Lab BONUS - Unit Testing Asynchronous Methods

Objectives

This lab shows some techniques to unit test asynchronous code, including writing async tests and creating asynchronous fake objects.

Application Overview

The "Starter" folder contains the code files for this lab.

Visual Studio 2022: Open the "DataProcessor.sln" solution.
Visual Studio Code: Open the "Starter" folder in VS Code.

Note: The lab also contains a "Completed" folder with the finished solution. If you get stuck along the way or have issues with debugging, take a look at the code in the "Completed" folder for guidance.

This solution is a console application that processes data from a text file. For details on the application, please refer to the "Overview" and "Current Classes" section of Lab01.

The "DataProcessor.Library.Tests" project contains existing tests. These use xUnit as the testing framework. Run the tests as follows:

Visual Studio Code
For Visual Studio Code, we'll run the tests from the command line. (This is the most consistent experience. If you use Visual Studio Code full time, you will want to find a plug-in for your specific test framework.)

One way to do this in Windows 11 is to open the "DataProcessor.Library.Tests" folder in File Explorer, and then right-click a blank space in the folder. There should be an option such as "Open in Terminal" or "Open in PowerShell".

Alternately, you can open PowerShell or Terminal (if installed) and then navigate to the service folder using the "cd" (change directory) command.

```
PS C:..\LabBONUS\Starter\DataProcessor.Library.Tests>
```

From here, type `dotnet test` to run the unit tests.

```
PS C:\LabBONUS\Starter\DataProcessor.Library.Tests> dotnet test
```

You should get output similar to the following:

```
[xUnit.net 00:00:00.00] xUnit.net VSTest Adapter v2.8.2+699d445a1a (64-bit .NET
9.0.5)
[xUnit.net 00:00:00.57]   Discovering: DataProcessor.Library.Tests
[xUnit.net 00:00:00.60]   Discovered:  DataProcessor.Library.Tests
[xUnit.net 00:00:00.60]   Starting:    DataProcessor.Library.Tests
... (lots of stuff omitted)

Test summary: total: 9, failed: 5, succeeded: 4, skipped: 0, duration: 4.8s
Build failed with 5 error(s) in 14.4s
```

This shows a total of 9 tests. Some of them failed because they contain placeholder code.

Visual Studio 2022
If you are using Visual Studio 2022, you can use the integrated Test Explorer.

If you do not have the Test Explorer showing, use the "Test" menu and choose "Test Explorer" (or use "Ctrl+E, T" as a keyboard shortcut).

In the Test Explorer, use the "Run All Tests In View" button (usually on the far right), or press use "Ctrl+R, V" as a keyboard shortcut.

Note: your keyboard shortcuts may be different if you have third-party extensions installed (such as ReSharper).

The Test Explorer should show a total of 9 tests. Some of them are marked as "Failed" because they contain placeholder code.

Lab Goals

We have three goals:

1. Update the existing tests to properly await the async methods.
2. Create a fake object that can be used for testing.
3. Write tests that use the fake object.

Current Classes

DataProcessor.Library/DataParser

This is the class we want to test. This class parses a data file and logs errors using a separate logger class.

```
public async Task<IReadOnlyCollection<Person>> ParseData(IEnumerable<string> data)
{
    List<Person> processedRecords = [];
    foreach (var record in data)
    {
```

```
        var fields = record.Split(',');
        if (fields.Length != 6)
        {
            await logger.LogMessage("Wrong number of fields in record", record)
                .ConfigureAwait(false);
            continue;
        }

        int id;
        if (!Int32.TryParse(fields[0], out id))
        {
            await logger.LogMessage("Cannot parse Id field", record)
                .ConfigureAwait(false);
            continue;
        }

        DateTime startDate;
        if (!DateTime.TryParse(fields[3], out startDate))
        {
            await logger.LogMessage("Cannot parse Start Date field", record)
                .ConfigureAwait(false);
            continue;
        }

        int rating;
        if (!Int32.TryParse(fields[4], out rating))
        {
            await logger.LogMessage("Cannot parse Rating field", record)
                .ConfigureAwait(false);
            continue;
        }

        var person = new Person(id, fields[1], fields[2],
            startDate, rating, fields[5]);

        // Successfully parsed record
        processedRecords.Add(person);
    }
    return processedRecords;
}
```

The logger is injected through the constructor of the DataParser class.

```
public DataParser(ILogger logger)
{
    this.logger = logger ?? new NullLogger();
}
```

DataProcessor.Library/ILogger

This is the interface for the logger. We will use this to create fake/mock objects for testing:

```
public interface ILogger
{
    Task LogMessage(string message, string data);
}
```

DataProcessor.Library.Tests/TestData

This class contains hard-coded test data that can be used for testing.

