Unit testing

Asp.net core

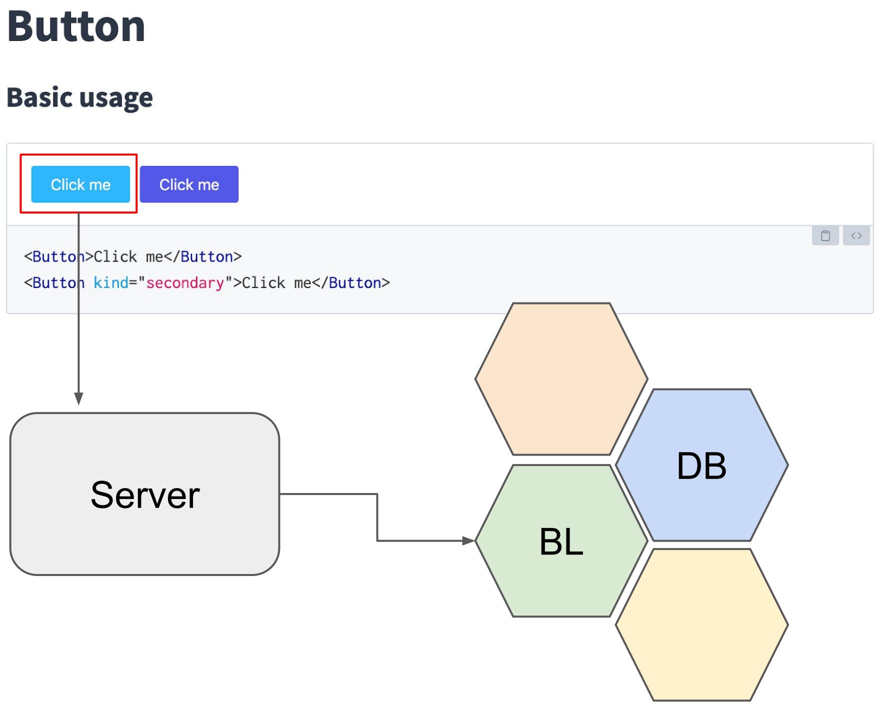
Uncle Bob Martin says

Am I suggesting 100% test coverage? No, I’m demanding it. Every single line of code that you write should be tested.

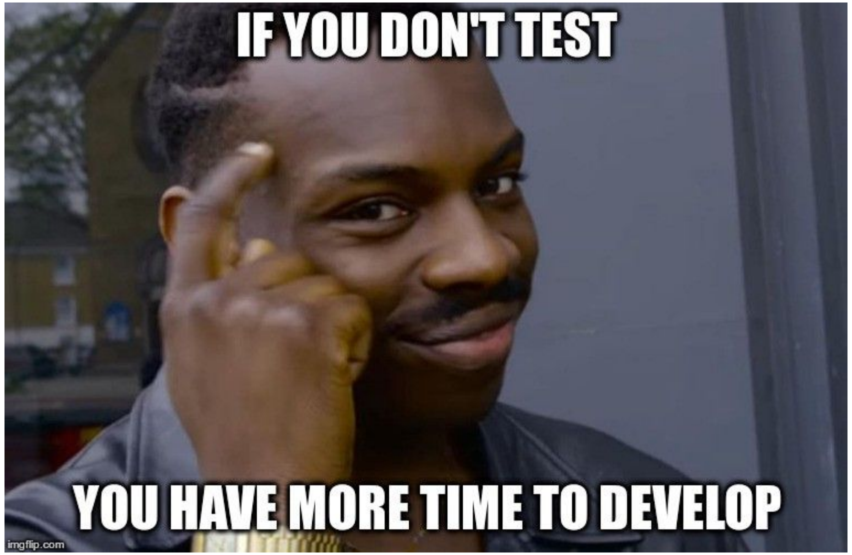
I don’t want management to mandate 100% test coverage. I want your conscience to mandate it as a point of honor.

How about: if you have 100%, you can refactor savagely whenever you want with confidence.

Have you ever test code ?



```
<Button>Click me</Button>
<Button kind="secondary">Click me</Button>
```



UI testing

- Check all the GUI elements for size, position, width, length, and acceptance of characters or numbers. For instance, you must be able to provide inputs to the input fields.
- Check you can execute the intended functionality of the application using the GUI
- Check Error Messages are displayed correctly
- Check for Clear demarcation of different sections on screen
- Check Font used in an application is readable
- Check the alignment of the text is proper
- Check the Color of the font and warning messages is aesthetically pleasing
- Check that the images have good clarity
- Check that the images are properly aligned
- Check the positioning of GUI elements for different screen resolution.

