**Chapter 8: Unit Testing parallel and asynchronous programs**

*"With great power comes great responsibility."*

In this chapter we will cover what is unit testing, why unit testing is needed and we will see how to write unit tests for methods that support parallel execution and also for methods that can be executed asynchronously.

**Structure**

* Unit Testing (What, Why and How)
* Unit Test for async program using XUnit
* Unit Test for Parallel program using XUnit
* Summary
* Exercise

**Objectives**

By end of this chapter reader should be able to understand

* What is unit testing and why unit tests are needed?
* How to write unit tests for async methods
* How to write unit tests for parallel methods

**Overview**

Before we get started on how to do unit testing first thing that we should understood what unit tests are and why unit testing is needed. A unit test is a method that calls a method in our application and validates the response of that method against a predefined value. This predefined value is called as mock data and the process of validating mock data with the output of calling method is called as assertion.

For example, if there is a method Divide in my application that takes two integers as input and returns division of those methods, a typical unit test for such method will look like below

public void TestDivide()

{

var mathClass = new MathClass();

int output = 3; //mock data

var result = mathClass.Divide(6, 2);

Assert.Equal(output, result);

}

In this example first we define the mock data i.e. expected output, then afteroect initialization we call actual method and store it’s result which is used to compare with expected output. The idea of writing unit tests is that even though underlying method(in this case Divide method of MathClass) may undergo some changes like a different library can be used to calculate division etc. but any of these changes should not change the output of this method.

One way to ensure that a change in method has not broken anything is to execute it manually and validate the output, however this is error prone as there could be many scenarios in real time applications. Another way to do this is write unit test(s) which can be excuted every time method is changed to ensure final output of method is not changed.

There are many frameworks available to write unit tests like the built-in one comes directly with visual studio, then we have third party frameworks like NUnit, XUnit etc. More or less each framework supports writing all kinds of unit tests so it’s upto developers on choosing which framework they want to use as all the frameworks does support writing any kind of unit tests including unit tests that support async/await.For the purpose of this book we will focus on using XUnit, however all the samples can be written in other frameworks as well.

**Basics of unit testing with XUnit**

XUnit is a unit testing library that comes with all the necessary methods to unit test our application code. When using XUnit any method that is annotated with keyword [Fact] becomes a test method. In general unit tests are created as part of separate class library project where we add a class file, add reference to the class that needs to be tested and then create a test method by annotating it with keyword [Fact]. Although unit test classes can be part of same project as the class that it is testing it is recommended to make it as a separate project for easy maintainability and segregation.

Let us create a class library project and add a simple class file that we are going to test, let us call it MathClass. Add a method Divide that accepts 2 parameters and returns division, with this MathClass class will look like below

public class MathClass

{

public int Divide(int numerator, int denominator)

{

if (denominator == 0)

{

throw new DivideByZeroException();

}

else

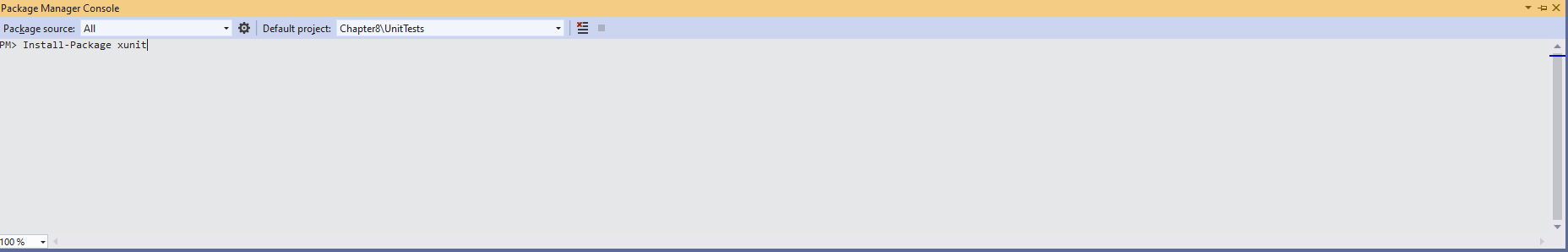
return numerator / denominator;

}

}

Now let’s add a unit test for this method, to start with let us create a class library project UnitTests and add a class, name it MathClassUnitTest. Now since we are using XUNnit we need to install XUnit package. Open package manager console and run below command as shown in Figure 8.1

Install-Package xunit



**Figure 8.1 – Install XUnit through package manger console**

Also, we need to install XUnit runner package that will help to run the test case through visual studio test runner. For this run below command

Install-Package xunit.runner.visualstudio

Now follow below steps to add a unit test

1. Add reference of MathClass by referencing Calculator project to unit test project
2. Add a method TestDivide that creates an object of MathClass and call divide method
3. To verify output with expected value we will make use of Assert class of XUnit
4. Annotate the method with keyword [Fact]

