upGrad ENTERPRISE

Unit Testing with xUnit

By
Narasimha Rao T
Microsoft.Net FSD Trainer
Professional Development Trainer
tnrao.trainer@gmail.com

1. What is Unit Testing?

- **Definition**: Unit testing is the practice of testing the smallest testable parts (units) of an application in isolation.
- Typically, a *unit* is a method or function.
- The goal is to verify correctness, handle edge cases, and ensure reliability.

2. Why Do We Write Unit Tests?

- **Benefits**:
  - Catch bugs early in the development cycle.
  - Ensure code correctness and maintainability.
  - Facilitate **refactoring** without fear of breaking existing logic.
  - Improve developer confidence.
  - Acts as documentation for how code should behave.
  - Enables **Test Driven Development (TDD)**.

3. Unit Testing Libraries in .NET Core

Common frameworks:

1. **xUnit** – Preferred framework for .NET Core.
2. **NUnit** – Popular and mature.
3. **MSTest** – Microsoft's testing framework.

Mocking libraries (for dependencies):

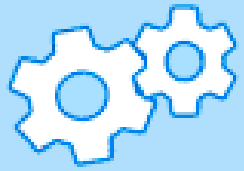
- **NSubstitute**
- **Moq**
- **FakeItEasy**

4. Writing Unit Tests with xUnit

- **Key Features**:
  - `[Fact]` → Write a test method without parameters.
  - `[Theory]` + `[InlineData]` → Parameterized tests.
  - Assertion methods (e.g., `Assert.Equal`, `Assert.Throws`).

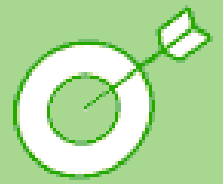
Arrange-Act-Assert (AAA)

ARRANGE



Setup the code to test

ACT



Perform the action you want to test

ASSERT

Check if the result matches your expectation

5. Create Unit Test Project using xUnit

Steps:

1. In terminal/VS:

```
dotnet new xunit -n MyApp.Tests
dotnet add MyApp.Tests reference MyApp
```

2. Folder structure:

```
MyApp/
└── MyApp.Tests/
```

6. Write Tests for Basic Utility Methods

Example: Testing an `Add` method in `Calculator.cs`.

```
public class Calculator
{
    public int Add(int a, int b) => a + b;
}
```

Test:

```
public class CalculatorTests {
    [Fact]
    public void Add_TwoNumbers_ReturnsSum() {
        var calc = new Calculator();
        var result = calc.Add(2, 3);
        Assert.Equal(5, result);
    }
}
```

7. Testing Async and Exception Scenarios

- **Async Tests:**

```
[Fact]
public async Task GetDataAsync_ReturnsValue()
{
    var service = new DataService();
    var result = await service.GetDataAsync();
    Assert.NotNull(result);
}
```

- Exception Handling:

```
[Fact]
public void Divide_ByZero_ThrowsException()
{
    var calc = new Calculator();
    Assert.Throws<DivideByZeroException>(() => calc.Divide(10, 0));
}
```

8. Mocking Dependencies with NSubstitute

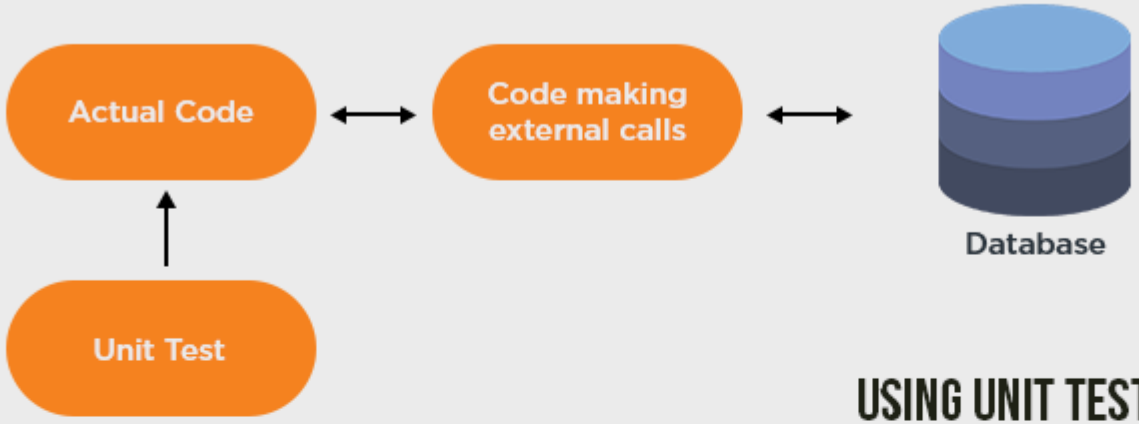
- **Why Mock?** To isolate the unit under test from external dependencies (e.g., DB, APIs).

Example:

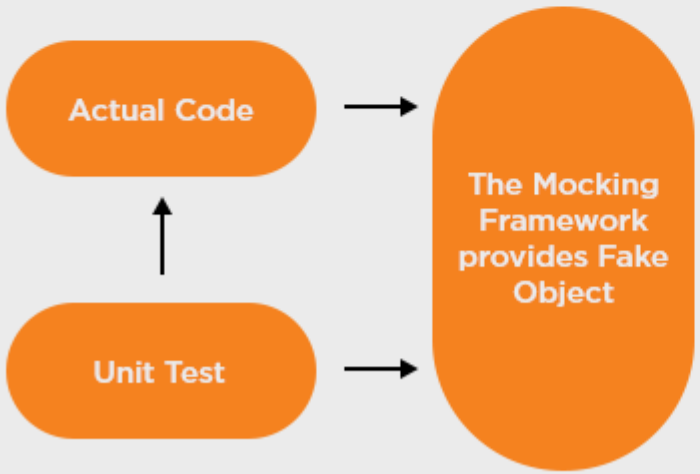
```
public interface IRepository
{
    string GetData();
}

public class Service
{
    private readonly IRepository _repo;
    public Service(IRepository repo) => _repo = repo;

    public string GetProcessedData() => _repo.GetData().ToUpper();
}
```



USING UNIT TESTING



USING MOCKING FRAMEWORK

9. Parameterized Tests with `Theory` + `InlineData`

```
[Theory]
[InlineData(2, 3, 5)]
[InlineData(10, 5, 15)]
public void Add_MultipleInputs_ReturnsExpected(int a, int b, int expected)
{
    var calc = new Calculator();
    var result = calc.Add(a, b);
    Assert.Equal(expected, result);
}
```

10. Naming Conventions & Test Organization

- **Convention**:

  ```
  MethodName_StateUnderTest_ExpectedOutcome
  ```

  Example: `Add_TwoPositiveNumbers_ReturnsSum`

- **Organization**:
  - Use folders matching the main project structure.
  - One test class per source class.
  - Group related tests logically.

Q & A

Narasimha Rao T
tnrao.trainer@gmail.com