Unit Testing with C#, including MVC



<https://www.codeproject.com/Articles/5275529/Unit-Testing-with-Csharp-including-MVC?msg=5753897#xx5753897xx>

20 Sep 2020[CPOL](http://www.codeproject.com/info/cpol10.aspx)

A discussion of what to unit test in your C# code and how to test it, including a section covering MVC

This article will tackle what code you should unit test and how you should test it, and also what you shouldn't be testing. As well as general C# code, there is also a section on MVC which includes explanations on how you can test code that relies on Session, cookies, etc.

* [**Download source - 800.4 KB**](https://www.codeproject.com/KB/tips/5275529/SourceCode.zip)

Introduction

This article is an introduction to unit testing and will seek to cover the paradigm of unit testing just as much as the technical nuts and bolts. So as well as covering how to unit test, this article will cover what to unit test and why, and more importantly what **not** to unit test and why. Towards the end of the article, I will discuss slightly more advanced web-related testing such as how you can test an MVC project, including code that relies on session state, cookies, email, etc.

The basis of the project we will be unit testing is a simple banking application.

Table of Contents

1. [The Basics of Unit Testing](https://www.codeproject.com/Articles/5275529/Unit-Testing-with-Csharp-including-MVC?msg=5753897#1)
   1. [What Can't be Unit Tested](https://www.codeproject.com/Articles/5275529/Unit-Testing-with-Csharp-including-MVC?msg=5753897#1.1)
2. [Bank System - First Attempt](https://www.codeproject.com/Articles/5275529/Unit-Testing-with-Csharp-including-MVC?msg=5753897#2)
3. [Bank System - Updated](https://www.codeproject.com/Articles/5275529/Unit-Testing-with-Csharp-including-MVC?msg=5753897#3)
4. [Unit Tests](https://www.codeproject.com/Articles/5275529/Unit-Testing-with-Csharp-including-MVC?msg=5753897#4)
   1. [Introduction](https://www.codeproject.com/Articles/5275529/Unit-Testing-with-Csharp-including-MVC?msg=5753897#4.1)
   2. [Unit Testing the Bank Service](https://www.codeproject.com/Articles/5275529/Unit-Testing-with-Csharp-including-MVC?msg=5753897#4.2)
   3. [Self-Shunting](https://www.codeproject.com/Articles/5275529/Unit-Testing-with-Csharp-including-MVC?msg=5753897#4.3)
   4. [Mocking Framework](https://www.codeproject.com/Articles/5275529/Unit-Testing-with-Csharp-including-MVC?msg=5753897#4.4)
   5. [Bank Service Unit Tests - Final Version](https://www.codeproject.com/Articles/5275529/Unit-Testing-with-Csharp-including-MVC?msg=5753897#4.5)
   6. [Testing Private Methods](https://www.codeproject.com/Articles/5275529/Unit-Testing-with-Csharp-including-MVC?msg=5753897#4.6)
5. [Unit Testing MVC Controllers](https://www.codeproject.com/Articles/5275529/Unit-Testing-with-Csharp-including-MVC?msg=5753897#5)
   1. [Unit Testing Model Validation](https://www.codeproject.com/Articles/5275529/Unit-Testing-with-Csharp-including-MVC?msg=5753897#5.1)
   2. [Unit Testing with Web Context Objects](https://www.codeproject.com/Articles/5275529/Unit-Testing-with-Csharp-including-MVC?msg=5753897#5.2)
   3. [Dependency Injection and IoC](https://www.codeproject.com/Articles/5275529/Unit-Testing-with-Csharp-including-MVC?msg=5753897#5.3)
6. [Code Coverage](https://www.codeproject.com/Articles/5275529/Unit-Testing-with-Csharp-including-MVC?msg=5753897#6)

The Basics of Unit Testing

Unit testing is the testing of the logic in a given method. When we write unit tests for a method, we want to ensure that all the possible scenarios the method is supposed to handle do what we expect them to do. These tests then become a kind of living specification document where if the working of a method is altered to satisfy a new requirement and it no longer satisfies the old requirements, the relevant unit test should fail, flagging up the issue for discussion. It could be that the method does need to satisfy existing conditions so should be refactored such that it does the new feature as well as the old ones, or it could be that the requirements of the function have changed such that the failing unit test covers a scenario no longer required so that unit test can be deleted. Regardless of the outcome of failing unit tests after new work, the unit tests are there to flag that there *is*an issue that needs to be looked into. And, of course, when all the tests pass, we gain confidence that the methods are all behaving just as we want them to. This confidence is especially important when working on large systems that have been developed by multiple people and you're unsure if changes you are making might be breaking things elsewhere.

Methods we want to test may have more than one path through them depending on different parameters, or have multiple outcomes we want to verify, and while it is tempting to cover all possible scenarios in a single test, it is often better to have a unit test for each thing you're trying to verify. For example, in our banking system when you transfer money between accounts, not only is the balance of the accounts in question updated, but a transaction is created for each account update too, and we could create a single test that asserts the money was transferred and that each account had a transaction record created, however it is better practice to split this out into three distinct tests so one will test that money was transferred, one will test that the source account's transaction was created, and the final one will test that the destination account's transaction was created. While this results in more tests overall, it also results in more granular tests that focus on very specific pieces of functionality and that is often preferred as if a test fails you know specifically what piece of functionality has failed.

What Can't be Unit Tested

There are two requirements regarding unit testing that control much about what can and can't be unit tested

* Unit tests and the code they run need to be completely **self-contained** and **not access any external resources**.
* Unit tests should run as **fast** as possible.

Let's explore in a bit more detail what these requirements mean.

By self-contained, I mean the code can't access any higher-level contexts such as a web context which supplies things like a Request, Response, Cookies, Session, etc. This is because our unit tests are running inside the context of a test runner (be that Microsoft's test runner or a third party test runner like NUnit). If you are unit testing a method in an MVC controller that uses the Session state, that code will only work when being executed by IIS as it is IIS that is providing you with a session. When you run that code as a unit test, the test runner is not giving you a web context to run inside so any reference to Session is going to be null. Likewise any access of Request, Response, etc. is going to fail.

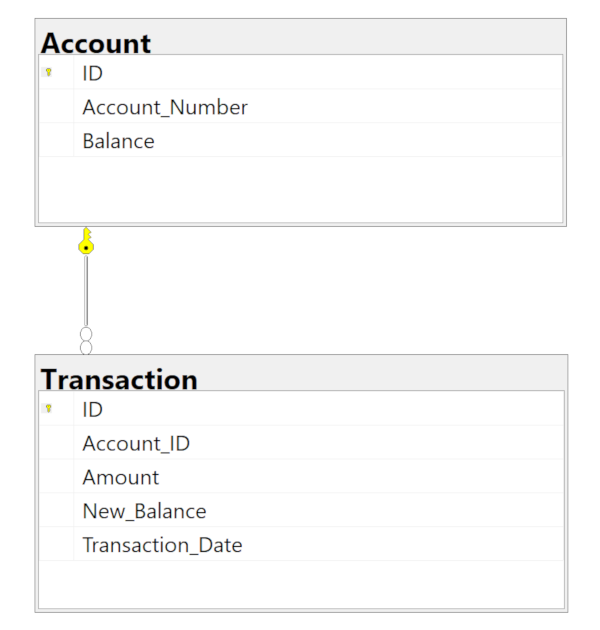
By external resources, I mean pretty much anything...files, databases, the network, SMTP servers, web APIs, Active Directory, literally anything but basic code. Most developers see unit tests as something they run on their local machine, but if you work for a company that has an automated delivery pipeline, those unit tests are going to be run by some server somewhere as a part of that delivery process. So your local machine might have access to a certain database or API, but does that server? How do you configure it such that the server running your tests uses a suitable test database and isn't running test code on the production database? That's the main reason you can't access external resources, but another is that it ties into the requirement that your unit tests should run quickly. Again, some companies will insist you use a localised deployment pipeline that will run your tests after each local deployment, and do you really want to wait while your unit tests access databases and send emails every time you deploy locally?