With this unit test class will look like below

using Calculator;

using Xunit;

public class MathClassUnitTest

{

[Fact]

public void TestDivide()

{

var mathClass = new MathClass();

int output = 3; //mock data

var result = mathClass.Divide(6, 2);

Assert.Equal(output, result);

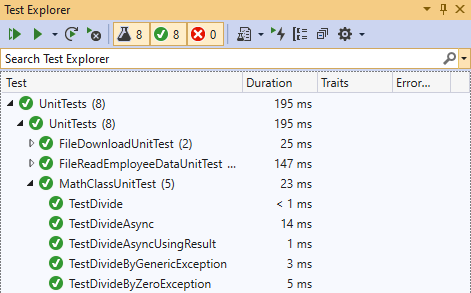
}

}

Here we are following AAA technique (Arrange, Act and Assert) where we start with initiaizing data and then invoke method and eventually validate output with expected result. In Visual studio 2019 to execute a test method application needs to be successfully built. So build the application and once it is successfully built there are multiple ways as mentioned below to run a unit test

**Executing unit tests**

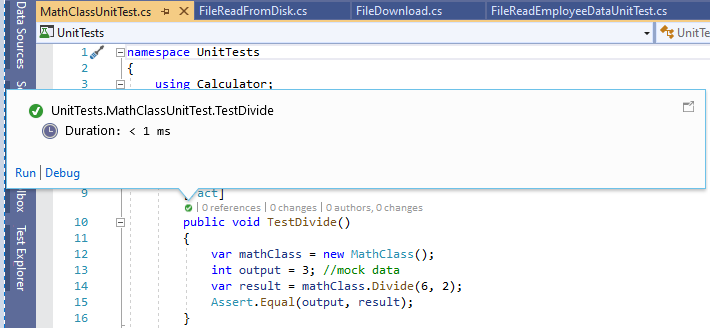
Visual studio test explorer lists dows all the unit tests available in our solutoin. Navigate to test explorer through View -> Test Explorer where we can see our test method as shown in Figure 8.2



**Figure 8.2 – Test explorer**

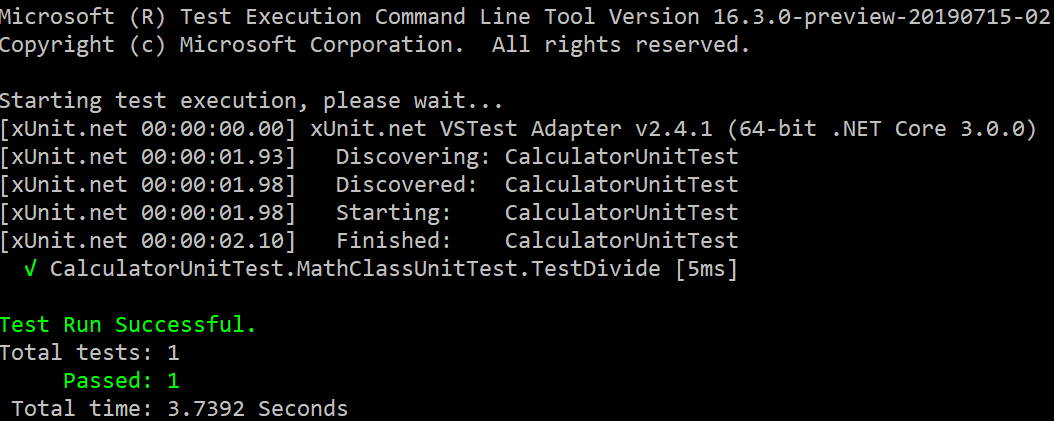
Now right click on the test divide method and click Run to execute the test, if the test passes blue icon will turn green , if test fails then it will turn red with error message at bottom of the test explorer. Similarly, we can debug test method.

Another way to run/debug a unit test is by clicking on the blue icon just above the unit test method as shown in Figure 8.3



**Figure 8.3 – Run/Debug unit test**

Another option to run unit tests is through developer command prompt by passing unit test dll as a parameter to vstest.console.exe. So, in our case open VS 2019 developer command prompt and run vstest.console.exe UnitTests.dll to run all the unit tests. Output will look something like in Figure 8.4



**Figure 8.4 – Run unit test through command line**

vstest.console.exe also gives many options to execute certain tests etc. and is very useful if there is a need to execute unit tests without visual studio. Most common scenario is while building continuous integration/continuous deployment (CI/CD) pipelines where one of the tasks is to execute all unit tests.

In this section we covered on how to create and configure unit test projects and various options to run unit tests. In next sections we will focus on how to write unit tests for asynchronous methods and parallel methods.

