Lab BONUS - Working with AggregateException

Objectives

This lab shows some techniques to work with AggregateExceptions thrown by Task, including getting partial exceptions with "await" and using a continuation to get the full exception. Also included, running multiple tasks concurrently and getting results.

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 retrieves an Order object using a set of services. Since we are dealing with exception handling, these services don't actually exist. (That's where the exceptions will come from.)

Start by running the console application.

Visual Studio Code
For Visual Studio Code, we'll run the application from the command line. (This is the most consistent experience.)

One way to do this in Windows 11 is to open the "ProductOrder.Viewer" 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:\Lab03\Starter\ProductOrder.Viewer>
```

From here, type "dotnet run" to run the application.

```
PS C:\Lab03\Starter\ProductOrder.Viewer> dotnet run
```

You should get output similar to the following:

```
Getting Order ID: 8
---
```

```
Exception Type:
    System.NotImplementedException
Exception Message:
    This has not been implemented yet
Inner Exception:
    NONE
---

Press any key to continue...
```

Visual Studio 2022
If you are using Visual Studio 2022, run the application in the debugger (using F5, the toolbar button, or "Debug->Start Debugging" from the menu).

A console window will open with the same output as above:

```
Getting Order ID: 8
---
Exception Type:
    System.NotImplementedException
Exception Message:
    This has not been implemented yet
Inner Exception:
    NONE
---

Press any key to continue...
```

Lab Goals

We have several goals:

1. Add functionality to the GetOrderAsync method. For this, we will make several concurrent service calls.
2. Await a Task that throws an AggregateException to see how it is automatically unwrapped to show the first exception.
3. Write a continuation so that we can see the all of the exceptions that are contained in the AggregateException.

Relevant Files/Classes

ProductOrder.Library/Order.cs
This contains the "Order" class as well as the "OrderReader" class.

The "OrderReader" class has a "GetOrderAsync" method that assembles the order. This is the primary method that we will be working in.

```
public Task<Order> GetOrderAsync(int orderId)
{
    var order = new Order();
    try
    {
        throw new NotImplementedException("This has not been implemented yet");
    }
    catch (Exception ex)
    {
        logger.LogException(ex);
    }

    return Task.FromResult<Order>(order);
}
```

We will fill in the code inside the "try" block. (See the "Hints" below for more details.)

This class also has a "GetOrderDetailsAsync" method that makes a service call to get several of the order fields.

ProductOrder.Library/Customer.cs
This file contains the "Customer" and "CustomerReader" classes. The OrderReader (above) will use the "CustomerReader" class to get the customer on the order.

ProductOrder.Library/Product.cs
This file contains the "Product" and "ProductReader" classes. The OrderReader (above) will use the "ProductReader" class to get the products on the order.

ProductOrder.Library/ConsoleExceptionLoggers.cs
This class logs exceptions to the console. It already knows how to display different types of exceptions. This will let us see different outputs as we go.

ProductOrder.Viewer/Program.cs
This is the console application. Here is the "Main" method:

```
static async Task Main(string[] args)
{
    int orderId = 8;

    Console.Clear();
    Console.WriteLine($"Getting Order ID: {orderId}");

    var logger = new ConsoleExceptionLogger();
    var orderReader = new OrderReader(logger);
```

```
    var order = await orderReader.GetOrderAsync(orderId);
    if (order.Id == orderId)
    {
        OutputOrder(order);
    }

    Console.WriteLine();
    Console.WriteLine("Press any key to continue...");
    Console.ReadKey();
}
```

We won't make any changes to this method, but you can see that it uses the OrderReader.

Hints

All of the code changes are in the "Order.cs" file, specifically in the "GetOrderAsync" method.

- To create the order, you will need data from the following methods:
    - OrderReader.GetOrderDetailsAsync()
    - Customer.GetCustomerForOrderAsync()
    - Product.GetProductsForOrderAsync()
- Rather than awaiting each method individually, run them and grab on to the resulting task.
- Use "Task.WhenAll" to see when all tasks have completed.
- Use the task results to create the order.

Additional Information

- With the method above, each Task will throw it's own exception. The task returned by "Task.WhenAll" has an AggregateException with all three exceptions.
- When using "await" with "Task.WhenAll", only the first of the inner exceptions is shown.
- By using a continuation on the "Task.WhenAll", you can get to the AggregateException directly and pass it to the logger.

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

Building the Order: Step-By-Step

1. Open the "Order.cs" file and review the "GetOrderAsync" method.