Not requiring external resources also helps with repeatability. Let's say you need certain records in a database to exist for your tests to run, or your tests create rows that you then need to make sure are deleted, the management of these things can quickly get out of hand, but if your tests don't require any database state management, then it's not a problem, you can re-run the same test over and over.

At this point, you might be thinking that unit testing is worthless as all of your code accesses external resources, and this is where the paradigm of unit testing kicks in. One of the most common questions regarding unit testing is "Here is my code, how do I unit test it?" and the answer is almost always "You can't". Unit testing isn't something you can retro-fit into your code, if you want to unit test, you have to write your code from the start with unit testing in mind. The remainder of this article will focus on how this is achieved.

Bank System - First Attempt

I'm going to start off with a basic repository to manage tables in a banking-related database and also write a banking service that uses this repository. During this article when I say *service* I'm not referring to a Windows Service but simply a class that offers a related set of functions. I'll first do it in a way that many probably would when not thinking about unit testing. Our system is very simple with only two tables; one that holds account information and one that keeps a history of account transactions.



**Repository Code**

This is a class that uses Entity Framework to interface with the tables in our Bank database.

**Note**: This article isn't a tutorial on how to write repositories or systems in general. In the real world, you would probably have repositories for each table, maybe using a generic pattern and maybe also a unit of work pattern. For simplicity, this article is bundling the limited number of required methods into a single class.

| **Function** | **Description** |
| --- | --- |
| GetAccount | Returns an Account object that represents the account. There are two overloads, one retrieves the account by ID and one by account number. |
| SetBalance | Updates the Account table to set the Balance field to the given value. |
| AddTransaction | Adds a row to the Transaction table indicating what account was amended, the amount the balance was amended by and what the new balance is. |

Hide   Shrink    Copy Code

using System;

using System.Linq;

using UnitTestArticle.Services.Exceptions;

namespace UnitTestArticle.Repositories

{

public class BankRepositoryBad : IDisposable

{

private BankContext context;

private bool disposed = false;

public BankRepositoryBad(BankContext context)

{

this.context = context;

}

public BankRepositoryBad()

: this (new BankContext())

{

}

public Account GetAccount(string accountNumber)

{

return this.context

.Accounts

.FirstOrDefault(a => a.Account\_Number == accountNumber.Trim());

}

public Account GetAccount(int id)

{

return this.context

.Accounts

.FirstOrDefault(a => a.ID == id);

}

public void SetBalance(int id, decimal balance)

{

Account account = GetAccount(id);

if (account == null)

{

throw new AccountNotFoundException(id);

}

account.Balance = balance;

context.SaveChanges();

}

public void AddTransaction(int id, decimal amount, decimal newBalance)

{

context.Transactions.Add(new Transaction

{

Account\_ID = id,

Amount = amount,

New\_Balance = newBalance,

Transaction\_Date = DateTime.Now

});

context.SaveChanges();

}

protected virtual void Dispose(bool disposing)

{

if (!this.disposed)

{

if (disposing)

{

this.context.Dispose();

}

}

this.disposed = true;

}

public void Dispose()

{

Dispose(true);

GC.SuppressFinalize(this);

}

}

}

**Service Code**

This class has functions that carry out tasks related to managing the bank system such as transferring money between accounts, amending balances, etc. It contains the logic related to these tasks and it uses the bank repository to update the actual bank database.

| **Function** | **Description** |
| --- | --- |
| GetAccount | Returns an Account object for the account with the supplied account number. |
| UpdateAccountBalance | Updates the balance of an Account by the given amount. If the amount is positive money is added, if it is negative money is withdrawn. |
| TransferMoney | Transfers money between two accounts, ensuring the funds are available. |

Hide   Shrink    Copy Code

using System;

using UnitTestArticle.Repositories;

using UnitTestArticle.Services.Exceptions;

namespace UnitTestArticle.Services

{

public class BankServiceBad : IDisposable

{

private BankRepositoryBad repo;

private bool disposed = false;

public BankServiceBad()

{

repo = new BankRepositoryBad();

}

public Account GetAccount(string accountNumber)

{

return repo.GetAccount(accountNumber);

}

public void UpdateAccountBalance(Account account, decimal amount)

{

if (account == null)

{

throw new ArgumentNullException(nameof(account));

}

repo.SetBalance(account.ID, account.Balance += amount);

repo.AddTransaction(account.ID, amount, account.Balance);

if (account.Balance < 0)

{

var reportingService = new ReportingService();

reportingService.AccountIsOverdrawn(account.ID);

}

}

public void TransferMoney(string sourceAccountNumber,

string destinationAccountNumber, decimal transferAmount)

{

if (transferAmount <= 0)

{

throw new InvalidAmountException();

}

Account sourceAccount = repo.GetAccount(sourceAccountNumber);

if (sourceAccount == null)

{

throw new AccountNotFoundException(sourceAccountNumber);

}

Account destinationAccount = repo.GetAccount(destinationAccountNumber);

if (destinationAccount == null)

{

throw new AccountNotFoundException(destinationAccountNumber);

}

if (sourceAccount.Balance < transferAmount)

{

throw new InsufficientFundsException();

}

*// remove transferAmount from destination account*

repo.SetBalance(sourceAccount.ID, sourceAccount.Balance -= transferAmount);

*// record the transaction*

repo.AddTransaction(sourceAccount.ID, -transferAmount, sourceAccount.Balance);

*// add transferAmount to source account*

repo.SetBalance

(destinationAccount.ID, destinationAccount.Balance += transferAmount);

*// record the transaction*

repo.AddTransaction

(destinationAccount.ID, transferAmount, destinationAccount.Balance);

}

protected virtual void Dispose(bool disposing)

{

if (!this.disposed)

{

if (disposing)

{

this.repo.Dispose();

}

}

this.disposed = true;

}

public void Dispose()

{

Dispose(true);

GC.SuppressFinalize(this);

}

}

}

**Calling Code**

Hide   Copy Code

using (var bankService = new BankServiceBad())

{

string accountNumber = "1111111111";

var account = bankService.GetAccount(accountNumber);

if (account == null)

{

throw new AccountNotFoundException(accountNumber);

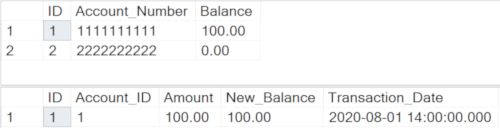
}

*// Add 100 to account 1111111111*

bankService.UpdateAccountBalance(account, 100);

}

After the code above runs, we will have 100 added to the balance of account 1111111111 and an entry in the Transaction table to show 100 was added.



We have two main classes here, BankRepositoryBad and BankServiceBad, with the service being called by client code and the service then calling the repository. Some methods on the service are basic "pass through" methods (like GetAccount), they just return the results of a call to the repository, however methods like TransferMoney actually have some basic logic like validation, working out how the account balances change, creating transactions, etc., and UpdateAccountBalance has an optional call to another service if the account becomes overdrawn.

Let's focus on the possibility of unit testing the repository first. Is there any actual business logic in these methods? Not really, they're just wrappers around calls to Entity Framework; almost all of the code in them is Entity Framework code, they just read, update or create rows. **Can** we unit test this class? No, it relies on external resources like a database. **Should** we unit test this class? No, there is no real logic in the class and the code is mainly code that runs third-party code, Entity Framework in this case. If we were to unit test this code, we'd really just be testing Microsoft's code, not our own, and we should assume that third-party code works, our unit tests should focus solely on our own code.

