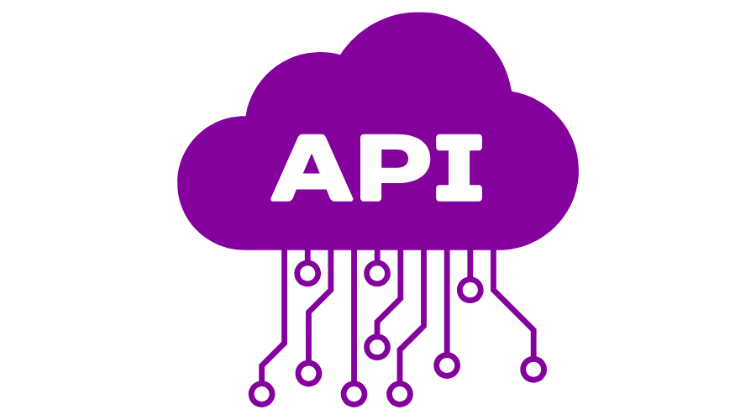
# Workshop: Basic API Functional Testing with NUnit and RestSharp



The **EShop** is a comprehensive **e-commerce application** with multiple components, including **user management**, **product management**, **order processing**, and more. Here's an overview of the key aspects and functionalities of the project:

### Project Overview

#### Technologies Used:

* **Node.js**: A JavaScript runtime used for server-side development.
* **Express.js**: A web application framework for Node.js, used to build the API.
* **MongoDB**: A NoSQL database used to store data.
* **Mongoose**: An ODM (Object Data Modeling) library for MongoDB and Node.js.
* **bcrypt**: A library to hash passwords for security.
* **dotenv**: A module to load environment variables from a .env file.



#### Key Functionalities:

* **User Management**: Handles user registration, login, and role-based access control.
* **Product Management**: Manages product information, including CRUD operations.
* **Order Processing**: Manages the order lifecycle, including creation, updating, and deletion.
* **Cart Management**: Handles shopping cart operations for users.
* **Coupon Management**: Manages discount coupons for promotional purposes.
* **Enquiries**: Handles customer enquiries and their statuses.
* **Blog Management**: Manages blog posts and categories.

### 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 **EShop.zip** file, which contains all the necessary files.
2. **Unzip** the **EShop.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>.

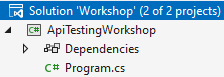
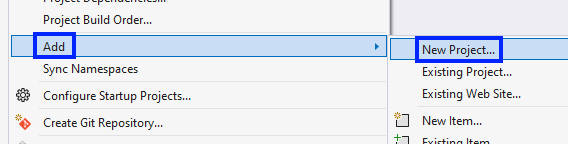
You will learn how to use **NUnit** and **RestSharp** to perform **basic functional testing** on API endpoints.

## Setting Up the Project:

### Create a New C# Project

Open Visual Studio and create a new C# Console Application project named ApiTestingWorkshop.

Right-click the solution in Solution Explorer and select **Add** 🡪 **New Project**.

 🡪

Choose **Unit Test Project** and name it ApiTests.

Your project should have this architecture for now:



### Install NUnit and RestSharp via NuGet

Right-click on the ApiTests project and select **Manage NuGet Packages**.

Search for and install the following packages:

* NUnit
* NUnit3TestAdapter
* Microsoft.NET.Test.Sdk
* RestSharp

### Configure the Project for Unit Testing

Ensure that the ApiTests project references NUnit and is set up to run tests. Check that the NUnit3TestAdapter is properly installed.

### Add Global Constants

Create new class GlobalConstants.cs. This class is a static utility class that contains essential **constants** and **methods** for the application. It includes:

* **BaseUrl:** A constant string that defines the API's base URL ("<http://localhost:5000/api>").
* **AuthenticateUser Method:** A method that handles user authentication by sending a POST request to the API. It returns an **authentication token** if successful, or fails the test with an error message if authentication fails.

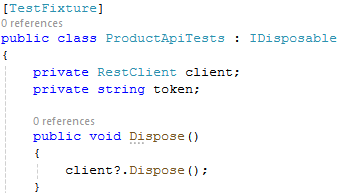
This class centralizes key **API configurations** and **authentication logic** for easy reuse across the application.



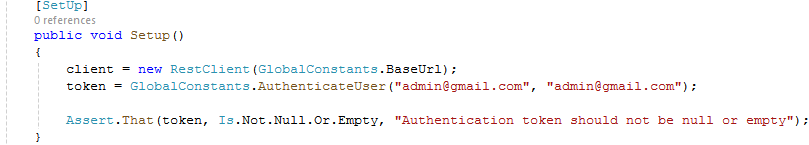
## CRUD Operations for Products Testing

Create a new test class named ProductApiTests.cs in the ApiTests project.

The ProductApiTests class is a test fixture, marked with the [TestFixture] attribute, designed to test the **Product API**. It includes a RestClient for making **API requests** and implements the IDisposable interface to **ensure proper cleanup of resources**. The Dispose() **method** disposes of the RestClient instance to prevent resource leaks, maintaining a clean test environment.

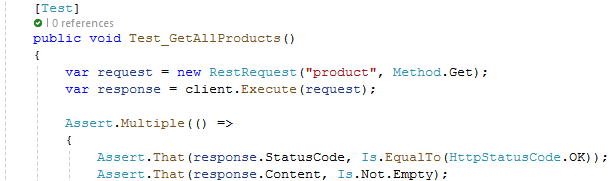


The Setup **method** is a pre-test initialization method that runs before each test. It creates a RestClient using the **base API URL** and authenticates the user by obtaining an **authentication token**. The method also includes an assertion to ensure that the **token is valid** (**not null or empty**), **setting up a secure and consistent environment for the tests**.



