API Testing with RestSharp and NUnit: A Student Guide

Welcome to this comprehensive guide on API testing using RestSharp and NUnit! In this tutorial, we'll focus on testing the FakeStoreAPI, an open API that mimics an e-commerce platform. By the end of this guide, you'll be able to write and execute various API tests for CRUD operations.

Table of Contents

Objectives

By the end of this tutorial, you will be able to:

- Set up a test project with RestSharp and NUnit
- Write tests for all CRUD operations (GET, POST, PUT, DELETE)
- Test APIs with query parameters
- Validate API responses with various assertion techniques
- Create a structured and maintainable test suite

Prerequisites

- Basic knowledge of C# programming
- Understanding of REST API concepts (endpoints, HTTP methods, status codes)
- Visual Studio 2022 or VSCode with .NET 6+ SDK installed
- Internet connection to access the FakeStore API

Setting Up Your Environment

Visual Studio 2022 Setup

1. **Open Visual Studio 2022**

2. **Create a new project**

- Go to `File` → `New` → `Project`
- Search for "NUnit Test Project" and select it
- Name your project `FakeStoreApiTests` and choose a location
- Select `.NET 6.0` (or later) as your target framework
- Click `Create`

3. **Install Required NuGet Packages**

- Right-click on your project in Solution Explorer
- Select `Manage NuGet Packages`
- Search for and install the following packages:
  - `RestSharp` (latest stable version)
  - `Newtonsoft.Json` (latest stable version)
  - NUnit packages should already be installed

VSCode with .NET CLI Setup

1. **Create a directory for your project**

```
mkdir FakeStoreApiTests
cd FakeStoreApiTests
```

2. **Create a new NUnit project using .NET CLI**

```
dotnet new nunit
```

3. **Add required packages**

```
dotnet add package RestSharp
dotnet add package Newtonsoft.Json
```

4. **Open the project in VSCode**

```
code .
```

Understanding the FakeStore API

The FakeStore API provides endpoints to simulate a simple e-commerce platform. We'll focus on the `/products` endpoint, which supports all CRUD operations:

- `GET /products` - Returns all products
- `GET /products/{id}` - Returns a specific product
- `POST /products` - Creates a new product

- `PUT /products/{id}` - Updates a product
- `DELETE /products/{id}` - Deletes a product

Products have the following structure:

```
{
  "id": 1,
  "title": "Product Name",
  "price": 109.95,
  "description": "Product description",
  "category": "category name",
  "image": "image URL",
  "rating": {
    "rate": 3.9,
    "count": 120
  }
}
```

Creating the Test Project

Let's start by creating some model classes and basic setup for our tests.

Step 1: Create Product Model Class

Create a new class file named `Product.cs`:

```
using Newtonsoft.Json;

namespace FakeStoreApiTests.Models
{
    public class Product
    {
        [JsonProperty("id")]
        public int Id { get; set; }

        [JsonProperty("title")]
        public string Title { get; set; }

        [JsonProperty("price")]
        public decimal Price { get; set; }

        [JsonProperty("description")]
        public string Description { get; set; }

        [JsonProperty("category")]
        public string Category { get; set; }

        [JsonProperty("image")]
        public string Image { get; set; }
```

```
        [JsonProperty("rating")]
        public Rating Rating { get; set; }
    }

    public class Rating
    {
        [JsonProperty("rate")]
        public double Rate { get; set; }

        [JsonProperty("count")]
        public int Count { get; set; }
    }
}
```

Step 2: Create Base Test Class

Create a file named `BaseApiTests.cs` for common setup:

```
using NUnit.Framework;
using RestSharp;

namespace FakeStoreApiTests
{
    public class BaseApiTests
    {
        protected RestClient Client;
        protected const string BaseUrl = "https://fakestoreapi.com";

        [SetUp]
        public void Setup()
        {
            Client = new RestClient(BaseUrl);
        }

        [TearDown]
        public void Cleanup()
        {
            Client.Dispose();
        }
    }
}
```

Writing Tests for GET Operations

Let's start by testing the GET operations to retrieve products from the API.

