Unit Testing in .NET

Session #05

dotnet training

Agenda

Entity Framework Core

Unit Testing in dotnet

xUnit vs NUnit vs MSTest

Code Coverage

Mocks

Unit Testing in WebAPI

Unit Testing in dotnet

Unit Testing in .NET

Getting Started

- Completed integrated on .NET platform
- Can be executed using dotnet CLI
- Can be executed live on Visual Studio
- Can be executed by request on Visual Studio Code
- Some best practices: https://learn.microsoft.com/en-us/dotnet/core/testing/unit-testing-best-practices

xUnit vs Unit vs MSTest

xUnit vs NUnit vs MSTest

Frameworks

- Most used Unit Testing Frameworks for C#
- They have similar feature set but with different way to implement them on your code
- Selecting between each other can be a personal or community-driven choice
- Nowadays, most used framework is xUnit and is the one with more active community

NUnit

Frameworks

- NUnit is an open-source testing framework ported from JUnit
- Latest version is NUnit 3, following all JUnit specs
- Uses Annotations to define tests, test cases and other main topic in unit testing

NUnit

Frameworks

- NUnit is an open-source testing framework ported from JUnit
- Latest version is NUnit 3, following all JUnit specs
- Uses Annotations to define tests, test cases and other main topic in unit testing
- Fully integrated on .NET foundation
- Can be a good choice for developers moving from JAVA

XUnit

Frameworks

- Open-source testing framework based on the .NET framework.
- Can be used on all .NET languages (C#, VB, F#)
- The creators of NUnit created xUnit as they wanted to build a better and easier to use framework rather than adding incremental features to the NUnit framework.
- Has a big community and for now is the most used testing framework on .NET

XUnit

Frameworks

- The popular attributes [SetUp] and [TearDown] are also not a part of the xUnit framework. For initialization, constructor of the test class is used, whereas, for de-initialization, IDisposable interface is used.
- Non-parameterized tests are implemented under the [Fact] attribute, whereas the [Theory] attribute is used if you plan to use parameterized tests.
- xUnit framework can locate the test methods, irrespective of the location of the tests, so you don’t need to mark code as [TestClass]

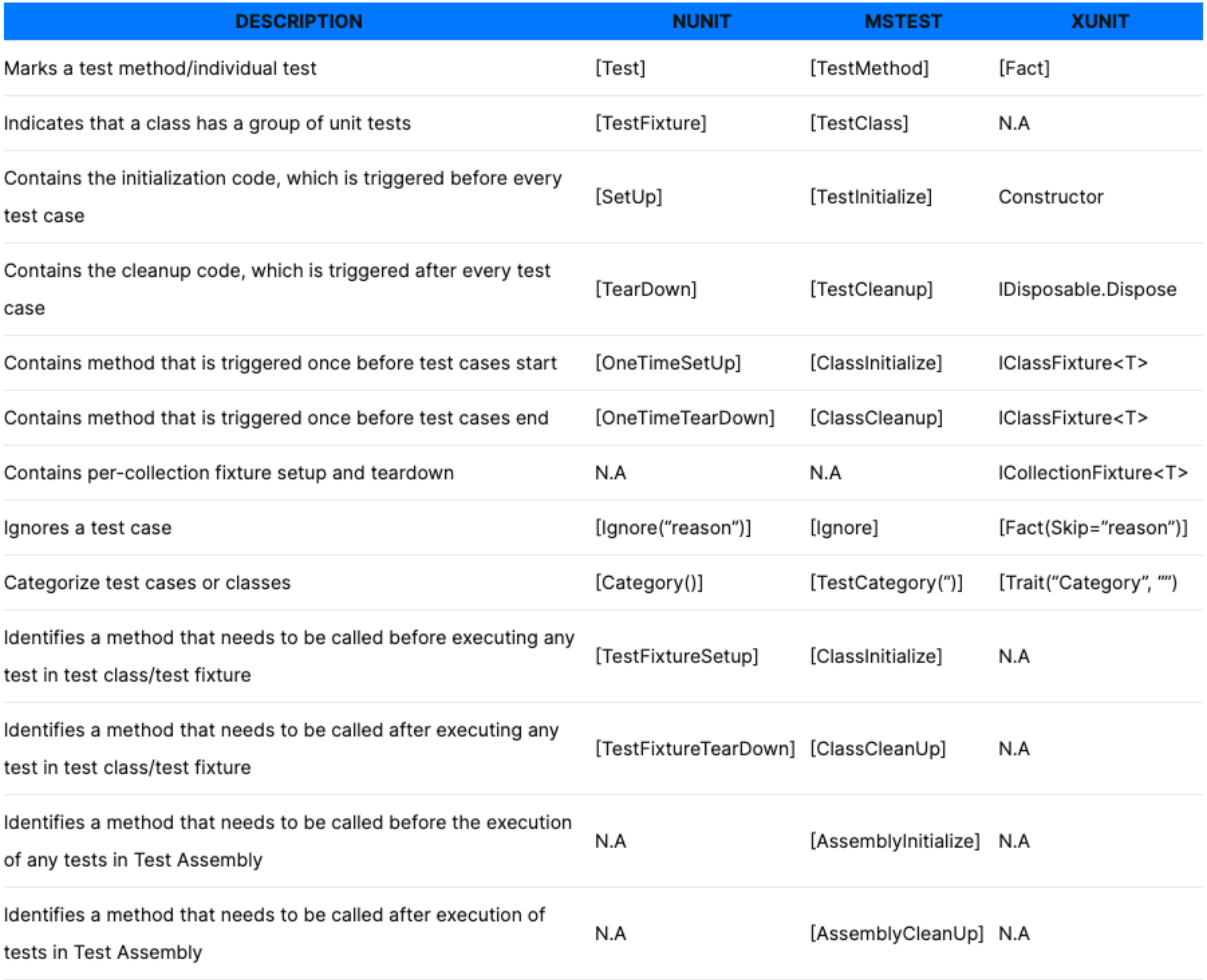
MSTest

Frameworks

- MSTest is the default test framework that is shipped along with Visual Studio
- The initial version of MSTest (V1) was not open-source, newer version is open-source
- MSTest can be run from the console
- VSTest running MSTest inside Visual Studio (simplifying a bit…:))