### GET Request Testing:

The Test\_GetAllProducts **method** is a unit test that verifies the functionality of the **API endpoint** responsible for retrieving **all products**. It sends a **GET request to the "product" endpoint**, confirms the response **status code** is **200 OK** and ensures the **response content is not empty**.

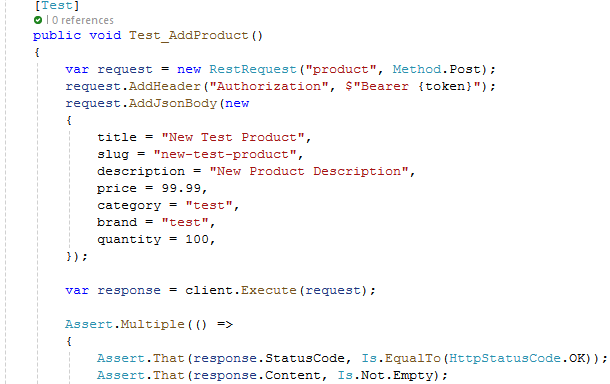


**Verifies** that specific product titles, such as "**Smartphone Alpha**" and "**Wireless Headphones**," **are included in the response**. Checks that the **prices** of these products match expected values. The test uses Assert.Multiple to **execute all assertions together**, ensuring comprehensive validation of the API's product retrieval feature.

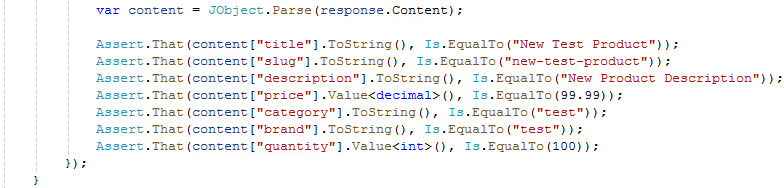


### POST Request Testing:

The Test\_AddProduct **method** is a unit test that validates the API's ability to add a new product. It sends a **POST request** to the "product" endpoint with the necessary product details, including title, slug, description, price, category, brand, and quantity. The request is authenticated using a Bearer token.Then **verifies the response**, ensuring that the **status code** is **200 OK** and the response content accurately reflects the data sent in the request.

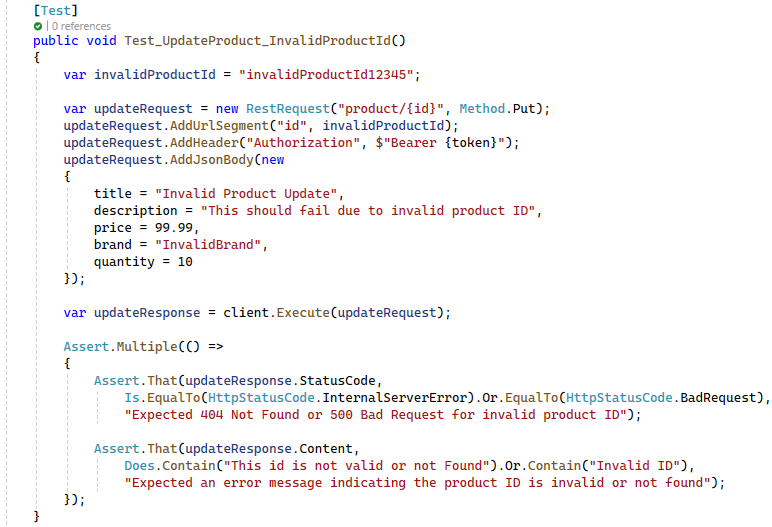


**After all it asserts multiple product attributes**, such as title, price, and quantity, to confirm that the product was correctly added to the system. This test ensures that the API properly processes and stores new product data.



### PUT Request Testing:

This test case, Test\_UpdateProduct\_InvalidProductId, is a **negative test** designed to validate the behavior of the API when an invalid or non-existent product ID is used to update a product. The test sends an HTTP PUT request with an invalid product ID and expects the API to respond with a **400 Bad Request** or **500 Internal Server Error**, depending on the server's error handling. Additionally, the test checks for an **appropriate error message**, such as "**This id is not valid or not Found**" in the response content, indicating that the provided product ID is invalid.



### DELETE Request Testing:

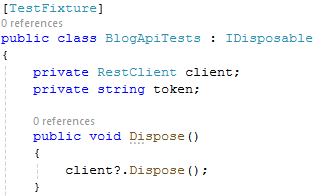
The Test\_DeleteProduct **method** is a unit test that validates the API's capability to **delete a specific product**. The test begins by **retrieving the list of products** and **searching for the product with the slug** "**electric-bike**". Once identified, it sends a **DELETE** **request** to **remove the product**, ensuring the request is **authenticated**. After confirming the deletion with a **200 OK** **status code**, the test performs an **additional check** by attempting to retrieve the deleted product to verify that it has indeed been removed from the database.



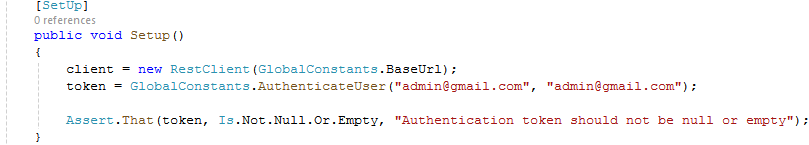
## CRUD Operations for Blogs Testing

Create a new test class named BlogApiTests.cs in the ApiTests **project**.