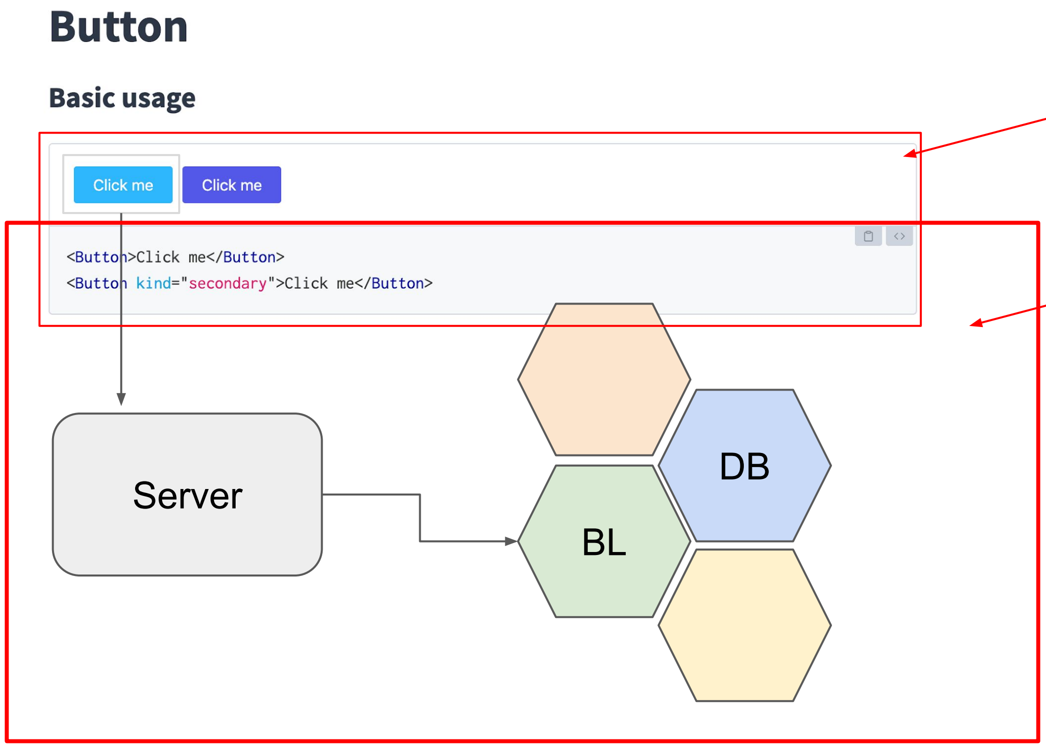
UI testing approaches

Manual Based Testing

Record and Replay

Model Based Testing

A model is a graphical description of a system's behavior. It helps us to understand and predict the system behavior. Models help in a generation of efficient test cases using the system requirements.



Button

Basic usage

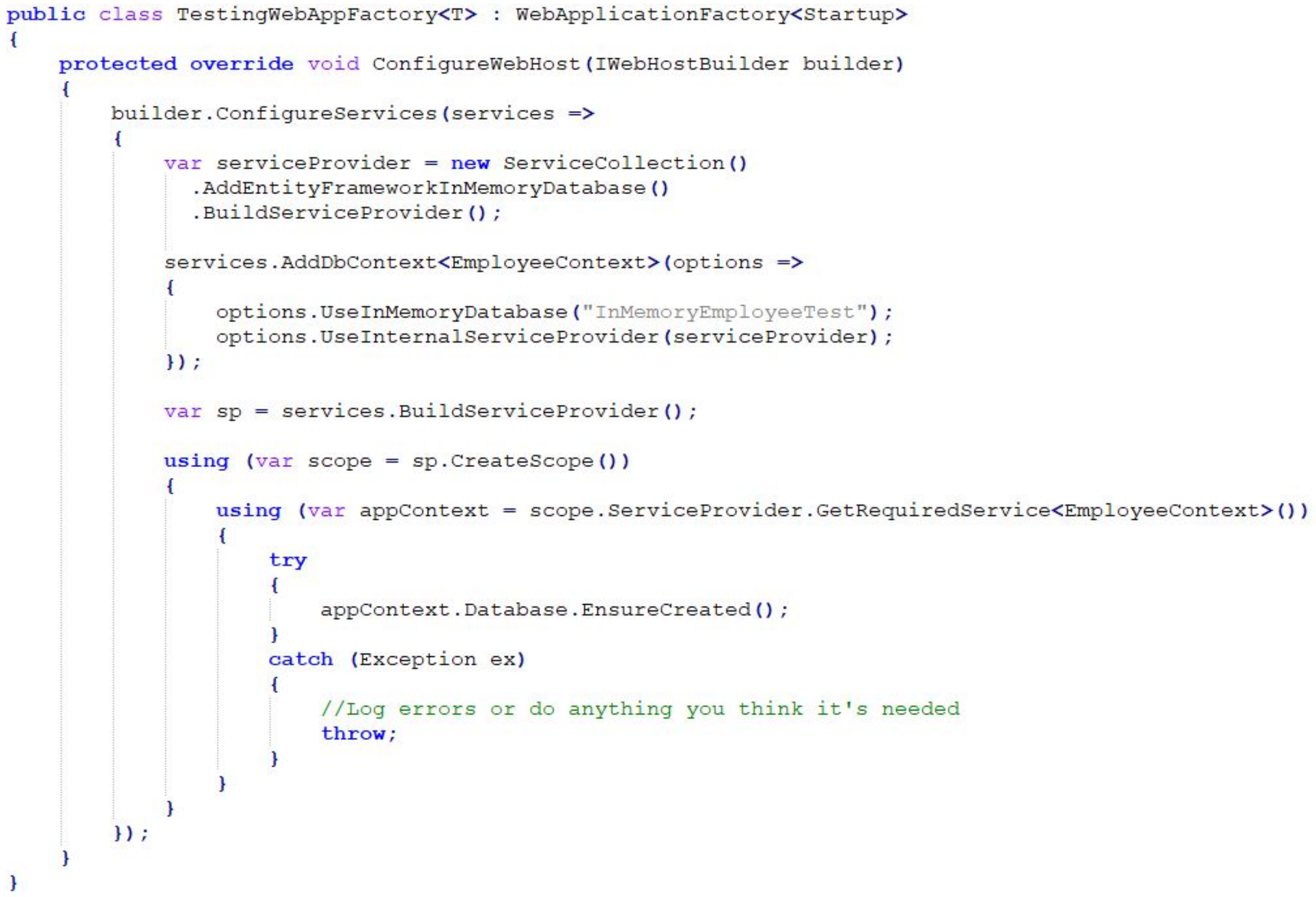
```
<Button>Click me</Button>
<Button kind="secondary">Click me</Button>
```