DataProcessor.Library.Tests/FakeLogger

This is a stub for our fake logging class.

```
public class FakeLogger //: ILogger
{
}
```

This will need to be filled in for testing.

DataProcessor.Library.Tests/DataParserLoggerTests

This class contains tests to make sure the logger is called appropriately.

Here is a sample test:

```
[Fact]
public void ParseData_WithBadRecord_LoggerIsCalledOnce()
{
    // Arrange
    Mock<ILogger> mockLogger = new();
    DataParser parser = new(mockLogger.Object);

    // Act
    parser.ParseData(TestData.BadRecord);

    // Assert
    mockLogger.Verify(m =>
        m.LogMessage(It.IsAny<string>(), TestData.BadRecord[0]),
        Times.Once());
}
```

This test is working, but it needs some fixing to make sure it stays that way.

DataProcessor.Library.Tests/DataParserTests

This class contains tests to make sure that good records are returned and bad records are not. These tests have not been implemented.

Sample:

```
[Fact]
public void ParseData_WithMixedData_ReturnsGoodRecords()
{
    Assert.Fail("Test not implemented");
}
```

Hints

In DataParserLoggerTests.cs:

- Update the tests so that they `await` the call to the ParseData method.

In FakeLogger.cs

- Implement the `ILogger` interface.
- This does not need to do anything, but you should avoid returning "null".

In DataParserTests.cs:

- Implement the unit tests based on the method name.
- When calling the `DataParser.ParseData` method, use the static fields in the `TestData` class. Example:

```
parser.ParseData(TestData.GoodRecord)
```

- For "Mixed" data, use the TestData.Data field.

If you want more assistance, step-by-step instructions are included below. Otherwise, if you'd like to challenge yourself, **STOP READING NOW**

Updating Existing Tests: Step-By-Step

1. Open the `DataParserLoggerTests` test class.

This class contains several existing tests that call an asynchronous method (`ParseData`). These tests do not wait for the asynchronous method to complete. While they do currently pass, this is primarily due to luck. If the

`ParseData` method changes, the timing could change, and these tests would no longer pass.

It is best to make sure we await the asynchronous methods to ensure that they finish.

2. Review the first unit test.

```
[Fact]
public void ParseData_WithBadRecord_LoggerIsCalledOnce()
{
    // Arrange
    Mock<ILogger> mockLogger = new();
    DataParser parser = new(mockLogger.Object);

    // Act
    parser.ParseData(TestData.BadRecord);

    // Assert
    mockLogger.Verify(m =>
        m.LogMessage(It.IsAny<string>(), TestData.BadRecord[0]),
        Times.Once());
}
```

Let's do a walkthrough of what this test is doing before updating it.

This test calls the `ParseData` method with 1 bad record. We expect that the logger will be called 1 time for this. This test uses Moq (a mocking framework) to check the behavior.

In the Arrange section, a mock of `ILogger` is created using Moq (a mock is an in-memory implementation of the interface that we can use for testing). Then the `ILogger` part of the mock is passed to the `DataParser` class constructor.

In the Act section, we call the `ParseData` method.

In the Assert, we use the mock object's `Verify` method. In this case, we check to make sure that the `LogMessage` method on the logger is called one time (based on the `Times.Once()` parameter). And that one of the parameters is the bad record.

We won't go into the full details of the Moq framework. If you're interested, a quickstart is available here: https://github.com/devlooped/moq/wiki/Quickstart.