The BlogApiTestsclass is a test fixture, marked with the **[**TestFixture**]** attribute, designed to test the **Blog API**. It includes a RestClient for making **API requests** and implements the IDisposable interface to **ensure proper cleanup of resources**. The Dispose**()** **method** disposes of the RestClient instance to prevent resource leaks, maintaining a clean test environment.

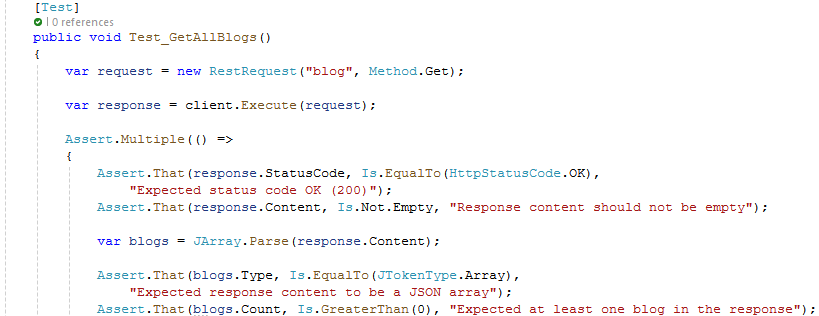


The **Setup** **method** is a pre-test initialization method that runs before each test. It creates a RestClient using the **base API URL** and authenticates the user by obtaining an **authentication token**. The method also includes an assertion to ensure that the **token is valid** (**not null or empty**), **setting up a secure and consistent environment for the tests**.



### GET Request Testing:

The Test\_GetAllBlogs method verifies the functionality of the API endpoint responsible for retrieving **all blog** posts. It begins by sending a **GET request** to the "**blog**" endpoint and then performs several assertions to **ensure** **the** **response is valid**. The test checks that the response status code is **200 OK** and that the content **is not empty**. It further validates that the response content is a JSON array **containing at least one blog** post.



For each blog in the array, the test asserts that critical fields such as **title**, **description**, **author**, and **category** **are** **present and not empty**. This ensures that the API returns **complete and correctly formatted blog data**.



### POST Request Testing:

The Test\_AddBlog **method** sends a **POST request** to the "**blog**" endpoint with the necessary blog details, including the **title**, **description**, and **category**, and uses a **Bearer token for authentication**. The test then checks that the response status code is **200 OK** and that the response **content is not empty**. It further asserts that the **returned blog data matches the input values** for the title, description, and category, and ensures that the author field is present and not empty.

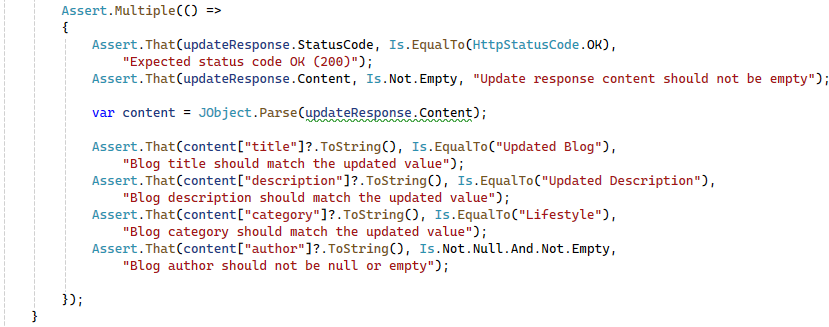


### PUT Request Testing:

The Test\_UpdateBlog **method** first **retrieves all blogs** with a **GET request** and identifies the blog titled "**10 Tips for a Healthier Lifestyle**" for updating. This title is included in seeded data. After ensuring the blog exists, the test extracts its ID and sends a **PUT request to update** theblog's **title**, **description**, and **category**, using a **Bearer token for authentication**.



The test then verifies that the response status code is **200 OK** and that the response content is not empty. It further **asserts** that the updated blog's title, description, and category **match the new values** provided, and that the author field remains valid.

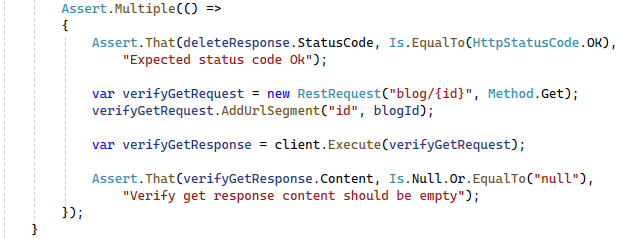


### DELETE Request Testing:

The Test\_DeleteBlog **method** begins by sending a **GET request** to retrieve all blogs, then identifies the blog titled "**The Evolution of Entertainment in the Digital Age**" for deletion. After confirming the blog exists, the test extracts its ID and sends a **DELETE request**, using a **Bearer token for authentication**.

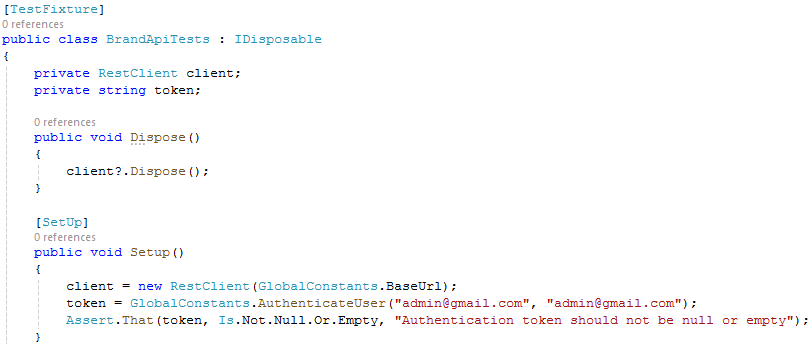


The test checks that the response **status code** is **200 OK**, indicating a **successful deletion**. The test attempts to retrieve the deleted blog again to **confirm that it has been removed**, ensuring the content is either **null or empty**.