Now let's look at the service class. **Should** we unit test this class? Yes, it implements business rules such as validation, what happens when you transfer money, what happens when you amend an account and so on. **Can** we unit test this class? No, because it relies on the repository which in turn relies on a database which is an external resource. There is no way to unit test these methods without them updating a database of sorts. The trick to unit testing our service class is to eliminate the dependency on the repository in a way that lets us still test our business logic, and we achieve that by using interfaces to abstract away the repository.

Bank System - Updated

The updated bank system is going to include an interface (IBankRepository) that the repository is going to implement. The methods and what they do will remain the same as before, the only difference is the addition of the interface.

Hide   Copy Code

using System;

namespace UnitTestArticle.Interfaces

{

public interface IBankRepository : IDisposable

{

Account GetAccount(string accountNumber);

Account GetAccount(int id);

void SetBalance(int id, decimal balance);

void AddTransaction(int id, decimal amount, decimal newBalance);

}

}

Hide   Shrink    Copy Code

using System;

using System.Linq;

using UnitTestArticle.Interfaces;

namespace UnitTestArticle.Repositories

{

public class BankRepository : IBankRepository

{

private BankContext context;

private bool disposed = false;

public BankRepository(BankContext context)

{

this.context = context;

}

public BankRepository()

: this (new BankContext())

{

}

public Account GetAccount(string accountNumber)

{

return this.context

.Accounts

.FirstOrDefault(a => a.Account\_Number == accountNumber.Trim());

}

public Account GetAccount(int id)

{

return this.context

.Accounts

.FirstOrDefault(a => a.ID == id);

}

public void SetBalance(int id, decimal balance)

{

Account account = GetAccount(id);

if (account == null)

{

throw new ApplicationException("Account not found");

}

account.Balance = balance;

context.SaveChanges();

}

public void AddTransaction(int id, decimal amount, decimal newBalance)

{

context.Transactions.Add(new Transaction

{

Account\_ID = id,

Amount = amount,

New\_Balance = newBalance,

Transaction\_Date = DateTime.Now

});

context.SaveChanges();

}

protected virtual void Dispose(bool disposing)

{

if (!this.disposed)

{

if (disposing)

{

this.context.Dispose();

}

}

this.disposed = true;

}

public void Dispose()

{

Dispose(true);

GC.SuppressFinalize(this);

}

}

}

Next, we'll amend the BankService to reference the IBankRepository interface instead of the concrete BankRepository class, and we'll create an IBankService interface for the service as well.

Hide   Copy Code

using System;

namespace UnitTestArticle.Interfaces

{

public interface IBankService : IDisposable

{

Account GetAccount(string accountNumber);

void TransferMoney(string sourceAccountNumber,

string destinationAccountNumber, decimal transferAmount);

void UpdateAccountBalance(Account account, decimal amount);

}

}

Hide   Shrink    Copy Code

using System;

using UnitTestArticle.Interfaces;

using UnitTestArticle.Repositories;

using UnitTestArticle.Services.Exceptions;

namespace UnitTestArticle.Services

{

public class BankService : IBankService, IDisposable

{

private IBankRepository repo;

private bool disposed = false;

public BankService()

: this (new BankRepository())

{

}

public BankService(IBankRepository repo)

{

this.repo = repo;

}

public Account GetAccount(string accountNumber)

{

return repo.GetAccount(accountNumber);

}

public void UpdateAccountBalance(Account account, decimal amount)

{

if (account == null)

{

throw new ArgumentNullException(nameof(account));

}

repo.SetBalance(account.ID, account.Balance += amount);

repo.AddTransaction(account.ID, amount, account.Balance);

if (account.Balance < 0)

{

var reportingService = new ReportingService();

reportingService.AccountIsOverdrawn(account.ID);

}

}

public void TransferMoney(string sourceAccountNumber,

string destinationAccountNumber, decimal transferAmount)

{

if (transferAmount <= 0)

{

throw new InvalidAmountException();

}

Account sourceAccount = repo.GetAccount(sourceAccountNumber);

if (sourceAccount == null)

{

throw new AccountNotFoundException(sourceAccountNumber);

}

Account destinationAccount = repo.GetAccount(destinationAccountNumber);

if (destinationAccount == null)

{

throw new AccountNotFoundException(destinationAccountNumber);

}

if (sourceAccount.Balance < transferAmount)

{

throw new InsufficientFundsException();

}

*// remove transferAmount from destination account*

repo.SetBalance(sourceAccount.ID, sourceAccount.Balance - transferAmount);

*// record the transaction*

repo.AddTransaction(sourceAccount.ID, -transferAmount, sourceAccount.Balance);

*// add transferAmount to source account*

repo.SetBalance

(destinationAccount.ID, destinationAccount.Balance + transferAmount);

*// record the transaction*

repo.AddTransaction

(destinationAccount.ID, transferAmount, destinationAccount.Balance);

}

protected virtual void Dispose(bool disposing)

{

if (!this.disposed)

{

if (disposing)

{

this.repo.Dispose();

}

}

this.disposed = true;

}

public void Dispose()

{

Dispose(true);

GC.SuppressFinalize(this);

}

}

}

If you look at the code in our methods, they all reference "repo" which is of type IBankRepository, so **we have broken the direct dependence between the code in these methods and the concrete repository class**.

**Note also the constructors** on the BankService class; one accepts a specific instance of something that implements IBankRepository, and the default one creates an instance of the concrete BankRepository class. This means we have two ways of creating the class; we can create the class while supplying our own implementation of IBankRepository.

Hide   Copy Code

var service = new BankService(somethingThatImplementsIBankRepository);

or we can create the class using the default empty constructor:

Hide   Copy Code

var service = new BankService();

and that constructor creates a concrete BankService internally.

The ability to supply the class with the objects it is dependent on is called *inversion of control* as we are providing the class its dependants rather than the traditional method of the service itself deciding what it depends on. The importance of this will become clear when we get to the unit tests, which we'll look at now.

Unit Tests

Introduction

For this article, I am using the test runner that comes with Visual Studio, if you use a third-party framework like NUnit, then the basics are largely the same.

Related tests are grouped together in a test class maintaining a one-to-one relationship where each relevant class in your project has an equivalent test class. If I had a Maths class, then the test class would probably look something like this:

Hide   Shrink    Copy Code

using Microsoft.VisualStudio.TestTools.UnitTesting;

namespace UnitTestArticle.Tests

{

[TestClass]

public class MathsTests

{

[TestInitialize]

public void Init()

{

*// put code in here that you want to have run before each test*

}

[TestMethod]

public void AddTest()

{

*// Arrange*

int number1 = 4;

int number2 = 6;

Maths m = new Maths();

*// Act*

int result = m.Add(number1, number2);

*// Assert*

Assert.AreEqual(10, result);

}

[TestMethod]

public void SubtractTest()

{

*// Arrange*

int number1 = 4;

int number2 = 6;

Maths m = new Maths();

*// Act*

int result = m.Subtract(number1, number2);

*// Assert*

Assert.AreEqual(-2, result);

}

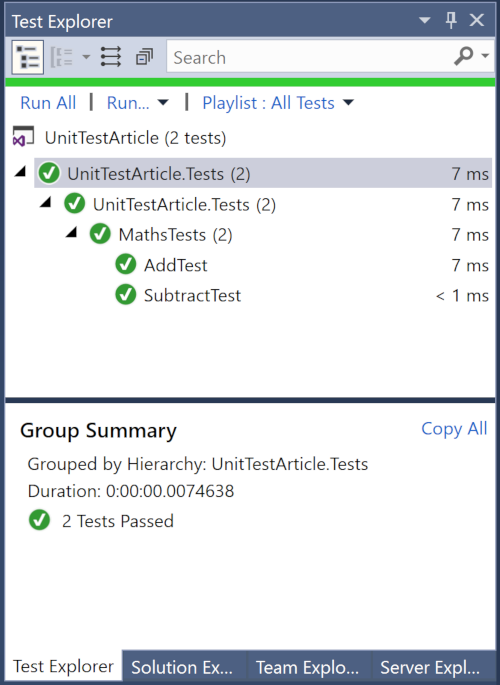
}

}

The test class itself is marked with the [TestClass] attribute and each test is marked [TestMethod]. These attributes tell the test runner where the tests are and what to run as tests. Many frameworks also allow you to specify methods that run before tests, after tests and so on; in the example above, our Init method is marked [TestInitialize] which means it will be called before each test method. These methods are typically used for setting up any data, etc. you want to use in your tests.