UI tests

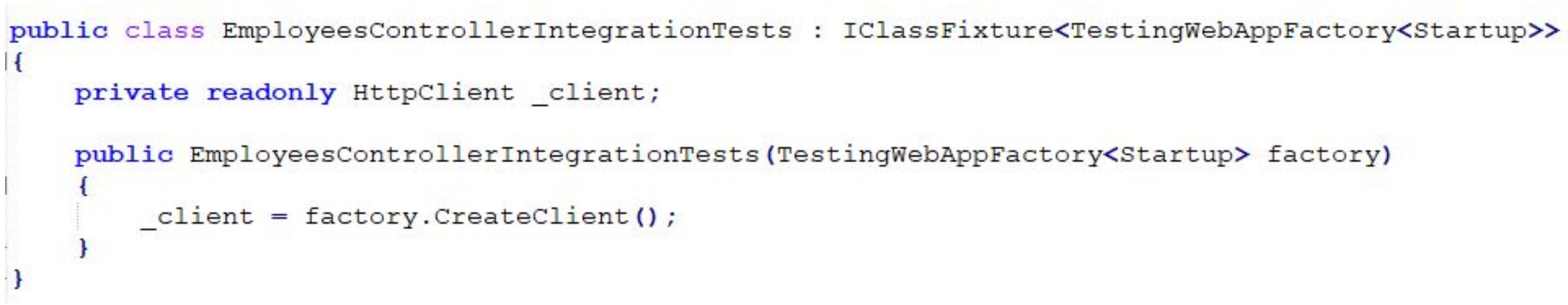
Integration tests

Integration testing

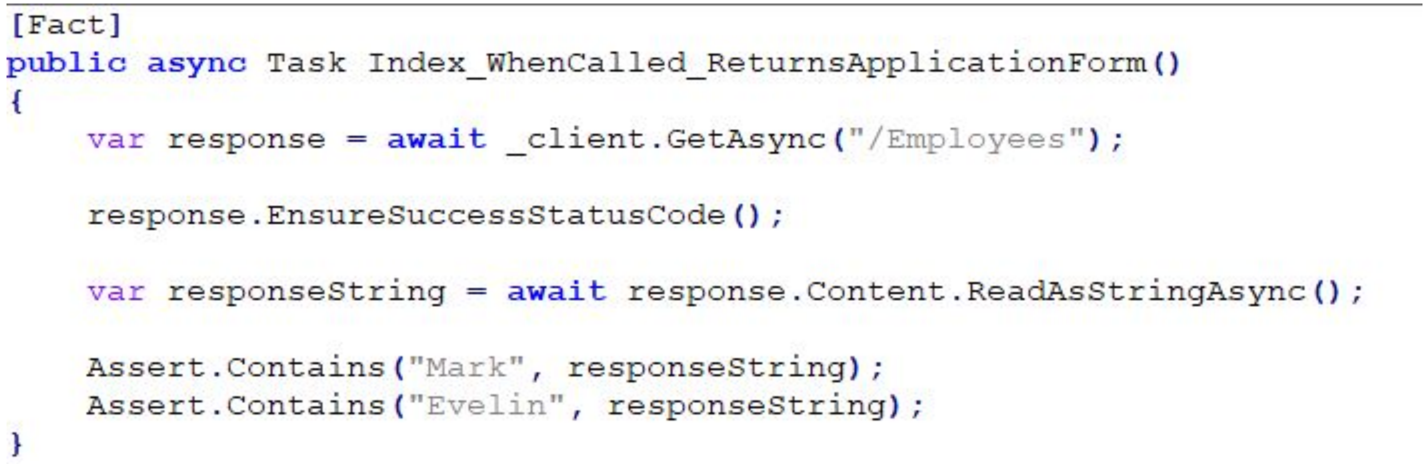
INTEGRATION TESTING is a level of software testing where individual units are combined and tested as a group. The purpose of this level of testing is to expose faults in the interaction between integrated units. Test drivers and test stubs are used to assist in Integration Testing.



