Async and Await

Estimated time for completion: 45 minutes

Overview:

In this lab you will learn how to make use of the new **async/await** keywords along with the other enhancements to asynchronous programming that have been included in .NET 4.5

Goals:

- Learn how to use async/await
- Learn how to use Task.Run to start a task
- Learn how to implement standard progress behaviour
- Learn how to build an asynchronous API
- Learn how to use Task.WhenAny

Lab Notes:

This lab requires Visual Studio 2012

Part 1: Async and Await

For this lab you will be updating a route planner application to use asynchronous processing. Currently the application can get noticeably blocked if the number of way points exceeds eight.

1. Open the project **Before\AsyncAndAwait.**sln in Visual Studio 2012. Compile and run the project. Select a start location, end location and at least two way points, and hit the calculate route button. You will see the ideal route displayed below.
2. Now increase the number of waypoints to nine or ten, and hit the calculate route button. You will see that it takes a noticeable amount of time to produce the result, and the user interface is unresponsive until the route is produced.
3. The goal is to make the user interface responsive whilst the calculation is in progress. There are two files in the source code that you will be working in, **RoutePlanner\ViewModels\RoutePlannerViewModel.cs** and **Geographical\TripCalculator.cs**. The **RoutePlannerViewModel.cs** contains the UI logic, and the **TripCalculator.cs** contains the code to calculate a route. Find the method **InternalCalculateRoute** inside the **RoutePlannerViewModel.cs** file, this is the method that is called when the user hits the calculate route button. This then further delegates to the **CalculateShortestRoute** method of the trip calculator to calculate the shortest route, it is this method that is causing the UI to freeze, as this method can take a long time to complete. This method therefore needs to be turned into an asynchronous method to allow the UI code to take advantage of async/await.

4. Locate the **CalculateShortestRoute** method, and rename it **CalculateShortestRouteAsync** and you are done.(;-), just kidding) This method now needs to return a Task<Trip> as opposed to just a trip
5. Modify the body of the method to create a Task<Trip> that calculates the route in the background, and return this task. Use **Task.Run** to create the background task.

```
public Task<Trip> CalculateShortestRouteAsync(MapLocation start,
             MapLocation end, MapLocation[] wayPoints)
{
 Trip trip = new Trip(start, end, wayPoints);

 return Task.Run(() => SequentialShortestPath(trip));
}
```

6. The code in the main should compile, but you will get an error in **RoutePlannerViewModel.cs** as we are still attempting to call the synchronous version.
7. Now modify the method **InternalCalculateRoute** to call the asynchronous version, utilising async and await.
8. Now re-run the application and you should see it is now responsive even when calculating routes with more than eight way points.

```
private async void InternalCalculateRoute(object o)
 {
   Route = null;
   IsCalculating = true;
   Progress = 0;
   ShortestDistanceSoFar = 0;

   Route = await tripCalculator
               .CalculateShortestRouteAsync(Start, End,
                                    PlacesToVisit.ToArray());

   IsCalculating = false;
}
```

Solution: After Part 1\AsyncAndAwait.sln

Part 2: Progress and Cancellation

In this part of the lab you will extend your asynchronous api to include progress and cancellation, to allow both feedback to the user and the ability to cancel a long running route calculation.

1. Now that you have a responsive UI even when the calculation is in progress you have the opportunity to add cancel functionality. .NET 4.0 introduced the type **CancellationTokenSource** as a standard way to offer cancellation behaviour to your asynchronous operation. The cancellation button is already present in the UI, all you

need to do is create a field inside the **RoutePlannerViewModel** class of this type, and when a new calculation is invoked create an instance of the cancellation token source.

2. Now modify the **InternalCancel** method inside the view model to signal cancellation.

```
private CancellationTokenSource CtsRouteCalculation;

private void InternalCancel(object o)
{
 CtsRouteCalculation.Cancel();
}
```

3. Issuing the cancellation has no effect unless there is code actively checking for cancellation, you will now need to modify your asynchronous calculation method to accept a **CancellationToken**. Modify the caller to pass the cancellation token, obtained via the **Token** property on the **CancellationTokenSource**.

```
   CtsRouteCalculation = new CancellationTokenSource();

   Route = await tripCalculator
               .CalculateShortestRouteAsync(Start, End,
                                            PlacesToVisit.ToArray(),
                                            CtsRouteCalculation.Token);
```

4. Modify the **TripCalculator.CalculateShortestRouteAsync** to flow the cancellation token into the **SequentialShortestPath** method. Modify the **SequentialShortestPath** method to check to see if cancellation has been requested, and throw an OperationCanceledException if it has, in other words call **ThrowIfCancellationRequested** on the cancellation token.

```
private static Trip SequentialShortestPath(Trip trip,CancellationToken ct)
{
   . . .
   foreach (MapLocation[] wayPoints in
                                        trip.WayPoints
                                       .ToArray()
                                       .HeapPermute())
   {
                  . . .

     ct.ThrowIfCancellationRequested();
   }

 return shortestTrip;
}
```

5. Run the code, cancellation will work to a degree but you will be left with an unhandled exception.
6. A cancellation exception is clearly a recoverable exception so wrap the await statement with a try/catch block, to catch and move on if an **OperationCanceledException** is thrown.

