**QA Back-End Test Automation**

**Regular Exam**

The **BookStore** app focuses on providing functionality for managing **book-related data**, such as **books** 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>.

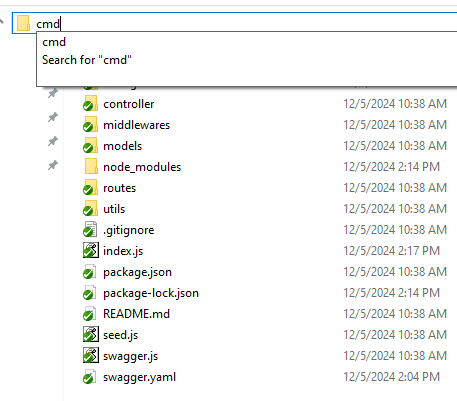
Follow these steps to get the application running locally.

1. **Download** the **BookStore**.zip file, which contains all the necessary files.
2. **Unzip** the **BookStore**.zip file into your preferred directory on your machine.

A red box with black text

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 Enter
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 a book on the server, running it a second time will fail as that book 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 BookTests.cs (Task 1) and **BookCategoryTests.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 Books Testing (150 Points)

### Get All Books (30 Points)

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

Conditions to Test:

* **Response Assertions:**
  + The HTTP response status code should be **200 OK**.
  + The response content should not be empty.
* **Data Structure Assertions:**
  + The response content should be a **JSON array**.
  + The JSON array should contain at **least one book**.
* **Book Fields Assertions (for each book):**
  + Each book's **title** should **not** be **null** or **empty**.
  + Each book's **author** should **not** be **null** or **empty**.
  + Each book's **description** should **not** be **null** or **empty**.
  + Each book’s **price** should **not** be **null** or **empty**.
  + Each book’s **pages** should **not** be **null** or **empty**.
  + Each book’s **category** should **not** be **null** or **empty**.

### Get Book by Title (30 Points)

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

Conditions to Test:

* **Response Assertions:**
  + The HTTP response status code should be **200 OK**.
  + The response content should **not** be **empty**.
* **Data Assertions:**
  + Verify that a book with the title "**The Great Gatsby**" is returned in the response.
* **Book Fields Assertions:**
  + The **author** of the book with the title "**The Great Gatsby**" should be "**F. Scott Fitzgerald**".

### Add Book (30 Points)

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

In order to create a new book you need to create a valid book object with **title**, **author**, **description**, **price**, **pages** 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 to Test:

* **Get all categories**
* **Create a new book**
* **Response Assertions:**
  + The HTTP response status code should be **200 OK**.
  + The response content should **not** be **empty**.
* **Retrieve the id of the created book from the response**
* **Get details of the book**
* **Response Assertions:**
  + The HTTP response status code should be **200 OK**.
  + The response content should **not** be **empty**.
* **Book Fields Assertions:**
  + The **title** in the response should match the input value.
  + The **author** in the response should match the input value.
  + The **price** should match the input value.
  + The **pages** should match the input value.
  + The **category** should not be 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).

### Update Book (30 Points)

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

Conditions to Test:

* **Get all books**
* **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 book with the title "**The Catcher in the Rye**" should exist in the response.
* **Get the id of the book.**
* **Update the book with new title and author.**
* **Update Request Assertions:**
  + The HTTP response status code for the PUT request should be **200 O**K.
  + The PUT request response content should not be empty.
* **Updated Fields Assertions:**
  + The updated title should be "**Updated Book Title**".
  + The updated author should be "**Updated Author**".

### Remove Book by Id (30 Points)

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

Conditions to Test:

* **Get all books**
* **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 book with the title "**To Kill a Mockingbird**" should exist in the response.
* **Get the id of the book.**
* **Delete the book**
* **Response Assertions:**
  + The HTTP response status code should be **200 OK** for successful deletion.
* **Get the details of the deleted book**
* **Response Assertions:**
  + The response content should be “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 **book 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 book category with the title "**Fictional Literature**".
* **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).
  + The title in the response should match the input value ("**Fictional Literature**").

#### Step 2: Get all categories

* **Method:** GET /category
* **Description:** Fetch all book 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**.
  + The new category should be present in the list, with the **correct ID**.

#### Step 3: Edit the category

* **Method:** PUT /category/{categoryId}
* **Description:** Update the category title to "**Updated Fictional Literature**" using a PUT request.
* **Expected Status Code:** 200 OK
* **Assertions:**
  + The HTTP response code should be **200 OK**.

#### Step 4 Verification (after edit):

* **Method**: GET /category/{categoryId}
* **Description:** Fetch the updated category to ensure the title 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 title should be updated to "**Updated Fictional Literature**".

#### Step 5: 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 6: 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.