Test Case 1: Get All Products

Create a new file `ProductGetTests.cs`:

```
using System.Collections.Generic;
using System.Net;
using FakeStoreApiTests.Models;
using NUnit.Framework;
using RestSharp;

namespace FakeStoreApiTests
{
    [TestFixture]
    public class ProductGetTests : BaseApiTests
    {
        [Test]
        public void GetAllProducts_ReturnsSuccessStatusCode()
        {
            // Arrange
            var request = new RestRequest("products", Method.Get);

            // Act
            var response = Client.Execute(request);

            // Assert
            Assert.That(response.StatusCode, Is.EqualTo(HttpStatusCode.OK));
            Assert.That(response.Content, Is.Not.Empty);
        }

        [Test]
        public void GetAllProducts_ReturnsListOfProducts()
        {
            // Arrange
            var request = new RestRequest("products", Method.Get);

            // Act
            var response = Client.Execute<List<Product>>(request);

            // Assert
            Assert.That(response.Data, Is.Not.Null);
            Assert.That(response.Data.Count, Is.GreaterThan(0));

            // Verify the first product has required properties
            var firstProduct = response.Data[0];
            Assert.That(firstProduct.Id, Is.GreaterThan(0));
            Assert.That(firstProduct.Title, Is.Not.Empty);
            Assert.That(firstProduct.Price, Is.GreaterThan(0));
            Assert.That(firstProduct.Category, Is.Not.Empty);
        }
    }
}
```

Test Case 2: Get Product by ID

Add the following tests to `ProductGetTests.cs`:

```
[Test]
public void GetProductById_WithValidId_ReturnsProduct()
{
    // Arrange
    int productId = 1;
    var request = new RestRequest($"products/{productId}", Method.Get);

    // Act
    var response = Client.Execute<Product>(request);

    // Assert
    Assert.That(response.StatusCode, Is.EqualTo(HttpStatusCode.OK));
    Assert.That(response.Data, Is.Not.Null);
    Assert.That(response.Data.Id, Is.EqualTo(productId));
    Assert.That(response.Data.Title, Is.Not.Empty);
    Assert.That(response.Data.Description, Is.Not.Empty);
    Assert.That(response.Data.Price, Is.GreaterThan(0));
}

[Test]
public void GetProductById_WithInvalidId_ReturnsNotFound()
{
    // Arrange
    int invalidProductId = 999999; // Assuming this ID doesn't exist
    var request = new RestRequest($"products/{invalidProductId}", Method.Get);

    // Act
    var response = Client.Execute(request);

    // Assert
    // Note: FakeStoreAPI returns empty string with 200 OK for non-existent items,
    // but in a real API you might expect 404 Not Found
    Assert.That(response.StatusCode, Is.EqualTo(HttpStatusCode.OK));
    Assert.That(response.Content, Is.EqualTo(""));
}
```

Test Case 3: Get Product Categories

```
[Test]
public void GetProductCategories_ReturnsAllCategories()
{
    // Arrange
    var request = new RestRequest("products/categories", Method.Get);

    // Act
    var response = Client.Execute<List<string>>(request);

    // Assert
    Assert.That(response.StatusCode, Is.EqualTo(HttpStatusCode.OK));
    Assert.That(response.Data, Is.Not.Null);
```

```
    Assert.That(response.Data.Count, Is.GreaterThan(0));
}

[Test]
public void GetProductsByCategory_ReturnsProductsInCategory()
{
    // Arrange
    string category = "electronics"; // Known category in FakeStoreAPI
    var request = new RestRequest($"products/category/{category}", Method.Get);

    // Act
    var response = Client.Execute<List<Product>>(request);

    // Assert
    Assert.That(response.StatusCode, Is.EqualTo(HttpStatusCode.OK));
    Assert.That(response.Data, Is.Not.Null);
    Assert.That(response.Data.Count, Is.GreaterThan(0));

    // Verify all returned products are in the requested category
    foreach (var product in response.Data)
    {
        Assert.That(product.Category, Is.EqualTo(category));
    }
}
```

Writing Tests for POST Operations

Now let's test creating new products with POST requests.