Attributes

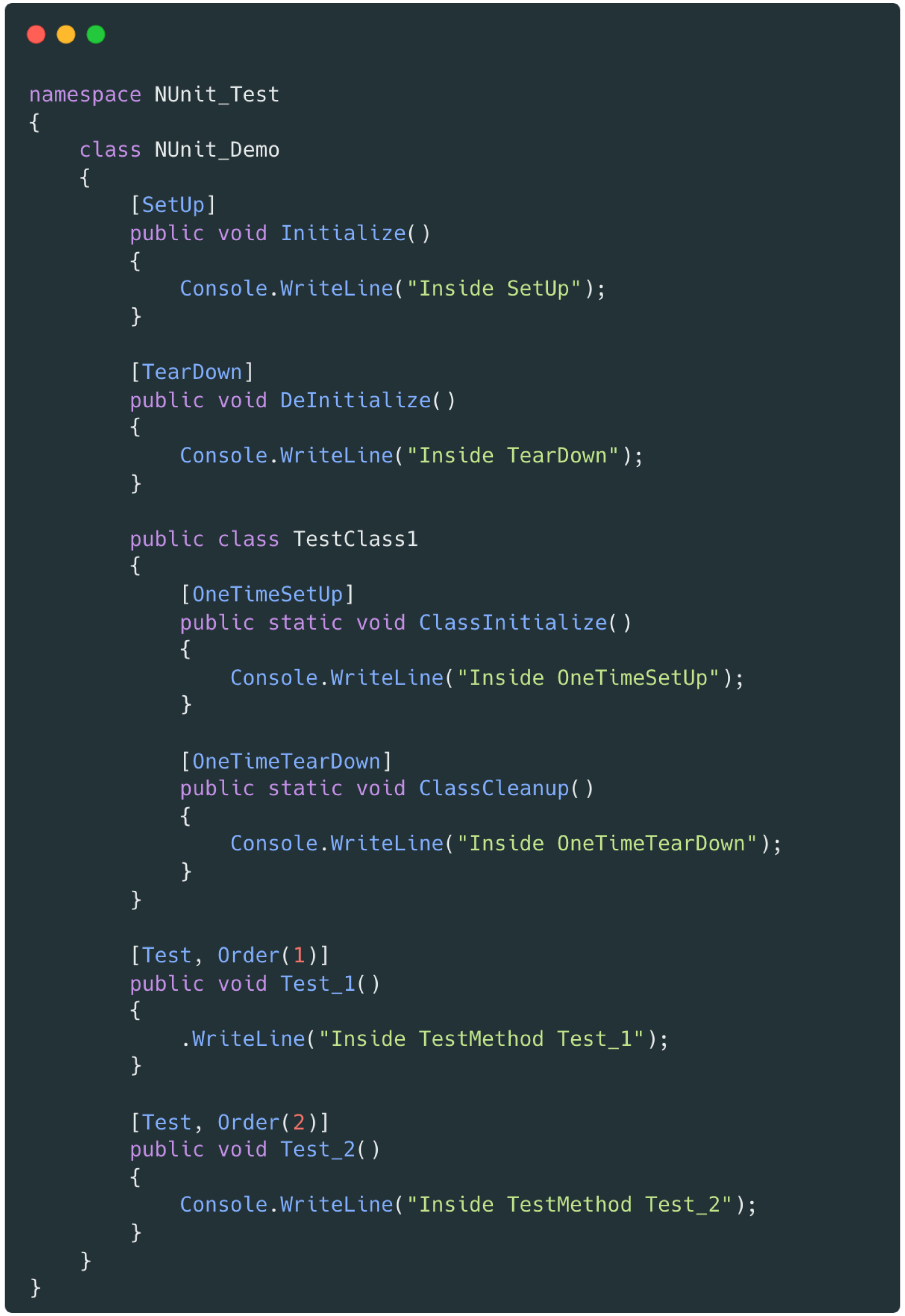
Frameworks



DESCRIPTION	NUNIT	MSTEST	XUNIT
Marks a test method/individual test	[Test]	[TestMethod]	[Fact]
Indicates that a class has a group of unit tests	[TestFixture]	[TestClass]	N.A
Contains the initialization code, which is triggered before every test case	[SetUp]	[TestInitialize]	Constructor
Contains the cleanup code, which is triggered after every test case	[TearDown]	[TestCleanup]	IDisposable.Dispose
Contains method that is triggered once before test cases start	[OneTimeSetUp]	[ClassInitialize]	IClassFixture<T>
Contains method that is triggered once before test cases end	[OneTimeTearDown]	[ClassCleanup]	IClassFixture<T>
Contains per-collection fixture setup and teardown	N.A	N.A	ICollectionFixture<T>
Ignores a test case	[Ignore(“reason”)]	[Ignore]	[Fact(Skip=”reason”)]
Categorize test cases or classes	[Category()]	[TestCategory(“)]	[Trait(“Category”, “”)
Identifies a method that needs to be called before executing any test in test class/test fixture	[TestFixtureSetup]	[ClassInitialize]	N.A
Identifies a method that needs to be called after executing any test in test class/test fixture	[TestFixtureTearDown]	[ClassCleanUp]	N.A
Identifies a method that needs to be called before the execution of any tests in Test Assembly	N.A	[AssemblyInitialize]	N.A
Identifies a method that needs to be called after execution of tests in Test Assembly	N.A	[AssemblyCleanUp]	N.A

NUnit

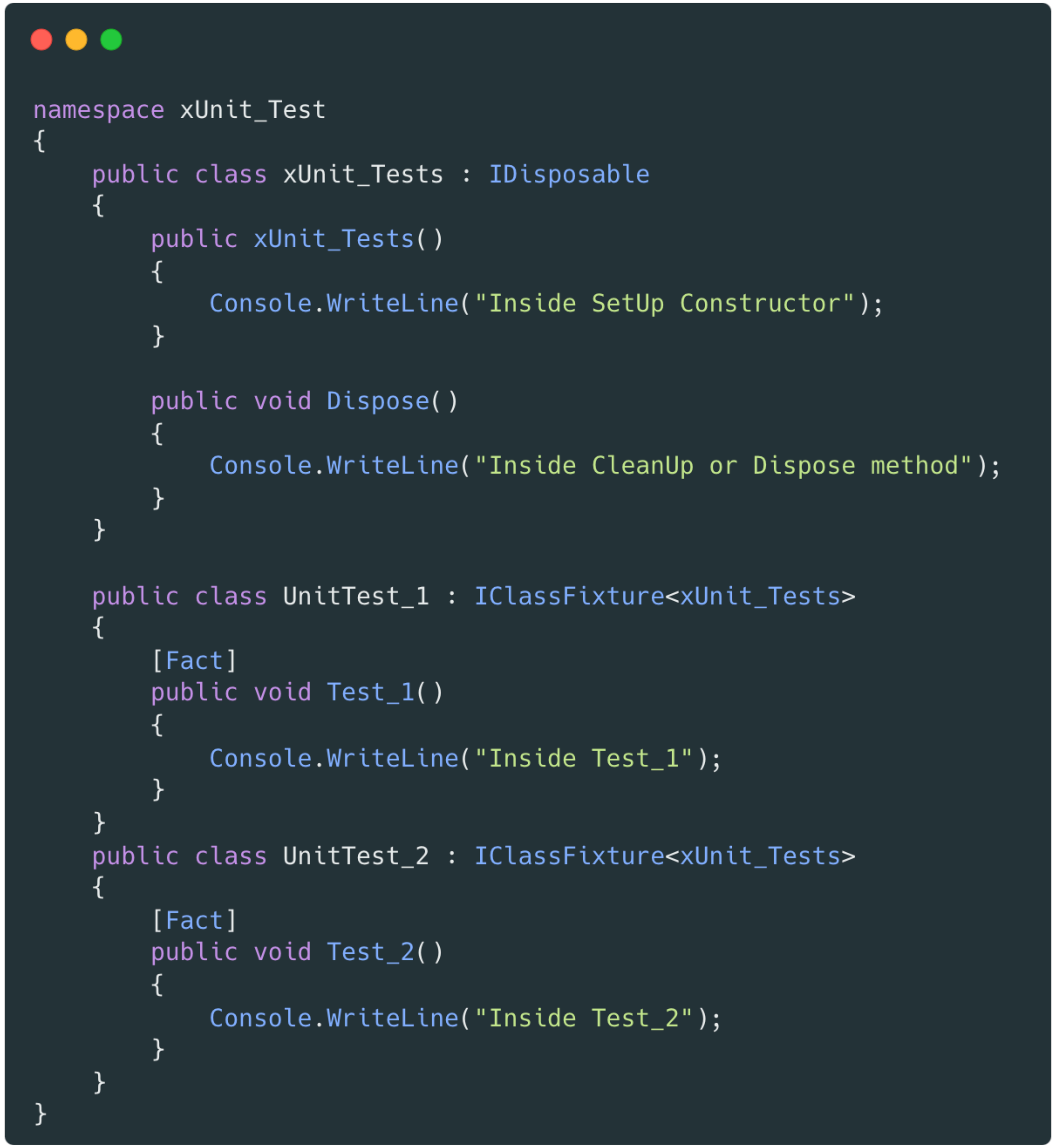
Example



```
namespace NUnit_Test
{
    class NUnit_Demo
    {
        [SetUp]
        public void Initialize()
        {
            Console.WriteLine("Inside SetUp");
        }

        [TearDown]
        public void DeInitialize()
        {
            Console.WriteLine("Inside TearDown");
        }

        public class TestClass1
        {
            [OneTimeSetUp]
            public static void ClassInitialize()
            {
                Console.WriteLine("Inside OneTimeSetUp");
            }

            [OneTimeTearDown]
            public static void ClassCleanup()
            {
                Console.WriteLine("Inside OneTimeTearDown");
            }
        }

        [Test, Order(1)]
        public void Test_1()
        {
            .WriteLine("Inside TestMethod Test_1");
        }

        [Test, Order(2)]
        public void Test_2()
        {
            Console.WriteLine("Inside TestMethod Test_2");
        }
    }
}
```

