and uses HMR (Hot Module Replacement) which detects any changes in the development code and replaces only that part of the code in the dev server instead of transpiling the entire app for faster development experience. Building a custom component library is time-consuming, so the here React-Bootstrap is used to develop responsive components. The client app is also integrated with

Redux Toolkit for state management. Sending asynchronous API requests to the backend is made easier with the built-in tool in Redux Toolkit called RTK Query. Additionally, CSS is used for styling the components further.

## 6.2. Use case Diagram

CA use case diagram shows the basic interaction between the system and various actors. The use case diagram in Figure 6.2 outlines how the web application interacts with the use. All the components and the use cases are explained below.

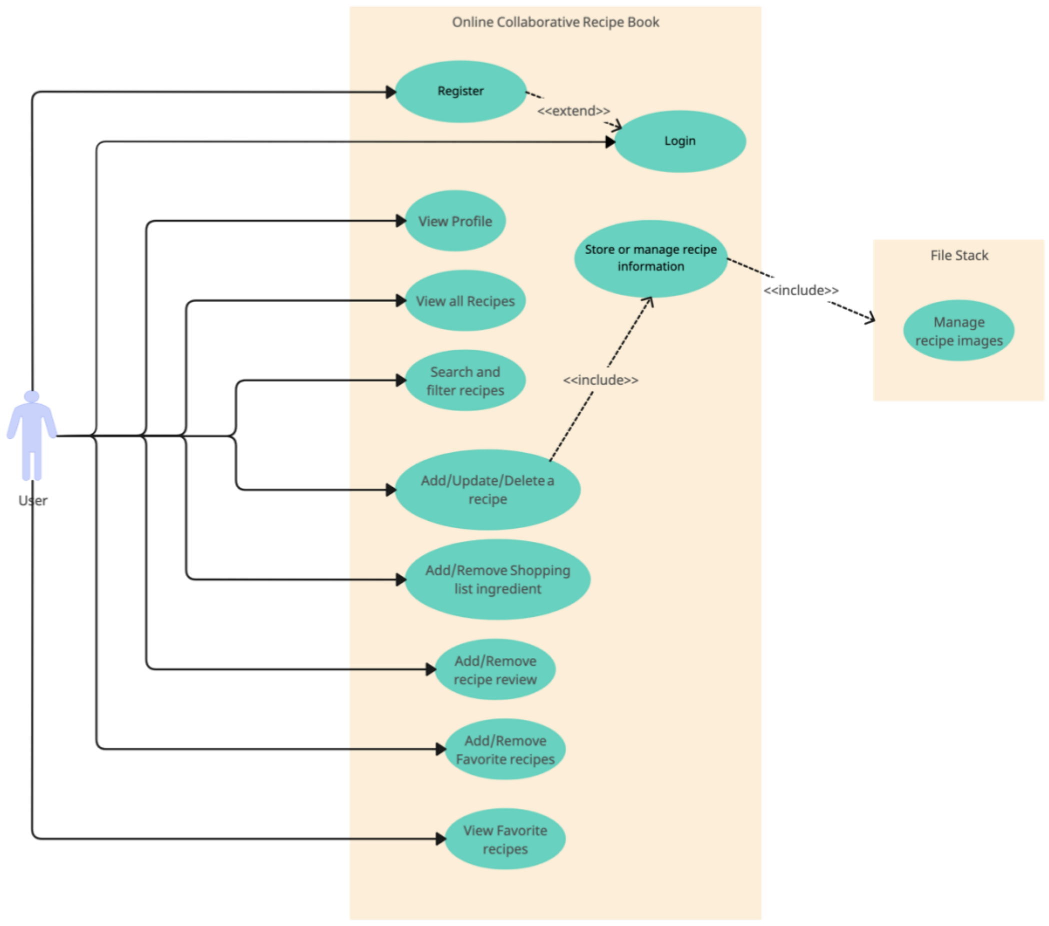

Online Collaborative Recipe Book
Register
<<extend>>
Login
View Profile
Store or manage recipe information
View all Recipes
<<include>>
File Stack
Manage recipe images
Search and filter recipes
<<include>>
User
Add/Update/Delete a recipe
Add/Remove Shopping list ingredient
Add/Remove recipe review
Add/Remove Favorite recipes
View Favorite recipes

*Figure 6.2: Use case diagram for the Online Collaborative Recipe Book*

**User:** The user of this application can search, filter and view the recipes without logging or registering into the app. To add, edit or delete user must have an account. To make a shopping list, add/remove recipes to/from favorites and to add/delete reviews the users must have an account.

**Register:** User must register to the application by providing first name, last name, email and password.

**Login:** User who are registered with the application can log in to the application by entering email and password used during registration.

**View Profile:** Upon logging to the system user can view the profile which contains uploaded recipes and link to his shopping list, link to view his favorite recipes.

**Add/Edit/Delete Recipe:** The logged-in user can add a recipe, edit or delete recipes belonging to him.

**Search and Filter Recipes:** The user can search for recipes and can also filter recipes even if the user doesn't have an account.

**Add/Remove Recipe Review:** Each individual recipe can be reviewed, and the user can provide feedback in addition to star ratings, and these ratings and comments will be stored in the MySQL database.

**Add/Remove Favorite Recipes:** The logged-in user can add or remove recipes to/from favorites.

**Add/Remove Shopping list Ingredients:** The logged-in user can add ingredients to the shopping list or remove ingredients from the shopping list.

**Managing Recipe Images:** All the recipe images that the user uploads are saved in the Filestack cloud storage, and a link is provided on which the image is available, and the file stack sends the links to Client server.

## 6.3. Database Design

A well-organized and structured database is essential for the smooth functioning and successful operation of an application. For this application, effective data management and effortless retrieval of data are vital. A well-designed database plays an important role in storing and retrieving data ensuring low overhead, data integrity, scalability and overall performance. Figure 6.3 shows the schema of the database that is designed for the recipe application. The relationships between tables are clearly defined and the constraints are strictly enforced to ensure that the data is consistent and accurate. This also helps in efficiently querying the database and retrieving related information. There are dedicated tables defined to store various categories for recipes to categorize like 'Diet', 'Cuisine' and 'Allergen' aiding in accurate search and filtering functionality of the web application. The detailed information on each table of the database in Figure 6.3 is discussed below.