The test methods themselves are structured in an "Arrange, Act, Assert" format. Typically, we initially do all of the things we need to do to set up the data for the test in the "Arrange" section. Next we have the "Act" section where we call the method we are testing, and finally the "Assert" section where we verify the results. The Assert class provides a range of methods that let us test things of interest, such as if things are null or not, if values match, if references match and so on.

We run our tests by selecting the **Test**menu in Visual Studio, then the **Run**submenu and then **All Tests**. Alternatively, we can run tests via the "**Test Explorer**" window.



Green ticks mean the tests have passed, any tests that have failed will have red crosses. There is a "**Run All**" link in the test explorer as well, and if you want to debug your unit test, you can use the **Debug** menu in the **Test** menu. You can also right click the body of a test method in the code editor and select to either run or debug it from the context menu.

Unit Testing the Bank Service

**The way we are going to unit test our** bank service is to use inversion of control to supply the service with a repository that doesn't rely on external resources. **What allows us to do this is the fact that our service relies on an interface rather than a concrete class and inversion of control allows us to supply that concrete class ourselves, so we get to test the logic o**f our code without needing an actual database. I'm going to go through two ways of doing this. The first is called self-shunting and I'll cover the basics of it but I'm not going to go too in-depth as there are better solutions. The main advantage of self-shunting is that it doesn't rely on any third-party libraries and it gives you a good degree of flexibility.

Self-Shunting

I said above that to unit test the service code, we need to provide a repository that doesn't rely on external resources, and with self-shunting, that repository is actually going to be the test class itself.

Below, we have a test class that also implements IBankRepository. The methods are all in the "Self-shunt" region and they use List<T> structures to store data rather than a database.

Hide   Shrink    Copy Code

using Microsoft.VisualStudio.TestTools.UnitTesting;

using System;

using System.Collections.Generic;

using System.Linq;

using UnitTestArticle.Interfaces;

using UnitTestArticle.Services;

using UnitTestArticle.Services.Exceptions;

namespace UnitTestArticle.Tests.Services

{

[TestClass]

public class BankServiceTestsSelfShunt : IBankRepository *// our test class*

*// implements IBankRepository*

{

#region Self-shunt

*// Our test class is going to implement a mocked version of IBankRepository*

*// using Lists to hold the data rather than database tables. The code in this*

*// region is a List based implementation of the IBankRepository methods*

private List<Account> Accounts;

private List<Transaction> Transactions;

Account IBankRepository.GetAccount(string accountNumber)

{

return Accounts.FirstOrDefault(a => a.Account\_Number == accountNumber);

}

Account IBankRepository.GetAccount(int id)

{

return Accounts.FirstOrDefault(a => a.ID == id);

}

void IBankRepository.SetBalance(int id, decimal balance)

{

Accounts.FirstOrDefault(a => a.ID == id).Balance = balance;

}

void IBankRepository.AddTransaction(int id, decimal amount, decimal newBalance)

{

Transactions.Add(new Transaction

{ Account\_ID = id, Amount = amount, New\_Balance = newBalance });

}

void IDisposable.Dispose()

{

}

#endregion

private BankService bankService;

[TestInitialize]

public void Init()

{

*// This is called at the start of every test*

Accounts = new List<Account>();

Transactions = new List<Transaction>();

*// Add two test accounts to the Accounts collection*

Accounts.Add(new Account { ID = 1, Account\_Number = "test1", Balance = 0 });

Accounts.Add(new Account { ID = 2, Account\_Number = "test2", Balance = 0 });

bankService = new BankService(this);

}

[TestMethod]

public void GetAccount\_WithValidAccount\_ReturnsAccount()

{

*// Arrange*

*// Act*

Account account = bankService.GetAccount("test1");

*// Assert*

Assert.IsNotNull(account);

Assert.AreEqual(1, account.ID);

Assert.AreEqual("test1", account.Account\_Number);

}

[TestMethod]

public void TransferMoney\_WithInsufficientFunds\_AccountsUpdated()

{

*// Arrange*

bankService.GetAccount("test1").Balance = 50;

*// Act*

bankService.TransferMoney("test1", "test2", 10);

*// Assert*

Assert.AreEqual(40, ((IBankRepository)this).GetAccount("test1").Balance);

Assert.AreEqual(10, ((IBankRepository)this).GetAccount("test2").Balance);

}

}

}

Let's look at the TransferMoney\_WithSufficientFunds\_AccountsUpdated test method as there are a few things I want to unpack. First of all is the naming convention. Naming your test methods could easily be an article in and of itself, and the best naming convention for you is the one that works for you, but the convention I use is the method name, an underscore, the particular case we are testing, an underscore, and then the result I expect. As we are potentially going to be testing the same methods for multiple scenarios, we can't just have the test method name mimic the method they are testing, and this convention also groups the tests for the same method together in the test runner. How you name yours is up to you, I'm not saying this is how you should name them, just that this is how I name them.

Next is the "Arrange, Act, Assert" format we discussed previously. In our arrange section, we are setting the source account to have a balance of 50. Next we have the "Act" section where we call the TransferMoney method we are testing, and finally the "Assert" section where we verify the results.

Our test makes sure test1 has a balance 50, does a transfer to test2 then asserts that test1 has 40 and test2 has 10.

Let's take a deeper look at what appears to be one of the simpler tests which is GetAccount\_WithValidAccount\_ReturnsAccount.

Hide   Copy Code

[TestMethod]

public void GetAccount\_WithValidAccount\_ReturnsAccount()

{

*// Arrange*

*// Act*

Account account = bankService.GetAccount("test1");

*// Assert*

Assert.IsNotNull(account);

Assert.AreEqual(1, account.ID);

Assert.AreEqual("test1", account.Account\_Number);

}

We call GetAccount to retrieve test1s account then test we didn't get a null object back, that the account ID was 1 and that "test1" was the account number. This looks like a valid test on the face of it, but when we look at the code in the BankService method we are calling:

Hide   Copy Code

public Account GetAccount(string accountNumber)

{

return repo.GetAccount(accountNumber);

}

The only reason repo.GetAccount returned the Account object was because of the code in my self-shunted class (remember "repo" references the test class itself so repo.GetAccount is calling the GetAccount method on the test class). So this test will only work if my self-shunt code does what is expected. So are we really testing the service method? Or are we just testing the self-shut code? The answer is that we're just testing the self-shut code so this unit test actually has no value. If we want to test the service code alone, then we have to focus on what the service code alone is doing, which is calling a method on the repo.

So let's add some functionality to our self-shut code to track when methods are called. We'll do this by having a variable called GetAccountCalled that will be incremented every time GetAccount is called.

