**System Design Document**
Project Name: RecipeConnect
Team: DevourDevs
Sprint 2

## 1. Introduction
This document outlines the system design for the implementation of RecipeConnect app. The system is developed using the MERN (MongoDB, Express.js, React, Node.js) stack and follows MVC architecture.

## 2. System Overview
The software will consist of the following key features:

- **User Authentication**: Users can register and login so that they can create and manage their recipes.
- **Recipe Browsing**: All users (both logged-in and non-logged-in) can view all recipes on the landing/home page. Users can view details about recipes, including ingredients, instructions, and related information.
- **Filtering and Searching:** Users can filter recipes based on dietary information, dish types, and cooking time. Users can also do a general search for recipes.
- **Recipe Rating:** Logged-in users can rate meals based on their experience.
- **Add Recipes to Favourite List:** Logged-in users can save a recipe to their favourite list
- **Ingredient-Based Search**: Logged-in users can input ingredients to get recipe suggestions.

## 3. CRC cards:
### 3.1 UserModel Class (models/userModel.js)

<table>
<tr><td colspan="2">Class Name: UserModel</td></tr>
<tr><td colspan="2">Parent Class: None<br>Subclass: None</td></tr>
<tr><td>Responsibilities:<br>● Define the schema for user data (username, password, email)<br>● Ensure email and username are unique<br>● Check for existing user with given username<br>● Hash password using bcrypt before saving users to database<br>● Validate user credentials during</td><td>Collaborators:<br>● AuthController - to handle user registration and login logic</td></tr>
</table>

| | |
|---|---|
| login (i.e. compare hashed password)<br>● Save user data to the database | |

**3.2 AuthController Class (controllers/authController.js)**

| Class Name: **AuthController** | |
|---|---|
| Parent Class: None<br>Subclass: None | |
| Responsibilities:<br>● register: handle user registration, validate input, check if user with username and email exists<br>● login: validate credentials, generate JWT, send response (including JWT) to client, redirect to home page (with login status)<br>● me: checking the user's login status and provides information of the logged-in user (userId, username) | Collaborators:<br>● UserModel - to create a new user instance, provides methods to check if user exists, and validate user credentials<br>● AuthMiddleware - to handle authentication checks for protected routes. |

**3.3 AuthMiddleware Class (middlewares/authMiddleware.js)**

| Class Name: **AuthMiddleware** | |
|---|---|
| Parent Class: None<br>Subclass: None | |
| Responsibilities:<br>● Protect routes that require user authentication<br>● Verify the user's JWT token to ensure they are logged in<br>● Redirect users to login page if they are not logged in when performing actions requiring authorization | Collaborators:<br>● authController - manage the authentication process<br>● mealController - to ensure that actions like creating/editing recipes are only accessible by logged-in users. |

### 3.4 MealModel Class (models/mealModel.js)

| Class Name: **MealModel** | |
|---|---|
| Parent Class: None<br>Subclass: None | |
| Responsibilities:<br>● Define the schema for recipes | Collaborators:<br>● mealController - to handle requests for retrieving recipes and other recipe-related operations<br>● authMiddleware - to verify user's identity before allowing protected recipe features such as adding to favourites, recipe rating |

### 3.5 MealController Class (controllers/mealController.js)

| Class Name: **MealController** | |
|---|---|
| Parent Class: None<br>Subclass: None | |
| Responsibilities:<br>● Fetch recipe data from Spoonacular api<br>● Return recipe data to client<br>● Handle recipe-related operations such as favourites, rating, filtering and search | Collaborators:<br>● mealModel - interact with the database for operations on recipes<br>● authMiddleware - ensure only logged-in users can perform actions like rating or adding a recipe to favourite list<br>● favouriteModel - to manage favourite meals<br>● ratingModel - to manage meal ratings |

### 3.6 FavouriteModel Class (models/favouriteModel.js)

| Class Name: **FavouriteModel** | |
|---|---|
| Parent Class: None<br>Subclass: None | |
| Responsibilities: | Collaborators: |

| | |
|---|---|
| • Define schema for user favourites<br>• Store favourite recipes for each user | • mealModel - to manage adding and retrieving favourite recipes<br>• userModel - to associate users with their favourite recipes.<br>• authMiddleware - ensure only logged-in users can add recipes to favourite list |

**3.7 RatingModel Class (models/ratingModel.js)**

| Class Name: **RatingModel** | |
|---|---|
| Parent Class: None<br>Subclass: None | |
| Responsibilities:<br>• Define schema for recipe ratings<br>• Store total rating points and users who rated the recipe | Collaborators:<br>• mealController - to manage rating operations<br>• userModel - to associate users with their ratings.<br>• authMiddleware - ensure only logged-in users can rate a recipe |

**4. System Interaction with the Environment**

**4.1 Operating System:**

- The system can be developed on Windows and macOS.
- Assumes the development environment supports Node.js for backend and React + Vite for frontend

**4.2 Technology Stack:**

- Frontend: Developed using React (JavaScript) and Vite as a build tool.
- Backend: Developed using Node.js and Express.js for handling API and client requests
- Database: Used local MongoDB for data storage. Relies on Spoonacular API for recipe data.
- External API: The app relies on Spoonacular API (https://spoonacular.com/food-api) for recipe data.

