Exercise: API Testing with C#

This document defines the exercises and homework assignments for the

"QA Back-End Test Automation" Course @ SoftUni.

1. GitHub API endpoints

GitHub Issues provides the standard RESTful **API endpoints**, which you can access with HTTP client from **https://api.github.com**:

- **GET endpoints** – respond with **JSON** object as result.
  - **GET /repos/{user}/{repo}/issues** – returns the **issues** in given GitHub repo.
  - **GET /repos/{user}/{repo}/issues/{num}** – returns the specified **issue**.
  - **GET /repos/{user}/{repo}/issues/{num}/comments** – returns the **comments** for an issue.
  - **GET /repos/{user}/{repo}/issues/comments/{id}** – returns the specified **comment**.
- **POST / PATCH / DELETE endpoints** – all of them need **authentication**.
  - **POST /repos/{user}/{repo}/issues** – creates a new **issue**.
  - **PATCH /repos/{user}/{repo}/issues/{num}** – modifies the specified **issue**.
  - **POST /repos/{user}/{repo}/issues/{num}/comments** – creates a new **comment** for certain issue.
  - **PATCH /repos/{user}/{repo}/issues/comments/{id}** – modifies existing **comment**.
  - **DELETE /repos/{user}/{repo}/issues/comments/{id}** – deletes existing **comment**.

2. GitHub API Project

- **Projects:**
  - The solution consists of two projects: **RestSharpServices** and **TestGitHubApi.**
  - **RestSharpServices** is where you'll write **methods** that interact with the **GitHub API using RestSharp**.
  - **TestGitHubApi** is where you'll write tests to verify the functionality of the **RestSharpServices** methods.
- **Models:**
  - Located in the Models folder of the RestSharpServices project.
  - These classes represent the data that the GitHub API will send and receive.
  - These classes are POCOs (Plain Old CLR Objects) and are already provided to you.

Your main tasks will be to fill in the RestSharpServices.cs class with methods that use RestSharp to interact with GitHub's API and to create tests in the TestGitHubApi.cs file to ensure your code is working correctly.

3. Understanding Models

- Open the **Models folder** in the **RestSharpServices** project.
- Review the Issue.cs, Label.cs, and Comment.cs classes.
- Understand the properties of each class and how they map to the JSON structure returned by GitHub's API.

```
public class Label
{
    [JsonPropertyName("id")]
    2 references | 1/1 passing
    public int Id { get; set; }

    [JsonPropertyName("name")]
    2 references | 1/1 passing
    public string? Name { get; set; }
}
```

```
public class Comment
{
    [JsonPropertyName("id")]
    2 references | 1/1 passing
    public int Id { get; set; }

    [JsonPropertyName("body")]
    3 references | 2/2 passing
    public string ? Body { get; set; }
}
```

```
public class Issue
{

    [JsonPropertyName("id")]
    3 references | 3/3 passing
    public int Id { get; set; }

    [JsonPropertyName("number")]
    5 references | 3/3 passing
    public int Number { get; set; }

    [JsonPropertyName("title")]
    3 references | 2/2 passing
    public string? Title { get; set; }

    [JsonPropertyName("body")]
    0 references
    public string? Body { get; set; }
}
```

- If you want to know more about the additional properties of each one. Refer to GitHub API Documentation

4. Setting Up the RestSharp Client

- Open the **RestSharpServices.cs** file in the **RestSharpServices** project.
- Create a constructor in the **GitHubApiClient** class that initializes the **RestClient** with the base URL for GitHub's API and authentication details.

```
3 references
public class GitHubApiClient
{
    private RestClient client;

    1 reference
    public GitHubApiClient(string baseUrl, string username, string token)
    {
        var options = new RestClientOptions(baseUrl)
        {
            Authenticator = new HttpBasicAuthenticator(username, token)
        };

        this.client = new RestClient(options);
    }
```

- The **constructor** takes **three parameters:**
    - **baseUrl:** The root URL of the GitHub API. This is where all API requests will start from.
    - **username:** Your GitHub username.

- **token:** Your GitHub personal access token. This is used for authentication and allows you to interact with the GitHub API.