3. Update the first unit test.

Add `await` to the `ParseData` method call.

```
// Act
await parser.ParseData(TestData.BadRecord);
```

Change the signature of the test method from `void` to `async Task`.

```
[Test]
public async Task ParseData_WithBadRecord_LoggerIsCalledOnce()
```

Here's the completed test.

```
[Fact]
public async Task ParseData_WithBadRecord_LoggerIsCalledOnce()
{
    // Arrange
    Mock<ILogger> mockLogger = new();
    DataParser parser = new(mockLogger.Object);

    // Act
    await parser.ParseData(TestData.BadRecord);

    // Assert
    mockLogger.Verify(m =>
        m.LogMessage(It.IsAny<string>(), TestData.BadRecord[0]),
        Times.Once());
}
```

4. Make the same updates to the other three tests.

Change the method signatures and `await` the `ParseData` method in the other three unit tests.

Note: I won't put the completed code here, you can check the "Completed" folder of the lab for that.

5. Re-run the tests.

Re-run the unit tests from the command line or from the Visual Studio Test Explorer (as described at the beginning of the lab). This should show the same results as before.

These 4 tests still pass, but they are more resilient to changes to the code.

Creating a Fake Object with Asynchronous Methods: Step-By-Step

For the other unit tests, we will use a fake object (rather than a mock). As mentioned above, a mock is an in-memory implementation that can be use for testing; a fake is "real" implementation in our code that has fake or placeholder behavior.

Depending in the code and testing scenario, sometimes mocks work well, and sometimes fake objects are preferred. So we will look at fakes here.

1. Review the `FakeLogger` class.

As shown above, the FakeLogger class is currently empty.

```
public class FakeLogger //: ILogger
{
}
```

2. Implement the logger interface.

Remove the comment to specify that FakeLogger implements the `ILogger` interface.

```
public class FakeLogger : ILogger
{

}
```

To implement the interface, click on `ILogger`, then use "Ctrl+." in Visual Studio. This gives the option to "Implement interface". You can also use the Quick Actions lightbulb to get the same option.

This gives us a placeholder for the interface method.

```
public class FakeLogger : ILogger
{
    public Task LogMessage(string message, string data)
    {
        throw new NotImplementedException();
    }
}
```

3. Return a completed task.

We don't need this method to do anything. It is tempting to simply return `null` from this method.

```
public class FakeLogger : ILogger
{
    public Task LogMessage(string message, string data)
    {
        // AVOID THIS
        return null;
    }
}
```

Caution
It's tempting to return `null` from this method, particularly since the logger intentionally does nothing. But this is a bad practice can cause issues for callers.

We could create a new `Task` with a `TaskFactory`, but a better solution is to use the static `CompletedTask` property on the `Task` type.

Note: If you completed lab01, then this should look familiar.

```
public class FakeLogger : ILogger
{
    public Task LogMessage(string message, string data)
    {
        return Task.CompletedTask;
    }
}
```

This completes the `FakeLogger` class.

A couple of questions:

- Q: The `NullLogger` class in the library does the same thing. Isn't this duplicated code?
  A: While it is technically duplicated code, this logger belongs to the tests. It can change independently of whatever is in the library.
- Q: What if the async method needs to return a value?
  A: For this we can use the `FromResult` method on Task.

```
return Task.FromResult<string>("Test String");
```

Now that we have the fake object in place, we can test the `ParseData` method.

Writing Async Test Methods: Step-by-Step

1. Review the DataParserTests class.

The DataParserTests class contains 5 unit tests that have placeholder code.

```
[Fact]
public void ParseData_WithMixedData_ReturnsGoodRecords()
{
    Assert.Fail("Test not implemented");
}
```

2. Review the first test.

These tests use a 3 part naming scheme:

- Unit under test
- State being tested
- Expected result

```
[Fact]
public void ParseData_WithMixedData_ReturnsGoodRecords()
{
    Assert.Fail("Test not implemented");
}
```

For this test the unit under test is the "ParseData" method. The state being tested is that the method is called with both good and bad records ("Mixed Data"). The expected result is that the good records are returned.

3. "Arrange" the first test.

We need an instance of the `DataParser` class to run tests against. So, we will create an instance of the `FakeLogger` class and pass it to the `DataParser` constructor.

```
// Arrange
FakeLogger logger = new();
DataParser parser = new(logger);
```

4. "Act" in the first test.

For the "Act" section, we will call the `DataParser` method with `TestData.Data` (this is test data set that contains both good and bad records).