```
public Task<Order> GetOrderAsync(int orderId)
{
```

```
    var order = new Order();
    try
    {
        throw new NotImplementedException("This has not been implemented yet");
    }
    catch (Exception ex)
    {
        logger.LogException(ex);
    }

    return Task.FromResult<Order>(order);
}
```

We want to replace the code in the "try" block with the functionality to build the order object.

2. Review the Order class (in the same file).

The Order class has the following members:

```
public class Order
{
    public int Id { get; set; }
    public DateTime DateOrdered { get; set; }
    public DateTime DateFulfilled { get; set; }
    public Customer? Customer { get; set; }
    public List<Product>? Products { get; set; }
}
```

To fill in these properties, we can use the following methods:

- OrderReader.GetOrderDetailsAsync for
  - Id
  - DateOrdered
  - DateFulfilled
- Customer.GetCustomerForOrderAsync() for
  - Customer
- Product.GetProductsForOrderAsync() for
  - Products

3. Call the methods listed above and hang on to the tasks.

Inside the "try" block of the "GetOrderAsync" method, call the various methods required for the order and save each Task in a local variable.

Note: The OrderReader class has fields for the CustomerReader and the OrderReader, so we can use those to make our calls.

```
public Task<Order> GetOrderAsync(int orderId)
{
    var order = new Order();
    try
    {
        Task<Order> orderDetailsTask =
            GetOrderDetailsAsync(orderId);
        Task<Customer> customerTask =
            customerReader.GetCustomerForOrderAsync(orderId);
        Task<List<Product>> productTask =
            productReader.GetProductsForOrderAsync(orderId);
    }
    catch (Exception ex)
    {
        logger.LogException(ex);
    }

    return Task.FromResult<Order>(order);
}
```

4. Wait for all three tasks to complete.

Use "await Task.WhenAll" with the three tasks as the parameter. This will wait for all three tasks to complete. (This is still inside the "try" block.)

```
await Task.WhenAll(orderDetailsTask, customerTask, productTask)
    .ConfigureAwait(false);
```

Note: Also use "ConfigureAwait(false)" since we are in a class library.

We need to do a couple more things to finish this up. Add the "async" modifier to the method signature (if it isn't there already).

```
public async Task<Order> GetOrderAsync(int orderId)
```

Then change the return type from

```
return Task.FromResult<Order>(order);
```

to

```
return order;
```

We need to change the return value because of the nature of "await". When nothing in an asynchronous method is awaited, we need to explicitly return a Task (in this case, by using "Task.FromResult"). When something *is* awaited within a method, any return value is automatically wrapped in a Task. Because of this, we just return the value ("order") and rely on the compiler to take care of the rest.

5. Use the task results to complete the order object.

Since we awaited a "WhenAll", we know that all of the tasks are complete. This means that we can safely access the "Result" property on each task to fill in our order. (This is inside the "try" block.)

```
order = orderDetailsTask.Result;
order.Customer = customerTask.Result;
order.Products = productTask.Result;
```

Here's the completed method:

```
public async Task<Order> GetOrderAsync(int orderId)
{
    var order = new Order();
    try
    {
        Task<Order> orderDetailsTask =
            GetOrderDetailsAsync(orderId);
        Task<Customer> customerTask =
            customerReader.GetCustomerForOrderAsync(orderId);
        Task<List<Product>> productTask =
            productReader.GetProductsForOrderAsync(orderId);

        await Task.WhenAll(orderDetailsTask, customerTask, productTask)
            .ConfigureAwait(false);

        order = orderDetailsTask.Result;
        order.Customer = customerTask.Result;
        order.Products = productTask.Result;
    }
    catch (Exception ex)
    {
        logger.LogException(ex);
    }

    return order;
}
```

Awaiting the AggregateException

"Task.WhenAll" waits for a set of tasks to complete and returns another task. If any of the tasks throw an exception, then the returned task is "faulted" (meaning, it is marked as having generated an exception).

"Task.WhenAll" wraps the thrown exceptions in an AggregateException. This AggregateException has a collection of inner exceptions with all of the exceptions that are thrown. In our case, all three tasks throw an exception, so the AggregateException will contain three inner exceptions.

When we "await" a faulted task, it automatically unwraps the AggregateException and throws the first inner exception that it finds.

In the "catch" block, this unwrapped exception is logged to the console.

We can see this by running the application.

1. Run the application from inside Visual Studio (or by using "dotnet run" from the command line).

The output is as follows:

```
Getting Order ID: 8
---
Exception Type:
    System.Net.Http.HttpRequestException
Exception Message:
    No connection could be made because the target machine actively refused it.
Inner Exception:
    System.Net.Sockets.SocketException
    No connection could be made because the target machine actively refused it.
---

Press any key to continue...
```

This shows a single exception: an HttpRequestException. This is thrown because there are no services running for this application to connect to.

This is often the enough information for us to handle the error (and this is why "await" does this).

But we can get to all of the exceptions by adding a continuation to the task that is returned by "WhenAll".

Adding a Continuation to Get All of the Exceptions: Step-By-Step

To get around the unwrapping of the AggregateException that "await" does, we will add a continuation to the task returned by "Task.WhenAll".

1. Add a call to ".ContinueWith" and pass the task exception to the logger.