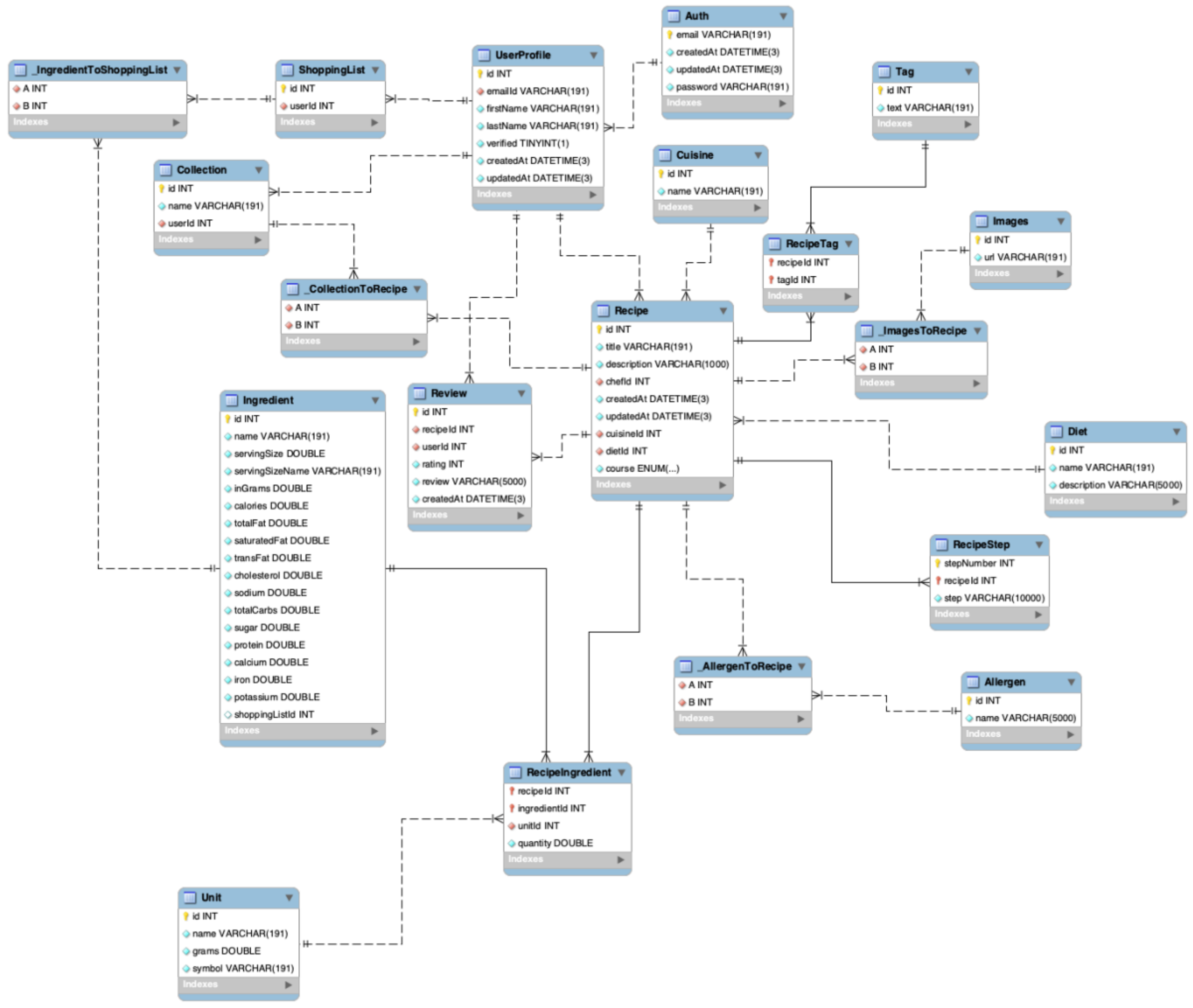


*Figure 6.3: Database Schema*

**Auth**: This table contains authentication information like email and password. This table is linked to User Profile table.

**UserProfile**: This table contains the user information such as id, emailId, firstName, lastName, verified and timestamps. Verified is a Boolean value which is by default set to false, when user verifies the email, this value is set to true. emailId as foriegnkey from Auth table. This has one-to-one relationship with the Auth table.

**Recipe**: This table consist of information recipes uploaded by users. It has the following attributes, id which is a unique id, title, description, chefId which is a foreignkey which maps to userProfile references id attribute, dietId, cuisineId which is foreignkey reference for tables Diet and Cuisine respectively, course which is an enum.

**Diet**: This table contains id, name, description. name corresponds to diet name and description contains information on particular diet and id is unique identifier.

**Cuisine**: This table consists of id which is unique id and name which is the cuisine name.

**Ingredient**: This table contains ingredient information such as id, name, servingSize, servingSizeName, inGrams, calories, totalFat, satuartedFat, transFat, cholesterol, sodium, totalCarbs, sugar, protein, calcium, iron, potassium, shoppingListId. The nutritional value of the ingredients is measured for a particular serving size of the ingredient.

**Unit**: This table has name, grams(measurement) and symbol to store unit information for the purpose of ingredient measurement.

**RecipeIngredient**: This table is a relationship between Recipe, Ingredient and Unit table which contains ingredient information for a particular recipe. It has recipeId, ingredientId and unitId which are foreignkey references to Recipe table, Ingredient table and Unit table respectively. It also has quantity as an attribute which store quantity of the ingredient used in the recipe.In this table recipeId and ingredientId are primary keys.

**Review**: This table has id, which is unique identifier, recipeId, userId which are foreignkey references to ids of tables Recipe and Ingredient respectively, rating which is an integer value, review which stores comments given by users.

**Tag**: This is used to store tags which can be used to tag recipes. It has id and text.

**RecipeTag**: This table is a relationship between Tag table and Recipe table. It only has attribute recipeId and tagId which are foreign key references to Tag and Recipe table respectively.

**Allergen**: This table stores name of the allergen and id which is used to uniquely identify a particular allergen.

**_AllergenToRecipe**: This table is automatically generated by Prisma to relate the Recipe and Allergen tables.

**Images**: This table contains all the image URLs.

**_ImagesToRecipe**: This table is automatically generated by Prisma to relate the Recipe and Images tables.

**RecipeStep**: This table contains steps for a particular recipe. It has step number which is an integer and recipeId which is a foreignkey reference to Id of Recipe table, step which stores recipe instructions. Step and recipeId are primary keys of this table.

**ShoppingList**: This table has id and userId which is a foreignkey reference to id of the UserProfile table.

**_IngredientToShoopingList**: This table is automatically generated by Prisma which maps shoppingList to Ingredient table. The purpose of this table is to map ingredients to a particular user's ShoppingList.

**Collection**: This table has id, name and userId. userId is the foreignkey reference to the UserProfile table.

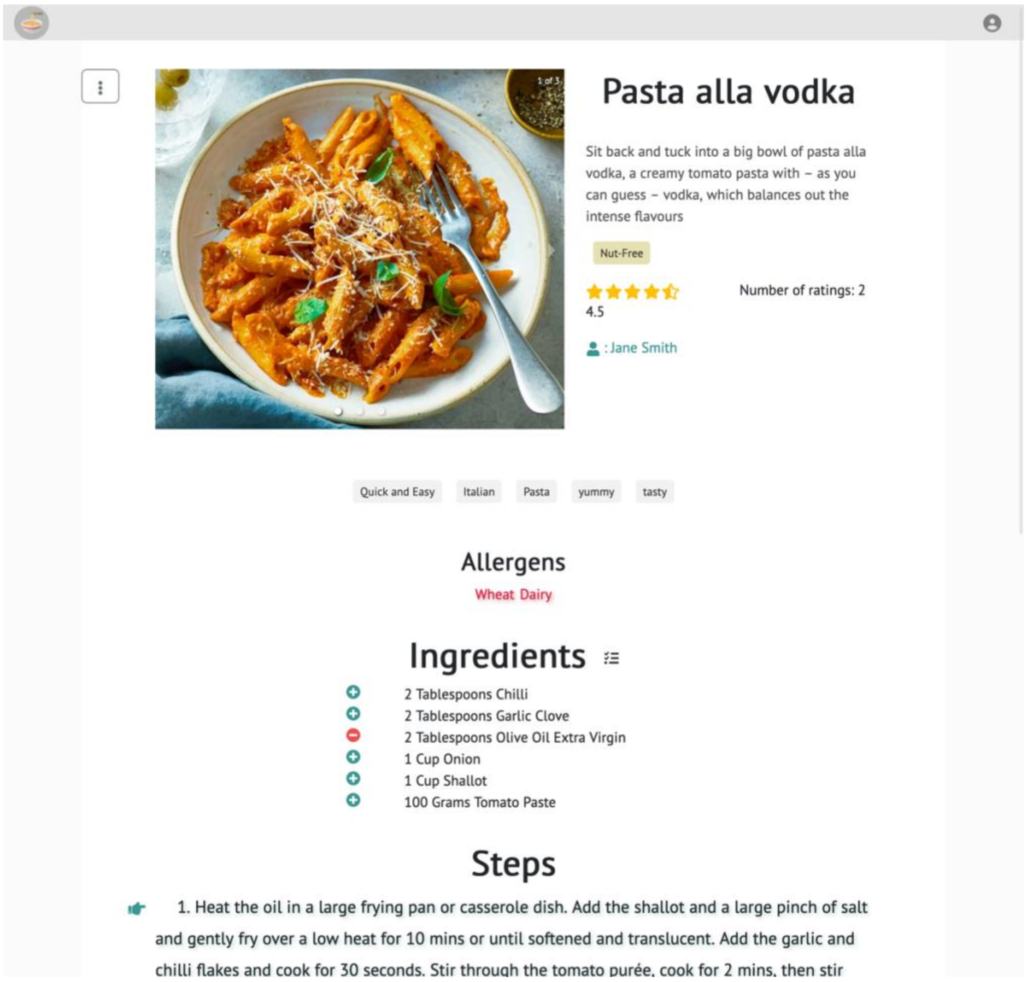
**_CollectionToRecipe**: This table maps recipes to Collection table which is automatically generated by Prisma.

# 7. Implementation

This section discusses about the project's implementation. This project is divided into two major parts. The first part is the user interface which is the front-end web application, and the second is the backend server. These two components are explained separately and maintain two separate code bases.

## 7.1. Front-end Implementation

The front-end for this web application (Figure 7.1) is built using ReactJs library. Taking maintainability and scalability into consideration, the whole front-end is built using functional components. Routing is performed using a React plugin called React Router which works seamlessly with React. For each route, when the browser is at a particular link, the respective JSX element and the child elements (in case of nested routes) are rendered. Each page is made up of several components which work together to provide a good user experience.