Hide   Copy Code

public class BankServiceTestsSelfShunt : IBankRepository *// our test class implements*

*// IBankRepository*

{

#region Self-shunt

*// Our test class is going to implement a mocked version of IBankRepository*

*// using Lists to hold the data rather than database tables. The code in this*

*// region is a List based implementation of the IBankRepository methods*

private List<Account> Accounts;

private List<Transaction> Transactions;

*// GetAccountCalled will store how many times that method has been called*

private int GetAccountCalled;

Account IBankRepository.GetAccount(string accountNumber)

{

*// increase the call count for this method*

GetAccountCalled++;

return Accounts.FirstOrDefault(a => a.Account\_Number == accountNumber);

}

*// ... rest of class as before*

Now let's tackle the unit testing of GetAccount from another direction:

Hide   Copy Code

[TestMethod]

public void GetAccount\_CallsRepo()

{

*// Arrange*

*// Act*

Account account = bankService.GetAccount("test1");

*// Assert*

Assert.IsTrue(GetAccountCalled > 0);

}

This might seem less intuitive but it is actually a far more valid test as the only thing we want to ensure GetAccount does is to pass the account number to the repository to get the results. We're not testing the repository here so we don't care what the repository does, we only want to ensure it is called, so that's the only thing we should test for. This is another example of focussing your unit tests on what the code you're testing should actually do and how you have to adapt your thinking when creating unit tests.

I'm going to abandon the self-shunt method at this point, I only brought it up to introduce you to the idea and to help address some issues we have when testing against mocked code. Below, we'll start to use a mocking framework to do this work for us which is a far better solution, and the one which is industry standard.

Mocking Framework

Mocking frameworks let us create objects that mimic or simulate the classes we are abstracting, but the behaviour of the methods on the mocked object are driven programmatically by the code in our tests. For this article, I am using [Moq](https://github.com/Moq/moq4/wiki/Quickstart" \t "_blank) which you can install as a [NuGet package](https://www.nuget.org/packages/Moq/). There are other mocking frameworks out there and they all largely do the same thing just with different syntax so feel free to use whichever framework you are comfortable with.

Let's start to write some tests using Moq:

Hide   Shrink    Copy Code

using Microsoft.VisualStudio.TestTools.UnitTesting;

using Moq;

using UnitTestArticle.Interfaces;

using UnitTestArticle.Services;

namespace UnitTestArticle.Tests.Services

{

[TestClass]

public class BankServiceTests

{

private BankService bankService;

private Mock<IBankRepository> repoMock;

private Account test1Account;

private Account test2Account;

[TestInitialize]

public void Init()

{

*// This is called at the start of every test*

*// Create two test accounts*

test1Account = new Account { ID = 1, Account\_Number = "test1", Balance = 0 };

test2Account = new Account { ID = 2, Account\_Number = "test2", Balance = 0 };

*// Mock the IBankRepository*

repoMock = new Mock<IBankRepository>();

*// Ensure GetAccount returns the relevant Account object*

repoMock.Setup(m => m.GetAccount("test1")).Returns(test1Account);

repoMock.Setup(m => m.GetAccount("test2")).Returns(test2Account);

bankService = new BankService(repoMock.Object);

}

[TestMethod]

public void TransferMoney\_WithSufficientFunds\_SourceAccountUpdated()

{

*// Arrange*

test1Account.Balance = 50;

*// Act*

bankService.TransferMoney("test1", "test2", 10);

*// Assert*

repoMock.Verify(m => m.SetBalance(1, 40), Times.Once);

}

}

}

If you look at the Init function, it creates two test account objects, then creates a mocked version of IBankRepository. The way a mocking framework works is that you tell it what happens when client code calls certain functions with certain arguments. In our Init method, we have:

Hide   Copy Code

repoMock.Setup(m => m.GetAccount("test1")).Returns(test1Account);

repoMock.Setup(m => m.GetAccount("test2")).Returns(test2Account);

We are telling the mocked IBankRepository object that whenever GetAccount is called with a parameter of "test1" to return our test1Account object, and for "test2" return test2Account. This allows us to completely abstract away our concrete repository class and dictate on a test-by-test basis how the mocked repo behaves when called.

The next thing the Init function does is to set the bankService variable to be an instance of our BankService class with the mocked repository passed in as a parameter.

Hide   Copy Code

bankService = new BankService(repoMock.Object);

As our BankService uses whatever is passed in as a repository, this lets us dictate how the code inside that service behaves by manipulating the results of calls to the mocked repository. So if we want the bank service to receive an account with a balance of 100 when GetAccount is called, we can do that. If we want the bank service to receive an account with a balance of -100 to test how it handles overdrawn accounts, we can do that too, and all without the need for a database, purely from the power of our mocking framework.

Looking at the TransferMoney\_WithSufficientFunds\_SourceAccountUpdated test method, we know the Init function is called before it runs so test1Account and test2Account have been initialised and the mocked repo has been told to return the appropriate Account when GetAccount is called. This method is testing a transfer so we set the balance of test1Account to 50, call TransferMoney to transfer 10 between test1 and test2 and we then assert that the service told the repo to SetBalance on test1 to 40 (40 being the starting balance of 50 minus the 10 we are transferring).

That is not all we want the TransferMoney method to do, we also want it to update the destination balance and also create a transaction for each account update, however to maintain a nice granular set of tests, we will write individual tests to cover each of those individual requirements.

When we used the self-shunting method of unit testing, we tested that GetAccount called the equivalent method on the repo at least once and we can easily replicate that kind of functionality using Moq.

Hide   Copy Code

[TestMethod]

public void GetAccount\_CallsRepo()

{

*// Arrange*

*// Act*

Account account = bankService.GetAccount("test1");

*// Assert*

repoMock.Verify(m => m.GetAccount("test1"), Times.AtLeastOnce);

}

We could also test GetAccount by testing that the account returned is the same one we told the mocked repo to return.

Hide   Copy Code

[TestMethod]

public void GetAccount\_WithValidAccount\_ReturnsAccount()

{

*// Arrange*

*// Act*

Account account = bankService.GetAccount("test1");

*// Assert*

Assert.IsNotNull(account);

Assert.AreSame(test1Account, account);

}

Let's flesh out our bank service tests a little more to cover the other TransferMoney scenarios and also add tests for the other methods. Note that the BankService class also uses a ReportingService class so we've made some amendments to abstract that class also.

Hide   Shrink    Copy Code

public class BankService : IBankService, IDisposable

{

private IBankRepository repo;

private IReportingService reportingService;

private bool disposed = false;

public BankService()

: this (new BankRepository(), new ReportingService())

{

}

public BankService(IBankRepository repo, IReportingService reportingService)

{

this.repo = repo;

this.reportingService = reportingService;

}

public void UpdateAccountBalance(Account account, decimal amount)

{

if (account == null)

{

throw new ArgumentNullException(nameof(account));

}

repo.SetBalance(account.ID, account.Balance += amount);

repo.AddTransaction(account.ID, amount, account.Balance);

if (account.Balance < 0)

{

reportingService.AccountIsOverdrawn(account.ID);

}

}

*// rest of code*

Bank Service Unit Tests - Final Version

Hide   Shrink    Copy Code

using Microsoft.VisualStudio.TestTools.UnitTesting;

using Moq;

using UnitTestArticle.Interfaces;

using UnitTestArticle.Services;

using UnitTestArticle.Services.Exceptions;

namespace UnitTestArticle.Tests.Services

{

[TestClass]

public class BankServiceTests

{

private BankService bankService;

private Mock<IBankRepository> repoMock;

private Mock<IReportingService> reportingMock;

private Account test1Account;

private Account test2Account;

[TestInitialize]

public void Init()

{

*// This is called at the start of every test*

*// Create two test accounts*

test1Account = new Account { ID = 1, Account\_Number = "test1", Balance = 0 };

test2Account = new Account { ID = 2, Account\_Number = "test2", Balance = 0 };

*// Mock the classes we are abstracting*

repoMock = new Mock<IBankRepository>();

reportingMock = new Mock<IReportingService>();

*// Ensure GetAccount returns the relevant Account object*

repoMock.Setup(m => m.GetAccount("test1")).Returns(test1Account);

repoMock.Setup(m => m.GetAccount("test2")).Returns(test2Account);

bankService = new BankService(repoMock.Object, reportingMock.Object);