- **RestClientOptions is a configuration object for the RestClient:**
  - **baseUrl** is assigned to the **RestClientOptions,** which means that all requests made by the **RestClient** will start with **this URL.**
  - **Authenticator** is set to a new instance of **HttpBasicAuthenticator,** which is a built-in authenticator in **RestSharp** that handles **HTTP Basic Authentication.** It takes the **username** and **token** and uses them to authenticate your **requests to the GitHub API.**
- **RestClient Initialization:**
  - Creates a new RestClient using the options we just configured. The RestClient is what actually sends HTTP requests and receives responses.

5. TestGitHubApi project

- A dedicated place where **all the tests** related to the GitHub API functionality will reside.
- **References the RestSharpServices project** to gain access to the classes and methods you wish to test.
- Has a **TestGitHubApi class** where you will write your test cases.

```
0 references
public class TestGitHubApi
{
    private GitHubApiClient client;


    [SetUp]
    0 references
    public void Setup()
    {
        client = new GitHubApiClient("https://api.github.com/repos/testnakov/", "your_username", "your_token");
    }
```

- The Setup() method initializes the client field with a new instance of GitHubApiClient, configured with the base URL of the GitHub API and the repository you are targeting.
- **https://api.github.com/repos/testnakov/** is the base URL passed to the GitHubApiClient.

6. Implementing API Methods and Tests

Your assignment is to implement a set of methods that interact with the GitHub API and then write tests to verify their functionality. Focus on ensuring that each method correctly performs the desired action, such as retrieving, creating, or deleting data. You have two options for approaching this task:

- **Incremental Approach:** Write one method, then immediately write the corresponding test to validate it. Repeat this process for each method. This approach helps you focus on one piece of functionality at a time and may make debugging easier.
- **Bulk Implementation Approach:** Write all the methods first, and once you're done, move on to write the tests for each method. This might suit you if you prefer to stay in the 'development zone' and focus on testing afterward.

Get All Issues Method

- `public List<Issue> GetAllIssues(string repo)`

- Retrieves a list of all issues for a specified repository. This method constructs a GET request to the GitHub issues endpoint and returns the issues if the response has content.

```
1 reference | 1/1 passing
public List<Issue> ? GetAllIssues(string repo)
{
    var request = new RestRequest($"{repo}/issues");
    var response = client.Execute(request);
    return response.Content != null ? JsonSerializer.Deserialize<List<Issue>>(response.Content) : null;
}
```

Test Get All Issues from a Repo

- **`public void Test_GetAllIssuesFromARepo()`**

```
[Test, Order (1)]
0 references
public void Test_GetAllIssuesFromARepo()
{
    string repo = "test-nakov-repo";
    var issues = client.GetAllIssues(repo);

        Assert.That(issues, Has.Count.GreaterThan(1), "There should be more than one issue.");

    foreach (var issue in issues)
    {
        Assert.That(issue.Id, Is.GreaterThan(0), "Issue ID should be greater than 0.");
        Assert.That(issue.Number, Is.GreaterThan(0), "Issue Number should be greater than 0.");
        Assert.That(issue.Title, Is.Not.Empty, "Issue Title should not be empty.");
    }
}
```

Get Issue by Number

- **`public Issue GetIssueByNumber(string repo, int issueNumber)`**
- Fetches a single issue from a repository based on the issue number. It sends a GET request to the specific issue's endpoint and deserializes the response into an Issue object.

```
2 references | 2/2 passing
public Issue ? GetIssueByNumber(string repo, int issueNumber)
{
    var request = new RestRequest($"{repo}/issues/{issueNumber}", Method.Get);
    var response = client.Execute(request);
    return response.Content != null ? JsonSerializer.Deserialize<Issue>(response.Content) : null;

}
```

Test Get Issue by Valid Number

- **`public void Test_GetIssueByValidNumber()`**

```
[Test, Order (2)]
0 references
public void Test_GetIssueByValidNumber()
{
    string repo = "test-nakov-repo";
    int issueNumber = 1;
    var issue = client.GetIssueByNumber(repo, issueNumber);

    Assert.IsNotNull(issue, "The response should contain issue data.");
    Assert.That(issue.Id, Is.GreaterThan(0), "The issue ID should be a positive integer.");
    Assert.That(issue.Number, Is.EqualTo(issueNumber), "The issue number should match the requested number.");
}
```