Pasta alla vodka

Sit back and tuck into a big bowl of pasta alla vodka, a creamy tomato pasta with – as you can guess – vodka, which balances out the intense flavours

Nut-Free

4.5

Number of ratings: 2

: Jane Smith

Quick and Easy Italian Pasta yummy tasty

Allergens

Wheat Dairy

Ingredients

2 Tablespoons Chilli
2 Tablespoons Garlic Clove
2 Tablespoons Olive Oil Extra Virgin
1 Cup Onion
1 Cup Shallot
100 Grams Tomato Paste

Steps

1. Heat the oil in a large frying pan or casserole dish. Add the shallot and a large pinch of salt and gently fry over a low heat for 10 mins or until softened and translucent. Add the garlic and chilli flakes and cook for 30 seconds. Stir through the tomato purée, cook for 2 mins, then stir

*Figure 7.1: Front end Recipe Details page*

### 7.1.1. Routing and Header Component

The Header component is rendered on every page which is defined in the main router. This is achieved using a built-in component '<Outlet />' in the parent route element which allows nested elements to be rendered when child routes are visited. The Header has a logo of the app on the left-hand side and a conditionally rendered component on the right (Figure 7.1). If the user is logged in to the application, it renders a dropdown with some links and if the user is not signed in the Header renders a button called 'Login/SignUp'. When this button is clicked a custom modal component called 'LoginSignUp' is rendered. This component is further broken down into two more components. The body of the modal is a conditionally rendered form. If the state 'isLogin' is set to true then the Login form is rendered, the Signup form is rendered otherwise. This state is set to true or false using a 'FormSwitcher' component having conditionally rendered buttons for login and register which sets the 'isLogin' state to true and false respectively when clicked on.

### 7.1.2. User Authentication

This application uses a global state management tool called Redux Toolkit to store various data in the states used in the application. One of the states used is the 'isAuthenticated' state. This state is set by a custom react hook which runs every time the Header component renders to check if an authentication token is available or not in the local storage. If the token is available, it will check with the backend API for validity. The 'isAuthenticated' state is set to 'true' or 'false' based on the server response.

### 7.1.3. User Profile and Recipe Management

Some features are only available to logged-in users. Users can access the profile page which has the details of the users like name, email and the list of recipes they published. This page also has a button to a publish recipe. There are two additional buttons on the profile page that navigate to a page with the users' shopping list and a page with the list of recipes logged-in users have added to their favorites. The recipe page containing all the information about a particular recipe can be viewed with or without logging in but only the logged-in user who has uploaded the particular recipe can delete or edit it. All the logged-in users have the option to add or remove a recipe from favorites.

The recipe page (Figure 7.1) also has an option to add recipe ingredients to the shopping list, or an option to remove them from the list if they already exist. This shopping list feature is unique to this project and is not found in any other similar websites or applications out there.

Another unique feature of this application is to track the recipe steps. When a user (logged-in or not) clicks on a step can highlight it indicating the user has reached the particular instruction. This feature is helpful for users to keep track of what they are doing when cooking the recipe. The reviews are listed at the bottom of this page. Each review card contains information like the ratings, review and the name of the critic. It also has the information on when was it added. In the review section, there is a button to add reviews which when pressed renders a modal to add a review. A user can review a recipe only once. The users have the option to delete a review that they gave if logged in.

#### 7.1.4. Reusable InfiniteScrollList Component

All the above pages use a custom-built InfiniteScrollList component. A react component can be made reusable by passing arbitrary inputs for it called props. Props for 'InfiniteScrollList' component are listed below.

i. recipe: It takes a boolean value suggesting if this component renders recipe cards.

ii. review: This prop is also a boolean value indicating if the children are review cards.

iii. loadData: It is a function passed as a prop that loads recipes or reviews page by page from back-end to give an infinite scrolling experience.

iv. searchValue: This prop is used to render both review cards and recipe cards. In the recipe search page, the component takes the search keyword as a prop to list only the recipes matching the search terms. In the recipe details page, this component is used to render a list of reviews which takes the id of the recipes. This information is used to list only the reviews given to a particular recipe with a specific ID. Also on the user profile page, this prop takes the ID of the user profile to fetch all the recipes published by the particular user.

v. token: It is a string value containing the authentication token of the logged-in user. This particular prop is only useful on the saved recipes page. The backend route fetching all saved recipes requires an authentication token to retrieve the saved recipes.

vi. filters: While searching for recipes, one can also filter recipes to refine the search results. This prop is responsible for providing the filter query required by the search route of the server.

In total this component is used on 5 pages for both review rendering purposes and recipe rendering purposes.

A recipe card component is a small component containing some details about recipes like the title, description, ratings etc. The review card has star ratings, comments and the names of the users who reviewed it.

#### 7.1.5. Interaction with Backend API

To interact with the backend Redux Toolkit library has a built-in tool called RTK Query. An individual API can be created using this, containing information on endpoints and how the data is fetched (Figure 6.1). Two types of operations can be performed in an API. Query operation is used to only fetch data from the backend API. It is mainly used to send GET requests to the backed endpoints. Mutation on the other hand is used when there are any changes made to the data on the server. This operation requires a URI or endpoint and method of the request sent and optionally takes the request body parameter. It is fairly easy to use the hooks created by the APIs in a react component. Note that the term API here does not refer to the backend API, but the API which is created using RTK Query to communicate with the server.

## 7.2. Back-end Implementation

The backend server is developed using Express.js framework to build the REST API for this project. Prisma is a new trending ORM that is used to simplify database interactions. It can be paired with any database or can be migrated to any database Relational or Non-relational. In this case, Prisma is used with MySQL which is a powerful open-source database. Some routes are protected and can be accessed only when authenticated. Authentication is performed using JWT (JSON Web Tokens). Finally, this RESTful API is integrated with Swagger/OpenAPI to document and test the routes.

### 7.2.1. Project Structure

The project uses MRCS directory structure as mentioned in section System Architecture. The codebase is categorized into 4 major types (Figure 6.1: Architecture Diagram). Models are the schema definitions of database entities. Routes are the endpoint definitions for the REST API which is mapped to controller functions. Controllers are the functions that handle the HTTP request run the business logic and respond with the status code and HTTP response. Services are the functions that deal with database transactions. These four sections are isolated, modularized and encapsulated.

### 7.2.2. Working and Flow

For instance, an HTTP request is sent from the client app to register a user to this application.

A route is defined to handle this request and run the controller responsible for registration. The controller processes this request and if all the criteria are met to create a new user - like the email provided hasn't already been used to register before, uses the service to create a new user in the database and sends a response back to the client. This highlights the cohesive functioning of all components within the backend system.

### 7.2.3. Authentication

Protected features which are developed in the application requires bearer token which will be generated by hashing user information like email, first name, last name etc. from the user profile. This token is required to access the protected routes which requires authentication to perform some tasks. The token should be valid and is verified by a middleware (which is similar to a controller function but runs prior to the controller to perform certain tasks, here verify the access token). If the token is valid the request is carried out successfully, otherwise the middleware responds with an error response and the controller function won't execute.

### 7.2.4. Essential Endpoints

Some of the essential API endpoints necessary for the application are listed below.

**1. Recipes route**

A GET request is sent to the route '/api/recipes' to fetch all the recipes in the database. This method does not require authentication and has some three query parameters. Two of the three parameters are necessary for pagination (retrieve recipes in chunks) which are page, and limit. The 'limit' parameter informs the controller how many recipes should be fetched per page. The 'page' query parameter lets the server know what page of the recipe array should be fetched. The 'searchValue' parameter is used to search and filter recipes.

**2. Recipe routes**

This particular route performs three major operations based on the method used by the HTTP request. If GET request is made to '/api/recipes/{id}' the server responds with the recipe details containing the recipe id. To update a particular recipe, an HTTP request is sent with PATCH method along with the updated recipe details in the request body. This route also performs a delete operation when the DELETE request is sent along with the recipe id in the URL. To delete and update recipes authentication token is required. To add a new recipe POST request is sent to '/api/recipes/new' route along with recipe details and an authentication token.

