Code Academy

Lecturer

Rokas Slaboševičius

the most interesting part API Unit Testing

Data

Today you will learn

01 Moq

02 Autofixture

03 xUnit testing

04 SpecimenBuilder

NUnit vs xUnit

- We will write the tests in the xUnit project, because the main difference between the NUnit and xUnit projects is that xUnit re-creates the data for each test, while NUnit reuses it, so the tests are not so well isolated from each other.

Moq and Autofixture

To test functionality, you need data to start with.

For mocking simple data there is a very good framework called Autofixture, it brings values into objects as a human would do, and all you need to do is to create an object in the test parameters.

```
public class UnitTest1
{
    [Theory, AutoData]
    public void Test1(Car car)
    {
        // Arrange
        var repositoryMock = new Mock<ICarRepository>();
        var mapperMock = new Mock<IMapper>();
        var loggerMock = new Mock<ILogger<CarController>>();
        var sut = new CarController(repositoryMock.Object, mapperMock.Object, loggerMock.Object);
        repositoryMock.Setup(x => x.GetCar(It.IsAny<Guid>())).Returns(car);
        // Act
        var testResponse = sut.GetCarById(It.IsAny<Guid>());
        //Assert
        Assert.Equal(testResponse, car);
    }
}
```

Moq and Autofixture

Next comes the mocking of services with the Moq framework.

Since services are operated through intrafaces with DI(dependency injection) we can create an **implementation of the service** for testing

In this case, we are testing the Controller, so when we create it, we call it **a sut (subject under test)** and through parameters we pass the **.Object** value, which represents the test interface implementation

```
public class UnitTest1
{
    [Theory, AutoData]
    public void Test1(Car car)
    {
        // Arrange
        var repositoryMock = new Mock<ICarRepository>();
        var mapperMock = new Mock<IMapper>();
        var loggerMock = new Mock<ILogger<CarController>>();
        var sut = new CarController(repositoryMock.Object, mapperMock.Object, loggerMock.Object);
        repositoryMock.Setup(x => x.GetCar(It.IsAny<Guid>())).Returns(car);
        // Act
        var testResponse = sut.GetCarById(It.IsAny<Guid>());
        //Assert
        Assert.Equal(testResponse, car);
    }
}
```

Moq and Autofixture

```
repositoryMock.Setup(x => x.GetCar(It.IsAny<Guid>())).Returns(car);
```

The next line is responsible for creating the test script we want.

Since unit tests are responsible for a small part of the code, we have to isolate the scripts we want to test.

The controller under test has two paths - when the car object is found in the repository and when it is not found.

We are trying to test the scenario where an object is found and returned.

We therefore need the repository to return the object.

In the lambda function, we indicate that we want to setup the **GetCar** method and also specify,

that any parameter can be passed to the **GetCar** method during the test. Finally, we say that in this repository call scenario, the object created in our test with Autofixture will be returned

Moq and Autofixture

At the end Assert whether what the controller returned is the same as what we said it would return in our script

```
public class UnitTest1
{
    [Theory, AutoData]
    public void Test1(Car car)
    {
        // Arrange
        var repositoryMock = new Mock<ICarRepository>();
        var mapperMock = new Mock<IMapper>();
        var loggerMock = new Mock<ILogger<CarController>>();
        var sut = new CarController(repositoryMock.Object, mapperMock.Object, loggerMock.Object);
        repositoryMock.Setup(x => x.GetCar(It.IsAny<Guid>())).Returns(car);
        // Act
        var testResponse = sut.GetCarById(It.IsAny<Guid>());
        //Assert
        Assert.Equal(testResponse, car);
    }
}
```

Moq and Autofixture

Pay attention to the names of the tests - they can be very long, but they must be descriptive of the scenario being tested.

Also, if you don't pass anything in the parameters, you can use the attribute

[Fact] instead of **[Theory, AutoData]**

```
[Theory, AutoData]
public void Repository_Returns_Car_Object_Correctly(Car car)
{
```

Moq and Autofixture

Testing another scenario where the repository cannot find an object in the database.

Note that I have indicated that the repository will return null.