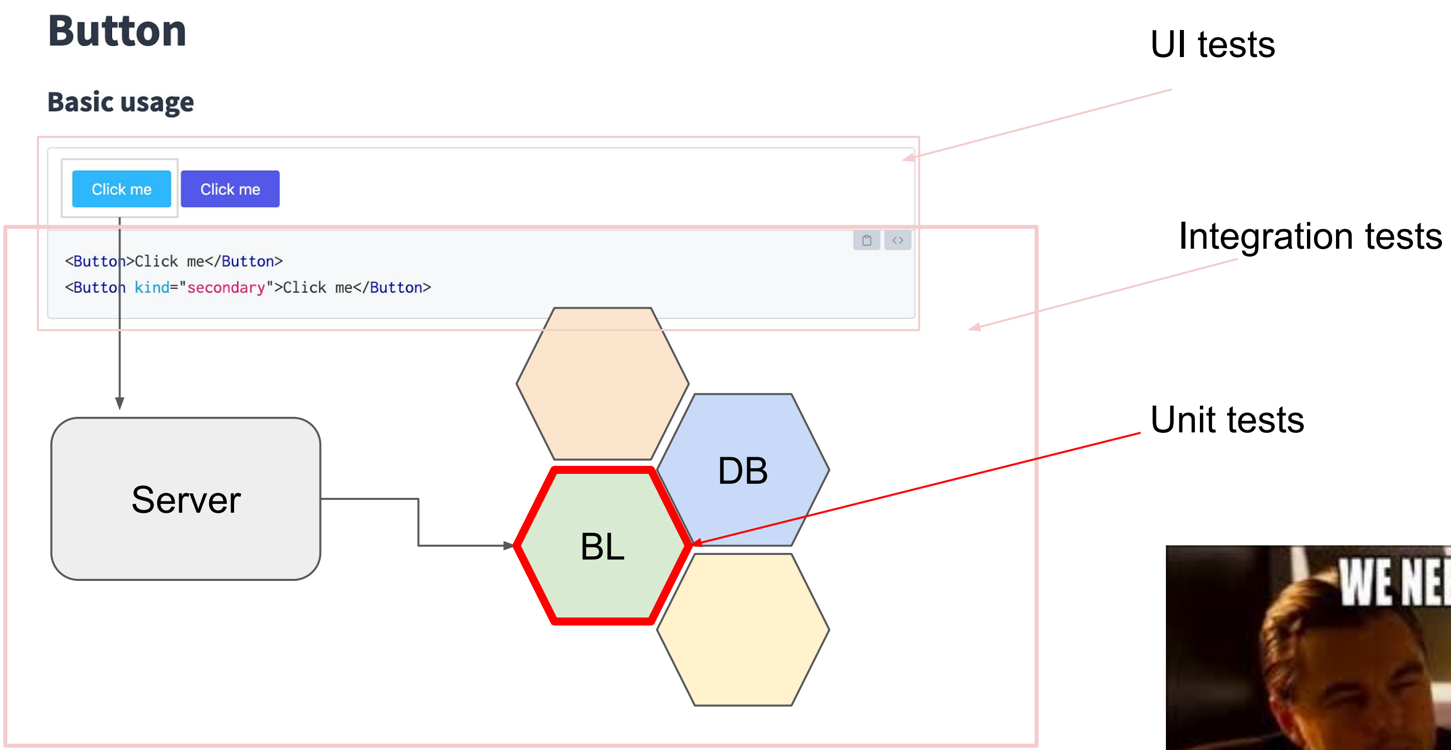
```
public class TestingWebAppFactory<T> : WebApplicationFactory<Startup>
{
    protected override void ConfigureWebHost(IWebHostBuilder builder)
    {
        builder.ConfigureServices(services =>
        {
            var serviceProvider = new ServiceCollection()
              .AddEntityFrameworkInMemoryDatabase()
              .BuildServiceProvider();

            services.AddDbContext<EmployeeContext>(options =>
            {
                options.UseInMemoryDatabase("InMemoryEmployeeTest");
                options.UseInternalServiceProvider(serviceProvider);
            });

            var sp = services.BuildServiceProvider();

            using (var scope = sp.CreateScope())
            {
                using (var appContext = scope.ServiceProvider.GetRequiredService<EmployeeContext>())
                {
                    try
                    {
                        appContext.Database.EnsureCreated();
                    }
                    catch (Exception ex)
                    {
                        //Log errors or do anything you think it's needed
                        throw;
                    }
                }
            }
        });
    }
}
```



```
public class EmployeesControllerIntegrationTests : IClassFixture<TestingWebAppFactory<Startup>>
{
    private readonly HttpClient _client;

    public EmployeesControllerIntegrationTests(TestingWebAppFactory<Startup> factory)
    {
        _client = factory.CreateClient();
    }
}
```