**3. Review routes**

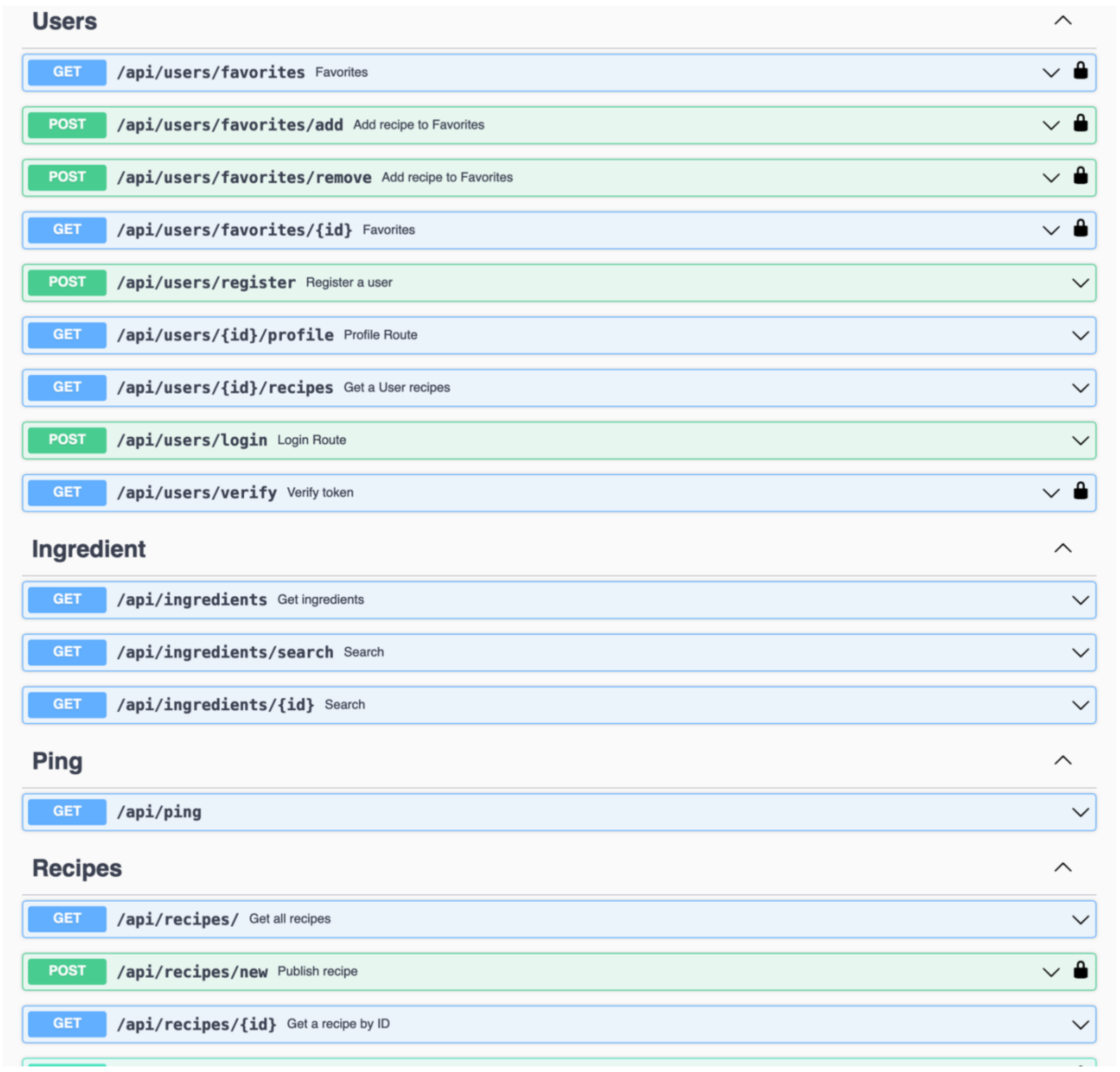
To leave a review on a recipe a POST request is sent to '/api/recipes/{id}/reviews/new' route. This route is protected and needs a token to post a review. Similarly, to remove the review a DELETE request is made to '/api/recipes/{id}/reviews/{reviewId}' along with the auth token to delete a review on the recipe. To get all reviews related to a recipe, GET request is sent to '/api/recipes/{id}/reviews' where 'id' is used to uniquely identify a recipe.

**4. Shopping list routes**

Client app retrieves shopping list having ingredients by requesting at '/api/users/shoppingList' route using GET method. This fetches only the shopping list of logged-in users. To add an ingredient to the users' lists POST request is sent to '/api/users/shoppingList/add' with ingredient id in the body of the request. Also removing an ingredient from the shopping list can be done by sending DELETE request to '/api/users/shoppingList/{id}' where 'id is the ingredient id present in the user's shopping list.

### 7.2.5. API Documentation

The backend API is designed by following RESTful convention ensuring a logical and structural approach to resource management. To make it easy to use, the API is documented using Swagger/OpenAPI documentation. It serves as a reference guide for developers suggesting how to use the API. The documentation is manually written for each route using YAML. The documentation can be accessed at '/api/docs' route of the backend server. It contains documentation for each endpoint providing information about the endpoint, method used (GET, POST, DELETE, PATCH, etc.), parameters required at the endpoints, and lists all the possible responses with examples. A developer can interact with the API through this and perform API testing. In Figure 7.2 the UI for the backend API that was built for this application.



Users

GET /api/users/favorites Favorites

POST /api/users/favorites/add Add recipe to Favorites

POST /api/users/favorites/remove Add recipe to Favorites

GET /api/users/favorites/{id} Favorites

POST /api/users/register Register a user

GET /api/users/{id}/profile Profile Route

GET /api/users/{id}/recipes Get a User recipes

POST /api/users/login Login Route

GET /api/users/verify Verify token

Ingredient

GET /api/ingredients Get ingredients

GET /api/ingredients/search Search

GET /api/ingredients/{id} Search

Ping

GET /api/ping

Recipes

GET /api/recipes/ Get all recipes

POST /api/recipes/new Publish recipe

GET /api/recipes/{id} Get a recipe by ID

*Figure 7.2: Swagger UI*

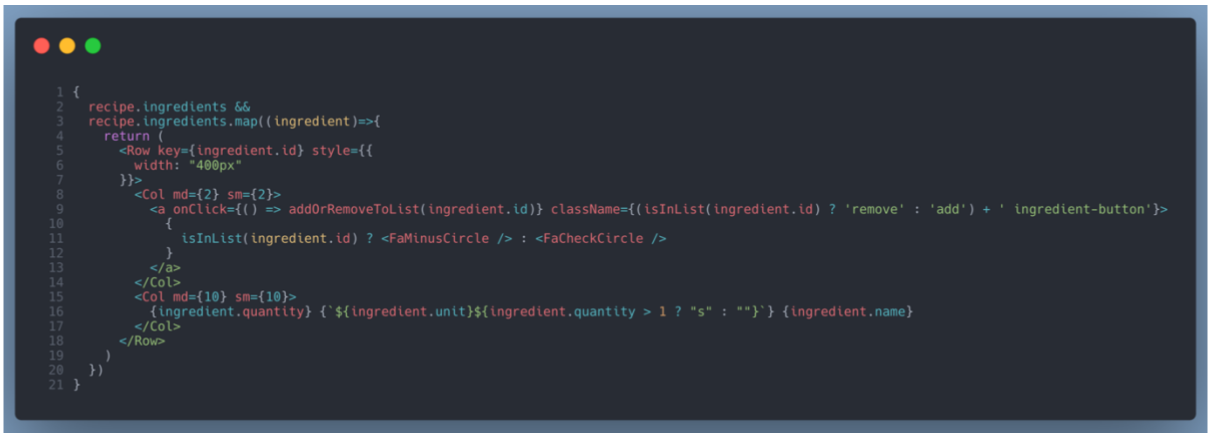
## 7.3. Code Snippets

Some of the code snippets that are unique or challenging to implement are listed and explained in this sub section.