Get All Labels for Issue

- `public List<Label> GetAllLabelsForIssue(string repo, int issueNumber)`
- Retrieves all labels for a particular issue within a repository. It sends a GET request to the issue's labels endpoint and deserializes the returned labels.

```
1 reference | 1/1 passing
public List<Label> ? GetAllLabelsForIssue(string repo, int issueNumber)
{
    var request = new RestRequest($"{repo}/issues/{issueNumber}/labels", Method.Get);
    var response = client.Execute(request);
    return response.Content != null ? JsonSerializer.Deserialize<List<Label>>(response.Content) : null;

}
```

Test Get All Labels for Issue

```
[Test, Order (3)]
0 references
public void Test_GetAllLabelsForIssue()
{
    string repo = "test-nakov-repo";
    int issueNumber = 6;
    var labels = client.GetAllLabelsForIssue(repo, issueNumber);

    // Perform your assertions
    Assert.That(labels.Count, Is.GreaterThan(0));

    foreach (var label in labels)
    {
        Assert.That(label.Id, Is.GreaterThan(0), "Label ID should be greater than 0.");
        Assert.That(label.Name, Is.Not.Empty, "Label Name should not be empty.");

        // Print the body of each label
        Console.WriteLine("Label: " + label.Id + " - Name: " + label.Name);
    }
}
```

Get All Comments for Issue

- `public List<Comment> GetAllCommentsForIssue(string repo, int issueNumber)`
- Gets all comments for a specified issue in a repository. The method sends a GET request to the issue's comments endpoint and returns a list of comments.

```
1 reference | 1/1 passing
public List<Comment> ? GetAllCommentsForIssue(string repo, int issueNumber)
{
    var request = new RestRequest($"{repo}/issues/{issueNumber}/comments", Method.Get);
    var response = client.Execute(request);
    return response.Content != null ? JsonSerializer.Deserialize<List<Comment>>(response.Content) : null;
}
```

Test Get All Comments for Issue

```
[Test, Order (4)]
0 references
public void Test_GetAllCommentsForIssue()
{
    string repo = "test-nakov-repo";
    int issueNumber = 6;
    var comments = client.GetAllCommentsForIssue(repo, issueNumber);

    // Perform your assertions
    Assert.That(comments.Count, Is.GreaterThan(0));

    foreach (var comment in comments)
    {
        Assert.That(comment.Id, Is.GreaterThan(0), "Comment ID should be greater than 0.");
        Assert.That(comment.Body, Is.Not.Empty, "Comment body should not be empty.");
        // Print the body of each label
        Console.WriteLine("Comment: " + comment.Id + " - Body: " + comment.Body);
    }
}
```

Create Issue

- **`public Issue CreateIssue(string repo, string title, string body)`**
- Creates a new issue in the given repository with a specified title and body. This method sends a POST request with the issue details and returns the created issue.

```
1 reference | 1/1 passing
public Issue ? CreateIssue(string repo, string title, string body)
{
    var request = new RestRequest($"{repo}/issues");
    request.AddJsonBody(new { title, body });
    var response = client.Execute(request, Method.Post);
    return response.Content != null ? JsonSerializer.Deserialize<Issue>(response.Content) : null;
}
```

Test Create Issue

```
[Test, Order(5)]
0 references
public void Test_CreateGitHubIssue()
{
    string repo = "test-nakov-repo";
    string expectedTitle = "Create Your Own Title";
    string body = "Give Some Description";
    var issue = client.CreateIssue(repo, expectedTitle, body);
    Assert.Multiple(() =>
    {
        Assert.That(issue.Id, Is.GreaterThan(0));
        Assert.That(issue.Number, Is.GreaterThan(0));
        Assert.That(issue.Title, Is.Not.Empty);
        Assert.That(issue.Title, Is.EqualTo(expectedTitle));
    });

    Console.WriteLine(issue.Number);
    lastCreatedIssueNumber = issue.Number;
}
```

Create Comment on GitHub Issue

- `public Comment CreateCommentOnGitHubIssue(string repo, int issueNumber, string` body)
- Posts a new comment to a specific issue in the repository. This method sends a POST request with the comment's body and returns the created comment.

```
1 reference | 0/1 passing
public Comment ? CreateCommentOnGitHubIssue(string repo, int issueNumber, string body)
{
    var request = new RestRequest($"{repo}/issues/{issueNumber}/comments");
    request.AddJsonBody(new { body });
    var response = client.Execute(request, Method.Post);
    return response.Content != null ? JsonSerializer.Deserialize<Comment>(response.Content) : null;
}
```

