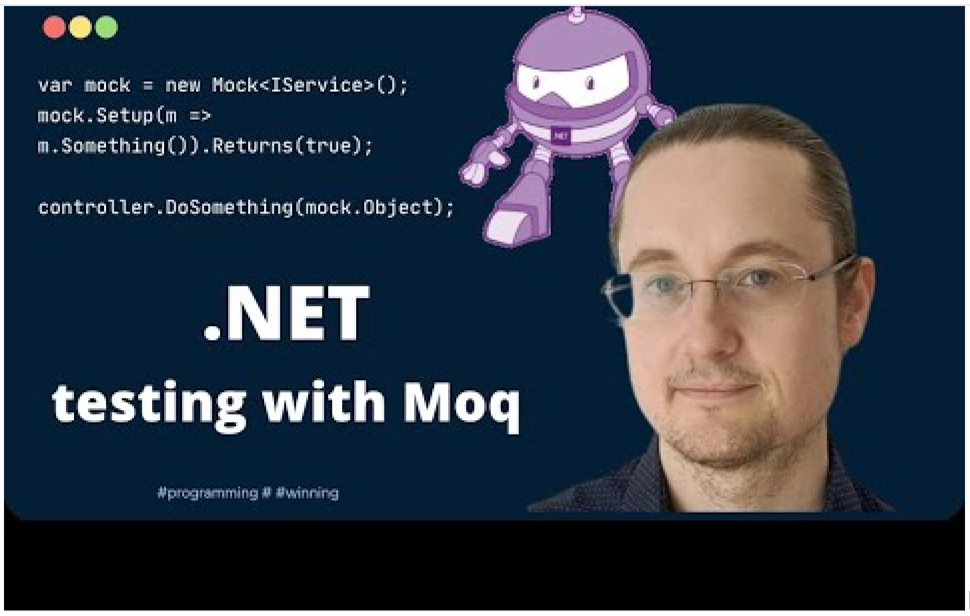
How YOU can Learn Mock testing in .NET Core and C# with Moq

Follow me on Twitter, happy to take your suggestions on topics or improvements /Chris

> Check out a video version



```
var mock = new Mock<IService>();
mock.Setup(m =>
m.Something()).Returns(true);

controller.DoSomething(mock.Object);
```

Introduction

> When we test we just want to test one thing - the business logic of the method. Often our method needs the help of dependencies to be able to carry out its job properly. Depending on what these dependencies answer - there might be several paths through a method. So what is Mock testing? It's about testing only one thing, in isolation, by mocking how your dependencies should behave.

In this article we will cover the following:

- **Why test**, it's important to understand why we test our code. Is it to ensure our code works? Or maybe we are adding tests for defensive reasons so that future refactors don't mess up the business logic?
- **What to test**, normally this question has many answers. We want to ensure that our method does what it says it does, e.g `1+1` equals `2`. We might also want to ensure that we test all the different paths through the method, the *happy path* as well as alternate/erroneous paths. Lastly, we might want to assert that a certain *behavior* takes place.

above.

References

- xUnit testing This page describes how to use xUnit with .Net Core
- nUnit testing This page describes how to use nUnit with .Net Core.
- dotnet test, terminal command description This page describes the terminal command `dotnet test` and all the different arguments you can call it with.
- dotnet selective test This page describes how to do selective testing and how to set up filters and query using filters.
- Get started with C# and VS Code
- .Net Core Series on NuGet, Serverless and much more
- Moq tutorial This is the GitHub repo for the Moq library. The README contains a tutorial.

Why test

As we mentioned already there are many answers to this question. So how do we know? Well, I usually see the following reasons:

- **Ensuring Quality**, because I'm not an all-knowing being I will make mistakes. Writing tests ensures that at least the worst mistakes are hard to tell whether it lends itself to be tested. Of course, I need to ask myself at this point whether this code should be tested. My advice here if it's not obvious what running the method will produce or if there is more than one execution path - it should be tested.