### 7.3.1. Code Snippet 1

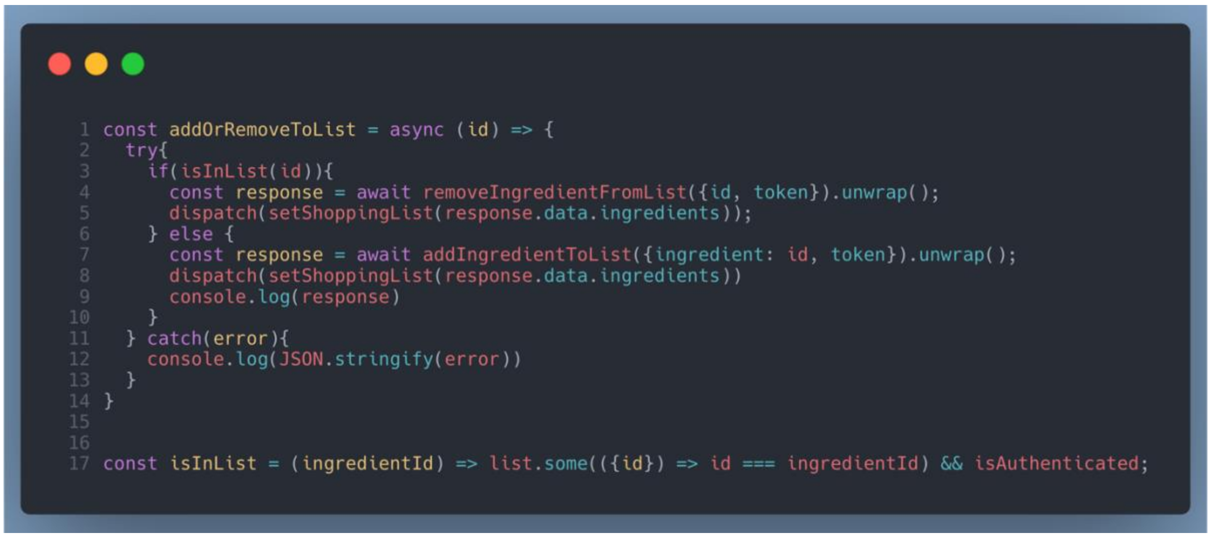
The below code snippet in Figure 7.3 is used to render ingredient information on the recipe details page. The code iterates through each recipe ingredient and creates a clickable link that is accessible only to the logged-in user. When the page initially loads the recipe details and the shopping list is loaded. The link on line number 9 when clicked adds or removes the particular ingredient to the shopping list. If the ingredient is present in the user's shopping list the icon in front of the ingredient name is red indicating a remove button. Otherwise, the button is green in color suggesting an add link. Which icon is to be used for the ingredient button is decided on line number 11. If the ingredient is not present in the shopping list the circular check mark icon is used. If it is already in the list the icon would be a circular minus.

The function to add an ingredient to the list and remove an ingredient from the shopping list is shown in Figure 7.4 below. It checks if the ingredient is present in the list using 'isInList' function which returns a Boolean value. If the returned value is true, the recipe is present in the list and the function proceeds to run a function that calls the backend route to remove the recipe from the user's shopping list. The function takes ingredient ID and auth token as parameters. If the value is false the function sends a backend request to add the ingredient to the list.



```
 1 {
 2   recipe.ingredients &&
 3   recipe.ingredients.map((ingredient)=>{
 4     return (
 5       <Row key={ingredient.id} style={{
 6         width: "400px"
 7       }}>
 8         <Col md={2} sm={2}>
 9           <a onClick={() => addOrRemoveToList(ingredient.id)} className={(isInList(ingredient.id) ? 'remove' : 'add') + ' ingredient-button'}>
10             {
11               isInList(ingredient.id) ? <FaMinusCircle /> : <FaCheckCircle />
12             }
13           </a>
14         </Col>
15         <Col md={10} sm={10}>
16           {ingredient.quantity} {`${ingredient.unit}${ingredient.quantity > 1 ? "s" : ""}`} {ingredient.name}
17         </Col>
18       </Row>
19     )
20   })
21 }
```

*Figure 7.3: Code snippet from Recipe component rendering ingredients*



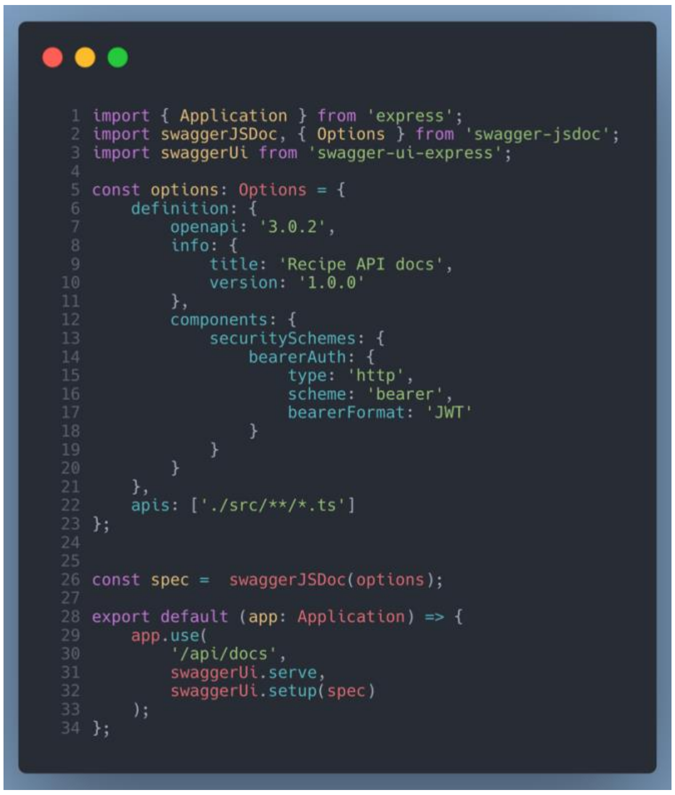
```
const addOrRemoveToList = async (id) => {
  try{
    if(isInList(id)){
      const response = await removeIngredientFromList({id, token}).unwrap();
      dispatch(setShoppingList(response.data.ingredients));
    } else {
      const response = await addIngredientToList({ingredient: id, token}).unwrap();
      dispatch(setShoppingList(response.data.ingredients))
      console.log(response)
    }
  } catch(error){
    console.log(JSON.stringify(error))
  }
}


const isInList = (ingredientId) => list.some(({id}) => id === ingredientId) && isAuthenticated;
```

*Figure 7.4: Code snippet from Recipe component for modifying shopping list*

### 7.3.2. Code Snippet 2

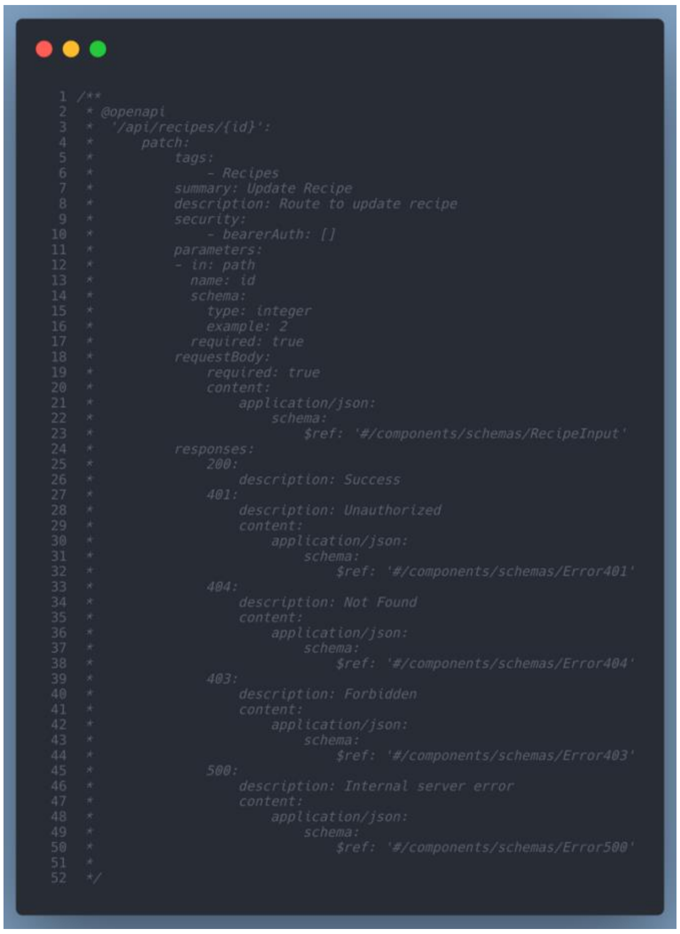
The piece of code in Figure 7.5 is a utility that deals with parsing and serving the swagger documentation of the back-end API. Initially, the necessary options are defined in the 'options' variable on line 5 of Figure 7.5. These options contain the information of the swagger specification like openapi version, api information like description and version. On line 12 a security component is defined which tells the swagger docs about the authentication method used by the API, here Bearer Authentication using JWT tokens. The next option ‘apis’ is an array of files in which the swagger definitions have been written. On line 22 the regular expression indicates the swagger options to check all the files with file extension 'ts' inside directories inside the src directory. Meaning, typescript files inside two nested directories. The specifications for the documentation containing all the information about the API are generated on line 26 using the library ‘swagger-jsdoc’ which takes the previously written options. The swagger UI is set up at the route ‘/api/docs’. The swagger-ui-express is another library used and is responsible for rendering the UI for the swagger definitions provided. On line 30 the route is defined where the swagger UI will be available.