Create a new file `ProductPostTests.cs`:

```
using System.Net;
using FakeStoreApiTests.Models;
using NUnit.Framework;
using RestSharp;

namespace FakeStoreApiTests
{
    [TestFixture]
    public class ProductPostTests : BaseApiTests
    {
        [Test]
        public void CreateProduct_WithValidData_ReturnsCreatedProduct()
        {
            // Arrange
            var newProduct = new Product
            {
                Title = "Test Product",
                Price = 99.99m,
                Description = "This is a test product created by API test",
                Category = "electronics",
```

```
                Image = "https://fakestoreapi.com/img/81QpkIctqPL._AC_SX679_.jpg",
                Rating = new Rating { Rate = 4.5, Count = 10 }
            };

            var request = new RestRequest("products", Method.Post);
            request.AddJsonBody(newProduct);

            // Act
            var response = Client.Execute<Product>(request);

            // Assert
            Assert.That(response.StatusCode, Is.EqualTo(HttpStatusCode.OK));
            Assert.That(response.Data, Is.Not.Null);
            Assert.That(response.Data.Id, Is.GreaterThan(0)); // FakeStoreAPI
assigns an ID
            Assert.That(response.Data.Title, Is.EqualTo(newProduct.Title));
            Assert.That(response.Data.Price, Is.EqualTo(newProduct.Price));
            Assert.That(response.Data.Description,
Is.EqualTo(newProduct.Description));
            Assert.That(response.Data.Category, Is.EqualTo(newProduct.Category));
        }

        [Test]
        public void CreateProduct_WithMissingRequiredFields_HandlesMissingData()
        {
            // Arrange
            var incompleteProduct = new
            {
                Title = "Incomplete Product"
                // Deliberately missing other fields
            };

            var request = new RestRequest("products", Method.Post);
            request.AddJsonBody(incompleteProduct);

            // Act
            var response = Client.Execute<Product>(request);

            // Assert
            // Note: FakeStoreAPI is lenient with missing fields, but a real API
might return 400 Bad Request
            Assert.That(response.StatusCode, Is.EqualTo(HttpStatusCode.OK));
            Assert.That(response.Data, Is.Not.Null);
            Assert.That(response.Data.Id, Is.GreaterThan(0));
            Assert.That(response.Data.Title, Is.EqualTo(incompleteProduct.Title));
        }

        [Test]
        public void CreateProduct_WithInvalidPrice_HandlesInvalidData()
        {
            // Arrange
            var invalidProduct = new
            {
                Title = "Invalid Product",
```

```
                Price = "not-a-number", // Invalid price
                Description = "This product has an invalid price",
                Category = "electronics"
            };

            var request = new RestRequest("products", Method.Post);
            request.AddJsonBody(invalidProduct);

            // Act
            var response = Client.Execute(request);

            // Note: FakeStoreAPI might not validate properly, but we're testing
our approach
            // A real API would likely return 400 Bad Request for invalid data

            // Assert
            // We'll just check the response status rather than making specific
assertions
            // about how the API handles invalid data
            Assert.That(response.IsSuccessful, Is.True);
        }
    }
}
```

Writing Tests for PUT Operations

Let's test updating existing products using PUT requests.

Create a new file `ProductPutTests.cs`:

```
using System.Net;
using FakeStoreApiTests.Models;
using NUnit.Framework;
using RestSharp;

namespace FakeStoreApiTests
{
    [TestFixture]
    public class ProductPutTests : BaseApiTests
    {
        private int existingProductId;
        private Product originalProduct;

        [SetUp]
        public void TestSetup()
        {
            // Get an existing product to update
            existingProductId = 1;
            var getRequest = new RestRequest($"products/{existingProductId}",
Method.Get);
            var getResponse = Client.Execute<Product>(getRequest);
```

```
            originalProduct = getResponse.Data;

            Assert.That(originalProduct, Is.Not.Null, "Setup failed: Could not
retrieve product to update");
        }

        [Test]
        public void UpdateProduct_WithValidData_ReturnsUpdatedProduct()
        {
            // Arrange
            var updatedProduct = new Product
            {
                Title = "Updated Product Title",
                Price = 199.99m,
                Description = "This is an updated product description",
                Category = originalProduct.Category,
                Image = originalProduct.Image
            };

            var request = new RestRequest($"products/{existingProductId}",
Method.Put);
            request.AddJsonBody(updatedProduct);

            // Act
            var response = Client.Execute<Product>(request);

            // Assert
            Assert.That(response.StatusCode, Is.EqualTo(HttpStatusCode.OK));
            Assert.That(response.Data, Is.Not.Null);
            Assert.That(response.Data.Id, Is.EqualTo(existingProductId));
            Assert.That(response.Data.Title, Is.EqualTo(updatedProduct.Title));
            Assert.That(response.Data.Price, Is.EqualTo(updatedProduct.Price));
            Assert.That(response.Data.Description,
Is.EqualTo(updatedProduct.Description));
        }

        [Test]
        public void UpdateProduct_WithPartialData_UpdatesOnlyProvidedFields()
        {
            // Arrange
            var partialUpdate = new
            {
                Title = "Partially Updated Product"
                // Only updating the title
            };

            var request = new RestRequest($"products/{existingProductId}",
Method.Put);
            request.AddJsonBody(partialUpdate);

            // Act
            var response = Client.Execute<Product>(request);

            // Assert
```

```
            Assert.That(response.StatusCode, Is.EqualTo(HttpStatusCode.OK));
            Assert.That(response.Data, Is.Not.Null);
            Assert.That(response.Data.Id, Is.EqualTo(existingProductId));
            Assert.That(response.Data.Title, Is.EqualTo(partialUpdate.Title));

            // FakeStoreAPI might replace other fields with defaults
            // or keep existing values depending on implementation
        }
    }
}
```

Writing Tests for DELETE Operations

Now let's test deleting products.

