ASP.NET Core - True Ultimate Guide

Section 19: Unit Testing [Advanced, Moq & Repository Pattern] - Notes

Fluent Assertions

Fluent Assertions is a .NET library that supercharges your unit tests by providing a more fluent, natural language syntax for assertions. Instead of using traditional, sometimes cryptic assertions like Assert.Equal or Assert.True, you write assertions that closely resemble how you would express expectations in plain English.

Benefits of Fluent Assertions

- **Readability:** Tests become more self-explanatory and easier to understand, even for developers who are not familiar with the codebase.
- **Maintainability:** Changes to the underlying code often result in more understandable test failures due to the descriptive nature of the assertions.
- **Rich API:** Offers a vast collection of assertion methods covering various scenarios (collections, strings, exceptions, and more), making it easier to write comprehensive tests.
- **Extensibility:** Allows you to create custom assertions for specific needs.

Important Fluent Assertions Methods with Examples

- **Basic Assertions:**

```
result.Should().Be(5);              // result should be equal to 5
result.Should().NotBe(10);          // result should not be equal to 10
result.Should().BeTrue();           // result should be true
result.Should().BeFalse();          // result should be false
result.Should().BeNull();           // result should be null
result.Should().NotBeNull();        // result should not be null
```

- **Collection Assertions:**

```
list.Should().HaveCount(3);          // list should have 3 elements
list.Should().Contain("apple");      // list should contain "apple"
list.Should().OnlyContain(x => x > 0);  // all elements in list should be greater than 0
list.Should().BeEquivalentTo(new[] { 1, 2, 3 }); // lists should contain the same elements (order doesn't matter)
```

- **String Assertions:**

```
name.Should().StartWith("John");      // name should start with "John"

name.Should().EndWith("Doe");        // name should end with "Doe"

name.Should().Contain("Middle");      // name should contain "Middle"

name.Should().MatchRegex(@"\d{3}-\d{3}-\d{4}"); // name should match a phone number pattern
```

- **Exception Assertions:**

```
Action act = () => someMethod();

act.Should().Throw<ArgumentException>(); // should throw ArgumentException

act.Should().Throw<Exception>()

   .WithMessage("Invalid operation");  // should throw an exception with a specific message
```

- **Type Assertions:**

```
object obj = new Person();

obj.Should().BeOfType<Person>();      // obj should be of type Person

obj.Should().BeAssignableTo<object>();   // obj should be assignable to object
```

AutoFixture

AutoFixture is another powerful library that helps with unit testing by automatically generating test data for your classes. It saves you time and effort in creating complex objects for your tests, especially when dealing with objects that have many properties or nested objects.

Benefits of AutoFixture

- **Test Data Generation:** Easily create instances of your classes with sensible default values for properties.
- **Customization:** You can customize the generated data to fit specific test scenarios.
- **Reduced Boilerplate:** Eliminates the need to manually create test data for each test.

Integrating AutoFixture with xUnit in ASP.NET Core

1. **Install Package:** Add the AutoFixture.Xunit2 NuGet package to your test project.
2. **Use the [AutoData] Attribute:** Decorate your test methods with [AutoData]. AutoFixture will automatically generate instances of the required types and pass them as arguments to your test methods.

Example with AutoFixture

```
public class PersonControllerTests
{
   [Theory, AutoData] // AutoFixture will create a Person instance for the test
   public void CreatePerson_ValidPerson_ReturnsOk(Person person, Mock<IPersonsService>
mockPersonsService)
   {
      // ... (rest of your test)
   }
}
```

Mocking

In unit testing, the goal is to test a specific unit of code (like a service class) in isolation from its dependencies. This helps you focus on the logic of the unit you're testing without worrying about external factors like database interactions or network calls. Mocking is a technique that enables this isolation.

Mocking involves creating substitute objects (mocks) that simulate the behavior of real dependencies. These mocks can be programmed to return specific data, throw exceptions, or track how they are used. This allows you to create controlled test scenarios and verify that your code interacts correctly with its dependencies.

Moq

Moq is a popular and intuitive mocking framework for .NET. It provides a fluent API to create mock objects easily.

