**QA Back-End Test Automation**

**Exam Preparation I**

You can check your solutions in [Judge](https://judge.softuni.org/Contests/4698/Exam-Prep-I).

The **RecipeBook** app focuses on providing functionality for managing **recipe-related data**, such as **recipes** and **categories**. **Before running the tests do not forget to start your API**.

## How to Run the Project

You should have installed **Docker**.

Follow these steps to get the application running in a Docker container.

1. **Download** the **RecipeBook.zip** file, which contains all the necessary files.
2. **Unzip** the **RecipeBook.zip** file into your preferred directory on your machine.
3. **Build** and **Run the Docker Containers**.

Ensure you have **Docker** and **Docker Compose** installed. Then, run the following command to build and start the containers:

**docker-compose up --build**

This command will load the Docker image into your local Docker environment.

1. **Access** the API

Once the containers are up and running, you can access the API at <http://localhost:5000/api>.

1. API **Documentation**

API documentation is available at <http://localhost:5000/api-docs>.

## CRUD Operations for Recipes Testing (150 Points)

Open ApiTesting.zip file. You can write your test inside the methods in RecipesTests.cs. Be careful not to change the names of the following methods.

#### Prerequisites

Make sure the following are installed on your system:

* **Docker**: To containerize and run the application.
* **Node.js**: Required to run and test the application locally if needed.

#### Steps to Set Up the Application in Docker

Unzip **RecipeBook.zip** file. **Navigate to the RrecipeBook folder** containing the Docker configuration files. **Build the Docker image** and **start the application** using Docker Compose.

**Swagger Documentation**: The project includes a swagger.yaml file, which likely describes the API endpoints. You can refer to this to understand the required routes and endpoints.

### Get All Recipes (30 Points)

Write a unit test for the **Test\_GetAllRecipes**() method. Verify that the API successfully retrieves all recipes.

**Conditions**:

* The API should return a **200 OK status code**.
* The response content should **not** be **empty**.
* The response should be in the form of a **JSON array**.
* The array should contain at **least one recipe**.
* Each recipe should have valid fields such as **title**, **author**, **ingredients**, and **instructions**, which must **not** be **null or empty**.
* **Response Assertions:**
  + Ensures the API returns an **HTTP** **200 OK status code**, indicating that the request to retrieve all recipes was successful.
  + Ensures the response body is **not empty**, confirming that the API returned content.
* **Data Structure Assertions:**
  + Ensures that the response content is in the form of a **JSON array**, as expected when returning a list of recipes.
  + Verifies that the **JSON array** contains at least one recipe.
* **Recipe Fields Assertions (for each recipe):**
  + Ensures each recipe has a **non-null**, **non-empty title field**.
  + Ensures the recipe has a **valid author**, confirming the recipe is associated with an author.
  + Ensures each recipe has a **list of ingredients**.
  + Ensures each recipe includes **instructions** for preparation.

### Get Recipe by Title (30 Points)

Write a unit test for the **Test\_GetRecipeByName**() method. Verify the ability to retrieve a specific recipe by its title.

**Conditions**:

* The API should return a **200 OK status** code.
* The response content **should not be empty**.
* The recipe with the title "**Chocolate Chip Cookies**" should exist.
* The retrieved recipe should have **valid values** for **fields** like author, ingredients, and instructions, none of which should be null or empty.
* **Response Assertions:**
  + Ensures the API returns an **HTTP 200 OK status code**, indicating success in retrieving recipes.
  + Ensures the response **content is not empty**.
* **Data Structure Assertions:**
  + Filters the recipes to find one with the **title** "**Chocolate Chip Cookies**".
* **Recipe Fields Assertions:**
  + Ensures that a recipe with the title "**Chocolate Chip Cookies**" **exists** in the list.
  + Ensures the author of the recipe is provided and is **not empty**.
  + Ensures the recipe has **ingredients** specified.
  + Ensures the recipe **includes** **instructions**.

### Add Recipe (30 Points)

Write a unit test for the **Test\_AddRecipe**() method. Test the creation of a new recipe.

**Conditions**:

* The API should successfully create the recipe and return a **200 OK status code**.
* The response should **not** be **empty**.
* The newly created recipe's fields such as **title**, **author**, **category**, **ingredients**, and **instructions** should **match the input data** and should not be null or empty.
* **Response Assertions:**
  + Verifies that the recipe was successfully added by checking for an **HTTP 200 OK response**.
  + Ensures that the response content is **not empty**.
* **Recipe Fields Assertions:**
  + Ensures the title of the newly added recipe matches the input value ("**Chocolate Cake**").
  + Ensures the **author** field is populated.
  + Verifies the **category** field is provided.
  + Ensures that the recipe has the **correct** **ingredients**.
  + Ensures that the recipe has preparation **instructions**.

### Update Recipe (30 Points)

Write a unit test for the **Test\_UpdateRecipe**() method. Test the ability to update an existing recipe.