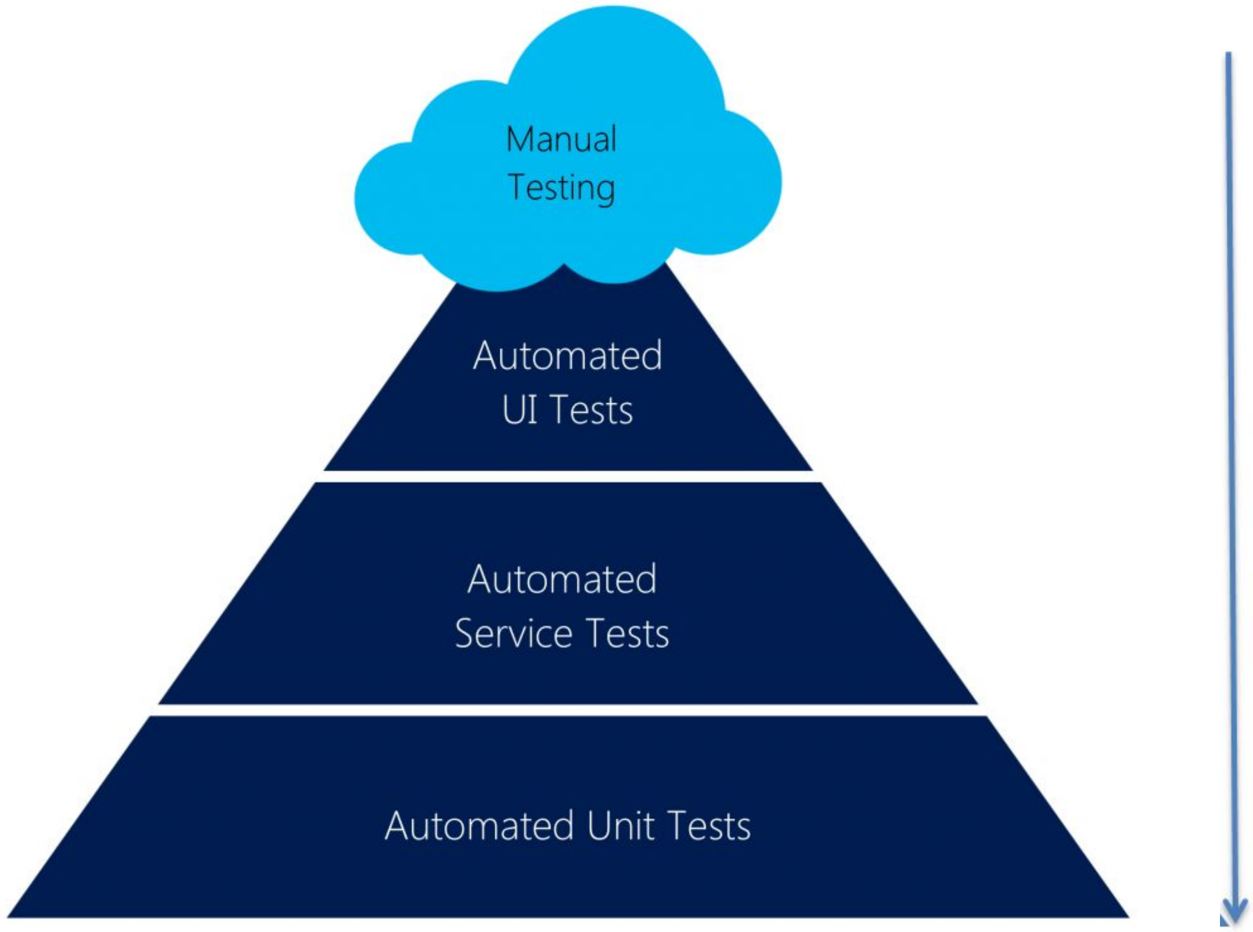
```
[Fact]
public async Task Index_WhenCalled_ReturnsApplicationForm()
{
    var response = await _client.GetAsync("/Employees");

    response.EnsureSuccessStatusCode();

    var responseString = await response.Content.ReadAsStringAsync();

    Assert.Contains("Mark", responseString);
    Assert.Contains("Evelin", responseString);
}
```