How Mocking Works Internally (with Moq)

1. **Create a Mock:** You start by creating a mock object for the interface of the dependency you want to replace.

```
var mockPersonRepository = new Mock<IPersonsRepository>();
```

2. **Set Up Behavior:** You configure how the mock should behave when its methods are called. This typically involves specifying the return values, throwing exceptions, or verifying the arguments passed to the methods.

```
mockPersonRepository.Setup(repo => repo.GetAllPersons())
          .ReturnsAsync(new List<Person> { /* your test data */ });
```

3. **Inject the Mock:** You inject the mock object into the class you're testing, either through constructor injection or property injection.

4. **Exercise Your Code:** You call the methods of the class under test, which will interact with the mock object instead of the real dependency.

5. **Verify Interactions:** You use Moq's verification features to check if the mock's methods were called as expected and with the correct parameters.

Code:

```
// PersonsServiceTest.cs (Constructor)
public PersonsServiceTest(ITestOutputHelper testOutputHelper)
{
  _fixture = new Fixture(); // AutoFixture for test data generation

  _personRepositoryMock = new Mock<IPersonsRepository>();
  _personsRepository = _personRepositoryMock.Object; // Get the mock object
```

```
   // Create a mock DbContext using EntityFrameworkCoreMock
   var dbContextMock = new DbContextMock<ApplicationDbContext>(
      new DbContextOptionsBuilder<ApplicationDbContext>().Options
   );

   // Mock the Countries DbSet with initial data
   dbContextMock.CreateDbSetMock(temp => temp.Countries, new List<Country> { });

   // Mock the Persons DbSet with initial data
   dbContextMock.CreateDbSetMock(temp => temp.Persons, new List<Person> { });

   //Create services based on mocked DbContext object
   _countriesService = new CountriesService(dbContextMock.Object);
   _personService = new PersonsService(_personsRepository); // Pass the mocked repository
to your service

   _testOutputHelper = testOutputHelper;
}
```

This code:

1. Creates mock objects for the IPersonsRepository interface and the ApplicationDbContext class.
2. Uses EntityFrameworkCoreMock to configure mock DbSet objects.
3. Initializes your services, passing the mock repository (_personsRepository) to your PersonsService.

```
//Example of setting up a mock method
_personRepositoryMock
 .Setup(temp => temp.AddPerson(It.IsAny<Person>()))
 .ReturnsAsync(person);
```

This code configures the mock repository to return the person object whenever the AddPerson method is called with any Person object.

Best Practices

- **Focus on Behavior:** Mock only the interactions you need to control in your test. Avoid over-mocking.
- **Loose Coupling:** Design your classes with dependency injection in mind, making it easy to swap out real dependencies for mocks.
- **Verification (Optional):** Use Verify to ensure that your code interacts with the mock as expected.
- **Readability:** Strive for clear and expressive setup and verification code.

Things to Avoid

- **Mocking Everything:** Don't mock classes that you are testing directly. The goal is to isolate the unit under test, not eliminate all dependencies.
- **Excessive Setup:** Avoid overly complex setups that obscure the intent of your tests.
- **Verifying Implementation Details:** Focus on verifying behavior, not specific implementation details.

Integration Tests

While unit tests focus on individual units in isolation, integration tests examine how different parts of your application work together. In ASP.NET Core MVC, this typically involves testing the interaction between controllers, views, services, and sometimes even external dependencies like databases or APIs.

Why Integration Tests Matter

- **Real-World Scenarios:** Integration tests simulate real user interactions, revealing potential issues that might not be caught by unit tests.
- **End-to-End Testing:** They help you verify that the entire flow of a request, from routing to model binding, validation, service calls, and view rendering, works correctly.

- **Database Interaction Testing:** Integration tests can test how your application interacts with a real (or in-memory) database, ensuring data persistence and retrieval are accurate.
- **Confidence in Deployment:** A strong suite of integration tests boosts your confidence when deploying your application, reducing the likelihood of unexpected errors in production.