xUnit

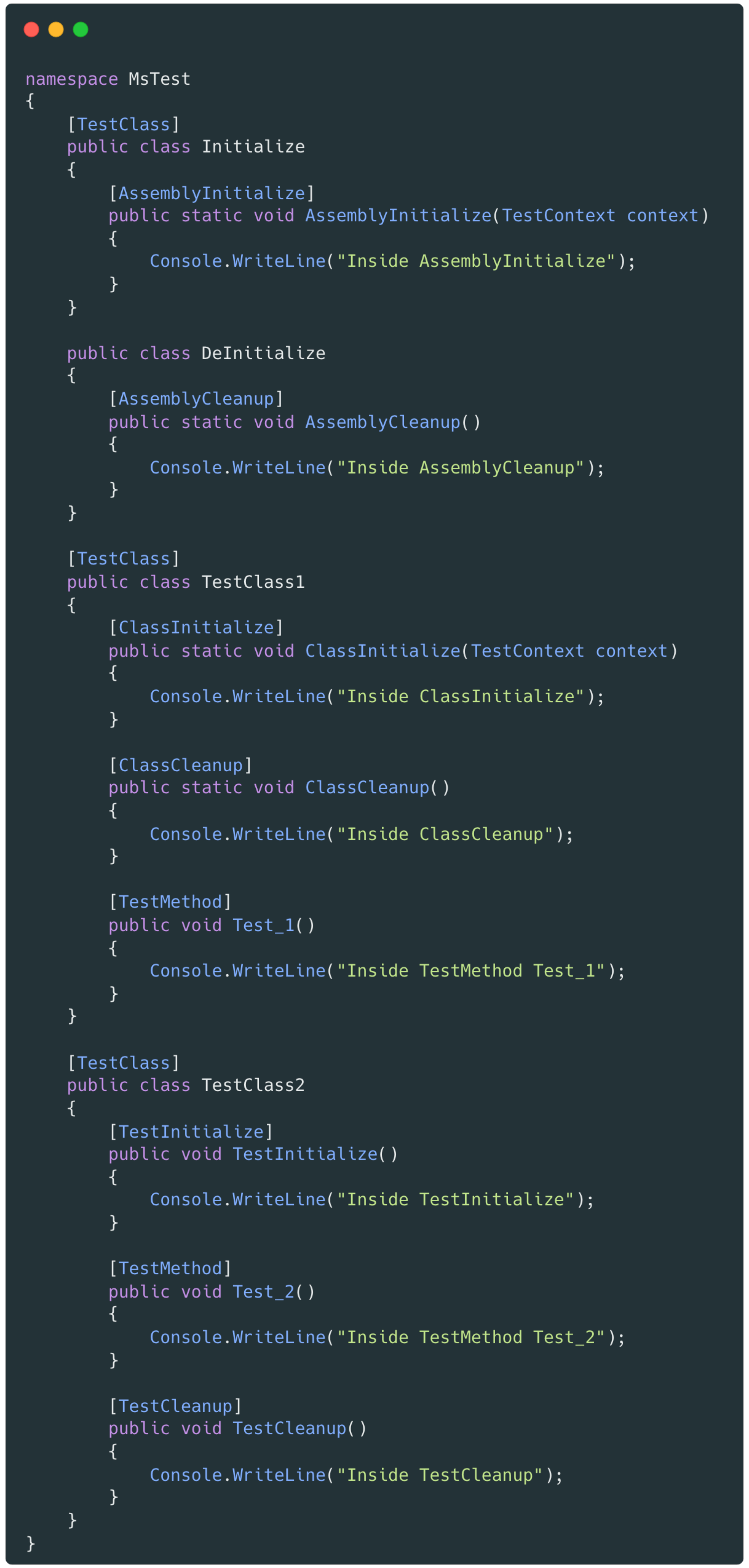
Example



```
namespace xUnit_Test
{
    public class xUnit_Tests : IDisposable
    {
        public xUnit_Tests()
        {
            Console.WriteLine("Inside SetUp Constructor");
        }

        public void Dispose()
        {
            Console.WriteLine("Inside CleanUp or Dispose method");
        }
    }

    public class UnitTest_1 : IClassFixture<xUnit_Tests>
    {
        [Fact]
        public void Test_1()
        {
            Console.WriteLine("Inside Test_1");
        }
    }
    public class UnitTest_2 : IClassFixture<xUnit_Tests>
    {
        [Fact]
        public void Test_2()
        {
            Console.WriteLine("Inside Test_2");
        }
    }
}
```