Button

Basic usage

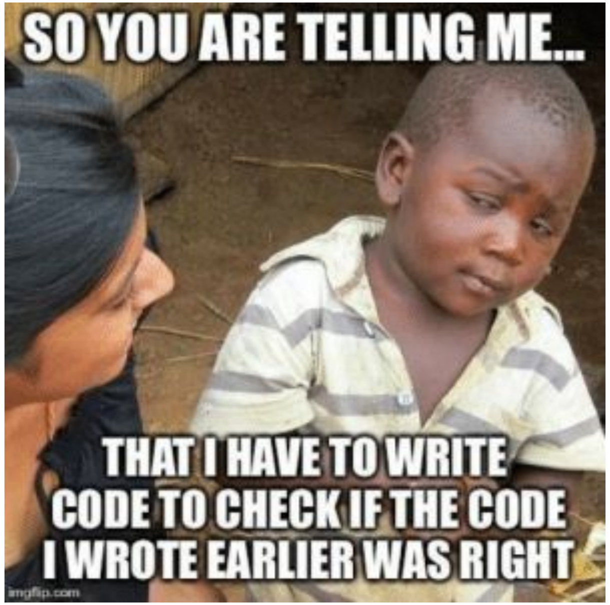
```
<Button>Click me</Button>
<Button kind="secondary">Click me</Button>
```



- More tests
- Shorter Execution Time
- Simple
- More stable
- Narrow Reach

What is unit testing ?

Unit testing is the testing of code to ensure that it performs the task that it is meant to perform. It tests code at the very lowest level possible — the individual methods of your classes


SO YOU ARE TELLING ME...
THAT I HAVE TO WRITE CODE TO CHECK IF THE CODE I WROTE EARLIER WAS RIGHT
imgflip.com

What is unit ?

Unit is as any discreet module of code that can be tested in isolation. It can be something as simple as a stand-alone routine, but it usually will be a single class and its methods.

A class is the primary, discrete code entity of many modern languages and, thus, are the base building blocks of your code. They are the data structures that, when used together, form a system.

Testable class

When unit testing a class, you are unit testing the given class and only the given class. Unit testing is always done in isolation — that is, the class under test needs to be completely isolated from any other classes or any other systems.

If you are testing a class and you need some external entity, then you are no longer unit testing. A class is only “testable” when its dependencies can be and are “faked”, and thus tested, without any of its real external dependencies.

So, if you are running what you think is a unit test, and that test needs to access a database, a file system, or any other external system, then you have stopped unit testing, and you’ve started integration testing.

Unit vs integration testing

Definition: Unit testing is the act of testing a single class in isolation, completely apart from any of its actual dependencies.

Definition: Integration testing is the act of testing a single class along with one or more of its actual external dependencies.

Fakes

Commonly, developers have used the term *mocking framework* to describe code that provides faking services to allow classes to be tested in isolation.

Fakes allow you to test a class in isolation by providing implementations of dependencies without requiring the real dependencies.

Definition: A fake class is any class that provides functionality sufficient to pretend that it is a dependency needed by a class-under test. There are two kinds of fakes: stubs and mocks.

Stub

A stub is a class that does the absolute minimum to appear to be an actual dependency for the class-under test. It provides no functionality required by the test, except to appear to implement a given interface or descend from a given base class.

When the class calls it, a stub usually does nothing. Stubs are entirely peripheral to testing the call and exist purely to enable the class under test to run.

A typical example is a stub that provides logging services.

Stub

Definition: A stub is a fake that has no effect on the passing or failing of the test and it exists purely to allow the test to run.

Mocks

Mocks are a bit more complicated. Mocks do what stubs do in that they provide a fake implementation of a dependency needed by the class under test (CUT).

A mock keeps a record of all the interactions with the CUT and reports back, passing the test if the CUT behaved correctly, and failing the test if it did not.

Definition: A mock is a fake that keeps track of the behavior of the class under test and passes or fails the test based on that behavior.

Unit testing will findbugs

Whether you do test-driven development and write your tests first, write your tests as you go along, or write tests long after the code has been written, unit testing will find bugs.