## CRUD Operations for Brands Testing:

Create a new test class named BrandApiTests.cs in the ApiTests project. This class is a test fixture designed to test the Brand API, using a RestClient for API requests. It implements IDisposable to clean up resources and prevent leaks. The Setup method initializes the RestClient and **authenticates** the user **before each test**, ensuring a secure and consistent testing environment by **validating** the **authentication token**.



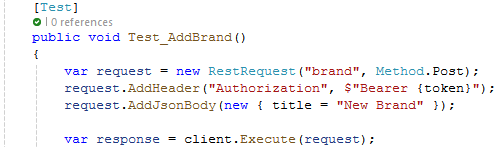
### GET Request Testing:

This method sends a **GET request** to the "**brand**" endpoint and performs several assertions to **ensure** **the** **response** is valid. The test checks that the status code is **200 OK** and that the **response content is not empty**. It confirms that the response is a **JSON array containing at least one brand** and **verifies that specific brand names**, such as "**TechCorp**" and "**GameMaster**," are included in the list. Additionally, the test ensures that each brand has a **non-null and non-empty ID and title**. The test also expects that **more than five brands are returned**.

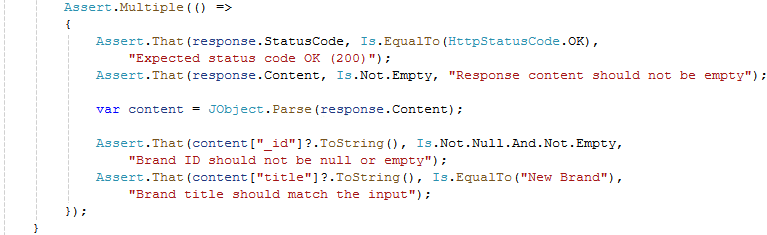


### POST Request Testing:

The Test\_AddBrand **method** sends a **POST request** to the "**brand**" endpoint, including a JSON body with the brand title "**New Brand**" and an **authorization token in the header**.

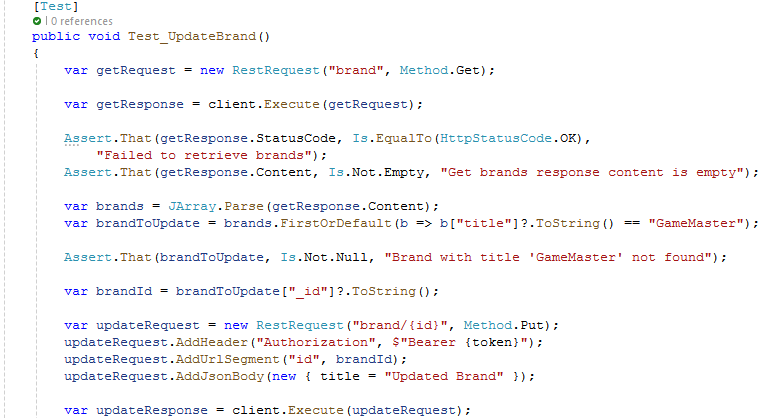


The test then checks that the **response status** code is **200 OK** and that the **response content** **is not empty**. It further asserts that the response contains a valid, **non-null brand ID** and that the **brand title** in the response **matches the input**.

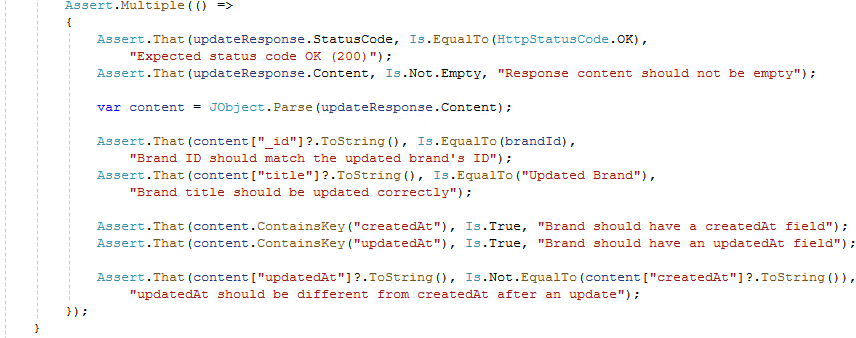


### PUT Request Testing:

The next method begins by sending a **GET request** to **retrieve all brands**, then identifies the brand titled "**GameMaster**" for **updating**. After confirming the brand exists, the test extracts its ID and **sends a PUT request** to update the brand's title to "**Updated Brand**," using a **Bearer token for authentication**.



The test checks that the **response status code** is **200 OK** and that the **response** **content is not empty**. It also verifies that the brand ID in the response matches the original ID, the title has been **updated correctly**, and that both createdAt and updatedAt fields are present, with the updatedAt value reflecting the **recent update**.