**Conditions**:

* First, retrieve the recipe "**Chocolate Cake**" and verify that it exists.
* The API should successfully update the recipe and return a **200 OK status code**.
* The updated recipe should have the new values ("**Double Chocolate Cake**" for the title and updated servings).
* The response content should reflect the updated values and should **not** be **empty**.
* **Response Assertions:**
  + Ensures the request to retrieve all recipes is successful with a **200 OK status**.
  + Ensures that the response content for retrieving recipes is not empty.
* **Recipe Existence Assertions:**
  + Ensures that the recipe to be **updated exists**.
* **Updated Fields Assertions:**
  + Ensures that the recipe update was successful with a **200 OK response**.
  + Ensures the response content for the **update** is **not** **empty**.
* **Updated Recipe Fields Assertions:**
  + Ensures the recipe title was updated correctly to "**Double Chocolate Cake**".
  + Verifies that the updated **servings** **field** **matches** **the** **input**.

### Remove Recipe by Id (30 Points)

Write a unit test for the **Test\_DeleteRecipe**() method. Test the ability to delete an existing recipe.

**Conditions**:

* First, retrieve the recipe "**Chicken Curry**" and ensure it exists.
* The API should successfully delete the recipe and return a **200 OK status code**.
* After deletion, trying to retrieve the recipe again should return **null** or an **empty** **result**, confirming that the recipe is deleted.
* **Get Request Assertions** 
  + Ensures that the retrieval of recipes before deletion is **successful**.
  + Ensures the response content is **not empty**.
* **Delete Request Assertions:**
  + Verifies that the recipe to be deleted **exists**.
  + Ensures the recipe deletion was successful with an **HTTP 200 OK status**.
* **Post-Deletion Verification:** 
  + Ensures that after deletion, trying to retrieve the recipe returns a **null** or **empty** **result**, confirming that the recipe was successfully deleted.

## Category Management Tests (150 Points)

You can write your test inside the methods in CategoryTests.cs. Be careful not to change the name of the following method.

### Test\_CategoryLifecycle (150 Points)

This test case ensures that the entire **lifecycle of a category** can be performed **successfully**. The lifecycle includes **creating** a category, **retrieving** it (both by listing all categories and by ID), **editing** the category, and finally **deleting** it.

#### Step 1: Create a new category

Ensure that a new category can be successfully created.

* Send a **POST request** to the category endpoint with the category details (name: Vegan Recipes).
* Validate the response.
* **Expected Behavior**:
  + The API should return a **200 OK status code**.
  + The response should contain a **valid category ID**.
  + The category name should match the input ("**Vegan Recipes**").

#### Step 2: Get all categories

Ensure the newly created category is included in the list of categories.

* Send a GET request to retrieve all categories.
* Parse the response and check for the existence of the newly created category.
* **Expected Behavior**:
  + The API should return a **200 OK status code**.
  + The response should **not** be **empty**.
  + The response should be a **JSON array** containing at least one category.
  + The newly created category should be present in the list.

#### Step 3: Get category by ID

Ensure that a specific category can be retrieved by its ID.

* Send a **GET request** with the newly created category's ID.
* Validate the category details in the response.
* **Expected Behavior**:
  + The API should return a **200 OK status code**.
  + The response should **not** be **empty**.
  + The category ID and name should match the expected values ("**Vegan Recipes**").

#### Step 4: Edit the category

Verify that a category can be updated successfully.

* Send a PUT request with the category ID and new name (name: **Healthy Vegan Recipes**).
* Validate the response to ensure the update was successful.
* **Expected Behavior:**
  + The API should return a **200 OK status** code.
  + The category name should be updated to "**Healthy Vegan Recipes**".

#### Step 5: Update category with invalid data (negative test)

Ensure the category cannot be updated with invalid data.

* Send a **PUT request** with invalid data (e.g., notName: "").
* Validate that the category name remains **unchanged**.
* **Expected Behavior:**
  + The API should return an appropriate response.
  + The category name should remain "**Healthy Vegan Recipes**" despite the invalid input.

#### Step 6: Delete the category

Ensure that a category can be **successfully deleted**.

* Send a **DELETE request** with the category ID.
* Validate the response to **confirm deletion**.
* Expected Behavior:
  + The API should return a **200 OK status code**.
  + The category should be **successfully** **deleted**.

#### Step 7: Verify category is deleted

Ensure the deleted category is no longer accessible.

* Send a **GET request** to retrieve the category by its ID.
* Validate that the response shows the category no longer exists.
* **Expected Behavior:**
  + The API should return null or empty content for the deleted category.
  + The category should no longer be retrievable.

## How to Submit Your Work

You need to submit your work on the SoftUni website in the Exam Section.

1. Archive the folder that contains your solution.
2. Upload the archive to the SoftUni website in the course section for your exam.