**Unit test async methods**

Unit testing asynchronous methods isn’t as straight forward as unit testing synchronous methods because asynchronous methods aren’t completed in one single call and if we test asynchronous methods like synchronous methods our test method won’t wait for the completion of asynchronous method and may end up asserting even before asynchronous method completion.

Fortunately, with XUnit writing unit tests is as easy writing any asynchronous method i.e. any asynchronous method that needs to be unit tested can be prefixed using keyword async. Let us see this with a simple example by creating a method that takes two parameters and returns division of those parameters (same as previous one) asynchronously. So, let’s add a new method DivideAsync to MathClassUnitTest class and let’s wrap Divide method into a Task and await on that as shown below

public async Task<int> DivideAsync(int numerator, int denominator)

{

var t = Task.Run(() =>

{

return Divide(numerator, denominator);

});

return await t;

}

Now add a test method TestDivideAsync in MathClassUnitTest, since we are writing unit test for sync method to assert the output we need to ensure that method execution is completed before assert and the way to do that is nothing different than calling any asynchronous method i.e. prefix the call to asynchronous method by await. Since we are awaiting on one of the method unit test method’s return type would be async task instead of void. So, our method definition will look like below

[Fact]

public async Task TestDivideAsync()

{

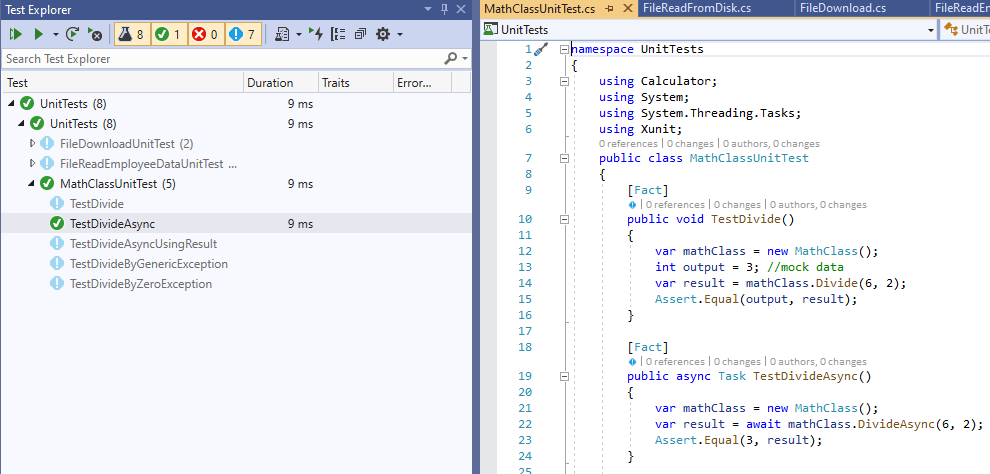
var mathClass = new MathClass();

var result = await mathClass.DivideAsync(6, 2);

Assert.Equal(3, result);

}

Build the application and run the unit test and it sould pass as shown in Figure 8.5



**Figure 8.5 – Unit test for async method**

Let’s tweak DivideAsync method i.e. method will still do division however we will use Math library from framework and this should not cause any change in unit test method. With this our method will look like below

public int Divide(int numerator, int denominator)

{

if (denominator == 0)

{

throw new DivideByZeroException();

}

else

{

int remainder;

return Math.DivRem(numerator, denominator, out remainder);

}

}

After this change running TestDivideAsync will still pass as only implementation of method is changed but the intent remains same. This is the biggest advantage of writing a unit test where we can ensure that any change to method implementation hasn’t broken any of the existing behavior.

Since XUnit support await we can write unit tests what can await with ease, however if we end up using a testing framework that doesn’t support await in unit tests then we need to follow the technique of calling asynchronous method from synchronous method i.e. by using GetAwaiter().GetResult() . So, let us add another test that synchronously calls DivideAsync method as shown below

[Fact]

public void TestDivideAsyncUsingResult()

{

var mathClass = new MathClass();

var result = mathClass.DivideAsync(6, 2).GetAwaiter().GetResult();

Assert.Equal(3, result);

}

Once we run this test it will pass, however this way is not at all recommended as this will possibly cause deadlock if we are mocking library code. So to be on the safer side we should use ConfigureAwait(false) i.e. return statement in our DivideAsync method should be changed to

return await t.ConfigureAwait(false);

This will ensure that any consumer of our library do not complain of deadlock while consuming it.

**Unit test exceptions in async methods**

It is a good practice that whenever we write unit tests, they should be written for both positive and negative scenario, specially exception cases. Obviously when a handled exception occurs in application our unit test should have the capability to assert against exception.