Key Elements of Integration Tests with xUnit

- **Test Server:** You create a test server instance using a custom WebApplicationFactory, which allows you to simulate your application's behavior in a test environment.
- **HTTP Client:** You use an HttpClient to send HTTP requests to the test server, mimicking how a real client (like a browser) would interact with your application.
- **Assertions:** Use assertions (e.g., from FluentAssertions or xUnit's built-in assertions) to validate the responses received from the server.

Best Practices

- **Focus on Integration:** Test the interactions between components, not the isolated behavior of individual units.
- **Database:**
  - **In-Memory:** Use an in-memory database (e.g., UseInMemoryDatabase) for faster tests and data isolation.
  - **Real Database (Optional):** For more realistic testing, use a test database with a separate schema or dataset.
- **Test Environment:** Configure your test server to use a "Test" environment to avoid accidentally affecting your development or production databases.
- **Clean Up:** If you're using a real database, ensure you clean up the test data after each test or test class to maintain data consistency.
- **Avoid External Dependencies:** If your application relies on external APIs or services, consider mocking or stubbing them for integration tests to avoid network dependencies and keep tests fast and reliable.
- **Clear Test Names:** Use descriptive names that explain the purpose and expected behavior of each test.

Code

CustomWebApplicationFactory:

```
public class CustomWebApplicationFactory : WebApplicationFactory<Program>
{
   protected override   1. github.com
```

github.com

```
 void ConfigureWebHost(IWebHostBuilder builder)
    {
        base.ConfigureWebHost(builder);


        builder.UseEnvironment("Test");   1. www.nuget.org
www.nuget.org
 // Set the environment to "Test"


        builder.ConfigureServices(services => {
            // Replace the default DbContext configuration with an in-memory database
            var descriptor = services.SingleOrDefault(temp => temp.ServiceType ==
typeof(DbContextOptions<ApplicationDbContext>));


            if (descriptor != null)
            {
                services.Remove(descriptor);   1. github.com
 MIT github.com


            }
            services.AddDbContext<ApplicationDbContext>(options =>
            {
                options.UseInMemoryDatabase("DatbaseForTesting");   1. github.com
github.com


            });
        });
    }
}
```

This class sets up a customized WebApplicationFactory for your integration tests:

1. **Inherits from WebApplicationFactory<Program>:** This base class provides the core functionality for creating a test server instance.

2. **ConfigureWebHost Override:** You override this method to customize the configuration of the test server.
3. **UseEnvironment("Test"):** Sets the ASPNETCORE_ENVIRONMENT variable to "Test", ensuring that the application loads any test-specific configuration settings from appsettings.Test.json.
4. **ConfigureServices:** Replaces the default database context configuration with an in-memory database provider for testing.

PersonsControllerIntegrationTest:

```
public class PersonsControllerIntegrationTest : IClassFixture<CustomWebApplicationFactory>
{
  private readonly HttpClient _client;   1. github.com
github.com


  // Constructor injection of the custom factory
  public PersonsControllerIntegrationTest(CustomWebApplicationFactory factory)
  {
    _client = factory.CreateClient(); // Create an HttpClient to interact with the test server
  }

  #region Index

  [Fact]
  public async Task Index_ToReturnView()
  {
    // Act: Send a GET request to the "/Persons/Index" endpoint
    HttpResponseMessage response = await _client.GetAsync("/Persons/Index");

    // Assert:
    // 1. Check if the response was successful (status code 2xx)
    response.Should().BeSuccessful();
```

```
      // 2. Read the response content as a string
      string responseBody = await response.Content.ReadAsStringAsync();


      // 3. Parse the HTML content using HtmlAgilityPack
      HtmlDocument html = new HtmlDocument();
      html.LoadHtml(responseBody);
      var document = html.DocumentNode;


      // 4. Assert that the response contains a table with the class "persons"
      document.QuerySelectorAll("table.persons").Should().NotBeNull();
    }


    #endregion
}
```

This test class uses the custom CustomWebApplicationFactory to create a test server instance.