Test Create Comment on GitHub Issue

```
[Test, Order (7)]
0 references
public void Test_GetCommentById()
{
    // Arrange
    string repo = "test-nakov-repo";
    int commentId = 1954407608;

    // Act
    Comment comment = client.GetCommentById(repo, commentId);

    // Assert
    Assert.IsNotNull(comment, "Expected to retrieve a comment, but got null.");
    Assert.That(comment.Id, Is.EqualTo(commentId), "The retrieved comment ID should match the requested comment ID.");
    // Assert.That(comment.Body, Is.EqualTo(expectedBody), "The retrieved comment body should match the expected body.");
}
```

Note: To keep track of the state across multiple test runs within the same execution context you can use static fields.

- `lastCreatedIssueNumber`

- This field stores the issue number of the most recently created issue in the GitHub repository. We use this number to refer to the issue in subsequent tests that might need to add comments to it, close it, or perform other operations.

- **`lastCreatedCommentId`**
  - Similarly, this field holds the ID of the most recently created comment. If we create a comment in one test, we can use this ID to reference that comment in later tests, such as when we want to edit or delete the comment.

```
0 references
public class TestGitHubApi
{
    private GitHubApiClient client;
    private static int lastCreatedIssueNumber;
    private static int lastCreatedCommentId;
```

Get Comment by Id

- **`public Comment GetCommentById (string repo, int commentId)`**
- Retrieves a specific comment by its ID from a repository's issue. It sends a GET request to the comment's endpoint and returns the comment if found.

```
1 reference | 1/1 passing
public Comment ? GetCommentById (string repo, int commentId)
{
    var request = new RestRequest($"{repo}/issues/comments/{commentId}", Method.Get);
    var response = client.Execute(request);
    return response.Content != null ? JsonSerializer.Deserialize<Comment>(response.Content) : null;
}
```

Edit Comment on Git Hub Issue

- **`public Comment ? EditCommentOnGitHubIssue( string repo, int commentId, string newBody)`**
- Updates the body of an existing comment by comment ID on a GitHub issue. This method sends a PATCH request with the new body content and returns the updated comment.

```
1 reference | 1/1 passing
public Comment ? EditCommentOnGitHubIssue( string repo, int commentId, string newBody)
{
    var request = new RestRequest($"{repo}/issues/comments/{commentId}", Method.Patch);
    request.AddJsonBody(new { body = newBody });
    var response = client.Execute(request);
    return response.IsSuccessful && response.Content != null ? JsonSerializer.Deserialize<Comment>(response.Content) : null;
}
```

Test Edit Comment on Git Hub Issue

```
[Test, Order (8)]
0 references
public void Test_EditCommentOnGitHubIssue()
{
    string repo = "test-nakov-repo";
    int commentId = 1954407608;   // You can only edit comments that you've created.
    string newBody = "This is the updated text of the comment.";

    // Act
    var updatedComment = client.EditCommentOnGitHubIssue(repo, commentId, newBody);

    // Assert
    Assert.IsNotNull(updatedComment, "The updated comment should not be null.");
    Assert.That(updatedComment.Id, Is.EqualTo(commentId), "The updated comment ID should match the original comment ID.");
    Assert.That(updatedComment.Body, Is.EqualTo(newBody), "The updated comment text should match the new body text.");
}
```

Delete Comment on Git Hub Issue

- **`public bool DeleteCommentOnGitHubIssue(string repo, int commentId)`**
- Executes a deletion of a specified comment from a GitHub issue. The method sends a DELETE request to the GitHub API and returns a boolean indicating whether the deletion was successful

```
1 reference | 1/1 passing
public bool DeleteCommentOnGitHubIssue(string repo, int commentId)
{
    var request = new RestRequest($"{repo}/issues/comments/{commentId}", Method.Delete);
    var response = client.Execute(request);
    return response.IsSuccessful;
}
```

Test Delete Comment on Git Hub Issue

```
[Test, Order (9)]
0 references
public void Test_DeleteCommentOnGitHubIssue()
{
    // Arrange
    string repo = "test-nakov-repo";
    int commentId = lastCreatedCommentId; // Use an actual comment ID that you have permission to delete.

    // Act
    bool result = client.DeleteCommentOnGitHubIssue(repo, commentId);

    // Assert
    Assert.IsTrue(result, "The comment should be successfully deleted.");
}
```