XUnit gives various overloads of Throw method along with asynchronous version to assert against any exception. Going back to our divide example since we already handled divide by zero exception, let us write a unit test for this scenario. In this unit test we will pass denominator as 0 and expected output would be a divide by zero exception, so our unit test will look like below

[Fact]

public async Task TestDivideByZeroException()

{

var mathClass = new MathClass();

var result = mathClass.DivideAsync(6, 0);

await Assert.ThrowsAsync<DivideByZeroException>(async () => await result);

}

Notice that we are awaiting on ThrowAsync as if we do not await this test will always pass irrespective of the exception, reason being same as not awaiting on any asynchronous method.

In this case we handled a very specific exception however if we want to handle generic exception then XUnit provides ThrowAnyAsync which can receive any exception and pass test case accordingly. So let us tweak Divide method a little to throw another exception based on some condition say when denominator is 1. Adding this condition method will look like below

public int Divide(int numerator, int? denominator)

{

if (denominator == 0)

{

throw new DivideByZeroException();

}

else if (!denominator.HasValue)

{

throw new ArgumentNullException ();

}

else

{

int remainder;

return Math.DivRem(numerator, denominator.Value, out remainder);

}

}

Now add a unit test that will receive ArgumentNullException exception using ThrowAnyAsync which looks like below

[Fact]

public async Task TestDivideByGenericException()

{

var mathClass = new MathClass();

var result = mathClass.DivideAsync(6, null);

await Assert.ThrowsAnyAsync<Exception>(async () => await result);

}

Build and run this test case and it will pass. This method will pass in either of the exception i.e. DivideByZeroException or ArgumentNullException. Just like ThrowAsync ThrowAnyAsync also needs to be awaited or else you end up with a test that is always passing.

**Unit test async method using mock data**

In most of the enterprise applications unit tests are primarily around service layer classes where will have outbound calls like a database call or loading file in memory etc. However, unit tests aren’t supposed to make outbound calls instead they should create mock data for all the outbound calls involved and then validate the business logic method that is unit tested for.

Let us see this with a simple example where we have method that downloads file from web asynchronously and then reads file content, apply some logic (in our case we will do a string reversal) and send back response. In this case file downloading is an external call and for unit testing it should mock with some predefined response.

Add a new class library project, let us call it FileIO. Let us start with adding a class FileDownload and we will use HttpClient of .Net which will be initialized through constructor , this class will look like below

namespace FileIO

{

public class FileDownload

{

HttpClient \_client;

public FileDownload(HttpClient client)

{

if (client != null)

\_client = client;

else

\_client = new HttpClient();

}

}

}

Add a method DownloadFileAsync, we will use GetAsync method of it to download file. Once it is downloaded use ReadAsStringAsync to retrieve data and apply string reversal before returning. This method will look like below

public async Task<string> DownloadFileAsync()

{

string url = "https://github.com/Ravindra-a/largefile/blob/master/README.md"; //Replace this with any URL

using (HttpResponseMessage response = await \_client.GetAsync(url)) // Should mock GetAsync for unit tests

{

if (response.IsSuccessStatusCode)

{

string result = await response.Content.ReadAsStringAsync();

// Now reverse this string - In enterprise appication this will be some business logic

StringBuilder reverseString = new StringBuilder();

for (int i = result.Length - 1; i >= 0; i--)

{

reverseString.Append(result[i]);

}

return reverseString.ToString();

}

else if (response.StatusCode == HttpStatusCode.NotFound)

{

throw new FileNotFoundException();

}

throw new Exception(); //For all other stauts codes

}

}

Now let us write a unit test for this method where focus is on mocking response from GetAsync and validating string reversal logic. So before even starting unit tests we need to create mock objects for all the dependencies and looking at our DownloadFileAsync method we have only one outbound call which is GetAsync of HttpClient class, so we some how need create a mock object of HttpClient.

Mock objects help in mimicking the behavior just like a real time call, however instead of returning actual data mock objets will return response as per dummy data we define, to do this we can use many frameworks like Moq, Fakeiteasy etc. , however in this example we will create HttpClient object (also as it is only one single mock object we do not need a full fledged framework) using it’s constructor that takes HttpMessageHandler, for this HttpMessageHandler we will create a mock object. Since SendAsync method of HttpMessageHandler allows us to customize the response we can use it in a way that will help us to cucstomize the behavior of HttpClient.