When you write a full suite of tests that define what the expected behavior is for a given class, anything in that class that isn’t behaving as expected will be revealed.

Unit testing will keep bugs away

A complete and thorough suite of unit tests will help to ensure that any bugs that creep into your code will be revealed immediately.

Make a change that introduces a bug, and your tests can reveal it the very next time you run your tests. If you find a bug that is outside the realm of your unit-test suite you can write a test for it to ensure that the bug never returns.

Unit testing saves time

Most developers believe that writing unit tests takes more time than it saves

Writing unit tests helps ensure that your code is working as designed, right from the start. Unit tests define what your code should do, and thus you won’t spend time writing code that does things that it shouldn’t do.

Every unit test becomes a regression test, ensuring that things continue to work as designed while you develop. They help ensure that subsequent changes don’t break things.

Unit testing gives peace of mind

You can run all those tests and know that your code works as it is supposed to. You can refactor and change the code, knowing that if you break anything, you’ll know right away.

Knowing the state of your code, that it works, and that you can update and improve it without fear is an excellent thing.

Unit testing documents the proper use of a class

One of the benefits of unit testing is that your tests can define for subsequent developers how the class should be used.

Unit tests become, in effect, simple examples of how your code works, what it is expected to do, and the proper way to use the code being tested. Consumers of your code can look to your unit tests for information about the proper way to make your code do what it is supposed to do.

Good practies

- Test cases should be small and isolated. As mentioned above, they should test something specific about the code.
- Try keeping test to assertion ratio near to 1. It makes it easy to identify any assertion which has failed. Having multiple assertions can make it cumbersome to verify which assertion went rogue.
- Always avoid test interdependence. Each test case should handle their own build up and tear down. Test runners don't generally run tests in any specified order. Thus, we can not assume anything based on the order in which we write the cases.
- A test case approaches the behavioral aspect of the code. So it should be easily comprehensible in the sense that what we are testing and what to do in case of failure.
- Mock as little as needed. Too many fakes create fragile tests which break when they code undergoes changes in production.
- Avoid mocking chatty interfaces. Any change in order of calling may break the test.
- It should be independent of external factors. You must not require a setup to run a unit test.
- We must maintain a clear naming convention in writing unit tests. This simply makes our code more comprehensible.
- While working on a continuous integration build, always add the unit test cases to the build so that whenever one test case fails, the whole build fails thus leaving no exemptions.

AAA

The AAA (Arrange-Act-Assert) pattern has become almost a standard across the industry. It suggests that you should divide your test method into three sections: arrange, act and assert. Each one of them only responsible for the part in which they are named after.

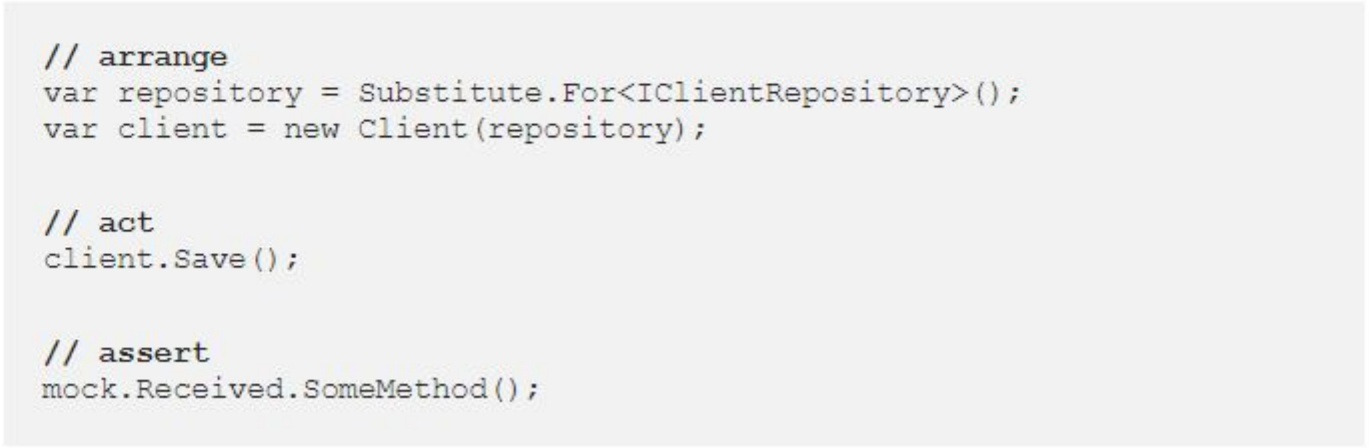
So the arrange section you only have code required to setup that specific test. Here objects would be created, mocks setup (if you are using one) and potentially expectations would be set.

Then there is the Act, which should be the invocation of the method being tested.

And on Assert you would simply check whether the expectations were met.