7. Re-run the code, to verify cancellation allows the application to continue running, post cancellation.
8. You will now have noticed that there is a progress bar visible whilst the calculation is in progress your task is it to populate this progress bar. The progress bar will be showing not just % progress but also showing the shortest calculated distance so far. In order to communicate progress between the trip calculator and the view model, you will need to create a class that contains both of these pieces of information.

```
public class RouteCalculationProgress
{
  public int Progress { get; private set; }
  public double ShortestRouteSoFar { get; private set; }

  public RouteCalculationProgress(int progress,
                                  double shortestRouteSoFar)
 {
   Progress = progress;
   ShortestRouteSoFar = shortestRouteSoFar;
 }
}
```

9. The standard way of reporting progress is to take advantage of the new interface created in .NET 4.5 called **IProgress<T>** this interface simply has a **Report** method on it that takes a value of type T, this is invoked by the code wanting to report the progress. Extend your asynchronous **CalculateShortestRouteAsync** method to now take a parameter of type **IProgress**<RouteCalculationProgress>, and flow it into the **SequentialShortestPath** method.
10. Modify the body of the method to invoke the **Report** method for each iteration of the loop. The **numberOfPermutations** variable has been included to help you work out % progress.

```
long numberOfPermutations = trip.WayPoints.Select((wp, i) => i +
   1).Aggregate((total, next) => total *= next);

long nPermutation = 0;
foreach (MapLocation[] wayPoints in trip.WayPoints.ToArray().HeapPermute())
{
    . . .
   int progress = (int) ((decimal) nPermutation/(decimal)
   numberOfPermutations*100.0m);

    progressObserver.Report(
                    new RouteCalculationProgress(progress,
                                        shortestTrip.TotalDistance));

  ct.ThrowIfCancellationRequested();
}
```

11. Now update the view model so that it supplies a progress observer. To do this we could create a new class that implements IProgress<T> but instead make use of the

Progress<T> adapter class that takes an Action<T> delegate, and it is that delegate that will actually receive the progress update.

```
Route = await tripCalculator
               .CalculateShortestRouteAsync(Start, End,
                PlacesToVisit.ToArray(),

                CtsRouteCalculation.Token,
                new Progress<RouteCalculationProgress>(
                 prg =>
                 {
                   Progress = prg.Progress;
                   ShortestDistanceSoFar = prg.ShortestRouteSoFar;
                  }));
```

12. You will have noticed that when you run the application and attempt to calculate a route that the application locks up, seems like you are back to square one. Why?
13. The reason is you are performing too many progress updates, each update requires posting an update onto the UI thread, and thus swamping it with updates. It makes sense to perhaps only update progress every whole % that way we guarantee to only perform 100 updates which the UI thread can easily handle. Modify the calculation method to do so.

```
int previousProgress = 0;
foreach (MapLocation[] wayPoints in trip.WayPoints.ToArray().HeapPermute())
{
   . . .
 if (previousProgress != progress)
 {
   progressObserver.Report(
      new RouteCalculationProgress(progress, shortestTrip.TotalDistance));
   previousProgress = progress;
 }
}
```

14. Now rerun the application and you will see a progress bar and an application that is still responsive.

Solution: After Part 2\ AsyncAndAwait.sln

Part 3: Task.WhenAny

In this part you will introduce another algorithm to calculate the shortest route, and execute them both at the same time, and present the user with the route produced by the quickest algorithm.

1. You may have noticed that once we get to trips that have in excess of 10 way points you are waiting a rather long time, we need to fix this as users aren't going to wait days to get a result. In this part of the lab you will deploy two algorithms to calculate the shortest route and whichever produces the answer first wins. The second algorithm is guaranteed to produce some answer in 5 seconds not necessarily the best but some valid route, this

algorithm is currently implemented on the **TripCalculator** type called **ImFeelingLucky**. It's called in the same way as the other algorithm.

2. Your task is to now invoke both tasks from inside the **InternalCalculateRoute** method and whichever comes back first wins. To do this you need to call both methods but don't **await** on them individually, utilise **Task.WhenAny** to produce the task you await on, providing it with the two tasks you previously created. Be careful the **await** on a **Task.WhenAn**y returns the task that completed not the result of the task. Set the Route property to the result of the first task to complete.

```
Task<Trip> completedTask = await Task.WhenAny(new[] {
                                    bruteForceTask, imFeelingLuckTask});
Route = completedTask.Result;
```

3. Once you have the first result, you obviously don't need the other task to continue, so issue a cancellation.
4. Re-run the application you should always now get a result in five seconds, whether it's good will depend on how lucky you are!

Solution: After Part 3\AsyncAndAwait.sln