Create a new file `ProductDeleteTests.cs`:

```
using System.Net;
using NUnit.Framework;
using RestSharp;

namespace FakeStoreApiTests
{
    [TestFixture]
    public class ProductDeleteTests : BaseApiTests
    {
        [Test]
        public void DeleteProduct_WithValidId_ReturnsSuccessStatus()
        {
            // Arrange
            int productId = 6; // Use an existing product ID
            var request = new RestRequest($"products/{productId}", Method.Delete);

            // Act
            var response = Client.Execute(request);

            // Assert
            Assert.That(response.StatusCode, Is.EqualTo(HttpStatusCode.OK));

            // Note: The FakeStoreAPI doesn't actually delete products
            // It just simulates the deletion and returns success
            // In a real API test, you might verify the product no longer exists
        }

        [Test]
        public void DeleteProduct_ThenGetProduct_VerifyProductIsDeleted()
        {
            // Arrange
            int productId = 7; // Use an existing product ID

            // Delete product
```

```
            var deleteRequest = new RestRequest($"products/{productId}",
Method.Delete);
            var deleteResponse = Client.Execute(deleteRequest);
            Assert.That(deleteResponse.StatusCode, Is.EqualTo(HttpStatusCode.OK));

            // Try to get the deleted product
            var getRequest = new RestRequest($"products/{productId}", Method.Get);
            var getResponse = Client.Execute(getRequest);

            // Assert
            // Note: Since FakeStoreAPI doesn't actually delete records,
            // this test is demonstrative - in a real API, you might
            // expect 404 Not Found or an empty response
            Assert.That(getResponse.StatusCode, Is.EqualTo(HttpStatusCode.OK));

            // In a real test, you might assert:
            // Assert.That(getResponse.StatusCode,
Is.EqualTo(HttpStatusCode.NotFound));
        }

        [Test]
        public void DeleteProduct_WithInvalidId_HandlesNonExistentProduct()
        {
            // Arrange
            int invalidProductId = 999999; // Assuming this ID doesn't exist
            var request = new RestRequest($"products/{invalidProductId}",
Method.Delete);

            // Act
            var response = Client.Execute(request);

            // Assert
            // Note: FakeStoreAPI might return success for non-existent IDs,
            // but a real API might return 404 Not Found

            // For testing purposes, we'll just check the response status
            Assert.That(response.StatusCode, Is.EqualTo(HttpStatusCode.OK));
        }
    }
}
```

Testing Query Parameters

The FakeStore API supports various query parameters like `limit` and `sort`. Let's test those.

Create a new file `ProductQueryTests.cs`:

```
using System.Collections.Generic;
using System.Net;
using FakeStoreApiTests.Models;
using NUnit.Framework;
```

```
using RestSharp;

namespace FakeStoreApiTests
{
    [TestFixture]
    public class ProductQueryTests : BaseApiTests
    {
        [Test]
        public void GetProducts_WithLimitParameter_ReturnsLimitedResults()
        {
            // Arrange
            int limit = 5;
            var request = new RestRequest("products", Method.Get);
            request.AddQueryParameter("limit", limit.ToString());

            // Act
            var response = Client.Execute<List<Product>>(request);

            // Assert
            Assert.That(response.StatusCode, Is.EqualTo(HttpStatusCode.OK));
            Assert.That(response.Data, Is.Not.Null);
            Assert.That(response.Data.Count, Is.EqualTo(limit));
        }

        [Test]
        public void
GetProducts_WithLimitAndSortParameters_ReturnsLimitedSortedResults()
        {
            // Arrange
            int limit = 3;
            string sort = "desc"; // Sort in descending order
            var request = new RestRequest("products", Method.Get);
            request.AddQueryParameter("limit", limit.ToString());
            request.AddQueryParameter("sort", sort);

            // Act
            var response = Client.Execute<List<Product>>(request);

            // Assert
            Assert.That(response.StatusCode, Is.EqualTo(HttpStatusCode.OK));
            Assert.That(response.Data, Is.Not.Null);
            Assert.That(response.Data.Count, Is.EqualTo(limit));

            // Verify products are sorted (assuming sorting by id with desc)
            for (int i = 0; i < response.Data.Count - 1; i++)
            {
                Assert.That(response.Data[i].Id,
Is.GreaterThanOrEqualTo(response.Data[i + 1].Id));
            }
        }

        [TestCase(0)]
        [TestCase(-1)]
        public void GetProducts_WithInvalidLimit_HandlesInvalidParameter(int
```

```
invalidLimit)
        {
            // Arrange
            var request = new RestRequest("products", Method.Get);
            request.AddQueryParameter("limit", invalidLimit.ToString());

            // Act
            var response = Client.Execute(request);

            // Assert
            // Note: The FakeStoreAPI may handle invalid parameters differently
            // than a real API. A real API might return 400 Bad Request.

            // Just verifying we get a response
            Assert.That(response.StatusCode, Is.EqualTo(HttpStatusCode.OK));
        }
    }
}
```

Advanced Testing Techniques

Let's add some more advanced tests to demonstrate additional techniques.

