**Test Concept for the Recipe Planner Application**

**Introduction**

The Recipe Planner Application is a web-based tool that allows users to manage recipes, including creating, viewing, and managing a list of recipes and their ingredients. This document outlines the test concept for the application’s backend and frontend components, focusing on the features and approaches to ensure functionality and reliability.

**Test Items**

**Components to Test**

1. **Backend**:
   * Endpoints provided by the RecipeController to manage recipes.
   * Mapper logic, specifically RecipeEntityMapper, to ensure accurate data transformation between entities and domain objects.
   * Service layer logic to validate business rules.
2. **Frontend**:
   * Form submission for adding recipes and their ingredients.
   * Integration with backend APIs.
   * Search bar functionality and navigation links in the navbar.

**Architecture Overview**

The application architecture includes:

* **Frontend**: React.js with components for recipe management.
* **Backend**: Spring Boot application providing REST APIs.
* **Database**: Persistent storage for recipes and ingredients.

**Features to Be Tested**

**Backend Features**

1. **Endpoints**:
   * GET /api/recipes: Retrieve all recipes.
   * GET /api/recipes/recipe/{recipeId}: Retrieve a recipe by ID.
   * POST /api/recipes: Add a new recipe.
2. **Mapper Logic**:
   * Transformations between RecipeEntity and Recipe domain objects.
3. **Service Layer**:
   * Validate business logic and ensure data consistency.

**Frontend Features**

1. **Add Recipe Form**:
   * Validate form inputs (e.g., name, description, and ingredients).
   * Submit form data to the backend using the fixed "Submit" button.
2. **Search Bar**:
   * Allow users to search for recipes by name.
   * Display filtered results dynamically.
3. **Navbar**:
   * Ensure navigation links direct users to the correct pages.

**Features Not to Be Tested**

* Non-functional requirements such as performance and scalability.
* Database integration (assumed functional).

**Approach**

**Testing Methods**

1. **Manual Testing**:
   * Use Postman to test backend endpoints.
   * Verify frontend UI functionality in the browser.
2. **Automated Testing**:
   * JUnit for backend unit tests (e.g., RecipeEntityMapperTest, service, and controller tests).
   * Cypress for frontend end-to-end tests (e.g., navbar navigation, search functionality, and form submission).

**Tools**

* **Postman**: For manual API testing.
* **JUnit**: For backend unit tests.
* **Cypress**: For frontend end-to-end tests.

**Pass/Fail Criteria**

**Pass Criteria**

* Endpoint responses match expected structures and data.
* Mapper, service, and controller tests pass for all scenarios (valid and null inputs).
* Frontend features function as expected, with correct API integration.

**Fail Criteria**

* Incorrect data transformation in mappers.
* API requests fail or return unexpected results.
* Frontend UI elements do not respond as expected (e.g., form submission fails).

**Test Deliverables**

* Test scripts for backend and frontend.
* Test reports (e.g., JUnit results, Cypress screenshots/videos).

**Testing Tasks**

1. Verify backend CRUD operations using Postman (GET all, GET by ID, POST).
2. Write and execute unit tests for mappers, service, and controller classes.
3. Test Search bar, and form submission on the frontend.
4. Use Cypress to test:
   * Navbar navigation links.
   * Search bar functionality.
   * Submit button in the Add Recipe page.

**Environmental Needs**

* **Software**:
  + Node.js and npm for frontend.
  + Java and Maven for backend.
  + Local development server for testing APIs.
* **Hardware**:
  + Machine with sufficient resources to run the application and Cypress.

**Schedule**

1. **Day 1**: Test backend endpoints with Postman and write JUnit tests for mappers and service layer.
2. **Day 2**: Implement and test the search feature on the frontend.
3. **Day 3**: Write and execute Cypress tests for frontend features.
4. **Day 4**: Compile test results and finalize the documentation.

**Reflection on Code Reviews and TDD**

The test-driven development (TDD) approach was instrumental in identifying potential edge cases early. Code reviews ensured consistent implementation and adherence to best practices, leading to more robust and maintainable code.