Unit Test Best Practices

Some aspects of Unit Testing best practices are listed here as all will not be possible to implement at present.

1. Test Case: Null Input Handling

Scenario: Ensure a method handles `null` input gracefully.
Test Steps:

- Call a method with `null` as an input.
- Assert that the method does not throw exceptions.
- Assert the return value is handled appropriately (e.g., return `null`, empty, or default).

```
[Test]
public void Method_ShouldHandleNullInputGracefully()
{
    var result = myService.Method(null);
    Assert.IsNull(result); // or other appropriate assertion
}
```

2. Test Case: Valid Input Produces Correct Output

Scenario: A method returns the correct value for valid inputs.
Test Steps:

- Call the method with valid inputs.
- Assert that the return value matches the expected output.

```
[Test]
public void Method_ShouldReturnCorrectValue_GivenValidInput()
{
    var input = "test";
    var expectedOutput = "expected";
    var result = myService.Method(input);
    Assert.AreEqual(expectedOutput, result);
}
```

3. Test Case: Exception Handling

Scenario: Ensure a method throws the correct exception for invalid inputs.
Test Steps:

- Pass invalid data to the method.
- Assert that it throws the expected exception.

```
[Test]
public void Method_ShouldThrowException_GivenInvalidInput()
{
    Assert.Throws<ArgumentException>(() =>
myService.Method(invalidInput));
}
```

4. Test Case: Boundary Value Test

Scenario: Test method behavior at boundary values.
Test Steps:

- Test the method with boundary input values.
- Assert that the method works correctly at boundary conditions (e.g., 0, min/max values).

```
[Test]
public void Method_ShouldThrowException_GivenInvalidInput()
{
    Assert.Throws<ArgumentException>(() =>
myService.Method(invalidInput));
}
```

4. Test Case: Boundary Value Test

Scenario: Test method behavior at boundary values.
Test Steps:

- Test the method with boundary input values.
- Assert that the method works correctly at boundary conditions (e.g., 0, min/max values).

```
[Test]
public void Method_ShouldHandleBoundaryValues()
{
    var minValue = 1;
```

```
    var maxValue = 100;
    var resultForMin = myService.Method(minValue);
    var resultForMax = myService.Method(maxValue);

    Assert.AreEqual(expectedForMin, resultForMin);
    Assert.AreEqual(expectedForMax, resultForMax);
}
```

5. Test Case: Database Interaction Test

Scenario: Ensure data is correctly saved to or fetched from the database.
Test Steps:

- Add a new entity to the database.
- Retrieve the entity and assert that the values match the expected results.
- Optionally rollback changes for isolated testing.

```
[Test]
public void DatabaseService_ShouldInsertAndRetrieveDataCorrectly()
{
    var entity = new Entity { Id = 1, Name = "Test Entity" };
    myRepository.Add(entity);

    var retrievedEntity = myRepository.GetById(1);
    Assert.AreEqual(entity.Name, retrievedEntity.Name);
}
```

6. Test Case: API Response Test

Scenario: Verify that the API endpoint returns the correct status code and data.
Test Steps:

- Send a request to the API endpoint.
- Assert the response status code and content.

```
[Test]
public async Task Api_ShouldReturnCorrectResponse()
{
    var response = await _httpClient.GetAsync("/api/resource");
    Assert.AreEqual(HttpStatusCode.OK, response.StatusCode);

    var content = await response.Content.ReadAsStringAsync();
```

```
    Assert.IsTrue(content.Contains("expectedData"));
}
```

7. Test Case: Dependency Mocking

Scenario: Test a method that depends on external services by mocking them.
Test Steps:

- Mock the dependencies and simulate responses.
- Assert that the method under test interacts with the mocks as expected.

```
[Test]
public void Service_ShouldCallDependency_WithCorrectArguments()
{
    var mockDependency = new Mock<IMyDependency>();
    mockDependency.Setup(d => d.DoSomething(It.IsAny<int>
())).Returns(true);

    var service = new MyService(mockDependency.Object);
    var result = service.Process(5);

    Assert.IsTrue(result);
    mockDependency.Verify(d => d.DoSomething(5), Times.Once);
}
```

8. Test Case: Performance Testing

NOTE: This is not as impactful for load/stress testing. This performance testing is only for a simple timing testing for key methods where it is applicable. Also build server may impact the performance and results may vary.

Scenario: Measure method execution time for performance benchmarking.
Test Steps:

- Call the method multiple times to assess performance.
- Assert that the execution time is within an acceptable threshold.

```
[Test]
public void Method_ShouldCompleteWithinTimeLimit()
{
    var stopwatch = new Stopwatch();
    stopwatch.Start();
```

```
    myService.Method();

    stopwatch.Stop();
    Assert.LessOrEqual(stopwatch.ElapsedMilliseconds, 100); // 100 ms
threshold
}
```