Chapter 6: ASP.NET 9 Web API Unit Testing with xUnit

6.1 Introduction

Unit testing and integration testing are essential practices for ensuring the reliability, maintainability, and correctness of your ASP.NET Core Web APIs. In this chapter, we focus on using xUnit along with the `WebApplicationFactory` and an in-memory database provider to build a comprehensive test suite for a sample `TodoApi` application.

Key objectives:

- Set up test projects for unit and integration tests.
- Write tests covering all CRUD operations and validation rules.
- Verify HTTP status codes, response content, and error scenarios.
- Utilize best practices such as the Arrange-Act-Assert (AAA) pattern.

6.2 Setting Up Test Projects

To begin, create a new test project alongside your Web API project:

```
cd src/TodoApi
dotnet new xunit -n TodoApi.Tests
cd TodoApi.Tests
dotnet add reference ../TodoApi/TodoApi.csproj
dotnet add package Microsoft.AspNetCore.Mvc.Testing
dotnet add package Microsoft.EntityFrameworkCore.InMemory
```

This sets up:

- **xUnit** for the test framework.
- **Microsoft.AspNetCore.Mvc.Testing** for `WebApplicationFactory<TEntryPoint>`.
- **Microsoft.EntityFrameworkCore.InMemory** for the in-memory database provider.

6.3 Configuring the In-Memory Database

In your test project, create a custom factory to configure the in-memory database:

```
public class CustomWebApplicationFactory<TStartup> : WebApplicationFactory<TSt
{
    protected override void ConfigureWebHost(IWebHostBuilder builder)
```

```
    {
        builder.ConfigureServices(services =>
        {
            // Remove the existing DbContext registration
            var descriptor = services.Single(
                d => d.ServiceType == typeof(DbContextOptions<TodoContext>));
            services.Remove(descriptor);

            // Add a new DbContext using InMemory provider
            services.AddDbContext<TodoContext>(options =>
            {
                options.UseInMemoryDatabase("InMemoryTodoTest");
            });

            // Ensure database is created
            var sp = services.BuildServiceProvider();
            using var scope = sp.CreateScope();
            var db = scope.ServiceProvider.GetRequiredService<TodoContext>();
            db.Database.EnsureCreated();
        });
    }
}
```

This factory:

- Removes the production DbContext registration.
- Registers an in-memory database named `InMemoryTodoTest`.
- Ensures the database is created and clean before tests run.

6.4 Writing Tests in `TodoApiTests.cs`

Create `TodoApiTests.cs` in the `TodoApi.Tests` project. We'll cover:

- Create Todo (with validation)
- Get All Todos
- Get Todo by ID
- Update Todo
- Delete Todo
- Validation rules (empty title, max length)

```
public class TodoApiTests : IClassFixture<CustomWebApplicationFactory<Program>>
```

```
[Fact]
public async Task CreateTodo_EmptyTitle_ReturnsBadRequest()
{
    // Arrange
    var newTodo = new { Title = "", IsComplete = false };

    // Act
    var response = await _client.PostAsJsonAsync("/api/todos", newTodo);

    // Assert
    response.StatusCode.Should().Be(HttpStatusCode.BadRequest);
}

[Fact]
public async Task GetAllTodos_ReturnsList()
{
    // Arrange
    await SeedTodo("Seed1");
    await SeedTodo("Seed2");
```

```
        // Act
        var response = await _client.GetAsync("/api/todos");
        var todos = await response.Content.ReadFromJsonAsync<List<TodoItem>>()

        // Assert
        response.StatusCode.Should().Be(HttpStatusCode.OK);
        todos.Should().HaveCountGreaterOrEqualTo(2);
    }

    [Fact]
    public async Task GetTodoById_ReturnsCorrectTodo()
    {
        // Arrange
        var seeded = await SeedTodo("ByIdTest");

        // Act
        var response = await _client.GetAsync($"/api/todos/{seeded.Id}");
        var todo = await response.Content.ReadFromJsonAsync<TodoItem>();

        // Assert
        response.StatusCode.Should().Be(HttpStatusCode.OK);
        todo.Title.Should().Be(seeded.Title);
    }

    [Fact]
    public async Task UpdateTodo_ReturnsNoContent()
    {
        // Arrange
        var seeded = await SeedTodo("ToUpdate");
        var update = new { Id = seeded.Id, Title = "Updated", IsComplete = tru

        // Act
        var response = await _client.PutAsJsonAsync($"/api/todos/{seeded.Id}",

        // Assert
        response.StatusCode.Should().Be(HttpStatusCode.NoContent);

        // Verify update
        var get = await _client.GetAsync($"/api/todos/{seeded.Id}");
        var todo = await get.Content.ReadFromJsonAsync<TodoItem>();
        todo.Title.Should().Be("Updated");
        todo.IsComplete.Should().BeTrue();
```

```
    }

    [Fact]
    public async Task DeleteTodo_ReturnsNoContent()
    {
        // Arrange
        var seeded = await SeedTodo("ToDelete");

        // Act
        var response = await _client.DeleteAsync($"/api/todos/{seeded.Id}");

        // Assert
        response.StatusCode.Should().Be(HttpStatusCode.NoContent);

        var get = await _client.GetAsync($"/api/todos/{seeded.Id}");
        get.StatusCode.Should().Be(HttpStatusCode.NotFound);
    }

    // Helper method to seed a todo
    private async Task<TodoItem> SeedTodo(string title)
    {
        var todo = new TodoItem { Title = title, IsComplete = false };
        var response = await _client.PostAsJsonAsync("/api/todos", todo);
        return await response.Content.ReadFromJsonAsync<TodoItem>();
    }
```

Notes on the test suite:

- Uses `WebApplicationFactory<Program>` for integration testing with real request/response flows
- Employs an **in-memory database** to isolate tests and ensure a fresh state.
- **Arrange-Act-Assert (AAA)** pattern is followed in each test.
- Validates both **successful** and **error** scenarios, including HTTP status codes and response conte

---

6.5 Running the Tests

Execute the tests via the .NET CLI:

```
dotnet test TodoApi.Tests
```

Test output should show all tests passing, confirming that CRUD operations and validation rules are correctly enforced by the API.

6.6 Summary

In this chapter, we covered how to:

- Configure xUnit test projects for an ASP.NET Core 9 Web API.
- Use `WebApplicationFactory` and an in-memory database for reliable integration testing.
- Write comprehensive tests covering all CRUD operations and validation rules.
- Follow best practices like AAA, clean database state, and status code verification.

With this setup, you can confidently evolve your API, knowing that tests guard against regressions and ensure expected behavior.