Since the `DataParser` method is asynchronous, we will want to `await` it so that we can get the result back (a read-only collection of Person objects).

```
// Act
var records = await parser.ParseData(TestData.Data);
```

In addition, we need to update the signature of the test method from "void" to "async Task".

```
public async Task ParseData_WithMixedData_ReturnsGoodRecords()
```

5. "Assert" the results.

For this we will remove the `Assert.Fail` to change it to something more useful.

In this case, the test data has 14 good records (mixed in wit 4 bad records). For simplicity, we will count the number of items returned.

```
// Assert
Assert.Equal(14, records.Count);
```

We could go into a deeper comparison, but this is what we need as an initial sanity check.

Here's the completed test:

```
[Fact]
public async Task ParseData_WithMixedData_ReturnsGoodRecords()
{
    // Arrange
    FakeLogger logger = new();
    DataParser parser = new(logger);

    // Act
    var records = await parser.ParseData(TestData.Data);

    // Assert
    Assert.Equal(14, records.Count);
}
```

6. Re-run the tests.

Re-run the unit tests. This time, you will see that one of the previously "failed" tests is now passing.

7. Implement the second test.

The second test is called "ParseData_WithGoodRecord_ReturnsOneRecord". Based on our naming, if we call the `ParseData` method with one good record (using the `GoodRecord` in the test data), we should get 1 record returned (meaning it was parsed successfully).

The steps are similar to the first test. Here's the completed test:

```
[Fact]
public async Task ParseData_WithGoodRecord_ReturnsOneRecord()
{
    FakeLogger logger = new();
    DataParser parser = new(logger);
```

```
    var records = await parser.ParseData(TestData.GoodRecord);

    Assert.Single(records);
}
```

Note: `Assert.Single` is used to check that there is only one item in the `records` collection.

Re-run the test. This test should now pass.

8. Create a factory method.

Since the "Arrange" section is the same for both of these tests (and all of the following), I like to create factory methods to get me an object I can use for testing.

In this case, a method called `GetParserWithFakeLogger` can give me a configured DataParser object.

```
private DataParser GetParserWithFakeLogger()
{
    FakeLogger logger = new();
    DataParser parser = new(logger);
    return parser;
}
```

Note: If you want to get fancy, we can do this same thing as an expression:

```
private DataParser GetParserWithFakeLogger() => new DataParser(new FakeLogger());
```

Probably don't do this, but if you want to get really fancy, we can use the implicit "new" for the DataParser. (I don't recommend this since it makes things pretty hard to read.):

```
private DataParser GetParserWithFakeLogger() => new(new FakeLogger());
```

Then the tests can be updated as follows:

```
[Fact]
public async Task ParseData_WithGoodRecord_ReturnsOneRecord()
{
    DataParser parser = GetParserWithFakeLogger();

    var records = await parser.ParseData(TestData.GoodRecord);
```

```
    Assert.Single(records);
}
```

Note: I prefer to use factory methods over setup methods in order to keep things obvious. Sometimes we lose track of what setup methods are doing. For more information, see this article: Unit Testing: Setup Methods or Not?

9. Update the other three tests.

The remaining tests are similar to the second test. The only difference is that they expect to have 0 records returned instead of 1 record. You can use `Assert.Empty(records)` to check for an empty collection.

Note: I won't put the completed code here, you can check the "Completed" folder of the lab for that.

10. Re-run the tests.

With all of the tests completed, there should now be 9 passing tests.

```
[xUnit.net 00:00:00.00] xUnit.net VSTest Adapter v2.8.2+699d445a1a (64-bit .NET
9.0.5)
[xUnit.net 00:00:00.62]   Discovering: DataProcessor.Library.Tests
[xUnit.net 00:00:00.65]   Discovered:  DataProcessor.Library.Tests
[xUnit.net 00:00:00.65]   Starting:    DataProcessor.Library.Tests
[xUnit.net 00:00:01.23]   Finished:    DataProcessor.Library.Tests
  DataProcessor.Library.Tests test succeeded (3.6s)

Test summary: total: 9, failed: 0, succeeded: 9, skipped: 0, duration: 3.6s
Build succeeded in 6.3s
```

Conclusion

So this has shown us how to write tests that call asynchronous methods and how to create fake objects that have asynchronous methods. These techniques can be extended to create mock objects that can be used for testing.

End of Lab BONUS - Unit Testing Asynchronous Methods