}

[TestMethod]

public void GetAccount\_WithValidAccount\_ReturnsAccount()

{

*// Arrange*

*// Act*

Account account = bankService.GetAccount("test1");

*// Assert*

Assert.IsNotNull(account);

Assert.AreSame(test1Account, account);

}

[TestMethod]

[ExpectedException(typeof(InsufficientFundsException))]

public void TransferMoney\_WithInsufficientFunds\_RaisesException()

{

*// Arrange*

test1Account.Balance = 50;

*// Act*

bankService.TransferMoney("test1", "test2", 100);

*// Assert*

}

[TestMethod]

public void TransferMoney\_WithSufficientFunds\_SourceAccountUpdated()

{

*// Arrange*

test1Account.Balance = 50;

*// Act*

bankService.TransferMoney("test1", "test2", 10);

*// Assert*

repoMock.Verify(m => m.SetBalance(1, 40), Times.Once);

}

[TestMethod]

public void TransferMoney\_WithSufficientFunds\_DestinationAccountUpdated()

{

*// Arrange*

test1Account.Balance = 50;

*// Act*

bankService.TransferMoney("test1", "test2", 10);

*// Assert*

repoMock.Verify(m => m.SetBalance(2, 10), Times.Once);

}

[TestMethod]

public void TransferMoney\_WithSufficientFunds\_SourceTransactionCreated()

{

*// Arrange*

test1Account.Balance = 50;

*// Act*

bankService.TransferMoney("test1", "test2", 10);

*// Assert*

repoMock.Verify(m => m.AddTransaction(1, -10, It.IsAny<decimal>()), Times.Once);

}

[TestMethod]

public void TransferMoney\_WithSufficientFunds\_DestinationTransactionCreated()

{

*// Arrange*

test1Account.Balance = 50;

*// Act*

bankService.TransferMoney("test1", "test2", 10);

*// Assert*

repoMock.Verify(m => m.AddTransaction(2, 10, It.IsAny<decimal>()), Times.Once);

}

[TestMethod]

public void UpdateAccountBalance\_WithPositiveAmount\_IncreasesBalance()

{

*// Arrange*

test1Account.Balance = 50;

*// Act*

bankService.UpdateAccountBalance(test1Account, 10);

*// Assert*

repoMock.Verify(m => m.SetBalance(1, 60), Times.Once);

}

[TestMethod]

public void UpdateAccountBalance\_WithNegativeAmount\_DecreasesBalance()

{

*// Arrange*

test1Account.Balance = 50;

*// Act*

bankService.UpdateAccountBalance(test1Account, -10);

*// Assert*

repoMock.Verify(m => m.SetBalance(1, 40), Times.Once);

}

[TestMethod]

public void UpdateAccountBalance\_WithPositiveBalance\_DoesNotReportOverdrawn()

{

*// Arrange*

test1Account.Balance = 50;

*// Act*

bankService.UpdateAccountBalance(test1Account, 10);

*// Assert*

reportingMock.Verify(m => m.AccountIsOverdrawn(1), Times.Never);

}

[TestMethod]

public void UpdateAccountBalance\_WithZeroBalance\_DoesNotReportOverdrawn()

{

*// Arrange*

test1Account.Balance = 0;

*// Act*

bankService.UpdateAccountBalance(test1Account, 0);

*// Assert*

reportingMock.Verify(m => m.AccountIsOverdrawn(1), Times.Never);

}

[TestMethod]

public void UpdateAccountBalance\_WithNegativeBalance\_ReportsOverdrawn()

{

*// Arrange*

test1Account.Balance = 10;

*// Act*

bankService.UpdateAccountBalance(test1Account, -20);

*// Assert*

reportingMock.Verify(m => m.AccountIsOverdrawn(1), Times.Once);

}

[TestMethod]

public void UpdateAccountBalance\_TransactionRecorded()

{

*// Arrange*

test1Account.Balance = 50;

*// Act*

bankService.UpdateAccountBalance(test1Account, 10);

*// Assert*

repoMock.Verify(m => m.AddTransaction(1, 10, 60), Times.Once);

}

}

}

If you look at the TransferMoney\_WithInsufficientFunds\_RaisesException test, you will see it does something interesting. It expects an exception to be raised during the Act stage and the way we test for that is by adding the [ExpectedException] attribute to the method. If that exception is raised, then the test is considered a pass, and if it isn't, the test is considered a fail.

Something else of interest is the It.IsAny notation.

Hide   Copy Code

repoMock.Verify(m => m.AddTransaction(2, 10, It.IsAny<decimal>()), Times.Once);</decimal>

This means that the value of that parameter is irrelevant, it just has to be the right type but it can have any value. We use this notation when we don't really care what the argument is, it isn't relevant to what we are testing. If you do care about the specific values, then you can supply them (as we did with 2 and 10). We can also use this technique in our Setup commands too, so if you don't care what the exact value of a parameter is we can use IsAny.

The code below will return test1Account only if GetAccount is called with "test1" as a parameter.

Hide   Copy Code

repoMock.Setup(m => m.GetAccount("test1")).Returns(test1Account);

However, the code below will return the new Account regardless of what parameter is supplied to GetAccount:

Hide   Copy Code

repoMock.Setup(m => m.GetAccount(It.IsAny<string>())).Returns(new Account

{ ID = 123, Balance = 1000, Account\_Number = "testAccount" });

Testing Private Methods

Our code isn't using any private methods, but if you have a private method you want to test, then you can't test it directly as you can't call it directly, instead you will need to create tests that call a public parent method in such a way that all features of the private method are also tested.

Unit Testing MVC Controllers

MVC strongly lends itself to unit testing as the controllers and models are basic .NET objects that can be instantiated and called on their own without any web context, and the return types from controller actions are also basic .NET classes which allow us to interrogate the response, and that's exactly what we need for unit testing. Let's write a basic form that allows us to transfer money between two accounts.

**Model**

Hide   Copy Code

using System.ComponentModel.DataAnnotations;

namespace UnitTestArticle.Models

{

public class TransferMoneyModel

{

[Display(Name ="Source Account Number")]

[Required]

[MinLength(10), MaxLength(10)]

public string SourceAccountNumber { get; set; }

[Display(Name = "Destination Account Number")]

[Required]

[MinLength(10), MaxLength(10)]

public string DestinationAccountNumber { get; set; }

[Display(Name = "Amount")]

[Required]

[Range(0.01,1000000)]

public decimal Amount { get; set; }

}

}

**View**

Hide   Copy Code

@model TransferMoneyModel

@{

ViewBag.Title = "Transfer Money";

}

<h2>Transfer Money</h2>

@Html.ValidationSummary()

@using (Html.BeginForm())

{

<div class="form-group">

@Html.LabelFor(m => m.SourceAccountNumber)

@Html.TextBoxFor(m => m.SourceAccountNumber, new { @class = "form-control" })

</div>

<div class="form-group">

@Html.LabelFor(m => m.DestinationAccountNumber)

@Html.TextBoxFor(m => m.DestinationAccountNumber, new { @class = "form-control" })

</div>

<div class="form-group">

@Html.LabelFor(m => m.Amount)

@Html.TextBoxFor(m => m.Amount, new { @class = "form-control", type="number" })

</div>

<button type="submit" class="btn btn-primary">Transfer</button>

}

**Controller**

Hide   Shrink    Copy Code

using System.Web.Mvc;

using UnitTestArticle.Interfaces;

using UnitTestArticle.Models;

using UnitTestArticle.Services;

using UnitTestArticle.Services.Exceptions;

namespace UnitTestArticle.Controllers

{

public class AccountController : Controller

{

private IBankService bankService;

public AccountController()