Since the controller has received a null value returns NotFound(), we need check whether the controller response.GetType() is NotFoundObjectResult

```
[Fact]
public void Repository_Returns_Null()
{
    // Arrange
    var repositoryMock = new Mock<ICarRepository>();
    var mapperMock = new Mock<IMapper>();
    var loggerMock = new Mock<ILogger<CarController>>();
    var sut = new CarController(repositoryMock.Object, mapperMock.Object, loggerMock.Object);
    repositoryMock.Setup(x => x.GetCar(It.IsAny<Guid>())).Returns((Car)null);
    // Act
    var testResponse = sut.GetCarById(It.IsAny<Guid>());
    //Assert
    var type = testResponse.Result.GetType();
    Assert.Equal("NotFoundObjectResult", type.Name);
}
```

Additional SpecimenBuilder

Sometimes the way Autofixture creates objects for us may not be suitable, in which case we need to create our own SpecimenBuilder, which specifies what the object should look like

SpecimenBuilder implementation

```
public class CarSpecimenBuilder : ISpecimenBuilder
{
    public object Create(object request, ISpecimenContext context)
    {
        if (request is Type type && type == typeof(Car))
        {
            return new Car { Color = "White", Id = Guid.NewGuid(), Model = "Bmw" };
        }
        return new NoSpecimen();
    }
}
```

SpecimenBuilder implementation

```
private readonly Fixture _fixture = new Fixture();

public CarControllerTests()
{
    _fixture.Customizations.Add(new CarSpecimenBuilder());
}

[Theory, AutoData]
public void Repository_Returns_Car_Object_Correctly()
{

    // Arrange
    var car = _fixture.Create<Car>();
    var repositoryMock = new Mock<ICarRepository>();
    var mapperMock = new Mock<IMapper>();
    var loggerMock = new Mock<ILogger<CarController>>();
    var sut = new CarController(repositoryMock.Object, mapperMock.Object, loggerMock.Object);
    repositoryMock.Setup(x => x.GetCar(It.IsAny<Guid>())).Returns(car);
    // Act
    var testResponse = sut.GetCarById(It.IsAny<Guid>());
    //Assert
    Assert.Equal(testResponse.Value, car);
}
```

Creating an AutoDataAttribute

To further optimise the code, we can create our own AutoDataAttribute.

Previously, we saw that the Theory attribute also used the AutoData attribute

```
[Theory, AutoData]
public void Repository_Returns_Car_Object_Correctly(Car car)
{
```

But the AutoData attribute does not use the SpecimenBuilder we created.

We create our own attribute.

Creating an AutoDataAttribute

This is what the class will look like:

Inheriting from AutoDataAttribute, we can use it as an attribute with Theory.

In the constructor, we specify to use base and add a return to the Fixture object we created earlier in the test class.

```
public class CustomDataAttribute : AutoDataAttribute
{
    public CustomDataAttribute() : base(() =>
    {
        var fixture = new Fixture();
        fixture.Customizations.Add(new CarSpecimenBuilder());
        return fixture;
    })
    { }
}
```

Creating an AutoDataAttribute

Change AutoData to CustomDataAttribute and return the Car object to the parameters.

Now this object will be created as we specified in the SpecimenBuilder and there is no need to create a Fixture in the tests themselves.

```
public class CarControllerTests
{

    [Theory, CustomDataAttribute]
    public void Repository_Returns_Car_Object_Correctly(Car car)
    {
        // Arrange
        var repositoryMock = new Mock<ICarRepository>();
        var mapperMock = new Mock<IMapper>();
        var loggerMock = new Mock<ILogger<CarController>>();
        var sut = new CarController(repositoryMock.Object, mapperMock.Object, loggerMock.Object);
        repositoryMock.Setup(x => x.GetCar(It.IsAny<Guid>())).Returns(car);
        // Act
        var testResponse = sut.GetCarById(It.IsAny<Guid>());
        //Assert
        Assert.Equal(testResponse.Value, car);
    }
```

Fun fact!

We can write the attribute name without the word Attribute, in this case it will be CustomData

```
[Theory, CustomData]
public void Repository_Returns_Car_Object_Correctly(Car car)
{
```

Task 1

- Test the programme for Lecture 3.
- Create both your SpecimenBuilder and attribute.