MSTest

Example

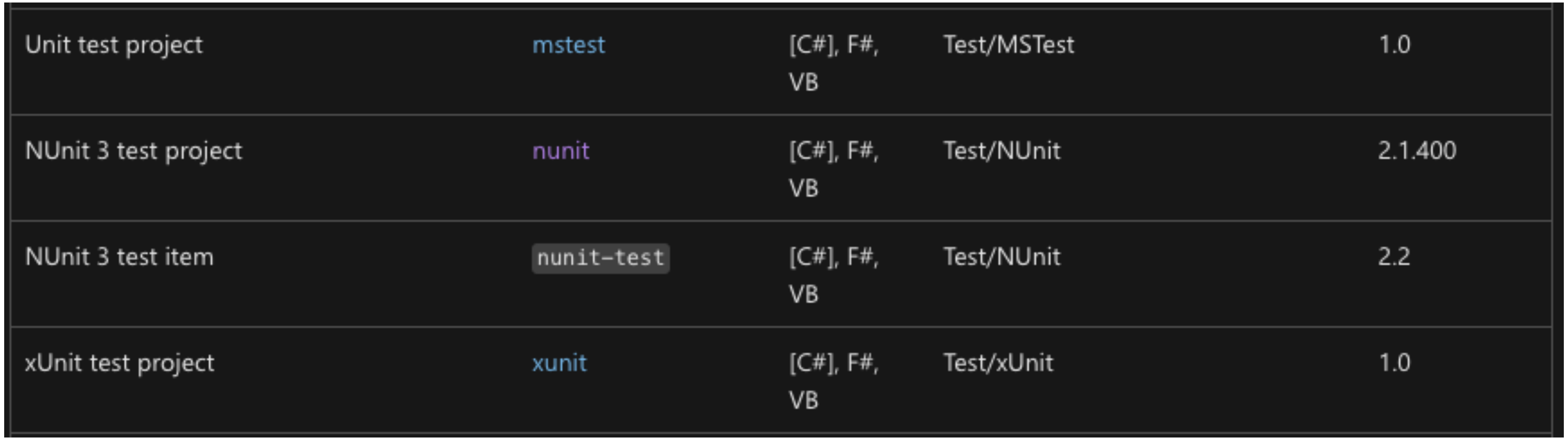


```
namespace MsTest
{
    [TestClass]
    public class Initialize
    {
        [AssemblyInitialize]
        public static void AssemblyInitialize(TestContext context)
        {
            Console.WriteLine("Inside AssemblyInitialize");
        }
    }

    public class DeInitialize
    {
        [AssemblyCleanup]
        public static void AssemblyCleanup()
        {
            Console.WriteLine("Inside AssemblyCleanup");
        }
    }

    [TestClass]
    public class TestClass1
    {
        [ClassInitialize]
        public static void ClassInitialize(TestContext context)
        {
            Console.WriteLine("Inside ClassInitialize");
        }

        [ClassCleanup]
        public static void ClassCleanup()
        {
            Console.WriteLine("Inside ClassCleanup");
        }

        [TestMethod]
        public void Test_1()
        {
            Console.WriteLine("Inside TestMethod Test_1");
        }
    }

    [TestClass]
    public class TestClass2
    {
        [TestInitialize]
        public void TestInitialize()
        {
            Console.WriteLine("Inside TestInitialize");
        }

        [TestMethod]
        public void Test_2()
        {
            Console.WriteLine("Inside TestMethod Test_2");
        }

        [TestCleanup]
        public void TestCleanup()
        {
            Console.WriteLine("Inside TestCleanup");
        }
    }
}
```

Create a Unit Test Project

Frameworks

- Using dotnet CLI templates

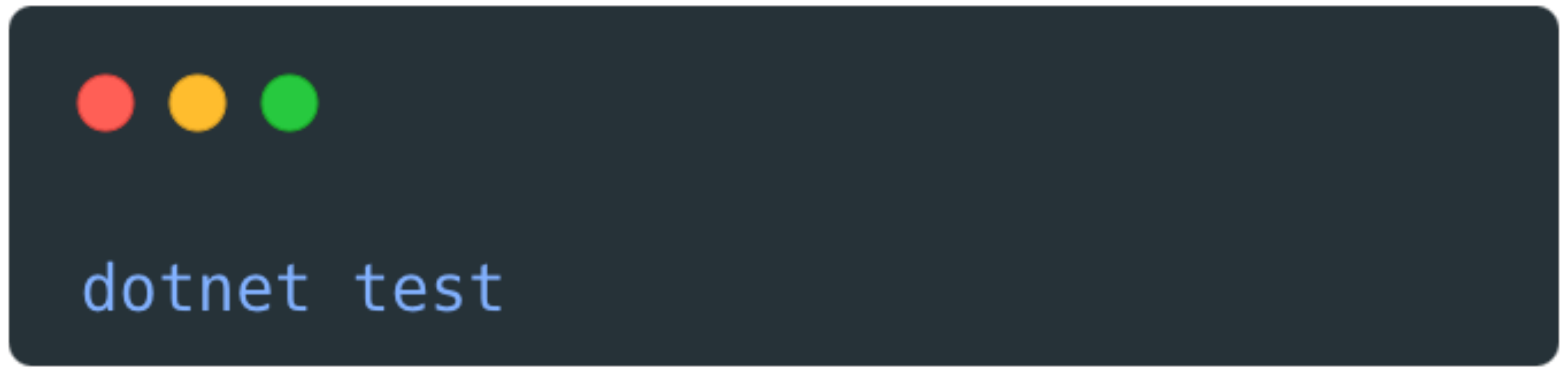


Unit test project	mstest	[C#], F#, VB	Test/MSTest	1.0
NUnit 3 test project	nunit	[C#], F#, VB	Test/NUnit	2.1.400
NUnit 3 test item	`nunit-test`	[C#], F#, VB	Test/NUnit	2.2
xUnit test project	xunit	[C#], F#, VB	Test/xUnit	1.0

Run a Unit Test Project

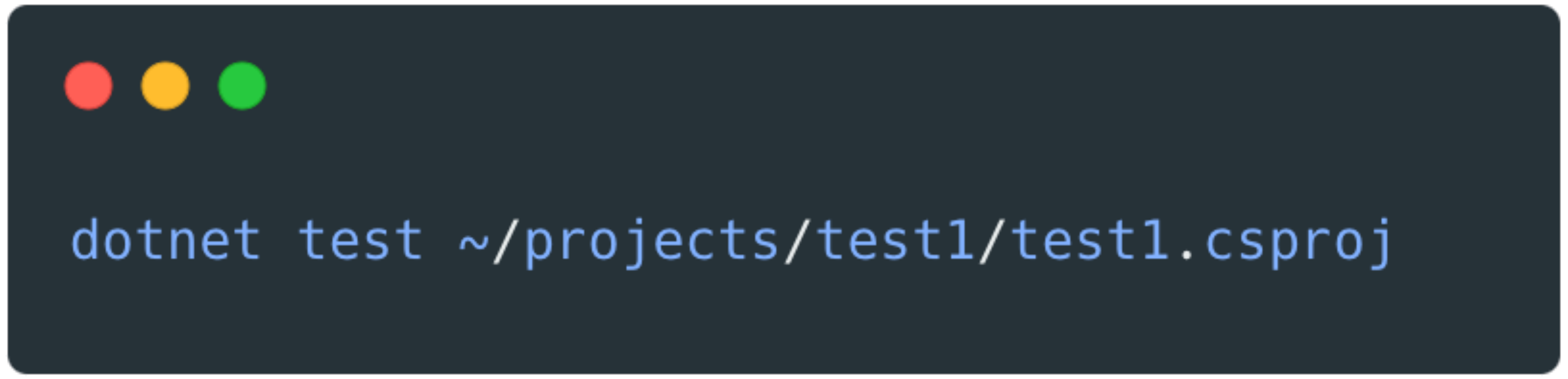
Frameworks

- Run the tests in the project in the current directory or all test projects in a solution



```
dotnet test
```