The specifications for the API are then set up and ready to access. The API route definition shown in Figure 7.6 is an example of how the swagger definition is written for a route. It is written inside a multi-line comment and in YAML. The definition must be started with the keyword ‘@openapi’ or ‘@swagger’ to be identified by the ‘swagger-jsdoc’ parser. It contains the path or the endpoint (line 3). The method used by the endpoint is mentioned in line 4. Each method has multiple properties, where endpoints with the same tags are grouped separately. The summary and description for a route can be specified as well. We can mention if it uses any authentication (line 9). The parameters are also specified which can be in path or query. Here the parameter in the path called 'id' is mandatory and should be specified in the path which is an integer. The request body is also defined on line 18 having some schema. The schema definition is written in some other file and is referenced on line 23. Finally, various responses expected from the endpoint are listed one below the other starting from line 25. Similarly, swagger definitions are written for each route.



```
import { Application } from 'express';
import swaggerJSDoc, { Options } from 'swagger-jsdoc';
import swaggerUi from 'swagger-ui-express';

const options: Options = {
    definition: {
        openapi: '3.0.2',
        info: {
            title: 'Recipe API docs',
            version: '1.0.0'
        },
        components: {
            securitySchemes: {
                bearerAuth: {
                    type: 'http',
                    scheme: 'bearer',
                    bearerFormat: 'JWT'
                }
            }
        }
    },
    apis: ['./src/**/*.ts']
};


const spec =  swaggerJSDoc(options);

export default (app: Application) => {
    app.use(
        '/api/docs',
        swaggerUi.serve,
        swaggerUi.setup(spec)
    );
};
```

*Figure 7.5: Code snippet to serve swagger documentation*



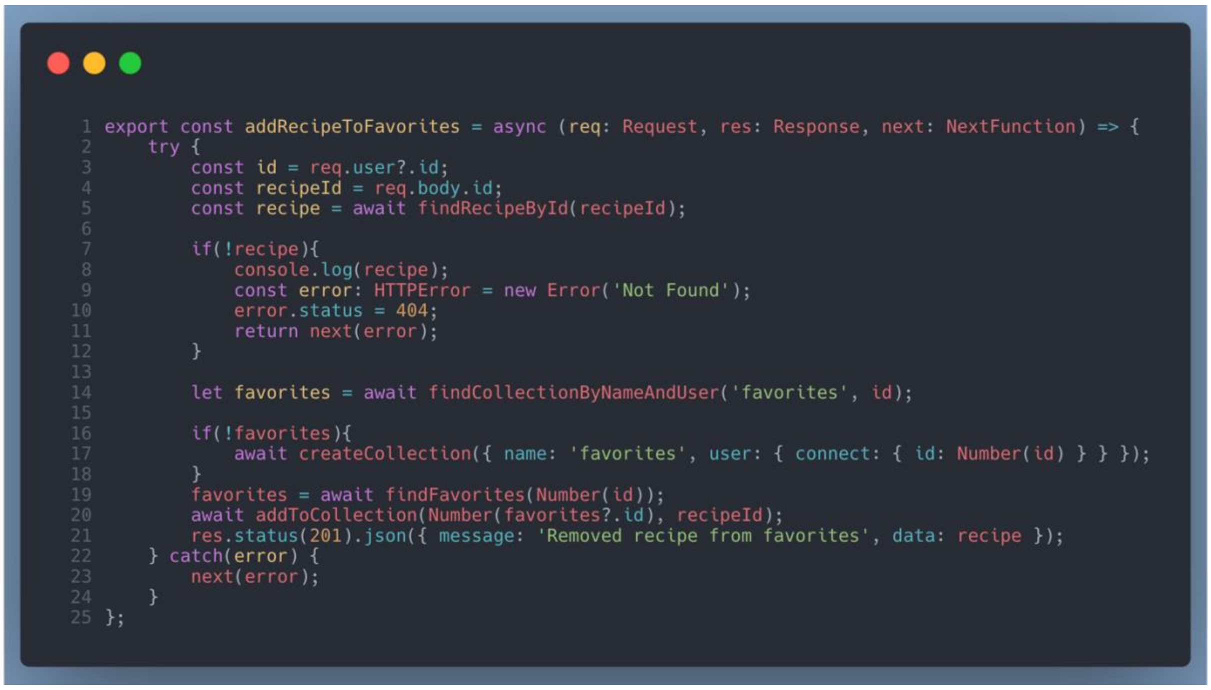
```
/**
 * @openapi
 *  '/api/recipes/{id}':
 *      patch:
 *          tags:
 *              - Recipes
 *          summary: Update Recipe
 *          description: Route to update recipe
 *          security:
 *              - bearerAuth: []
 *          parameters:
 *          - in: path
 *            name: id
 *            schema:
 *              type: integer
 *              example: 2
 *            required: true
 *          requestBody:
 *              required: true
 *              content:
 *                  application/json:
 *                      schema:
 *                          $ref: '#/components/schemas/RecipeInput'
 *          responses:
 *              200:
 *                  description: Success
 *              401:
 *                  description: Unauthorized
 *                  content:
 *                      application/json:
 *                          schema:
 *                              $ref: '#/components/schemas/Error401'
 *              404:
 *                  description: Not Found
 *                  content:
 *                      application/json:
 *                          schema:
 *                              $ref: '#/components/schemas/Error404'
 *              403:
 *                  description: Forbidden
 *                  content:
 *                      application/json:
 *                          schema:
 *                              $ref: '#/components/schemas/Error403'
 *              500:
 *                  description: Internal server error
 *                  content:
 *                      application/json:
 *                          schema:
 *                              $ref: '#/components/schemas/Error500'
 *
 */
```

*Figure 7.6: Sample swagger route definition*

### 7.3.3. Code Snippet 3

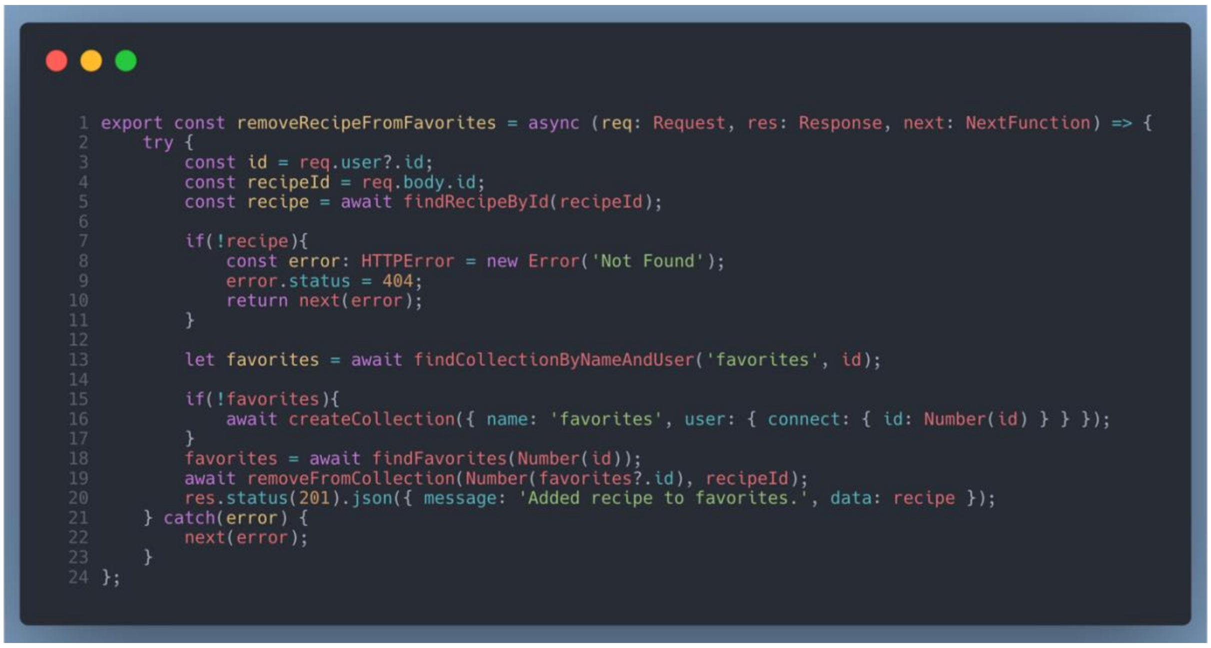
The controller responsible for adding recipes to users' favorites collection is shown in Figure 7.7. The route used to add a recipe to favorites is '/api/users/favorites/add'. When a post request is made, this controller handles the logic. Let us assume the user sending the request is already authenticated. The controller starts by parsing the body of the request in line number 3 to obtain the recipe id which is required to identify the recipe in the database (lines 3-6). If the recipe is not found in the database, the controller defines an error with 404 status code which stands for 'Not Found' and passes it to the server's error handler middleware. If the recipe exists, the function proceeds to find the user's favorites list on line 14. If this is the first time adding to favorites, the server creates a collection to store this information (line 17). On line 20 the service to link a recipe to favorites is run which takes the collection ID of the favorites list and the recipe ID itself.