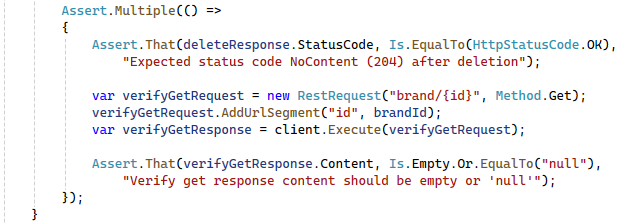


### DELETE Request Testing:

The Test\_DeleteBrand method begins by sending a **GET request** to **retrieve all brands** and identifies the brand titled "ViewTech" for deletion. After confirming that the brand exists, the test extracts its ID and sends a **DELETE request** to **remove the brand**, using a **Bearer token for authentication**. The test checks that the deletion response status code is **200 OK**, indicating the brand was **successfully deleted**.

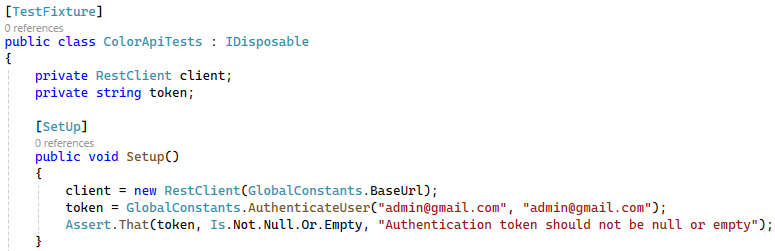


Finally, the test attempts to retrieve the **deleted brand to verify that it is no longer available**, ensuring the response content is **empty** or "**null**".



## CRUD Operations for Colors Testing:

Create a new test class named ColorApiTests.cs in the ApiTests project. This class is a test fixture designed to test the Color API, using a RestClient for API requests. It implements IDisposable to clean up resources and prevent leaks. The **Setup** method initializes the RestClient and **authenticates** the user **before each test**, ensuring a secure and consistent testing environment by **validating** the **authentication token**.



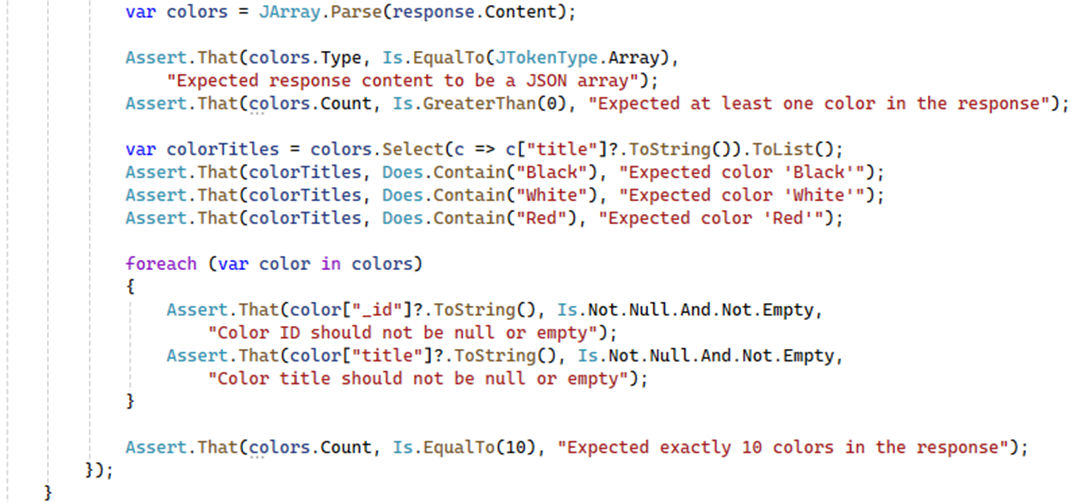
The Order **attribute** in this test class is being to specify the sequence in which the tests should be executed. Normally, **NUnit** (and most test frameworks) run tests in a random or **non-deterministic order**, since tests should ideally be independent of one another. However, in this case, we are going to use the [**Order**] attribute to explicitly **set the order of the tests**, which suggests that the tests may have some **dependencies** on one another, or that the order of execution is significant to the business logic being tested.

### GET Request Testing:

The Test\_GetAllColors **method** is a unit test that verifies the API's ability to **retrieve all colors**. It sends **a GET request** to the "**color**" endpoint and performs several assertions to ensure the **response is correct**. The test checks that the **response status code** is **200 OK** and that the response **content is not empty**.



It confirms that the content is a **JSON array** containing **at least one color** and specifically checks for the presence of common colors like "**Black**," "**White**," and "**Red**." The test also verifies that **each color has a valid**, **non-null ID and title**. Finally, it asserts that exactly **10 colors** are returned in the response, ensuring the completeness and accuracy of the API's color retrieval functionality.

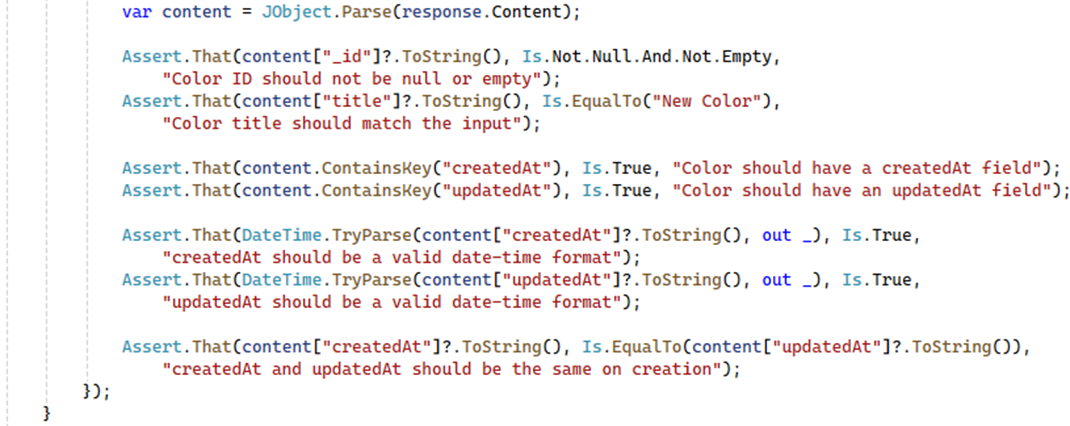


### POST Request Testing:

The next method sends a **POST request** to the "**color**" endpoint with a JSON body containing the **new color title**, "**New Color**" and includes an **authorization token in the header**. The test checks that the response status code is **200 OK** and that the response content is not empty.