So let’s add a new class FakeHttpMsgHandler to unit test project which is inherited from HttpMessageHandler and implement SendAsync as shown below

public class FakeHttpMsgHandler : HttpMessageHandler

{

private HttpResponseMessage \_response;

//Constructor

public FakeHttpMsgHandler(HttpResponseMessage response)

{

\_response = response;

}

protected override Task<HttpResponseMessage> SendAsync(HttpRequestMessage request, System.Threading.CancellationToken cancellationToken)

{

var taskCompletionSource = new TaskCompletionSource<HttpResponseMessage>();

taskCompletionSource.SetResult(\_response);

return taskCompletionSource.Task;

}

}

Now add a unit test class FileDownloadUnitTest and add reference to FileIO project. Add a unit test method DownloadFileSuccess as shown below

[Fact]

public async Task DownloadFileSuccess()

{

// Dummy response

HttpResponseMessage mockResponse = new HttpResponseMessage

{

StatusCode = HttpStatusCode.OK,

Content = new StringContent("Response from fake httpclient")

};

var msgeHandler = new FakeHttpMsgHandler(mockResponse);

var httpClient = new HttpClient(msgeHandler);

var fileDownloadObj = new FileIO.FileDownload(httpClient);

string expectedResult = "Response from fake httpclient";

//string reversal logic

char[] charArray = expectedResult.ToCharArray();

Array.Reverse(charArray);

//Call to method

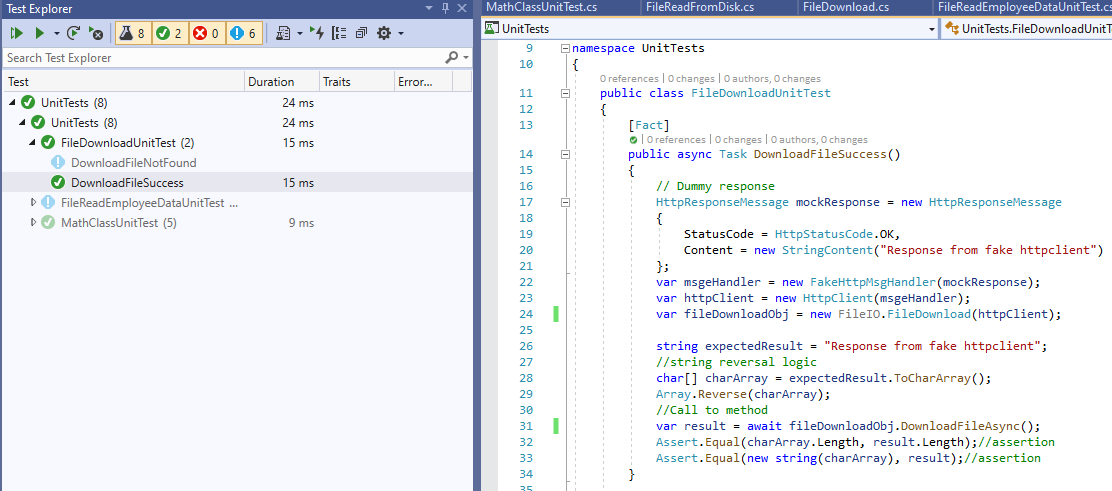
var result = await fileDownloadObj.DownloadFileAsync();

Assert.Equal(charArray.Length, result.Length);//assertion

Assert.Equal(new string(charArray), result);//assertion

}

Here we are calling our file download method which will respond with the mock response object, so the content will be “Response from fake httpclient” which will be reversed as per business rules with in DownloadFileAsync method. Finally, we are asserting that in our unit test. Once we run this test it will pass as shown in image



**Figure 8.6 – Unit test for async method**

Now let us add a negative test case where response from GetAsync is 404 (not found), so first thing we need to change is mock response which is be sending http status code 404. Then we will use ThrowsAsync to assert response against specific exception, in this case FileNotFoundException. With this our unit test will look like below

[Fact]

public async Task DownloadFileNotFound()

{

// Dummy response

HttpResponseMessage mockResponse = new HttpResponseMessage

{

StatusCode = HttpStatusCode.NotFound,

};

var msgeHandler = new FakeHttpMsgHandler(mockResponse);

var httpClient = new HttpClient(msgeHandler);

var fileDownloadObj = new FileIO.FileDownload(httpClient);

var result = fileDownloadObj.DownloadFileAsync();

await Assert.ThrowsAsync<FileNotFoundException>(async () => await result);

}

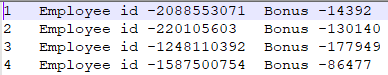
So here we ensured that the mock HttpResponseMessage object is returning a 404 and verified if our method is returning appropriate exception which is asserted through using XUnit’s ThrowAsync. This way we can add more unit tests. So, in this section we covered on how to create mock objects in case of dependencies so that our unit test focuses validating business logic.

**Unit test for parallel methods**

Till now we have seen various scenarios around unit testing asynchronous methods, in this section we will see how to unit test when there is parallel code or data parallelism i.e. say if there is method that uses parallel loops.