Removing recipes from favorites is also a similar process. A post request is sent to the route '/api/users/favorites/remove' with an authentication token in the header of the request. The controller mapped to this route is available in Figure 7.8. It is very similar to the previously discussed controller function. The only noticeable difference is seen on line number 19, a service to remove the recipe from the user's favorites collection.



```
 1 export const addRecipeToFavorites = async (req: Request, res: Response, next: NextFunction) => {
 2     try {
 3         const id = req.user?.id;
 4         const recipeId = req.body.id;
 5         const recipe = await findRecipeById(recipeId);
 6 
 7         if(!recipe){
 8             console.log(recipe);
 9             const error: HTTPError = new Error('Not Found');
10             error.status = 404;
11             return next(error);
12         }
13 
14         let favorites = await findCollectionByNameAndUser('favorites', id);
15 
16         if(!favorites){
17             await createCollection({ name: 'favorites', user: { connect: { id: Number(id) } } });
18         }
19         favorites = await findFavorites(Number(id));
20         await addToCollection(Number(favorites?.id), recipeId);
21         res.status(201).json({ message: 'Removed recipe from favorites', data: recipe });
22     } catch(error) {
23         next(error);
24     }
25 };
```

*Figure 7.7: Controller to add recipe to favorites*



```
export const removeRecipeFromFavorites = async (req: Request, res: Response, next: NextFunction) => {
    try {
        const id = req.user?.id;
        const recipeId = req.body.id;
        const recipe = await findRecipeById(recipeId);

        if(!recipe){
            const error: HTTPError = new Error('Not Found');
            error.status = 404;
            return next(error);
        }

        let favorites = await findCollectionByNameAndUser('favorites', id);

        if(!favorites){
            await createCollection({ name: 'favorites', user: { connect: { id: Number(id) } } });
        }
        favorites = await findFavorites(Number(id));
        await removeFromCollection(Number(favorites?.id), recipeId);
        res.status(201).json({ message: 'Added recipe to favorites.', data: recipe });
    } catch(error) {
        next(error);
    }
};
```

*Figure 7.8: Controller to remove recipe from favorites*

## 7.4. Libraries and Dependencies

The Table 7.1 contains all the third-party libraries that are used in the development of the application. The use of third-party APIs and libraries reduce the project development duration significantly since these libraries built for this purpose. The developer doesn’t have to be concerned about any bugs arising due to these libraries since all the libraries are tested and frequently updated by the maintainers.

*Table 7.1: Libraries and Plugins*

| **Library/Plugin Name** | **Usage** |
|---|---|
| Font awesome | Used in the client side for icons. |
| Redux Toolkit | Used in the front-end for state management and querying the backend API. |
| Bootstrap | Used to design front-end framework along with CSS. |
| React Bootstrap | It is custom component library. |
| React infinite scroll component | Used in the front-end to build infinite scroll list. |
| React-router-dom | Used in the front-end to navigate to different pages. |
| React select | Used in the front-end to select component inside a Recipe form. |
| Dotenv | Used in the application to store access keys, API key for file stack, jwt secret, database URI server port number. |
| JWT token | Used in the backend to generate user token. |
| Swagger-jsdoc | Used in the backend to parse the swagger |

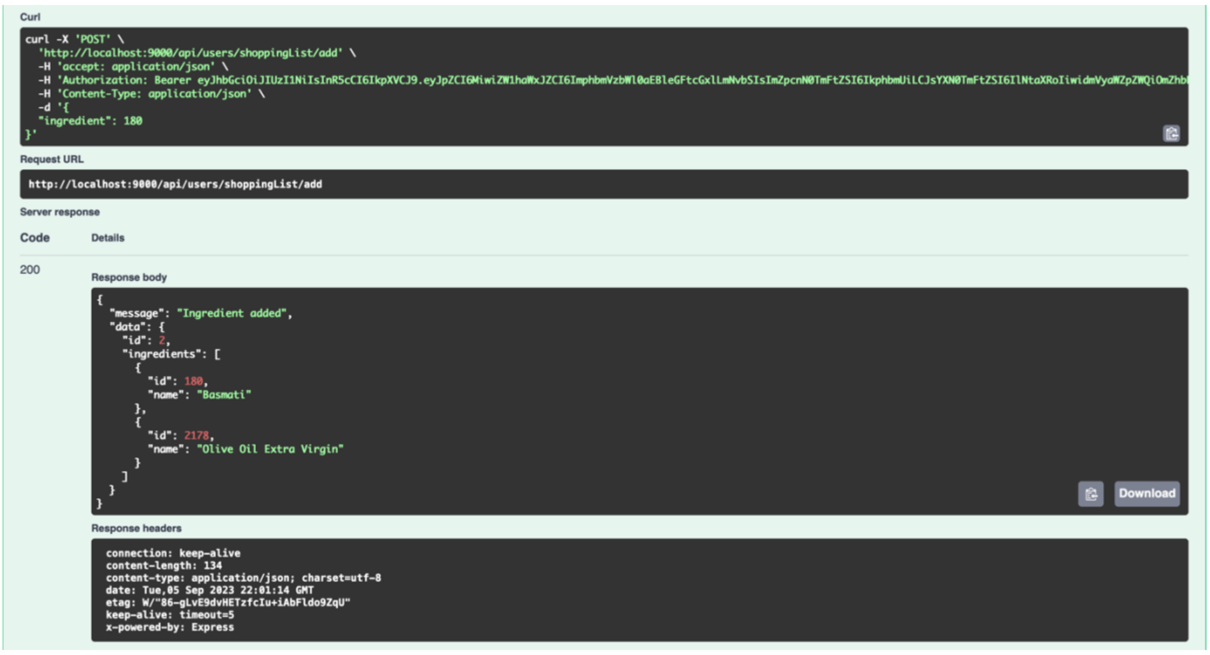
| | |
|---|---|
| | specification. |
| Swagger-UI-Express | Used in the backend to display Swagger docs on the browser. |
| Prisma | Used in the backend as ORM to interact with the database. |
| Puppeteer | Used in the backend to scrape the data to populate the database. |

# 8. Testing

As discussed above in system architecture, the application is broken into frontend(client) and backend components. As a result, all components were thoroughly tested to eliminate usability concerns and eliminate all bugs encountered during testing, ensuring that the quality of apps remains high.So, in this section, taking into account the insights provided by Ricca and Tonella's paper, "Analysis and Testing of Web Applications," presented at ICSE 2001 [20], testing was done using semi-automatic and manual testing, where functional testing was done using swagger to ensure all backend functionality was working properly or not, then UI testing to determine the accuracy of the user interface in the application, and finally usability testing for overall user experience.

## 8.1. Functional Testing

Functional testing is carried out in the development of this application to evaluate the functionality of each component. To test the Online Collaborative Recipe Book application Swagger-UI is used in which the requests are sent to the API endpoints to verify the intended results. In Figure 8.1, the request is sent using the POST request method to add ingredients to the Shopping list. Here, the ID of the ingredient is passed in the request body, '-d'. This request verifies if the response adds the ingredients to the Shopping list. Similarly, all the functionalities of the applications are tested to check if the response gives the expected scenarios.


Curl

```
curl -X 'POST' \
  'http://localhost:9000/api/users/shoppingList/add' \
  -H 'accept: application/json' \
  -H 'Authorization: Bearer eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJpZCI6MiwiZW1haWxJZCI6ImphbmVzbWl0aEBleGFtcGxlLmNvbSIsImZpcnN0TmFtZSI6IkphbmUiLCJsYXN0TmFtZSI6IlNtaXRoIiwidmVyaWZpZWQiOmZhb
  -H 'Content-Type: application/json' \
  -d '{
  "ingredient": 180
}'
```

Request URL

```
http://localhost:9000/api/users/shoppingList/add
```

Server response

Code | Details

200

Response body

```
{
  "message": "Ingredient added",
  "data": {
    "id": 2,
    "ingredients": [
      {
        "id": 180,
        "name": "Basmati"
      },
      {
        "id": 2178,
        "name": "Olive Oil Extra Virgin"
      }
    ]
  }
}
```

Response headers

```
connection: keep-alive
content-length: 134
content-type: application/json; charset=utf-8
date: Tue,05 Sep 2023 22:01:14 GMT
etag: W/"86-gLvE9dvHETzfcIu+iAbFldo9ZqU"
keep-alive: timeout=5
x-powered-by: Express
```

*Figure 8.1: API reference to add Ingredient to shopping list*

## 8.2. UI Testing

UI testing is done to verify the interaction of the user interface of the developed application.