It then parses the response to ensure the newly created color has a **valid**, **non-null ID** and that the **title matches the input**. The test also verifies that the response includes both createdAt and updatedAt fields, confirming they are in a valid date-time format and that these timestamps are **identical upon creation**.

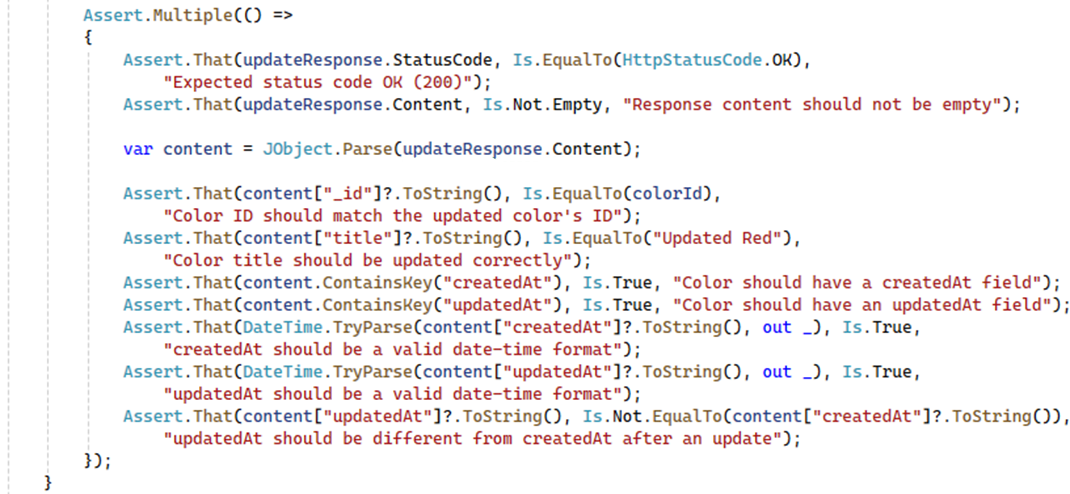


### PUT Request Testing:

The test begins by sending a **GET request** to **retrieve all available colors** and identifies the color titled "**Red**" for **updating**. After confirming that the color exists, the test extracts its ID and **sends a PUT request to update the color's title** to "**Updated Red**" using a **Bearer token for authentication**.



The test checks that the **response status code** is **200 OK** and that the **response content** is **not empty**. It further verifies that the color's ID remains the same, the title has been **updated correctly**, and both createdAt and updatedAt fields are present and in valid date-time format. The test also ensures that the updatedAt timestamp reflects the **recent update**, differing from the createdAt timestamp.

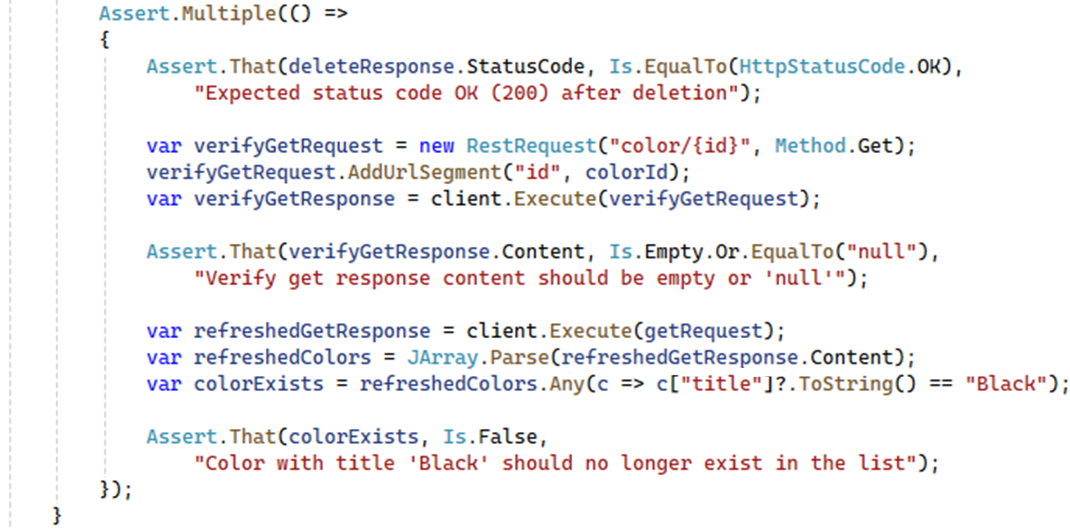
****

### DELETE Request Testing:

The Test\_DeleteColor **method** begins by sending a **GET request** to retrieve the list of colors and identifies the color titled "**Black**" for deletion. After confirming that the color **exists**, the test extracts its ID and **sends** a **DELETE** **request**, using a **Bearer token for authentication**.



