**QA Back-End Test Automation**

**Exam Preparation II**

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 must have installed **Node.js** – see detailed instructions here <https://nodejs.org/en/download/prebuilt-installer>.

Follow these steps to get the application running locally.

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.

A close up of a recipe book

Description automatically generated

1. Navigate to the unzipped folder in File Explorer and **Open in Terminal** there.
   1. Either using right-click **Open in Terminal**A screenshot of a computer

      Description automatically generated
   2. Or by writing **cmd** in the File Explorer path and pressing EnterA screenshot of a computer

      Description automatically generated
2. Execute the command **npm install** and wait for it to install the needed packages.
3. Execute the command **npm run start** and wait for it to start the application.
   1. **NOTE: Leave the cmd Terminal open, as closing it will close the server**.
4. Access the **API Documentation** available at <http://localhost:5000/api-docs>.
5. Keep in mind that Tests can affect the state of the data on the server, to ensure you’re working with the base data, you can reset the server by closing the Terminal and repeating steps **3** and **5**
   1. An example of a test that can affect the server data is one that deletes an object on the server, running it a second time will fail as that object would no longer be available.
   2. If you’re testing with swagger (<http://localhost:5000/api-docs>), keep in mind that it might be necessary to refresh the page after restarting the server, to avoid errors due to old data/examples.

## Solution Skeleton and Authentication

Unzip the Skeleton.zip file. You must write your tests inside the methods in RecipeTests.cs (Task 1) and **CategoryTests.cs (Task 2)**. Be careful not to change the names of the methods.

You are provided with a **Setup** function. Its role is to login to the application and save 2 variables you must use for your tests:

* **client** – initialized RestClient ready to be used for your tests. The base url for the API endpoints is already saved. In your tests you need to only add the name of the endpoint as first parameter.
* **token** – initialized authentication token ready to be used for your tests. When you need to execute request to an endpoint that requires authentication you need to add the token to the header like this:



## CRUD Operations for Recipes Testing (150 Points)

### 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**, **ingredients**, **instructions, cookingTme** and **category** which must **not** be **null or empty**.
* **Response Assertions:**
  + Ensure 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:**
  + Ensure 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):**
  + Ensure each recipe has a **non-null**, **non-empty title** field.
  + Ensure each recipe's **ingredients** field is a **JSON array**.
  + Ensure each recipe's **instructions** field is a **JSON array**.
  + Ensure each recipe has a **non-null, non-empty cookingTime** field.
  + Ensure each recipe has a **non-null, non-empty servings** field.
  + Ensure each recipe has a **non-null, non-empty category** field**.**

### Get Recipe by Title (30 Points)

Write a unit test for the **Test\_GetRecipeByName**() method.

**Conditions**:

* **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:**
  + Verify that a recipe with the **title** "**Chocolate Chip Cookies**" exists in the retrieved recipes.
* **Recipe Fields Assertions:**
  + Ensure the "**Chocolate Chip Cookies**" recipe has the correct **cookingTime**
  + Ensure the "**Chocolate Chip Cookies**" recipe has the correct **servings**
  + Ensure the "**Chocolate Chip Cookies**" recipe has the correct number of **ingredients**.
  + Ensure the "**Chocolate Chip Cookies**" recipe has the correct number of **instructions**.

### Add Recipe (30 Points)

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

To create a new Recipe, you need to create a valid object with **title**, **ingredients**, **instructions**, **cookingTime**, **servings** and **category**. The **category** property must contain a valid **id** of one of the existing categories. To do that you need to retrieve the list of categories and set the **category** property to have the value of the **id** of one of the categories returned.