```
await Task.WhenAll(orderDetailsTask, customerTask, productTask)
    .ContinueWith(task =>
        {
            logger.LogException(task.Exception);
        })
    .ConfigureAwait(false);
```

The "task.Exception" inside the continuation is the full AggregateException. The logger knows how to look at the inner exceptions for an AggregateException.

2. Restrict the continuation to only run when the task is faulted.

Add "TaskContinuationsOptions.OnlyOnFaulted" as a parameter to the "ContinueWith" method.

```
await Task.WhenAll(orderDetailsTask, customerTask, productTask)
    .ContinueWith(task =>
        {
            logger.LogException(task.Exception!);
        }, TaskContinuationOptions.OnlyOnFaulted)
    .ConfigureAwait(false);
```

This continuation will only run if the task is faulted. This is important because if the task is not faulted, then the "task.Exception" property would be null.

TaskContinuationOptions let us restrict when a continuation will run. They are really useful. Here are some of the options I've found useful:

- OnlyOnFaulted
- OnlyOnCanceled
- OnlyOnRanToCompletion (i.e., success)
- NotOnFaulted
- NotOnCanceled
- NotOnRanToCompletion

> *Note: There is also an exclamation point after "task.Exception". The compiler thinks that "task.Exception" may be null (which is why there are green squiggles under it). But since we are specifying "OnlyOnFaulted", we know that Exception will not be null. When we add the "!", we let the compiler know that this property will not be null, and the warning will be hidden.*

3. Re-run the application.

When we run the application, the output will be a bit different. I've truncated some of the results for this display:

```
Getting Order ID: 8
---
Exception Type:
    System.AggregateException
Exception Message:
    One or more errors occurred. (...)
Inner Exception:
    System.Net.Http.HttpRequestException
    No connection could be made...
Inner Exceptions Count: 3

InnerExceptions[0]:
    System.Net.Http.HttpRequestException
    No connection could be made...

InnerExceptions[1]:
    System.Net.Http.HttpRequestException
    No connection could be made...

InnerExceptions[2]:
    System.Net.Http.HttpRequestException
    No connection could be made...
---
---
Exception Type:
    System.AggregateException
Exception Message:
    One or more errors occurred. (...)
Inner Exception:
    System.Net.Http.HttpRequestException
    No connection could be made ...
Inner Exceptions Count: 1

InnerExceptions[0]:
    System.Net.Http.HttpRequestException
    No connection could be made ...
---

Press any key to continue...
```

We have a little bit of a problem here. The first Exception block (the one between the first set of "---" markers) is fine. This shows an AggregateException that has three inner exceptions, and each of those inner exceptions is displayed.

But the second Exception block shows another AggregateException. This is coming from the "orderDetailsTask".

Here's the relevant line:

```
order = orderDetailsTask.Result;
```

The "orderDetailsTask" is faulted (meaning, it threw an exception). If you try to access the "Result" property on a faulted task, an AggregateException is thrown that contains the appropriate inner exception(s).

5. Set up a guard clause to prevent use of a faulted task.

To prevent using faulted tasks, we can use the "task.IsCompletedSuccessfully" property to put a guard clause before the order assignments.

```
if (orderDetailsTask.IsCompletedSuccessfully &&
    customerTask.IsCompletedSuccessfully &&
    productTask.IsCompletedSuccessfully)
{
    order = orderDetailsTask.Result;
    order.Customer = customerTask.Result;
    order.Products = productTask.Result;
}
```

This will ensure that the order is only updated if *all* of the tasks complete successfully.

6. Re-run the application.

When we run the application, we now see the complete AggregateException (and just one AggregateException).

```
Getting Order ID: 8
---
Exception Type:
    System.AggregateException
Exception Message:
    One or more errors occurred. (...)
Inner Exception:
    System.Net.Http.HttpRequestException
    No connection could be made ...
Inner Exceptions Count: 3

InnerExceptions[0]:
    System.Net.Http.HttpRequestException
    No connection could be made ...

InnerExceptions[1]:
    System.Net.Http.HttpRequestException
    No connection could be made ...

InnerExceptions[2]:
```

```
    System.Net.Http.HttpRequestException
    No connection could be made ...
---

Press any key to continue...
```

Conclusion

So this has shown us that we can run multiple tasks concurrently. This is useful if we need to assemble an object from multiple services.

"Task.WhenAll" lets us know when all of the tasks are complete. We can use a continuation to get the complete AggregateException that otherwise gets truncated if we directly "await" it.

End of Lab BONUS - Working with AggregateException