Please see the updated installation step in README.md file for instructions to set up the database and run the app.

## 5. System Architecture

### 5.1 Frontend (client-side):

- Handles UI, render pages for users to interact with
- Sends HTTP requests to the Backend (via axios or fetch) to retrieve data (recipes, user information)
- Displays content based on whether the user is logged in or not. For example:
  + Only logged-in users can manage their recipes
  + All users (including non-logged-in) can see list of recipes on main pages

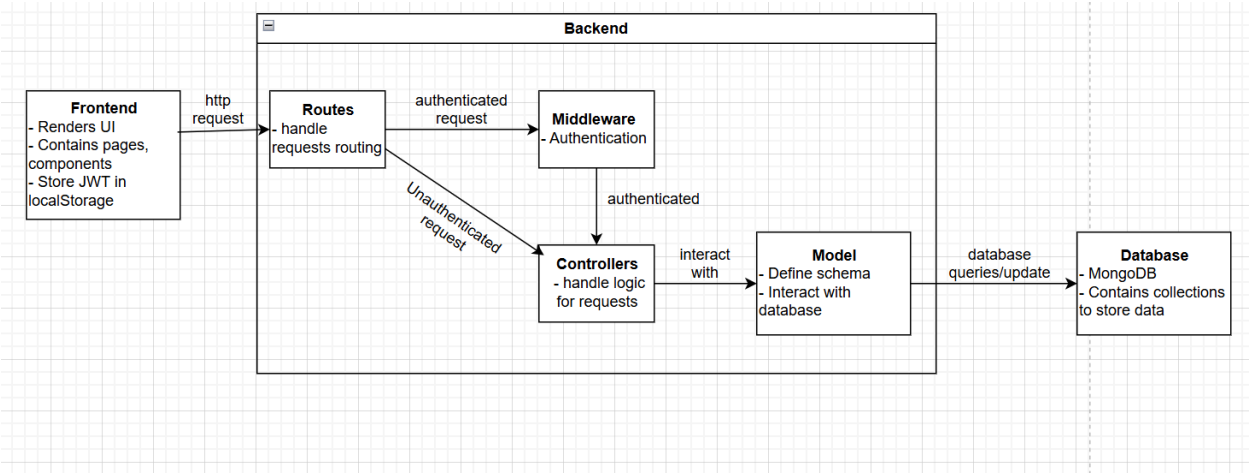
### 5.2 Backend (server-side):

- Manages logic for authentication, recipe operations, and serving API responses
- Authenticates users (with JWT and bcrypt) and verify requests for protected routes
- Interacts with the Database to retrieve and store user and recipe data.

### 5.3 Database:

- Stores user data and recipe content
- Provides data to the Backend via queries for things like retrieving recipes, user login logic.

### 5.4 Architecture Diagram:




## 6. System Decomposition:

### Frontend:

- Pages/Components: storing pages and UI components used in the app.
- App.jsx handles pages routing

- Logout will be handled on client side by removing JWT from local storage.

**Backend:**
- controllers: Handles authentication, recipe retrieval, favourites, ratings, and other recipe-related operations.
- middleware:
  + Authentication Middleware: Verifies JWT token on protected routes (e.g., recipe rating).
- models: Defines structured data models for users and recipe-related features.
- tests: Contains the test cases for backend testing using Jest

**Database:**
- The database is "recipeconnect"
- Current collections within the database:
  + Users: Stores hashed passwords, emails, and other profile data.
  + Meals: Stores recipe information

**7. Error Handling and Exceptional Cases:**
**Invalid user input during register:**
Password length must be at least 8: If this requirement is not met:
- Frontend will display an error message “Pass length must be at least 8” (or similar)

Confirmed password does not match provided password:
- Frontend will display an error message “Passwords do not match”

Username or email already taken:
- Backend will return 400 - Bad request
- Frontend will display an error message “Username/Email already existed”

**Authentication failure:**
Invalid JWT token: If the token is expired or invalid,
- Backend will return 401 - Unauthorized
- Frontend will redirect users to login page

Failed login: If credentials are incorrect during login,
- Backend will return 401 - Unauthorized
- Front end will display error message "Incorrect password or username"

**Database Errors:**
Connection Failures: If the backend cannot connect to database, backend will return 500 - Internal Server Error

Query Failure: If a query fails (e.g. no data found), backend will return 404 - Not found.

**Uncaught Errors:**

In the case of unexpected errors happening:

- Backend should provide a global error handler that will catch uncaught errors and return 500 - Internal Server Error
- Frontend should display appropriate error messages to users.