: this (new BankService())

{

}

public AccountController(IBankService bankService)

{

this.bankService = bankService;

}

public ActionResult Index()

{

return View();

}

[HttpGet]

public ActionResult TransferMoney()

{

return View();

}

[HttpPost]

public ActionResult TransferMoney(TransferMoneyModel model)

{

if (model.SourceAccountNumber == model.DestinationAccountNumber)

{

ModelState.AddModelError("SameAccount",

"The source and destination accounts must be different");

}

if (!ModelState.IsValid)

{

return View(model);

}

try

{

bankService.TransferMoney(model.SourceAccountNumber,

model.DestinationAccountNumber, model.Amount);

}

catch (InsufficientFundsException)

{

ModelState.AddModelError("InsufficientFunds",

"There were insufficient funds to complete the transfer.");

}

catch (AccountNotFoundException ex)

{

ModelState.AddModelError("AccountNotFound",

$"There was a problem finding account {ex.AccountNumber}.");

}

catch

{

ModelState.AddModelError("TransferFailed",

"There was a problem with the transfer, please contact your bank.");

}

if (!ModelState.IsValid)

{

return View(model);

}

return RedirectToAction("Index");

}

protected override void Dispose(bool disposing)

{

if (disposing)

{

bankService.Dispose();

}

base.Dispose(disposing);

}

}

}

When you create a new project with unit tests, you automatically get a "*Controllers*" folder created in the test project, so create a new class called AccountControllerTests in that folder. In our class below, we have a test method that tests the "happy path" of the TransferMoney action. In unit tests, the happy path is the scenario where everything works as intended, the inputs are all valid, and nothing goes wrong. When unit testing, it is important to test as many scenarios as you can, some of those paths will be testing fail conditions but some will also be happy paths.

Hide   Shrink    Copy Code

using Microsoft.VisualStudio.TestTools.UnitTesting;

using Moq;

using System.Web.Mvc;

using UnitTestArticle.Controllers;

using UnitTestArticle.Interfaces;

using UnitTestArticle.Models;

using UnitTestArticle.Services.Exceptions;

namespace UnitTestArticle.Tests.Controllers

{

[TestClass]

public class AccountControllerTests

{

private Mock<IBankService> bankServiceMock;

[TestInitialize]

public void Init()

{

bankServiceMock = new Mock<IBankService>();

}

[TestMethod]

public void TransferMoney\_HappyPath\_TransfersMoney()

{

*// Arrange*

var model = new TransferMoneyModel

{

SourceAccountNumber = "test1",

DestinationAccountNumber = "test2",

Amount = 123

};

var controller = new AccountController(bankServiceMock.Object);

*// Act*

RedirectToRouteResult result =

controller.TransferMoney(model) as RedirectToRouteResult;

*// Assert*

Assert.IsNotNull(result);

bankServiceMock.Verify(m => m.TransferMoney("test1", "test2", 123), Times.Once);

Assert.AreSame("Index", result.RouteValues["action"]);

}

}

}

Looking at the TransferMoney\_HappyPath\_TransfersMoney test method above, we set our model up with data, create an instance of the AccountController then call TransferMoney on that controller. The fact that we pass plain .NET objects to controllers is one of the ways MVC is easier to unit test over WebForms, where input is either read through the Request object, or from a server-side control, neither of which are easy to abstract. When TransferMoney works, it returns a redirect to the Index action so we assert that the response is a RedirectToRouteResult object (we do this by converting it using the "as" operator which returns null if the object can't be converted), we check that the TransferMoney method was called on our bank service, and finally that the action we are being redirected to is called Index. Strictly speaking, we could actually split this into two tests, one to ensure the bank service has been called and one to ensure the result is a redirect to Index, but for brevity, I'm keeping all happy path tests in a single test function. This is easier as our TransferMoney method only has one happy path, but that won't always be the case.

Let's flesh our test class out with some more tests.

Hide   Shrink    Copy Code

using Microsoft.VisualStudio.TestTools.UnitTesting;

using Moq;

using System.Web.Mvc;

using UnitTestArticle.Controllers;

using UnitTestArticle.Interfaces;

using UnitTestArticle.Models;

using UnitTestArticle.Services.Exceptions;

namespace UnitTestArticle.Tests.Controllers

{

[TestClass]

public class AccountControllerTests

{

private Mock<IBankService> bankServiceMock;

[TestInitialize]

public void Init()

{

bankServiceMock = new Mock<IBankService>();

}

[TestMethod]

public void TransferMoney\_HappyPath\_TransfersMoney()

{

*// Arrange*

var model = new TransferMoneyModel

{

SourceAccountNumber = "test1",

DestinationAccountNumber = "test2",

Amount = 123

};

var controller = new AccountController(bankServiceMock.Object);

*// Act*

RedirectToRouteResult result =

controller.TransferMoney(model) as RedirectToRouteResult;

*// Assert*

Assert.IsNotNull(result);

bankServiceMock.Verify(m => m.TransferMoney("test1", "test2", 123), Times.Once);

Assert.AreSame("Index", result.RouteValues["action"]);

}

[TestMethod]

public void TransferMoney\_WithSameAccount\_HasInvalidModelState()

{

*// Arrange*

var model = new TransferMoneyModel

{

SourceAccountNumber = "test1",

DestinationAccountNumber = "test1",

Amount = 123

};

var controller = new AccountController(bankServiceMock.Object);

*// Act*

ViewResult result = controller.TransferMoney(model) as ViewResult;

*// Assert*

Assert.IsNotNull(result);

Assert.IsTrue(controller.ModelState.Count > 0);

Assert.IsTrue(controller.ModelState.ContainsKey("SameAccount"));

}

[TestMethod]

public void TransferMoney\_WithInsufficientFunds\_HasInvalidModelState()

{

*// Arrange*

var model = new TransferMoneyModel

{

SourceAccountNumber = "test1",

DestinationAccountNumber = "test2",

Amount = 123

};

bankServiceMock.Setup(m => m.TransferMoney

(model.SourceAccountNumber, model.DestinationAccountNumber, model.Amount))

.Throws(new InsufficientFundsException());

var controller = new AccountController(bankServiceMock.Object);

*// Act*

ViewResult result = controller.TransferMoney(model) as ViewResult;

*// Assert*

Assert.IsNotNull(result);

Assert.IsTrue(controller.ModelState.Count > 0);

Assert.IsTrue(controller.ModelState.ContainsKey("InsufficientFunds"));

}

[TestMethod]

public void TransferMoney\_WithInvalidAccount\_HasInvalidModelState()

{

*// Arrange*

var model = new TransferMoneyModel

{

SourceAccountNumber = "test1",

DestinationAccountNumber = "test2",

Amount = 123

};

bankServiceMock.Setup(m => m.TransferMoney

(model.SourceAccountNumber, model.DestinationAccountNumber, model.Amount))

.Throws(new AccountNotFoundException("test1"));

var controller = new AccountController(bankServiceMock.Object);

*// Act*

ViewResult result = controller.TransferMoney(model) as ViewResult;

*// Assert*

Assert.IsNotNull(result);

Assert.IsTrue(controller.ModelState.Count > 0);

Assert.IsTrue(controller.ModelState.ContainsKey("AccountNotFound"));

}

}

}

For the various fail conditions, we configure the mocked bank service to throw the relevant exceptions to ensure the controller handles them correctly. For example, in the TransferMoney\_WithInsufficientFunds\_HasInvalidModelState test, we setup the mocked bank service like so:

Hide   Copy Code

bankServiceMock.Setup(m => m.TransferMoney(model.SourceAccountNumber,

model.DestinationAccountNumber, model.Amount))

.Throws(new InsufficientFundsException());

Any code in the controller that attempts to use the bank service to transfer that amount of money between those two accounts will have the bank service throw the InsufficientFundsException exception.