This verifies if the application is user-friendly and if the application works as expected. The developer has tested if the User interface elements such as forms, buttons, links, and layout of the application are consistent throughout the application. The navigation of the application is tested to ensure the users can navigate through the different pages of the application without any confusion. The application is also tested for its responsiveness across different screen sizes and devices. The application is tested to display proper error messages to help the users easily understand the issues to resolve the errors. The application forms and their fields are tested to verify if they are responding as required.

## 8.3. Usability Testing

By considering Lewis's 2012 Research paper "Usability Testing," [19] the author provides an in-depth analysis of user testing concepts, emphasizing user feedback and iterative improvement—a great resource for improving usability, mirroring the importance of user testing in web applications. For the testing component, user testing was conducted to ensure that the recipe application worked properly. So the type of testing in which different user test applications are referred to as usability testing, and by using this, the users have interacted with the recipe website to find all of the usability issues, all of the aesthetic of the application, CSS issues, feedback, and at the end got overall user experience of the project. During this testing phase, four participants completed specific tasks such as performing a search operation with filters, adding ingredients to a shopping list, tracing the steps, creating the recipe and determining whether all of the tags are properly listed for that user or not, and leaving ratings and comments on a specific recipe. During this testing, users' interactions with the application to identify issues was observed, and user discovered some usability bugs, including a few major and minor issues such as leaving ratings and comments being confusing, the shopping list not properly updating when there is a large set of data, and some alignment issues, and at the end of testing, all the discovered bugs were prioritized and performed bug fixes.

# 9. Conclusion

The Online Collaborative Recipe Book web application is developed and tested. All the essential requirements mentioned in the Requirements section is implemented and thoroughly tested. As for the recommended requirements, the shopping list feature which is a unique feature was implemented to make the application stand out from the rest of the similar websites. The optional requirement 'in-app chatting feature' was not implemented due to a narrow timeframe. This was communicated to the supervisor and was approved in a scrum call. A new feature requirement 'Progress Tracking' suggested by Prof. Paul Holmes, the second supervisor, during the interview that was conducted to get a second opinion, was then implemented to make the application feature rich.

During the development of the project, the author learned how to build a Full-Stack Web Application and test it. The author also learned to design RESTful API proper convention. Documenting the API is an essential part of the development phase. Using open-source documentation tools such as Swagger to write a good API reference was also a key skill obtained throughout the whole project.

Throughout the project development process, there were a lot of challenges faced by the developer that slowed down the progress. One of the challenging tasks was to scrape the ingredient dataset. Writing a script to perform the scraping was not the challenging part, but the whole scraping process itself. In order to extract all the data, the automated script had to run without any interruption. Though the whole process usually takes 5-6 hours to finish, the computer went to sleep mode when left unattended for a few minutes. This terminated the process leading to incomplete data scraping. Though the web scraper was running without the system going to sleep mode, the process failed once due to memory consumption. The script had to be rewritten in order to scrape the data chunk by chunk so the system to not run out of memory. The whole process of web scraping took roughly 3 days in total.

The main aim of the project has been achieved with novel technologies used to develop the web application. The final app has been documented and tested for bugs and usability before delivering the final product.

# 10. Future work

This section is primarily concerned with the future implementations of the recipe website that have already been discussed above. These are some of the most important and will have a significant impact on the recipe website that can be implemented in the future.

## 10.1. Recipe Forking and Versioning

At first, forking an existing recipe available on the recipe website which means copying the original user's recipe and then adding changes or creating their own version of the recipe. By enabling this option, users can build their own recipes without affecting the original recipe. This is similar to the forking GitHub's forking feature where a user can fork another user's repository.

To implement this, the developer may face some challenges such as data consistency, which suggests if the original recipe is changed, the forked recipe must not reflect changes from the original recipe. Versioning can also be one of the challenges if the user wants to discard the changes and revert back to the original version. To address all of these challenges, the developer must first employ a new database table schema or precisely, alter the recipe table schema in such a way that the original and forked recipes are stored in the same table and the forking, and the versioning information is stored in a different table. By generating a version history for recipes, users can see who made them and when they were made. Finally, for privacy reasons, it is desirable to include a privacy setting that allows users to make recipes public, private, or viewable by a limited number of people.

## 10.2. Recommendation and Ingredient Substitution

Ingredient recommendation or substitution can be an innovative feature that can be implemented as a later feature, such as if the user does not have any ingredients, he should be listed with similar ingredients. This implementation can have some significant challenges that must be considered and highlighted too. The first is nutritional analysis, which involves determining the exact nutritional value of the original ingredient and then displaying similar ingredients with nearly identical nutritional values. The second is ingredient availability, if there is no similar ingredient available, the system should display an alternative. Also, a scalability issue will arise as the ingredient data grows, and scalability becomes critical. Consequently, it is crucial to maintain regular updates to the database, particularly in anticipation of the potential addition as well as availability of new ingredients in the near future.

To address all of these issues, a recommendation algorithm (content-based algorithm) must first be developed, followed by deep learning approaches for obtaining accurate nutritional values. We can overcome scalability difficulties by utilizing cloud-based storage, and finally, by implementing image processing for the ingredients to provide essentially identical recipes, we can reveal the new ingredient.

## 11. References

1. Chef's Pencil. (2022, May 1). *The Best 21 Cooking Websites*. Chef's Pencil. https://www.chefspencil.com/the-best-20-cooking-websites/
2. Yepis, E. (2023, June 13). *2023 Developer Survey results are in: the latest trends in technology and work from the Stack Overflow community*. Stackoverflow.blog. https://stackoverflow.blog/2023/06/13/developer-survey-results-are-in/
3. TypeScript, *Documentation - TypeScript for the New Programmer*. www.typescriptlang.org. https://www.typescriptlang.org/docs/handbook/typescript-from-scratch.html#a-typed-superset-of-javascript
4. OpenJS Foundatio. *Express 4 - API Reference*. Expressjs.com. https://expressjs.com/en/4x/api.html
5. Mozilla Foundation. *HTTP response status codes*. MDN Web Docs. https://developer.mozilla.org/en-US/docs/Web/HTTP/Status
6. *Definition, Usage, Examples and Types*. Prisma's Data Guide. https://www.prisma.io/dataguide/intro/what-are-databases
7. *Prisma Documentation*. Prisma. Retrieved September 5, 2023, from https://www.prisma.io/docs/concepts/components/prisma-client
8. OpenAPI Specification. (2021, February 15). *Home*. OpenAPI Initiative. https://www.openapis.org/
9. Meta Open Source. (2023). *React*. React.dev. https://react.dev/
10. *Vite*. (n.d.). Vitejs.dev. https://vitejs.dev/guide/why.html
11. Abramov, D. (2023, May 1). *Redux*. Redux.js.org. https://redux.js.org/introduction/getting-started
12. Nutritionix. (2019). *Nutritionix*. Nutritionix.com. https://www.nutritionix.com/
13. *Puppeteer Documentation*. (2019). Puppeteer. https://pptr.dev/
14. *Built-in React Hooks*. (n.d.). React.dev. https://react.dev/reference/react
15. Khosravi, K. (2020, June 3). *Why you should use SWC (and not Babel)*. LogRocket Blog. https://blog.logrocket.com/why-you-should-use-swc
16. Mozilla Foundataion. (2019, November 7). *Getting started with the Web*. MDN Web Docs. https://developer.mozilla.org/en-US/docs/Learn/Getting_started_with_the_web

17. *React Bootstrap*. React-Bootstrap.netlify.app. https://react-bootstrap.netlify.app/
18. React Router. *Feature overview v6.4.1*. Reactrouter.com. https://reactrouter.com/en/main/start/overview
19. Lewis, J. R. (2012). Usability testing. Handbook of human factors and ergonomics, 1267-1312.
20. Ricca, F., & Tonella, P. (2001, May). Analysis and testing of web applications. In Proceedings of the 23rd International Conference on Software Engineering. ICSE 2001 (pp. 25-34). IEEE.