The test checks that the response **status code** is **200 OK**, indicating the **deletion was successful**. To ensure the color has been properly deleted, the test **sends** a **GET** **request for the deleted color by ID**, verifying that the response content is **empty** or "**null**". Finally, the test retrieves the list of colors again to confirm that "**Black**" **no longer exists in the list**.



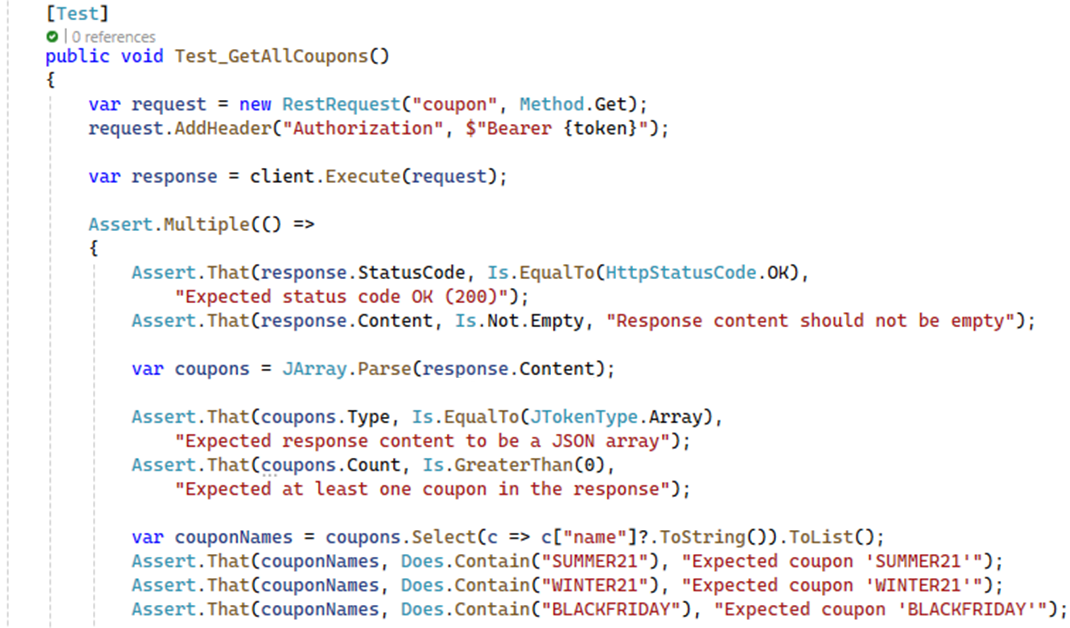
## CRUD Operations for Coupons Testing:

Create a new test class named CouponApiTests.cs in the ApiTests project. This class is a test fixture designed to test the Coupon API, using a RestClient for API requests. It implements IDisposable to clean up resources and prevent leaks. The **Setup** method initializes the RestClient and **authenticates** the user **before each test**, ensuring a secure and consistent testing environment by **validating** the **authentication token**.

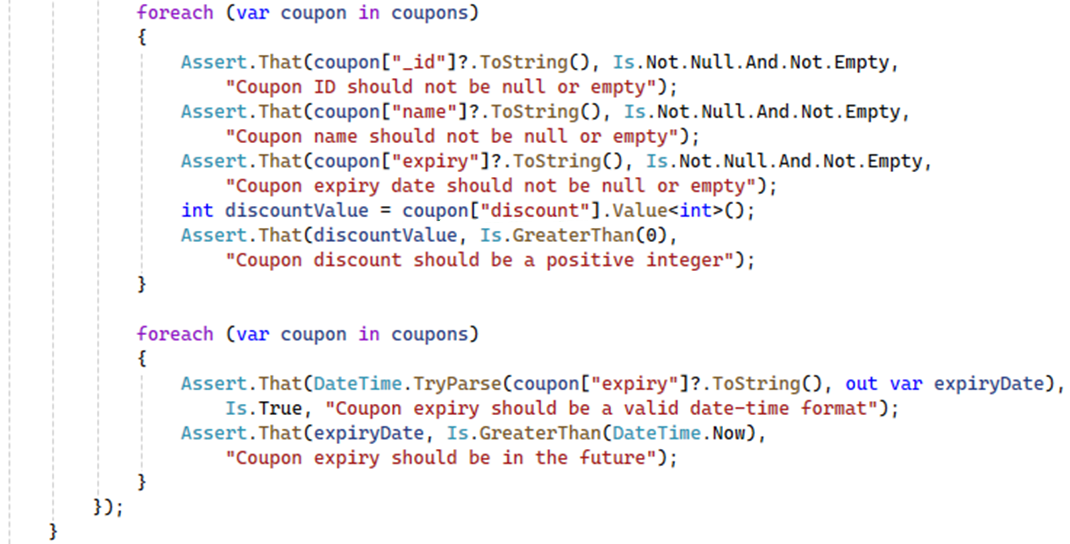


### GET Request Testing:

The Test\_GetAllCoupons **method** sends a **GET request** to the "**coupon**" **endpoint**, including an **authorization token in the header**, and performs several assertions to ensure the response is correct. The test checks that the **status code** is **200 OK** and that the **response content** is **not empty**. It confirms that the content is a **JSON array** containing at least one coupon and specifically checks for the presence of expected coupon names like "**SUMMER21**", "**WINTER21**", and "**BLACKFRIDAY**".



The test further verifies that **each coupon has a valid**, **non-null** **ID**, **name**, and **expiry date**, and that the **discount value** is a **positive integer**. Additionally, the test ensures that each coupon's expiry date is in a valid date-time format and is **set to a future date**. This comprehensive test ensures that the API correctly **retrieves** and **returns detailed coupon information**.