Our bank service is throwing exceptions to indicate invalid states but if the methods you mock return "false" to indicate failure rather than throw exceptions, then you would simply configure your mocked objects to return false instead.

Unit Testing Model Validation

The TransferMoneyModel I pass to the controller has MVC validation attributes attached to enforce conditions like mandatory fields, ranges, string lengths and so on however I haven't written any tests to cover these. While we have to remember that our tests should only cover our own logic and not third-party code and testing model validation is really just testing Microsoft's code, on the other hand, we might want to ensure that people haven't changed the validation attributes away from what we are expecting so you might want to add tests for model validation as well.

Unit Testing with Web Context Objects

The TransferMoney action we tested above doesn't rely on any web kind of web context, so it doesn't access the Request object, the Session object, cookies or other such things. If your code does rely on these things we handle it the exact same way we handled Entity Framework, we just abstract the code away behind interfaces that we can mock. Let's look at an updated action where we store the source account in the Session in case we want to default to it as a selection later. In addition to using the Session, we also send an email to the account holder with a confirmation. As emails rely on external resources such as SMTP servers and the network, this is also something we need to abstract away.

Rather than interact with the Session or SMTP directly, we're going to build a Session manager class that handles all interaction with the Session, and similarly we will have an Email service that does the same. Each of these new classes will also implement their own interfaces.

**Session Manager Interface**

Our Session manager is quite simple with a method to store values and one to retrieve them. For some added value, we are using generics to make Session storage strongly typed.

Hide   Copy Code

namespace UnitTestArticle.Interfaces

{

public interface ISessionManager

{

void Store<T>(string key, T value);

T Get<T>(string key);

}

}

**Session Manager**

Hide   Copy Code

using System.Web;

using UnitTestArticle.Interfaces;

namespace UnitTestArticle.Services

{

public class SessionManager : ISessionManager

{

public T Get<T>(string key)

{

return (T)HttpContext.Current.Session[key];

}

public void Store<T>(string key, T value)

{

HttpContext.Current.Session[key] = value;

}

}

}

**Email Service Interface**

Hide   Copy Code

namespace UnitTestArticle.Interfaces

{

public interface IEmailService

{

void SendTransferEmailConfirmation(string sourceAccountNumber,

string destinationAccountNumber, decimal transferAmount);

}

}

**Email Service**

I haven't bothered fleshing this out, but it would get the relevant email addresses, then construct and send the email via SMTP.

Hide   Copy Code

using UnitTestArticle.Interfaces;

namespace UnitTestArticle.Services

{

public class EmailService : IEmailService

{

public void SendTransferEmailConfirmation

(string sourceAccountNumber, string destinationAccountNumber, decimal transferAmount)

{

*// code to send email confirmation here*

}

}

}

**Account Controller**

We need to amend the controller to allow our two new services to be passed in, and I have also amended the TransferMoney action to store the source account number in the Session and to send the email. The updated controller is below:

Hide   Shrink    Copy Code

public class AccountController : Controller

{

private IBankService bankService;

private ISessionManager sessionManager;

private IEmailService emailService;

public AccountController()

: this (new BankService(), new SessionManager(), new EmailService())

{

}

public AccountController(IBankService bankService,

ISessionManager sessionManager, IEmailService emailService)

{

this.bankService = bankService;

this.sessionManager = sessionManager;

this.emailService = emailService;

}

[HttpPost]

public ActionResult TransferMoney(TransferMoneyModel model)

{

if (model.SourceAccountNumber == model.DestinationAccountNumber)

{

ModelState.AddModelError("SameAccount",

"The source and destination accounts must be different");

}

if (!ModelState.IsValid)

{

return View(model);

}

try

{

bankService.TransferMoney(model.SourceAccountNumber,

model.DestinationAccountNumber, model.Amount);

}

catch (InsufficientFundsException)

{

ModelState.AddModelError("InsufficientFunds",

"There were insufficient funds to complete the transfer.");

}

catch (AccountNotFoundException ex)

{

ModelState.AddModelError("AccountNotFound",

$"There was a problem finding account {ex.AccountNumber}.");

}

catch

{

ModelState.AddModelError("TransferFailed",

"There was a problem with the transfer, please contact your bank.");

}

if (!ModelState.IsValid)

{

return View(model);

}

this.sessionManager.Store("SourceAccountNumber", model.SourceAccountNumber);

this.emailService.SendTransferEmailConfirmation

(model.SourceAccountNumber, model.DestinationAccountNumber, model.Amount);

return RedirectToAction("Index");

}

**Unit Tests**

I've amended the happy path test for TransferMoney to verify that the session manager and the email service have been called. Again, we could actually have these as their own tests but I am putting them all in one test for brevity.

Hide   Shrink    Copy Code

[TestClass]

public class AccountControllerTests

{

private Mock<IBankService> bankServiceMock;

private Mock<ISessionManager> sessionManagerMock;

private Mock<IEmailService> emailService;

[TestInitialize]

public void Init()

{

bankServiceMock = new Mock<IBankService>();

sessionManagerMock = new Mock<ISessionManager>();

emailService = new Mock<IEmailService>();

}

[TestMethod]

public void TransferMoney\_HappyPath\_TransfersMoney()

{

*// Arrange*

var model = new TransferMoneyModel

{

SourceAccountNumber = "test1",

DestinationAccountNumber = "test2",

Amount = 123

};

var controller = new AccountController

(bankServiceMock.Object, sessionManagerMock.Object, emailService.Object);

*// Act*

RedirectToRouteResult result = controller.TransferMoney(model) as RedirectToRouteResult;

*// Assert*

Assert.IsNotNull(result);

*// assert TransferMoney was called on the bank service*

bankServiceMock.Verify(m => m.TransferMoney("test1", "test2", 123), Times.Once);

*// assert the source account number was stored in the session*

sessionManagerMock.Verify(m => m.Store("SourceAccountNumber", "test1"));

*// assert the confirmation email was sent*

emailService.Verify(m => m.SendTransferEmailConfirmation("test1", "test2", 123),

Times.Once);

*// assert the result is a redirect to Index*

Assert.AreSame("Index", result.RouteValues["action"]);

}

Dependency Injection and IoC

When I create controllers and other classes that have dependencies, I use a pattern where I have a constructor that accepts those dependencies and also a default parameterless constructor that specifies which concrete classes to use.

Hide   Copy Code

public BankService()

: this (new BankRepository(), new ReportingService())

{

}

public BankService(IBankRepository repo, IReportingService reportingService)

{

this.repo = repo;

this.reportingService = reportingService;

}

The parameterised constructor is using inversion of control which is a technique where a class is given their dependent classes by the calling code. This allows the site to use the proper version of classes under normal operation and the mocked version of classes when doing unit testing.

It is common to use "dependency injection" (DI) frameworks in MVC projects which is a framework that allows you to register which concrete class is to be used as the implementation for a given interface, and MVC has built-in support for using DI to create controllers. When DI is used in a project, the default parameterless constructor can be removed if your interfaces are all mapped to concrete classes in your DI framework, because when the controller is created by MVC, it looks at each interface in the constructor parameters and gets DI to provide the appropriate concrete class for that parameter. This process is recursive so if one of those classes (BankService for example) also has a constructor with parameters (IBankRepository and IReportingService) DI is used to resolve those parameters into concrete classes too. However when DI is not registered with the MVC framework, it uses the default parameterless constructor to create controllers which is why we use the pattern shown above.

Code Coverage

Code coverage is a metric that lets you know what percentage of your code is covered by your tests, and it can sometimes highlight scenarios you hadn't considered yourself. Some of the premium versions of Visual Studio have code coverage tools built-in, and there are plenty of third-party code coverage tools, however most have to be purchased, but there are some open source solutions available too.