- Run the tests in the test1 project

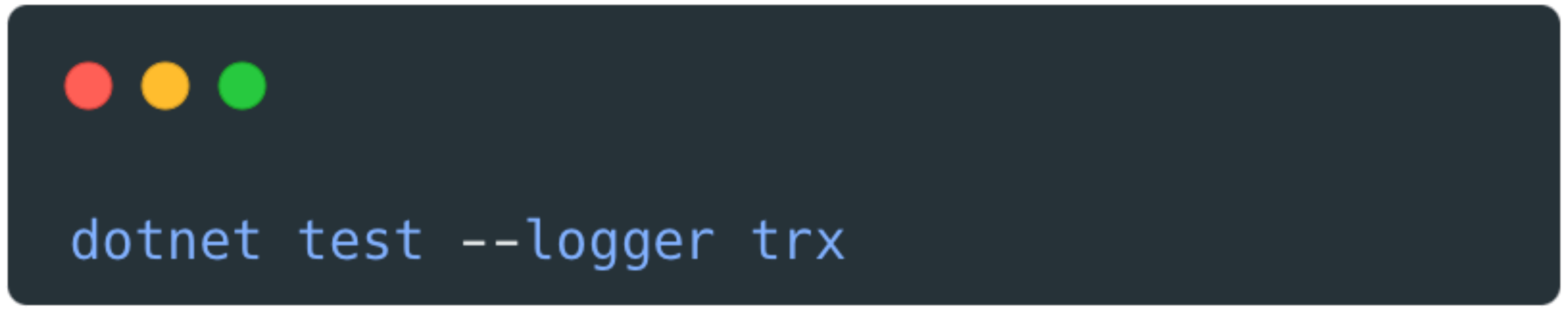


```
dotnet test ~/projects/test1/test1.csproj
```

Run a Unit Test Project

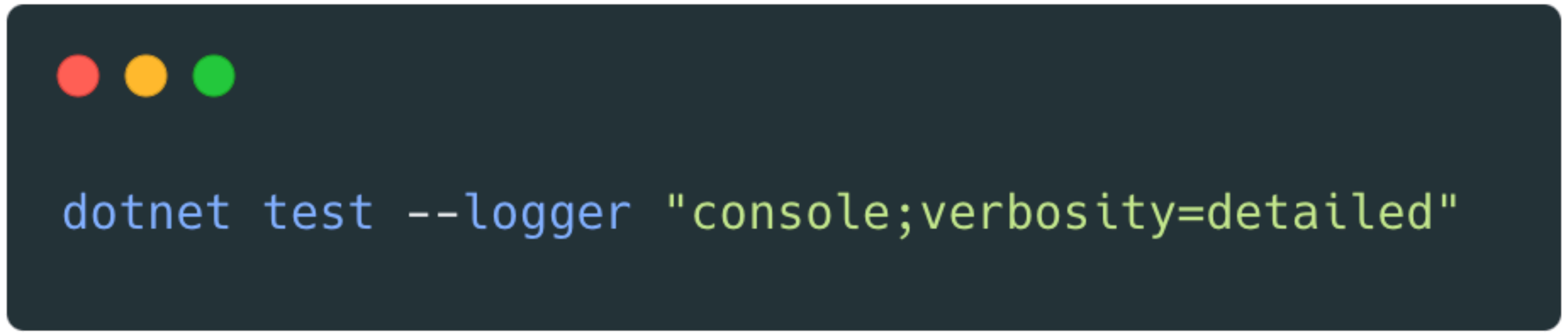
Frameworks

- Run the tests in the project in the current directory, and generate a test results file in the trx format



```
dotnet test --logger trx
```

- Run the tests in the project in the current directory, and log with detailed verbosity to the console



```
dotnet test --logger "console;verbosity=detailed"
```

Code Coverage

Code Coverage

Using xUnit

- There are two types of code coverage tools: Data Collectors and Report Generators

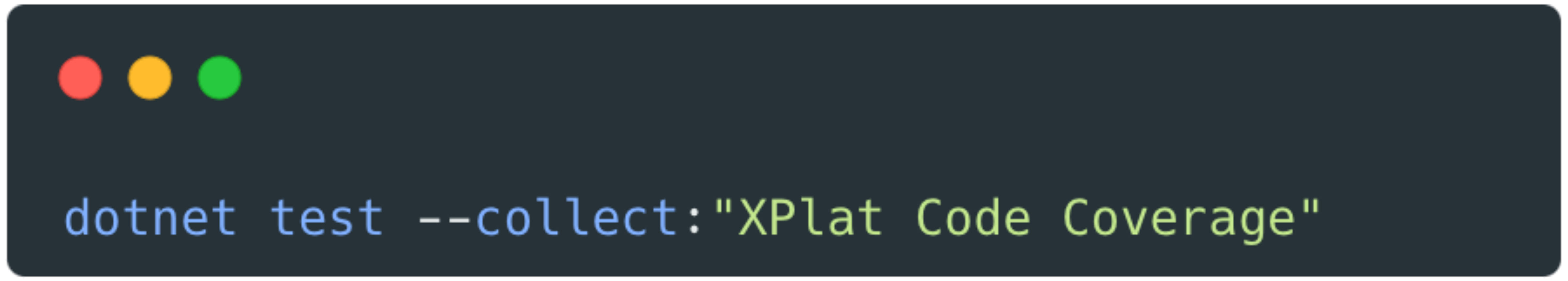
- **DataCollectors**: DataCollectors monitor test execution and collect information about test runs. They report the collected information in various output formats, such as XML and JSON

- **Report generators**: Use data collected from test runs to generate reports, often as styled HTML

Code Coverage: Collect

Using xUnit

- xUnit project is integrated with coverlet.collector by default
- With this integration you only need to use a parameter on dotnet CLI to get the data collector executed



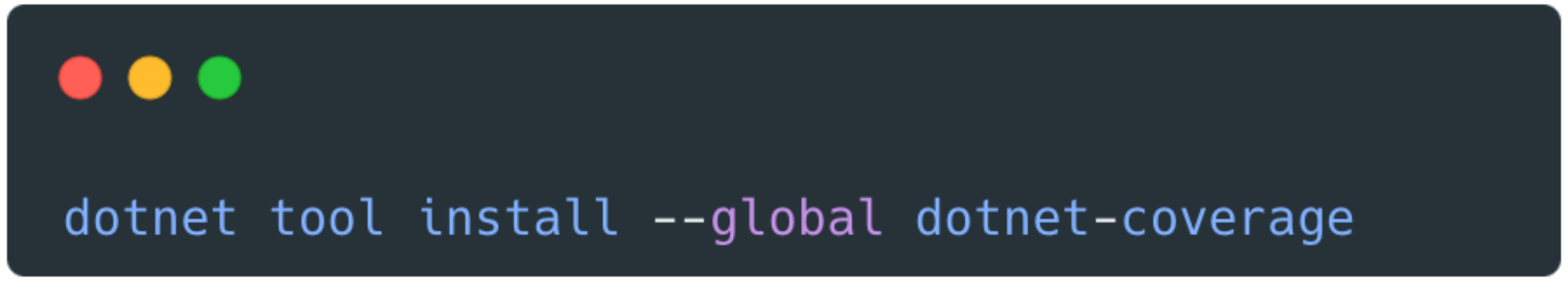
```
dotnet test --collect:"XPlat Code Coverage"
```