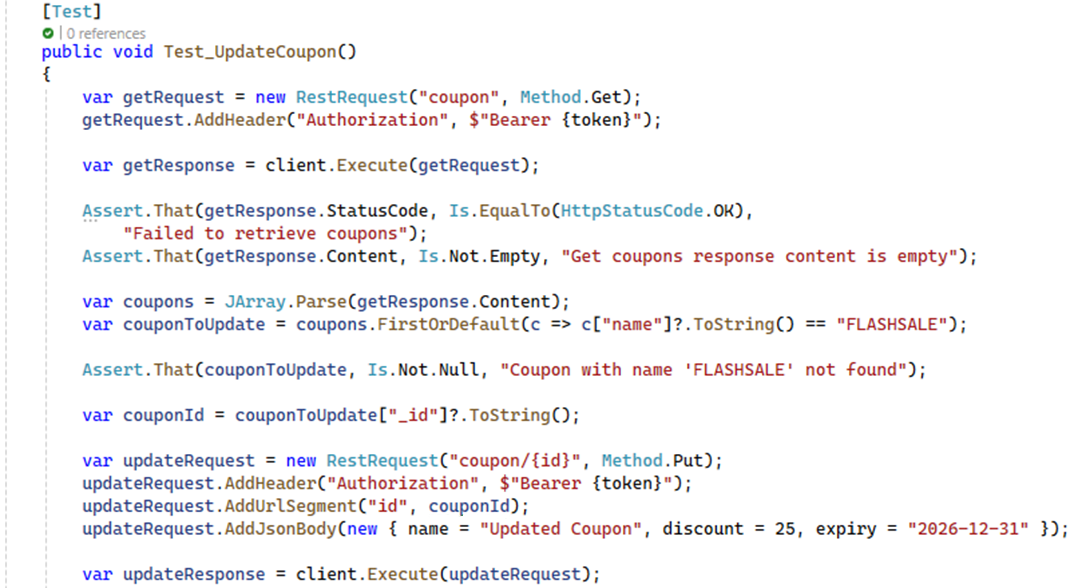
### POST Request Testing:

The test sends a **POST** **request** to the "**coupon**" **endpoint** with a **JSON body** containing the coupon's **name**, **discount value**, and **expiry date**. An **authorization token** is included in the **request header** to ensure proper authentication. The test then checks that the response status code is **200 OK** and that the **response content** is **not** **empty**. It parses the response to confirm that the newly created coupon has a **valid**, **non-null ID**, and that the **coupon's name**, **discount**, and **expiry** **date** **match** the **input values**. Additionally, the test verifies that the **expiry** **date** is in a **valid date-time** format and set in the future.

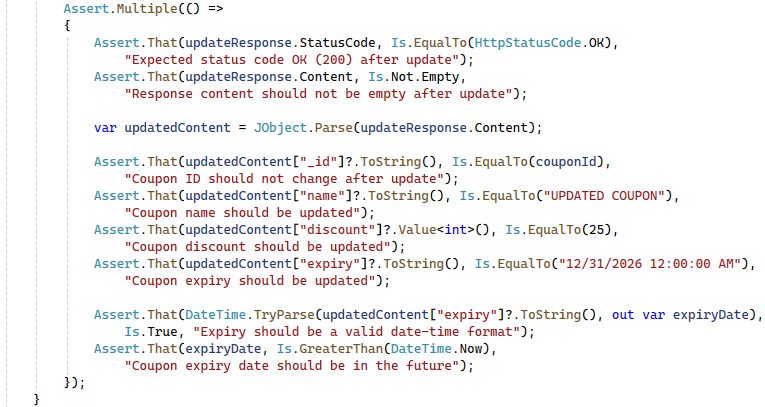


### PUT Request Testing:

The Test\_UpdateCoupon method begins by sending a **GET request** to **retrieve all coupons**, including an **authorization token** in the header for authentication. It then identifies the coupon titled "**FLASHSALE**" for updating. After confirming that the coupon exists, the test extracts its ID and sends a **PUT request to update** the coupon's **name**, **discount value**, and **expiry date**.



The test checks that the response status code is **200 OK** and that the response content is **not empty**. It verifies that the coupon's **ID remains unchanged**, ensuring that the update did not inadvertently create a new coupon. The test also confirms that the updated fields—**name**, **discount**, and **expiry date**—**match** the **new values** provided. Additionally, it ensures that the **expiry date** is in a **valid** **date-time** **format** and **set to a future date**.



### DELETE Request Testing:

The test starts by sending a **GET request** to retrieve the list of coupons, using a **Bearer token for authentication**. It identifies the coupon titled "**SPRING21**" for deletion. After confirming that the **coupon exists**, the test extracts its ID and **sends** a **DELETE request to remove the coupon**.



Following the deletion, the test performs **additional checks** to ensure that the coupon has been **successfully** **removed**. It sends a **GET request** for the specific coupon by its ID and also **retrieves the list of all coupons again**. The test then verifies that the **deleted coupon is no longer present in the list**, confirming that the **deletion** was **effective**.

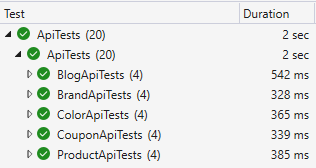


## Run the Tests:

In Visual Studio, open the **Test Explorer** (**Test** 🡪 **Windows** 🡪 **Test Explorer**).

Build the solution to discover the tests.

Run all tests to ensure they pass:



Explore the API and add your own tests.

Enjoy ☺