- **Being defensive**, you have a tendency to maintain software over several years. The people doing the maintaining might be you or someone else. One way to communicate what code is important is to write tests that absolutely should work regardless of what refactorings you, or anyone else, attempts to carry out.
- **Documentation**, documentation sounds like a good idea at first but we all know that out of sync documentation is worse than no documentation. For that reason, we tend to not write it in the first place, or maybe feel ok with high-level documentation only or rely on tools like Swagger for example. Believe it or not but tests are usually really good documentation. It's one developer to another saying, *this is how I think the code should be used*. So for the sake of that future maintainer, communicate what your intentions were/are.

What to test

So what should we test? Well, my first response here is all the paths through the method. The happy path as well as alternate paths.

My second response is to understand whether we are testing a function to produce a certain result like `1+1` equals `2` or whether it's more a *behavior* like - we should have been paid before we can ship the items in the cart.

≡

Demo - let's test it

What are we doing? Well, we have talked repeatedly about that Shopping Cart in an e-commerce application so let's use that as an example for our demo.

This is clearly a case of *behavior* testing. We want the Cart items to be shipped to a customer providing we got paid. That means we need to verify that the payment is carried out correctly and we also need a way to assert what happens if the payment fails.

We will need the following:

- **A `CartController`**, will contain logic such as trying to get paid for a cart's content. If we are successfully paid then ship the items in the cart to a specified address.
- **Helper services**, we need a few helper services to figure this out like:
  - `ICartService`, this should help us calculate how much the items in cart costs but also tell us exactly what the content is so we can send this out to a customer once we have gotten paid.
  - `IPaymentService`, this should charge a card with a specified sum
  - `IShipmentService`, this should be able to ship the cart content to a specific address

Creating the code

We will need two different .NET Core projects for this:

services.
- **a test project**, this project will contain all the tests and a reference to the above project.

The API project

For this project, this could be either an app using the template `mvc` , `webapp` or `webapi`

First, let's create a solution. Create a directory like so:

```
mkdir <new directory name>
cd <new directory name>
```

Thereafter create a new solution like so:

```
dotnet new sln
```

To create our API project we just need to instantiate it like so:

```
dotnet new webapi -o api
```

and lastly add it to the solution like so:

```
dotnet sln add api/api.csproj
```

Controllers/CartController.cs

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Mvc;
using Services;

namespace api.Controllers
{
  [ApiController]
  [Route("[controller]")]
  public class CartController
  {
    private readonly ICartService _cartService;
    private readonly IPaymentService _paymentService;
    private readonly IShipmentService _shipmentService;

    public CartController(
      ICartService cartService,
      IPaymentService paymentService,
      IShipmentService shipmentService
    )
    {
      _cartService = cartService;
      _paymentService = paymentService;
      _shipmentService = shipmentService;
    }

    [HttpPost]
    public string CheckOut(ICard card, IAddressInfo addressInfo)
    {
```

☰

```
        {
            _shipmentService.Ship(addressInfo, _cartService.Items
            return "charged";
        }
        else {
            return "not charged";
        }
    }
  }
}
```

Ok, our controller is created but it has quite a few dependencies in place that we need to create namely `ICartService` , `IPaymentService` and `IShipmentService` .

Note how we will not create any concrete implementations of our services at this point. We are more interested in establishing and testing the behavior of our code. That means that concrete service implementations can come later.

Services/ICartService.cs

Create the file `ICartService.cs` under the directory `Services` and give it the following content:

```
namespace Services
{
  public interface ICartService
  {
  }
}
```

This interface is just a representation of a shopping cart and is able to tell us what is in the cart through the method `Items()` and how to calculate its total value through the method `Total()` .

Services/IPaymentService.cs

Let's create the file `IPaymentService.cs` in the directory `Services` and give it the following content:

```
namespace Services
{
  public interface IPaymentService
  {
    bool Charge(double total, ICard card);
  }
}
```

Now we have a payment service that is able to take `total` for the amount to be charged and `card` which is debit/credit card that contains all the needed information to be charged.

Services/IShipmentService.cs

For our last service let's create the file `IShipmentService.cs` under the directory `Services` with the following content:

≡

```
using System.Generic;

namespace Services
{
  public interface IShipmentService
  {
    void Ship(IAddressInfo info, IEnumerable<CartItem> items);
  }
}
```

This contains a method `Ship()` that will allow us to ship a cart's content to the customer.

Services/Models.cs

Create the file `Models.cs` in the directory `Services` with the following content:

```
namespace Services
{
  public interface IAddressInfo
  {
    public string Street { get; set; }
    public string Address { get; set; }
    public string City { get; set; }
    public string PostalCode { get; set; }
    public string PhoneNumber { get; set; }
  }

  public interface ICard
  {
```

☰

```
    public DateTime ValidTo { get; set; }
  }

  public interface CartItem
  {
    public string ProductId { get; set; }
    public int Quantity { get; set; }
    public double Price{ get; set; }
  }
}
```

This contains some supporting interfaces that we need for our services.

Creating a test project

Our test project is interested in testing the behavior of `CartController` . First off we will need a test project. There are quite a few test templates supported in .NET Core like `nunit` , `xunit` and `mstest` . We'll go with `nunit` .

To create our test project we type:

```
dotnet new nunit -o api.test
```

Let's add it to the solution like so:

Thereafter add a reference of the API project to the test project, so we are able to test the API project:

```
dotnet add test/test.csproj reference api/api.csproj
```

Finally, we need to install our mocking library `moq` , with the following command:

```
dotnet add package moq
```

Moq, how it works

Let's talk quickly about our Mock library `moq` . The idea is to create a concrete implementation of an interface and control how certain methods on that interface responds when called. This will allow us to essentially test all of the paths through code.

Creating our first Mock

Let's create our first Mock with the following code:

```
var paymentServiceMock = new Mock<IPaymentService>();
```

The above is not a concrete implementation but a Mock object. A Mock can be:

- **Verified**, verification is something you carry out after your production code has been called. You carry this out to verify that a certain method has been called with specific arguments

Instruct our Mock

Now we have a Mock object that we can instruct. To instruct it we use the method `Setup()` like so:

```
paymentServiceMock.Setup(p => p.Charge()).Returns(true)
```

Of course, the above won't compile, we need to give the `Charge()` method the arguments it needs. There are two ways we can give the `Charge()` method the arguments it needs:

1. Exact arguments, this is when we give it some concrete values like so:

```
var card = new Card("owner", "number", "CVV number");

paymentServiceMock.Setup(p => p.Charge(114,card)).Returns(true)
```

2. General arguments, here we can use the helper `It` , which will allow us to instruct the method `Charge()` that *any* values of a certain data type can be passed through:

Accessing our implementation

We will need to pass an implementation of our Mock when we call the actual production code. So how do we do that? There's an `Object` property on the Mock that represents the concrete implementation. Below we are using just that. We first construct `cardMock` and then we pass `cardMock.Object` to the `Charge()` method.

```
var cardMock = new Mock<ICard>();

paymentServiceMock.Setup(p => p.Charge(It.IsAny<double>(),cardMoc
```

Add unit tests

Let's rename the default test file we got to `CartControllerTest.cs` . Next, let's discuss our approach. We want to:

- **Test all the execution paths**, there are currently two different paths through our CartController depending on whether `_paymentService.Charge()` answers with `true` or `false`
- **Write two tests**, we need at least two different tests, one for each execution path
- **Assert**, we need to ensure that the *correct* thing happens. In our case, that means if we successfully get paid then we should ship, so that

Let's write our first test:

```
// CartControllerTest.cs

[Test]
public void ShouldReturnCharged()
{
  // arrange
  paymentServiceMock.Setup(p => p.Charge(It.IsAny<double>(), carc

  // act
  var result = controller.CheckOut(cardMock.Object, addressInfoMc

  // assert
  shipmentServiceMock.Verify(s => s.Ship(addressInfoMock.Object,

  Assert.AreEqual("charged", result);
}
```

We have three phases above.

Arrange

Let's have a look at the code:

```
paymentServiceMock.Setup(p => p.Charge(It.IsAny<double>(), cardMc
```

`It.IsAny<double>()` and with a card object `cardMock.Object` then we should return `true` , aka `.Returns(true)` . This means we have set up a happy path and are ready to go to the next phase *Act*.

Act

Here we call the actual code:

```
var result = controller.CheckOut(cardMock.Object, addressInfoMock
```

As we can see above we get the answer assigned to the variable `result` . This takes us to our next phase, *Assert*.

Assert

Let's have a look at the code:

```
shipmentServiceMock.Verify(s => s.Ship(addressInfoMock.Object, it

Assert.AreEqual("charged", result);
```

Now, there are two pieces of assertions that take place here. First, we have a Mock assertion. We see that as we are calling the method `Verify()` that essentially says: I expect the `Ship()` method to have been called with an `addressInfo` object and a `cartItem` list and that it was called only once.

Next, we have a more normal-looking assertion namely this code:

```
Assert.AreEqual("charged", result);
```

It says our `result` variable should contain the value `charged` .

A second test

So far we tested the happy path. As we stated earlier, there are two paths through this code. The `paymentService` could decline our payment and then we shouldn't ship any cart content. Let's see what the code looks like for that:

```

[Test]
public void ShouldReturnNotCharged()
{
    // arrange
    paymentServiceMock.Setup(p => p.Charge(It.IsAny<double>(), ca

    // act
    var result = controller.CheckOut(cardMock.Object, addressInfo

    // assert
    shipmentServiceMock.Verify(s => s.Ship(addressInfoMock.Object


```

Above we see that we have again the three phases *Arrange*, *Act* and *Assert*.

Arrange

This time around we are ensuring that our `paymentService` mock is returning `false` , aka payment bounced.

```
paymentServiceMock.Setup(p => p.Charge(It.IsAny<double>(), cardMc
```

Act

This part looks exactly the same:

```
var result = controller.CheckOut(cardMock.Object, addressInfoMock
```

Assert

We are still testing two pieces of assertions - behavior and value assertion:

```
shipmentServiceMock.Verify(s => s.Ship(addressInfoMock.Object, it

```

Looking at the code above we, however, are asserting that `shipmentService` is not called `Times.Never()` . That's important to verify as

The second assertion just tests that the `result` variable now says `not charged` .

Full code

Let's have a look at the full code so you are able to test this out for yourself:

```
// CartControllerTest.cs

using System;
using Services;
using Moq;
using NUnit.Framework;
using api.Controllers;
using System.Linq;
using System.Collections.Generic;

namespace test
{
  public class Tests
  {
      private CartController controller;
      private Mock<IPaymentService> paymentServiceMock;
      private Mock<ICartService> cartServiceMock;

```

☰

```
        private Mock<IAddressInfo> addressInfoMock;
        private List<CartItem> items;

        [SetUp]
        public void Setup()
        {

            cartServiceMock = new Mock<ICartService>();
            paymentServiceMock = new Mock<IPaymentService>();
            shipmentServiceMock = new Mock<IShipmentService>();

            // arrange
            cardMock = new Mock<ICard>();
            addressInfoMock = new Mock<IAddressInfo>();

            //
            var cartItemMock = new Mock<CartItem>();
            cartItemMock.Setup(item => item.Price).Returns(10);

            items = new List<CartItem>()
            {
                cartItemMock.Object
            };

            cartServiceMock.Setup(c => c.Items()).Returns(items.AsE

            controller = new CartController(cartServiceMock.Object,
        }

        [Test]
        public void ShouldReturnCharged()
        {
            paymentServiceMock.Setup(p => p.Charge(It.IsAny<double>
            var result = controller.CheckOut(cardMock.Object, addre

            // assert
            // myInterfaceMock.Verify((m => m.DoesSomething()), Tim
            shipmentServiceMock.Verify(s => s.Ship(addressInfoMock.

            Assert.AreEqual("charged", result);
        }

        [Test]
        public void ShouldReturnNotCharged()
        {
            paymentServiceMock.Setup(p => p.Charge(It.IsAny<double>

            // act
            var result = controller.CheckOut(cardMock.Object, addre

            // assert
            shipmentServiceMock.Verify(s => s.Ship(addressInfoMock.
            Assert.AreEqual("not charged", result);
        }
    }
}
```

Final thoughts

So we have managed to test out the two major paths through our code but there are more tests, more assertions we could be doing. For example, we could ensure that the value of the Cart corresponds to what the customer is actually being charged. As well all know in the real world things are more service.

Summary

I've hopefully been able to convey some good reasons for why you should test your code. Additionally, I hope you think the library `moq` looks like a good candidate to help you with the more behavioral aspects of your code.