**Conditions**:

* **Get all Categories**
* **Create a new Recipe**
* **Response Assertions:**
  + The HTTP response status code should be **200 OK**.
  + The response content should **not** be **empty**
* **Retrieve the id of the created Recipe from the response**
* **Get the details of the Recipe**
* **Recipe Fields Assertions:**
  + Ensures the **title** of the newly added recipe matches the input value.
  + Ensure the **cookingTime** of the newly added recipe matches the input value.
  + Ensure the **servings** of the newly added recipe matches the input value.
  + Ensure the **category** is not empty.
    - The **id** of the category should match the input value (keep in mind that here the category is returned as an object, which has the **\_id** property you need).
  + Ensure the **ingredients** field is a **JSON array**.
    - The array should have the same **number of elements** as the input value for **ingredients**.
    - The **name** and **quantity** of each ingredient should be the same as the input values for **ingredients.**
  + Ensure the **instructions** field is a **JSON array**.
    - The array should have the same **number of elements** as the input value for **instructions**.
    - Ensure the **steps** in the instructions have the same values as the **steps** in the input values for **instructions.**

### Update Recipe (30 Points)

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

**Conditions**:

* **Get all Recipes**
* **Get Request Assertions:**
  + The HTTP response status code for the GET request should be **200 O**K.
  + The GET request response content should not be empty.
  + The Recipe with the title "**Spaghetti Carbonara**" should exist in the response.
* **Get the id of the Recipe**
* **Update** the **Recipe** withnew **title** and **servings**
* **Update Response Assertions:**
  + Ensure that the recipe update was successful with a **200 OK response**.
  + Ensure the response content for the **update** is **not** **empty**.
* **Update Recipe Fields Assertions:**
  + Ensure the recipe title was updated correctly to "**Spaghetti Bolognese**".
  + Verify 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**:

* **Get all Recipes**
* **Get Request Assertions:**
  + The HTTP response status code for the initial GET request should be **200 OK**.
  + The GET request response content should not be empty.
  + The recipe with the title "**Chicken Curry**" should exist in the response.
* **Get the Id of the recipe**
* **Delete the Recipe**
* **Delete Response Assertions:**
  + Ensure the recipe deletion was successful with an **HTTP 200 OK status**.
* **Post-Deletion Verification:** 
  + A subsequent GET request for the deleted recipe should return a response with content "null".

## 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

* **Method:** POST /category
* **Description:** A request is made to create a new category with the name "**Vegan Recipes**"
* **Expected Status Code:** 200 OK
* **Assertions:**
  + The HTTP response code should be **200 OK.**
  + The response body should contain a **non-null**, **non-empty** category ID (**\_id**).

#### Step 2: Get all categories

* **Method:** GET /category
* **Description:** Fetch all categories to ensure that the newly created category appears in the list of categories.
* **Expected Status Code:** 200 OK
* **Assertions:**
  + The HTTP response code should be **200 OK**.
  + The response content should **not be empty**.
  + The response should be a **JSON array**.
  + The array should contain at least one category.

#### Step 3: Get category by ID

* **Method:** GET /category/{categoryId}
* **Description:** Retrieve the category by the ID that was returned during the creation step.
* **Expected Status Code:** 200 OK
* **Assertions:**
  + The HTTP response code should be **200 OK.**
  + The response content should **not be empty**.
  + The returned category should have the **same ID** as the one created.
  + The category name should be "**Vegan Recipes**".

#### Step 4: Edit the category

* **Method:** PUT /category/{categoryId}
* **Description:** Update the category name to "**Healthy Vegan Recipes**" using a PUT request.
* **Expected Status Code:** 200 OK
* **Assertions:**
  + The HTTP response code should be **200 OK**.

#### Step 5: Verification (after edit):

* **Method**: GET /category/{categoryId}
* **Description:** Fetch the updated category to ensure the name has been changed.
* **Expected Status Code:** 200 OK
* **Assertions:**
  + The HTTP response code should be **200 OK.**
  + The response content should **not be empty.**
  + The category name should be updated to "**Healthy Vegan Recipes**".

#### Step 6: Delete the category

* **Method:** DELETE /category/{categoryId}
* **Description:** Delete the category by its ID using a DELETE request.
* **Expected Status Code:** 200 OK
* **Assertions:**
  + The HTTP response code should be **200 OK**.

#### Step 7: Verify the deleted category cannot be found

* **Method: GET /category/{categoryId}**
* **Description: Attempt to retrieve the deleted category to confirm that it has been removed.**
* **Assertions:**
  + The response content should be **“null”**.

## 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.