In TPL as we know we have Parallel.For and Parallel.ForEach which helps us in executing a method concurrently without bothering about creating threads and manage them, however with parallel code the thing that needs to be managed is handling data parallelly i.e. shared resources should be synchronized either through synchronization constructs like locks, semaphores etc. or through the data structures available for parallelism like ConcurrentBag, BlockingCollection etc.

In either case from a unit testing standpoint parallel execution of code do not change much on how unit tests are written, but they are absolutely necessary as it helps to test the concurrency logic and validate that shared resources aren’t corrupted. Let us see this with an example where we load a huge file that holds employee data (employee ID and bonus), each employee record is separate by a new line. This file will typically look something like below



**Figure 8.7 – Employee file**

We will load this into memory and then parallelly process each record, apply some business rules and return the final output which will be a subset of list of employees based on the filtering applied in business rules. So, to our FileIO class library project let us add an employee class and add below contents

public class Employee

{

public int EmployeeID { get; set; }

public int Bonus { get; set; }

}

Now to read a file we will use a StreamReader class, however as we would be writing unit tests, so we need to mock this StreamReader such that mock data that is configured in unit tests is returned instead of actually reading file from disk. So, we will add an interface IFileReader which will be used to read file using StreamReader and also used later to mock, let’s us add an interface IFileReader that looks like below

public interface IFileReader

{

StreamReader GetFileReader(string filePath);

}

The advantage of using interface is that it can be mocked very well with any mocking framework. We can also go ahead and do dependency injection but for now to keep it simple and focus on unit testing we will not do that.

Now add a class that implements this interface which basically gives us an object of StreamReader, let us call this class FileStreamReader and it will look like below

public class FileStreamReader : IFileReader

{

public StreamReader GetFileReader(string filePath)

{

return new StreamReader(filePath);

}

}

Now let us add our class where we read a file with list of employees, process records parallely and apply business rules, let us call this class FileReadFromDisk and add below code to it

public class FileReadFromDisk

{

private readonly IFileReader \_streamReader;

//Thread safe collection to store exceptions occurred during parallel processing

ConcurrentBag<Exception> errors = new ConcurrentBag<Exception>();

public FileReadFromDisk(IFileReader streamReader)

{

this.\_streamReader = streamReader;

}

}

Then we will create two methods that return Task

1. To read data from file and load into a list of employees. Here we will use something called as BlockingCollection which is a thread safe collection and allows to read and write to collection concurrently without corrupting collection. We will read line by line and add it to this collection, later we will read from this collection concurrently.This method will look like below

public Task ReadDataFromFile(string filePath, BlockingCollection<string> employeeData)

{

return Task.Factory.StartNew(() =>

{

using (StreamReader sr = this.\_streamReader.GetFileReader(filePath))

{

while (!sr.EndOfStream)

{

employeeData.Add(sr.ReadLine());

}

}

// Notify consumers that addition is completed

employeeData.CompleteAdding();

});

}

1. To serialize data that is loaded in employeeData collection into a collection of Employee type, here we will use Parallel.ForEach to iterate through employeeData collection and load into collection of Employee type. For collection of Employee type we will use ConcurrentBag as it is a thread safe collection and allows to add data concurrently without corrupting the list. This method will look like below

private Task SerializeEmployeeData(BlockingCollection<string> employeeData, ConcurrentBag<Employee> employeeDetails)

{

return Task.Factory.StartNew(() =>

{

try

{

Parallel.ForEach(employeeData.GetConsumingEnumerable(), line =>

{

// String manipulation

string[] lineFields = line.Split('\t');

int employeeID, bonus;

int.TryParse(lineFields[1].Substring(lineFields[1].IndexOf('-') + 1), out employeeID);

int.TryParse(lineFields[2].Substring(lineFields[2].IndexOf('-') + 1), out bonus);

employeeDetails.Add(new Employee { EmployeeID = employeeID, Bonus = bonus });

});

}

catch (Exception ex)

{

errors.Add(ex);

}

});

}

In above method we are also doing exception handling as normally in Parallel.ForEach loop if there is an exception in one of the iteration it won’t process any of the subsequent iterations, however what we want to do here is to handle the exception for every iteration so that we can process all iterations. At the end of this method we will handle any raised exceptions accordingly, here we are using ConcurrentBag collection of type Excepions that was declared earlier to accumulate exceptions during each iteration. Again the reason to use ConcurrentBag is to achive thread safety.