- This command generates a coverage.cobertura.xml that can be used by a report generator

Code Coverage: Collect

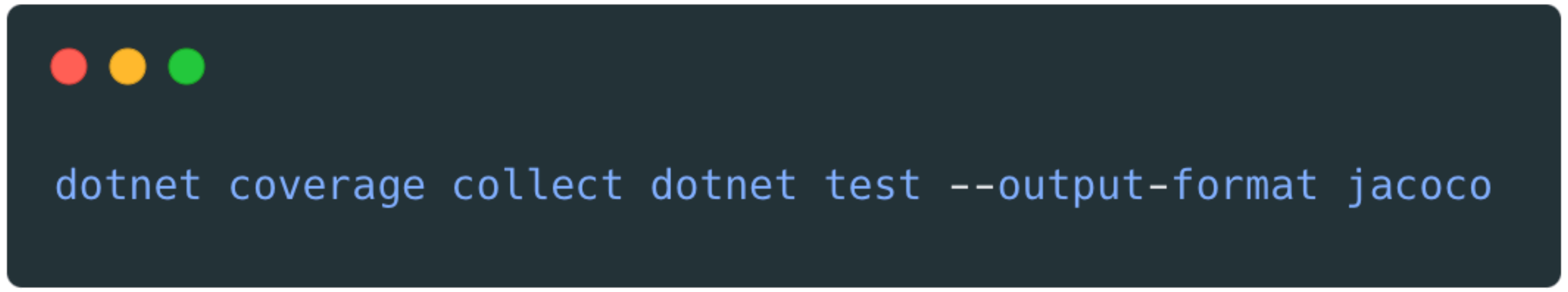
Using xUnit

- You can use other formats when collecting unit test results, like JaCoCo
- To do this you can use a tool named ‘dotnet-coverage’ that can be installed this way:



```
dotnet tool install --global dotnet-coverage
```

- After install you can use it this way to collect JaCoCo format

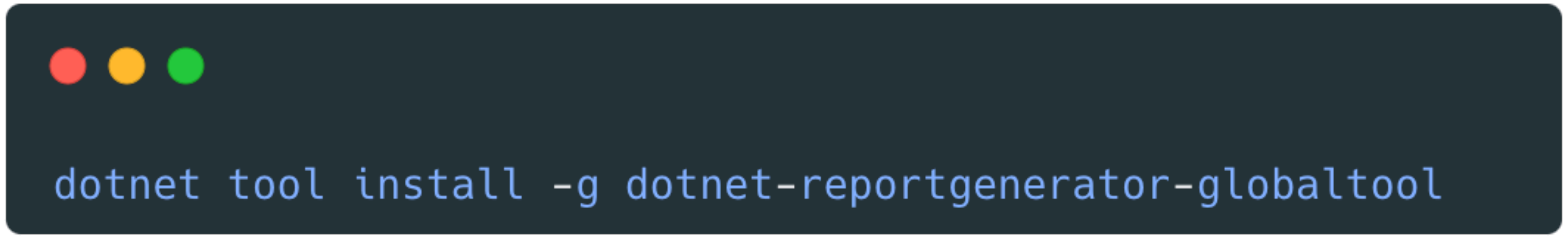


```
dotnet coverage collect dotnet test --output-format jacoco
```

Code Coverage: Report Generator

Using xUnit

- Either Cobertura or JaCoCo format, are standard formats allowing you to use several report generators available in the market, like CodeCov
- These formats can automatically interpreted by most used CI/CD platforms like Azure DevOps, GitHub, etc
- To generate a report locally, most use tool on dotnet is ReportGenerator
- You can install it using dotnet CLI

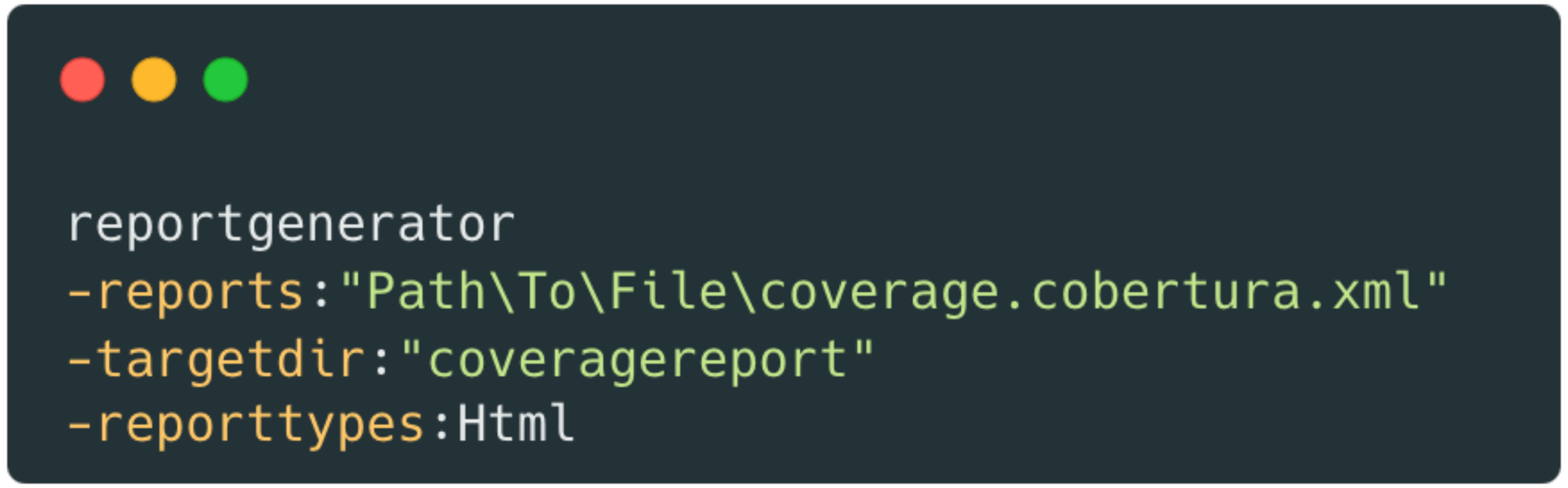


```
dotnet tool install -g dotnet-reportgenerator-globaltool
```

Code Coverage: Report Generator

Using xUnit

- After install, you can generate the report using this command



```
reportgenerator
-reports:"Path\To\File\coverage.cobertura.xml"
-targetdir:"coveragereport"
-reporttypes:Html
```

- This example generates a HTML report but other types are available like CSV, Markdown, LaTex, GitHub, Azure Pipelines, …