Testing Response Times

Create a new file `PerformanceTests.cs`:

```
using System.Diagnostics;
using NUnit.Framework;
using RestSharp;

namespace FakeStoreApiTests
{
    [TestFixture]
    public class PerformanceTests : BaseApiTests
    {
        [Test]
        public void GetAllProducts_ResponseTimeIsAcceptable()
        {
            // Arrange
            var request = new RestRequest("products", Method.Get);
            var stopwatch = new Stopwatch();

            // Act
            stopwatch.Start();
            var response = Client.Execute(request);
            stopwatch.Stop();

            // Assert
            Assert.That(response.IsSuccessful, Is.True);
```

```
            // Assuming an acceptable response time of 1 second
            // Adjust based on API performance expectations
            Assert.That(stopwatch.ElapsedMilliseconds, Is.LessThan(1000),
                "API response took longer than 1 second");
        }

        [Test]
        public void MultipleRequests_AverageResponseTimeIsAcceptable()
        {
            // Arrange
            var request = new RestRequest("products", Method.Get);
            var stopwatch = new Stopwatch();
            int requestCount = 5;
            long totalTime = 0;

            // Act
            for (int i = 0; i < requestCount; i++)
            {
                stopwatch.Restart();
                var response = Client.Execute(request);
                stopwatch.Stop();

                Assert.That(response.IsSuccessful, Is.True);
                totalTime += stopwatch.ElapsedMilliseconds;
            }

            long averageTime = totalTime / requestCount;

            // Assert
            // Assuming an acceptable average response time of 500ms
            Assert.That(averageTime, Is.LessThan(500),
                $"Average API response time ({averageTime}ms) exceeded
threshold");
        }
    }
}
```

Testing Headers and Status Code Validation

Create a new file `HeaderTests.cs`:

```
using System.Linq;
using System.Net;
using NUnit.Framework;
using RestSharp;

namespace FakeStoreApiTests
{
    [TestFixture]
    public class HeaderTests : BaseApiTests
    {
```

```
        [Test]
        public void Request_HasProperContentType()
        {
            // Arrange
            var request = new RestRequest("products", Method.Get);

            // Act
            var response = Client.Execute(request);

            // Assert
            Assert.That(response.IsSuccessful, Is.True);

            // Check that response has Content-Type header and it's
application/json
            var contentTypeHeader = response.ContentType;
            Assert.That(contentTypeHeader, Does.Contain("application/json"));
        }

        [Test]
        public void Request_WithCustomHeaders_SendsHeadersCorrectly()
        {
            // Arrange
            var request = new RestRequest("products", Method.Get);
            request.AddHeader("Accept", "application/json");
            request.AddHeader("X-Custom-Header", "TestValue");

            // Act
            var response = Client.Execute(request);

            // Assert
            Assert.That(response.IsSuccessful, Is.True);

            // Note: We can't easily verify that the headers were sent correctly
            // without server-side logging, but we can verify that the request
succeeded
        }
    }
}
```

Running Your Tests

Running Tests in Visual Studio 2022

1. Open the Test Explorer window by going to `Test` → `Test Explorer`
2. Click on `Run All Tests` or run individual tests by clicking on them

Running Tests Using .NET CLI (VSCode)

1. Open a terminal in your project directory
2. Run the following command:

```
dotnet test
```

Conclusion

Congratulations! You've now learned how to:

- Set up a test project with RestSharp and NUnit
- Create model classes to represent API responses
- Write tests for all CRUD operations (GET, POST, PUT, DELETE)
- Test API endpoints with query parameters
- Implement advanced testing techniques like performance testing and retry logic

This knowledge provides you with a solid foundation for API testing in real-world projects. Remember that while the FakeStore API is a great practice environment, real APIs may have different behaviors, especially around authentication, validation, and error handling.

Additional Resources

- RestSharp Documentation
- NUnit Documentation
- FakeStore API Documentation
- HTTP Status Codes
- REST API Best Practices