Now add a method ReadFileandProcessTask that will take 2 parameters

1. filePath as string – This is the path of the file that needs to be loaded
2. bonusAmountRule as int- This is a variable that is used in business rules to filter employees list

This method will further call the tasks defined above and will implement business rules to filter out employee list.Code for this method will look like below

public List<Employee> ReadFileandProcessTask(string filePath, int bonusAmountRule)

{

// Using ConcurrentBag for thread safety

var employeeDetails = new ConcurrentBag<Employee>();

// Blocking collection so that multiple consumers do not end up corrupting data

var employeeData = new BlockingCollection<string>();

// Single Producer

var readLines = ReadDataFromFile(filePath, employeeData);

// Multiple Consumers

var processLines = SerializeEmployeeData(employeeData, employeeDetails);

Task.WaitAll(readLines, processLines);

// Throw the exceptions here after the loop completes.

if (errors.Count > 0)

{

throw new AggregateException(errors);

}

//Business logic - Get all users with bonus greater than 50000

return employeeDetails.Where(x => x.Bonus >= bonusAmountRule).ToList();

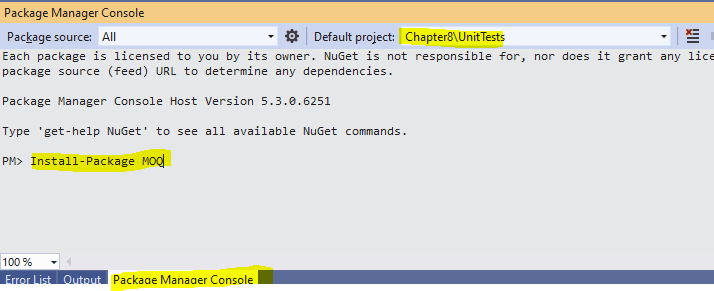
}

If we look at closely we are using producer and consumer pattern here where one producer is reading file and there are multiple consumers that are notified once producer completes its tasks.

If we do a build at this stage FileIO class library should build successfully.

Now let’s add unit tests for ReadFileandProcessTask, but before we start writing unit test let us add mocking framework that we planned to use which is MOQ. So open Package Manager Console, under Default Project: select UnitTests project and run below command (see Figure 8.8)

Install-Package Moq



**Figure 8.8 – Package Manager Console command to install Moq library**

Add a class FileReadEmployeeDataUnitTest to UnitTests project and add first unit test which will be around testing the logic in the application i.e. pass a list of mock employee data and see if we are getting appropriate filter results. So first add a private method that will return mock data, code of that method will look like below

private StringBuilder GetMockFileData ()

{

// We will follow the pattern used for manipulation in SerializeEmployeeData i.e. data separate by tab

StringBuilder mockFileData = new StringBuilder();

mockFileData.AppendLine("1 Employee id -1 Bonus -14392");

mockFileData.AppendLine("2 Employee id -2 Bonus -130140");

mockFileData.AppendLine("3 Employee id -3 Bonus -177949");

mockFileData.AppendLine("4 Employee id -4 Bonus -86477");

mockFileData.AppendLine("5 Employee id -5 Bonus -202725");

mockFileData.AppendLine("6 Employee id -6 Bonus -203595");

mockFileData.AppendLine("7 Employee id -7 Bonus -43698");

return mockFileData;

}

Now add the unt test method in which we will first create a mock object of file stream and use MemoryStream to load mock content into stream and return an object of StreamReader that takes this MemoryStream as parameter. Moq library gives a method called Setup which is used to return mock data when any particular method is called i.e. in ReadFileandProcessTask we want to return mock data from GetMockFileData when StreamReader reads the file and calls methods like ReadLine and EndOfStream, so we will use Setup method of Moq library to return a StreamReader that loads mock data we created.

Once this is done we will call the ReadFileandProcessTask followed by asserts, with this code for unit test will look like below

[Fact]

public void EmployeeDetailsEmployeesWithHigherBonusFound()

{

//Setup mocking data

string mockPath = "mockPath";

StringBuilder content = GetMockFileData();

MemoryStream ms = new MemoryStream(Encoding.UTF8.GetBytes(content.ToString()));

Mock<IFileReader> reader = new Mock<IFileReader>();

//Using Moq to respond with mock data for file stream

reader.Setup(sr => sr.GetFileReader(mockPath)).Returns(new StreamReader(ms));

FileReadFromDisk sut = new FileReadFromDisk(reader.Object);

//Call the app

List<Employee> employeesWithHigherBonus = sut.ReadFileandProcessTask(mockPath, 200000);