Mocks

Mocks

Unit Testing

- When doing unit testing there 2 types of mocks that are mostly used: services and databases
- Since .NET rely a lot on dependency injection, generate this mocks and use them on your tests is quite simple
- For mocking databases, Entity Framework Core is an excellent solution since have out-of-the-box a InMemory provider
- For mocking services, the most used library is Moq
- But since you’re using dependency injection, you can directly set new services on your scope and mock them

Unit Testing in WebAPI

Web API Minimal

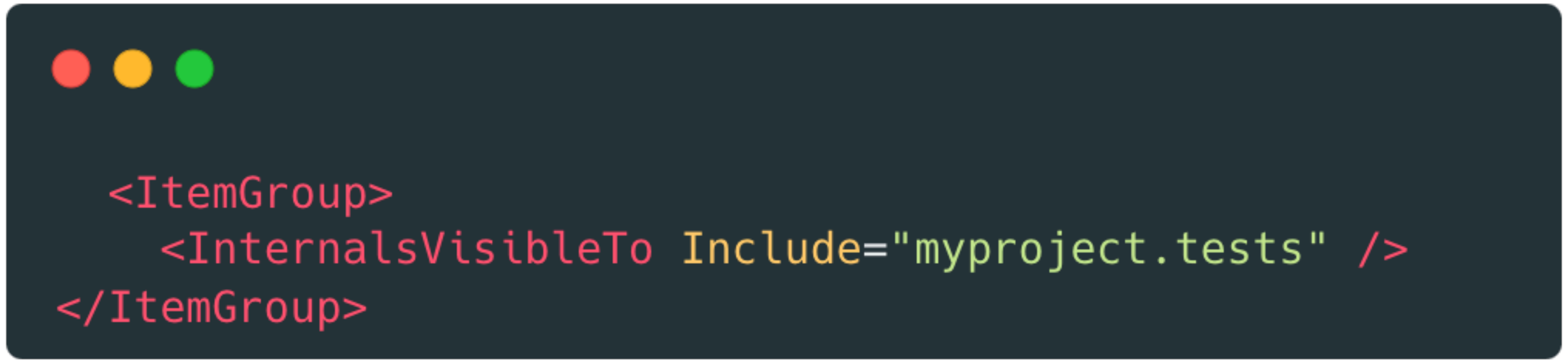
Unit Testing

- To execute unit testing on your Web API using non-minimal API approach, is like testing any other code
- When uni testing Minimal API, you need to perform some specific tasks to allow you to run this tests

Web API Minimal: Make Internals Visible

Unit Testing

- First, you need to make your main project (WebAPI) internals visible to unit testing project, changing a property on .csproj file



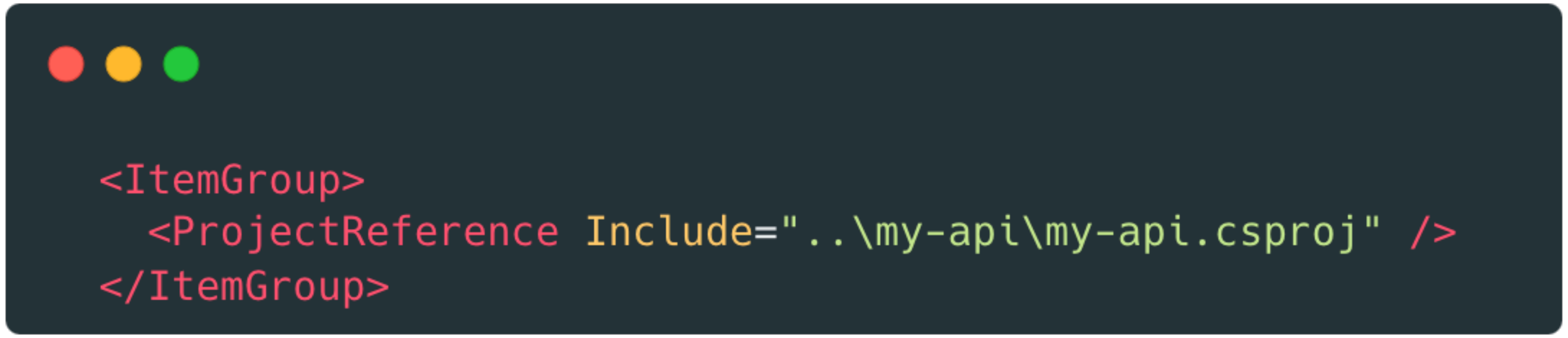
```
  <ItemGroup>
    <InternalsVisibleTo Include="myproject.tests" />
</ItemGroup>
```

- This is needed since on Minimal API you don’t have a class that runs your main code

Web API Minimal: Reference Project

Unit Testing

- Then you need to create a reference on your unit test project to your main project to be able to execute its code

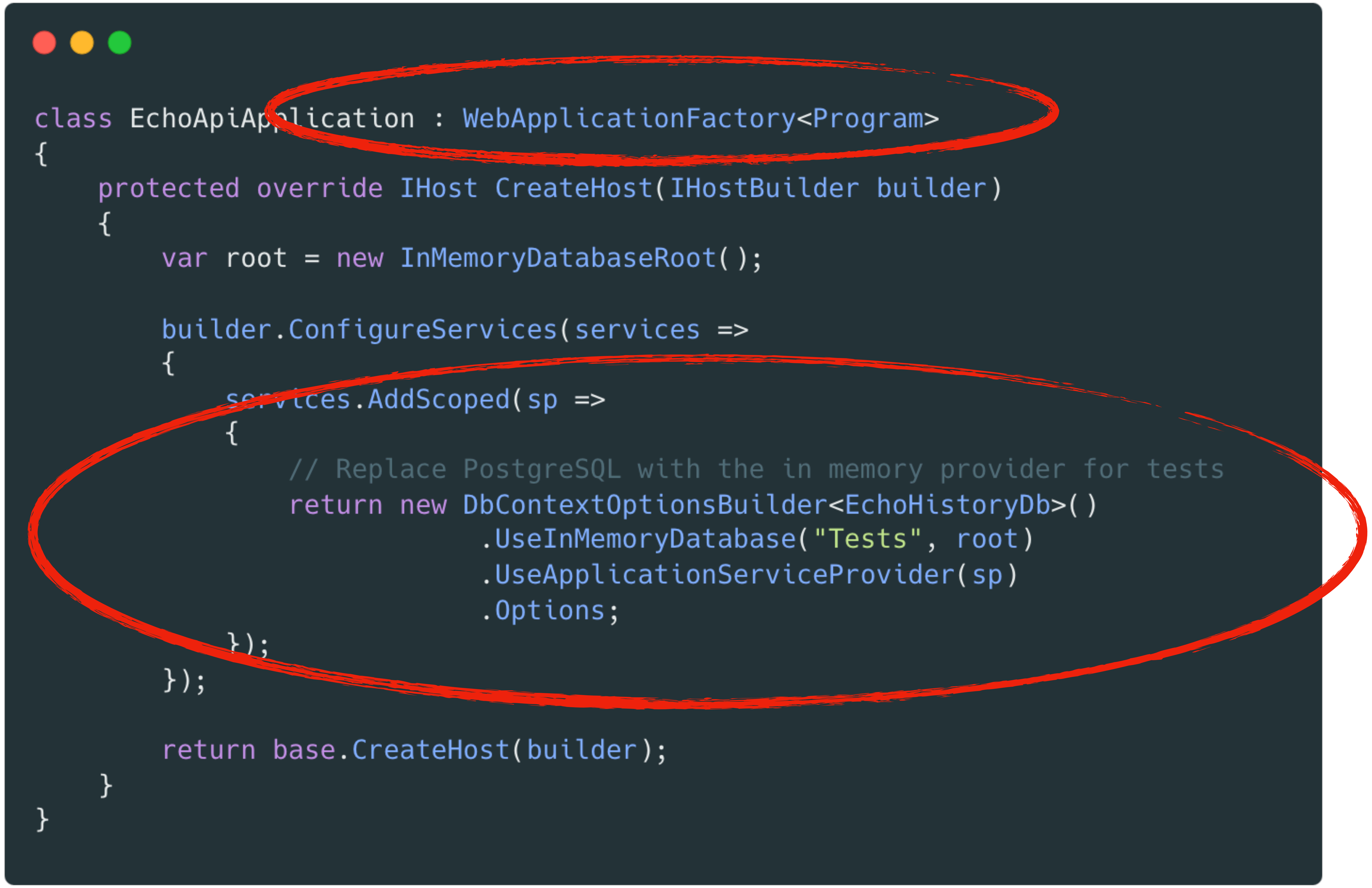


```xml
<ItemGroup>
  <ProjectReference Include="..\my-api\my-api.csproj" />
</ItemGroup>
```