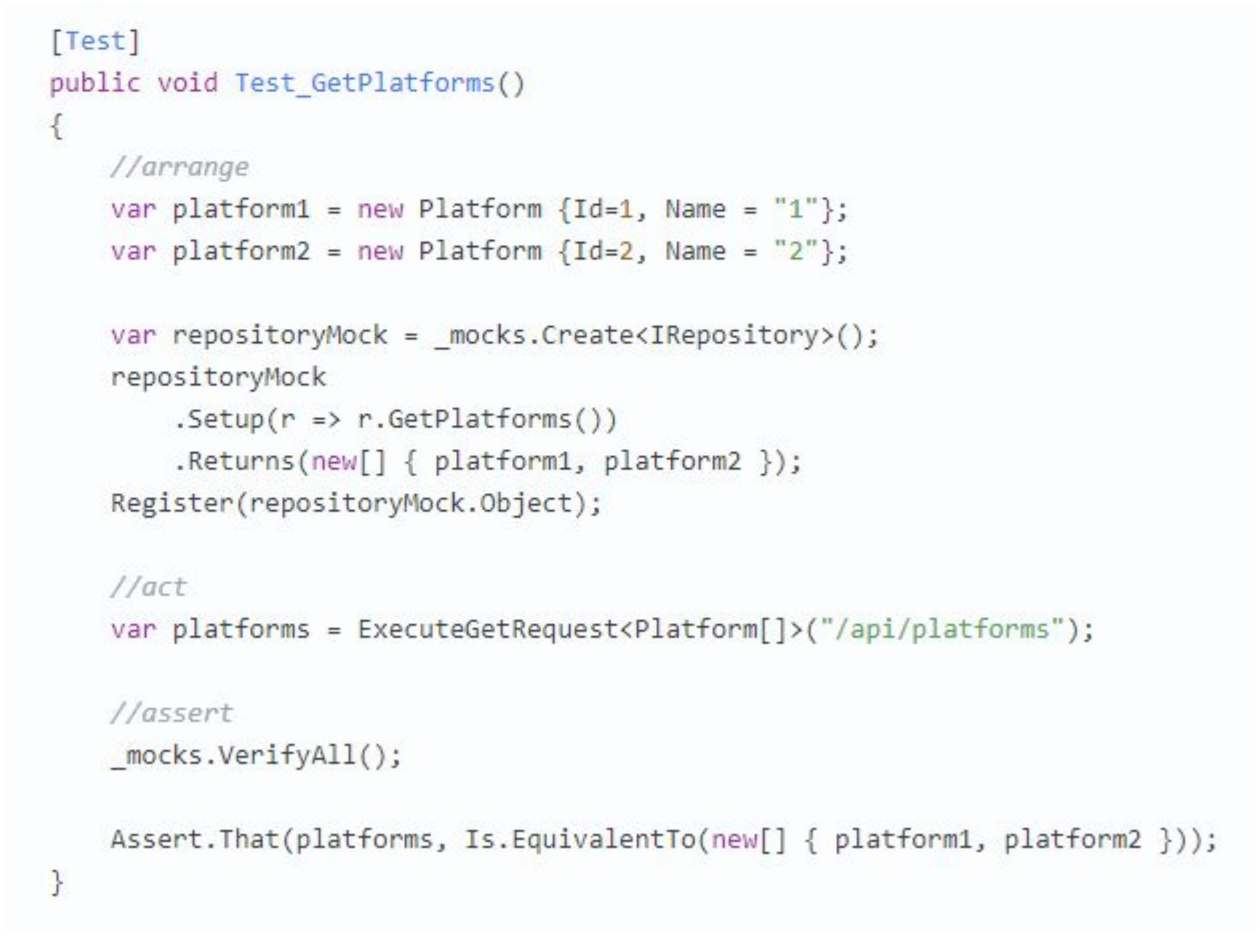
AAA

Following this pattern does make the code quite well structured and easy to understand



```
// arrange
var repository = Substitute.For<IClientRepository>();
var client = new Client(repository);

// act
client.Save();

// assert
mock.Received.SomeMethod();
```

AAA



```
[Test]
public void Test_GetPlatforms()
{
    //arrange
    var platform1 = new Platform {Id=1, Name = "1"};
    var platform2 = new Platform {Id=2, Name = "2"};

    var repositoryMock = _mocks.Create<IRepository>();
    repositoryMock
        .Setup(r => r.GetPlatforms())
        .Returns(new[] { platform1, platform2 });
    Register(repositoryMock.Object);

    //act
    var platforms = ExecuteGetRequest<Platform[]>("/api/platforms");

    //assert
    _mocks.VerifyAll();

    Assert.That(platforms, Is.EquivalentTo(new[] { platform1, platform2 }));
}
```