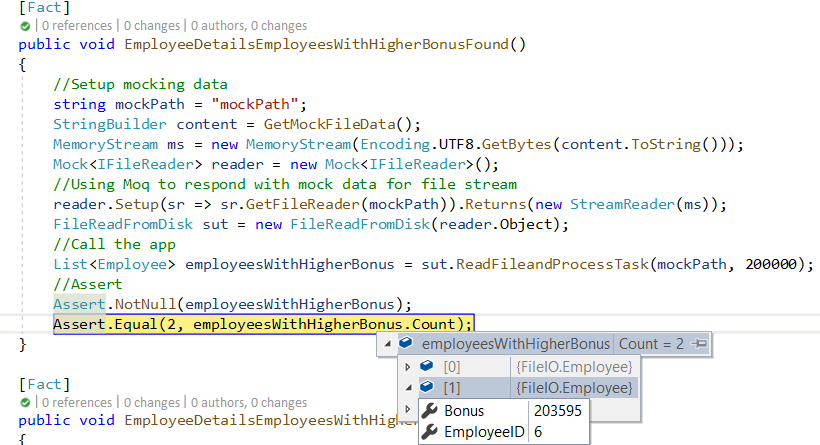
//Assert

Assert.NotNull(employeesWithHigherBonus);

Assert.Equal(2, employeesWithHigherBonus.Count);

}

Once we run this we will see that test will pass as shown in Figure 8.9 as our mock data has 2 records that has bonus greater than 200000



**Figure 8.9 – Unit test in debug mode with output**

Now let us add another unit test where our method returns an empty object, so for this we will pass String.Empty to our MeMoryStream and rest remains same as previous test and this time we will use Assert.Empty of XUnit as there won’t be any employees in the response from ReadFileandProcessTask. With this our unit test would look like below

[Fact]

public void EmployeeDetailsEmployeesWithHigherBonusNotFound()

{

//Setup mocking data

string mockPath = "mockPath";

MemoryStream ms = new MemoryStream(Encoding.UTF8.GetBytes(String.Empty));

Mock<IFileReader> reader = new Mock<IFileReader>();

//Using Moq to respond with mock data for file stream

reader.Setup(sr => sr.GetFileReader(mockPath)).Returns(new StreamReader(ms));

FileReadFromDisk sut = new FileReadFromDisk(reader.Object);

//Call the app

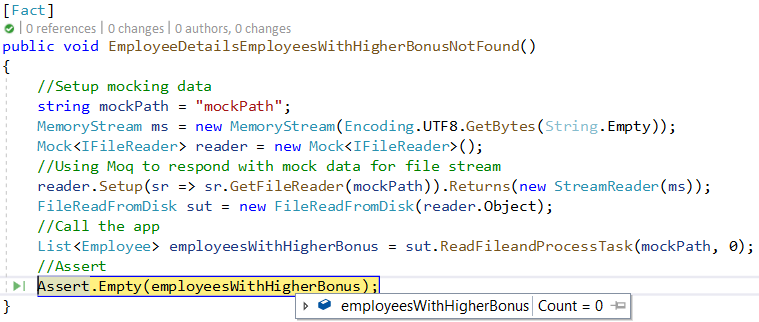
List<Employee> employeesWithHigherBonus = sut.ReadFileandProcessTask(mockPath, 0);

//Assert

Assert.Empty(employeesWithHigherBonus);

}

Once we run this unit test it will pass as shown in Figure 8.10



**Figure 8.10 – Unit test in debug mode with output**

Now let us try a scenario where the parallel loop throws an exception and write a unit test for that. To simulate this scenario we will create a mock data that breaks the pattern of our records which will throw exception and since we are continuing the parallel loop and accumulating exceptions in AggregateException we need to Assert output with AggregateException using Throw method of XUnit. Code for this unit test will look like below

[Fact]

public void EmployeeDetailsProcessingFailed()

{

//Setup mocking data

StringBuilder mockFileExceptionData = new StringBuilder();

// Record to throw exception

mockFileExceptionData.AppendLine("Exception record");

mockFileExceptionData.AppendLine("1 Employee id -1 Bonus -14392");

string mockPath = "mockPath";

MemoryStream ms = new MemoryStream(Encoding.UTF8.GetBytes(mockFileExceptionData.ToString()));

Mock<IFileReader> reader = new Mock<IFileReader>();

reader.Setup(sr => sr.GetFileReader(mockPath)).Returns(new StreamReader(ms));

//Call the app

FileReadFromDisk sut = new FileReadFromDisk(reader.Object);

//Assert

var ex = Assert.Throws<AggregateException>(() => sut.ReadFileandProcessTask(mockPath, 10000));

Assert.Single(ex.InnerExceptions);

//Asserting inner exception

Assert.Equal((new IndexOutOfRangeException()).GetType().Name, ex.Flatten().InnerExceptions[0].GetType().Name);

}

So in this we are asserting against Throw method of Assert and using Flatten method of AggregateException to getting the actual exception. And asserting. As others this oce you run this test it should pass.

So in this section we primarily covered on the things that needs to be taken care while unit testing parallel methods and went through small real time scenario and wrote u nit tests for it.

**Unit test async Void methods**

**Summary**

**Exercise**

1. Change return type of one of the asynchronous unit test to async void and check the output ? Does it always fail or works same way, if yes why ?