- This is needed since you want to execute main project code

Web API Minimal: Mock your API

Unit Testing

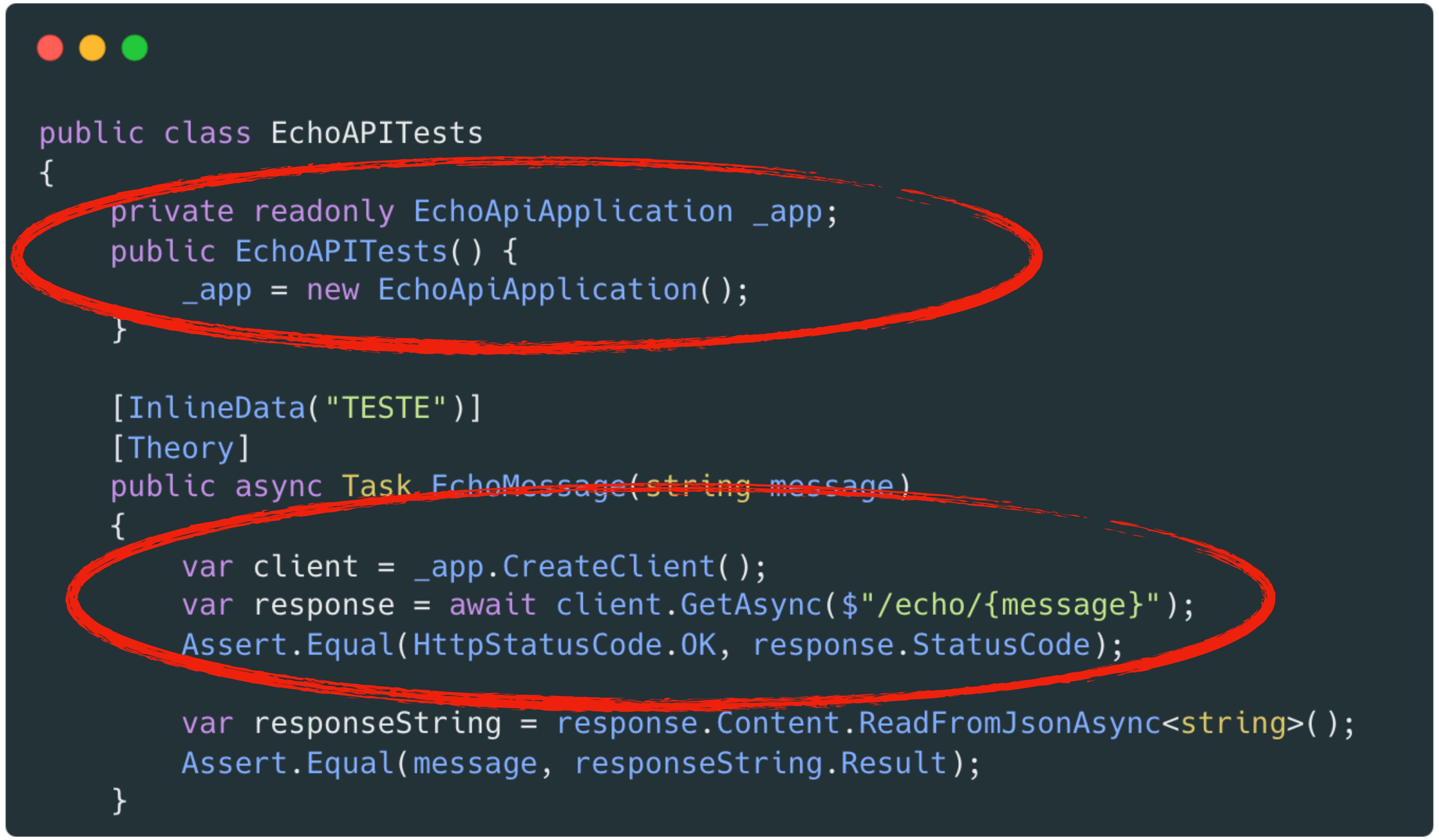


```
class EchoApiApplication : WebApplicationFactory<Program>
{
    protected override IHost CreateHost(IHostBuilder builder)
    {
        var root = new InMemoryDatabaseRoot();

        builder.ConfigureServices(services =>
        {
            services.AddScoped(sp =>
            {
                // Replace PostgreSQL with the in memory provider for tests
                return new DbContextOptionsBuilder<EchoHistoryDb>()
                              .UseInMemoryDatabase("Tests", root)
                              .UseApplicationServiceProvider(sp)
                              .Options;
            });
        });

        return base.CreateHost(builder);
    }
}
```

Web API Minimal: Execute your API

Unit Testing

- Finally on your unit test class you can generate a WebApplication object that will reference your API



```
public class EchoAPITests
{
    private readonly EchoApiApplication _app;
    public EchoAPITests() {
        _app = new EchoApiApplication();
    }

    [InlineData("TESTE")]
    [Theory]
    public async Task EchoMessage(string message)
    {
        var client = _app.CreateClient();
        var response = await client.GetAsync($"/echo/{message}");
        Assert.Equal(HttpStatusCode.OK, response.StatusCode);

        var responseString = response.Content.ReadFromJsonAsync<string>();
        Assert.Equal(message, responseString.Result);
    }
```

Q&A

Run Unit Tests on your WebAPI

Lab #05

dotnet training

Run Unit Tests on your WebAPI

Lab #05

- Learning Objectives
    - Create a unit test
    - Use dependency injection for make unit testing easier
    - Create a mock for your database
    - Run dotnet test
    - Get code coverage

- MD Link: https://github.com/tasb/dotnet-training/blob/main/labs/lab05.md
- HTML Link: https://tasb.github.io/dotnet-training/labs/lab05.html