1. **IClassFixture<CustomWebApplicationFactory>:** This interface tells xUnit to create a single instance of CustomWebApplicationFactory and share it among all tests in this class. This ensures that the test server is created only once, improving performance.
2. **Constructor Injection:** The constructor receives the factory and uses it to create an HttpClient that can send requests to the test server.
3. **Index_ToReturnView Test:**
    - **Act:** Sends a GET request to the Persons/Index endpoint.
    - **Assert:**
        - Checks if the response status code indicates success (2xx).
        - Parses the HTML response body using HtmlAgilityPack.
        - Asserts that the response contains a <table> element with the class "persons". This verifies that the Index view is rendering correctly.

Notes

- **Purpose:** Integration tests verify interactions between components, not isolated unit behavior.
- **Test Server:** Use WebApplicationFactory to create a test server instance.

- **In-Memory Database:** Use an in-memory database for testing to isolate data and improve speed.
- **Test Environment:** Set the environment to "Test" for test-specific configurations.
- **Clean Up:** Ensure proper cleanup of test data (especially if using a real database).
- **Mocking:** Consider mocking external dependencies for faster and more reliable tests.

Key Points to Remember

xUnit Advanced Topics

- **[Theory] and [InlineData]:**
  - [Theory] marks a test method that should be executed with multiple data sets.
  - [InlineData(...)] provides the data sets to use for the test.
- **[ClassFixture]:**
  - Shares a fixture instance (e.g., a test database connection) across all tests in a class.
  - Improves performance by avoiding redundant setup/teardown.
- **Custom Assertions:** Create your own assertions by extending the Xunit.Assert class.
- **Test Collections:** Group related tests using [Collection] and [CollectionDefinition] attributes.
- **Parallelization:** xUnit can run tests in parallel to improve execution speed.

Mocking (Moq)

- **Purpose:** Isolate the unit under test by simulating the behavior of dependencies.

- **Key Methods:**
  - Setup(expression): Configures how a mock method should behave.
  - Returns(value) or ReturnsAsync(value): Specifies the return value.
  - Throws(exception) or ThrowsAsync(exception): Simulates an exception being thrown.
  - Verify(expression, times): Ensures a method was called the expected number of times.

- **Best Practices:**
  - Mock only what's necessary.
  - Design for dependency injection.
  - Use clear and expressive setup and verification code.

AutoFixture

- **Purpose:** Automatically generates test data for your classes.
- **Key Features:**
  - [AutoData] attribute: Provides auto-generated instances to your test methods.
  - Customization: Control how data is generated using customizations and builders.
- **Benefits:**
  - Saves time writing test data.
  - Encourages testing with a variety of inputs.

FluentAssertions

- **Purpose:** Provides a more fluent and readable syntax for assertions.
- **Key Features:**
  - Method chaining for expressive assertions (e.g., result.Should().Be(5);)
  - Rich API with assertions for various scenarios (collections, exceptions, strings, etc.).

Repository Implementation & Unit Testing

- **Purpose:** Repositories handle data access logic (interaction with the database).
- **Interfaces:** Define interfaces (e.g., IPersonsRepository) to abstract data access and facilitate mocking.
- **Unit Tests:**
  - Focus on testing the repository's logic in isolation.
  - Use mocks for database interactions.
  - Cover all CRUD operations and edge cases.

Controller Unit Testing

- **Purpose:** Test controller actions and their interactions with services and models.
- **Mock Services:** Use mocks to isolate controllers from external dependencies.
- **Test Scenarios:**
    - Verify correct action results are returned (views, JSON data, redirects, etc.).
    - Check if the controller interacts with services as expected.
    - Test model validation and error handling.

Integration Tests

- **Purpose:** Test how multiple components (controllers, views, services, and sometimes even a real database) work together.
- **WebApplicationFactory:** Create a test server instance to simulate real requests.
- **HttpClient:** Use an HTTP client to send requests to the test server.
- **In-Memory Database:** Often use an in-memory database for testing.
- **Test Environment:** Set the ASPNETCORE_ENVIRONMENT to "Test" for test-specific configuration.

Interview Tips

- **Demonstrate Understanding:** Explain the purpose and benefits of each tool and technique.
- **Code Examples:** Be prepared to write or analyze code snippets showcasing these concepts.
- **Best Practices:** Discuss the best practices for each topic and common pitfalls to avoid.
- **Real-World Scenarios:** Connect these concepts to real-world testing challenges and how you would solve them.