Getting Started with Async / Await

By Jon Goldberger • March 13, 2017

Jon
Goldberger

Async programming is all the rage in mobile app development for good reasons. Using async methods for long running tasks, like downloading data, helps keep your user interface responsive, while not using async methods, or the improper use of `async / await`, can cause your app's UI to stop responding to user input until the long running task completes. This can result in a poor user experience, which can then lead to poor reviews on the app stores, which is never good for business.

Today we'll take a look at the use of async and how to utilize it to prevent jerky and unexpected behaviors in a `ListView`.

What is async/await?

The `async` and `await` keywords were introduced in .NET 4.5 to make calling async methods easier and to make your async code more easily readable. The `async / await` API is syntactic sugar that uses the TPL (Task Parallel Library) behind the scenes. If you wanted to start a new task and have code run on the UI thread after the task completes prior .NET 4.5, your code would have looked something like this:

```
// Start a new task (this launches a new thread)
Task.Factory.StartNew (() => {
    // Do some work on a background thread, allowing the UI to remain responsive
    DoSomething();
// When the background work is done, continue with this code block
}).ContinueWith (task => {
    DoSomethingOnTheUIThread();
// the following forces the code in the ContinueWith block to be run on the
// calling thread, often the Main/UI thread.
}, TaskScheduler.FromCurrentSynchronizationContext ());
```

That's not very pretty. Using `async / await`, the above becomes:

```
await DoSomething();
DoSomethingOnTheUIThread();
```

The above code gets compiled behind the scenes to the same TPL code as it does in the first example, so as noted, this is just syntactic sugar, and how sweet it is!

Using Async: Pitfalls

In reading about using async/await, you may have seen the phrase "async all the way" thrown around, but what does that really mean? Simply put, it means that any method that calls an async method (i.e. a method that has the `async` keyword in its signature) should use the `await` keyword when calling the async method. Not using the `await` keyword when calling an async method can result in exceptions that are thrown being swallowed by the runtime, which can cause issues that are difficult to track down. Using the `await` keyword requires that the calling method also use the `async` keyword in its signature. For example:

```
async Task CallingMethod()
{
    var x = await MyMethodAsync();
}
```

This poses a problem if you want to call an async method using the `await` keyword when you can't use the `async` modifier on the calling method, for instance if the calling method is a method whose signature can't use the `async` keyword or is a constructor or a method that the OS calls, such as `GetView` in an Android `ArrayAdapter` or `GetCell` in an iOS `UITableViewDataSource`. For example:

```
public override View GetView(int position, View convertView, ViewGroup parent)
{
    // Can't use await keyword in this method as you can't use async keyword
    // in method signature due to incompatible return type.
}
```

As you may know, an async method has to return either `void`, `Task`, or `Task<T>`, and returning `void` should only be used when making an event handler async. In the case of the `GetView` method noted above, you need to return an Android `View`, which can't be changed to return `Task<View>` as the OS method that calls it obviously does not use the `await`

keyword and so can't handle a `Task<T>` being returned. Thus you can't add the `async` keyword to the above method and therefore can't use the `await` keyword when calling an async method from the above method.

To get around this, one might be tempted, as I have been in the past, to just call a method from `GetView` (or similar method where the signature can't be changed regardless of the platform) as an intermediate method, and then call the async method from the intermediate method:

```
public override View GetView(int position, View convertView, ViewGroup parent)
{
    IntermediateMethod();
    // more code
}

async Task IntermediateMethod()
{
     await MyMethodAsync();
}
```

The problem here is that `IntermediateMethod` is now an async method and thus should be awaited just like the `MyMethodAsync` method needed to be. So, you have gained nothing here, as `IntermediateMethod` is now async and should be awaited. In addition, the `GetView` method will continue running all of the code after calling `IntermediateMethod()`, which may or may not be desirable. If the code following the call to `IntermediateMethod()` depends on the results of the `IntermediateMethod()`, then it isn't desirable. In such a scenario, you may be tempted to use the `Wait()` method call (or `Result` property) on the async task, e.g.:

```
public override View GetView(int position, View convertView, ViewGroup parent)
{
    IntermediateMethod().Wait();
    // more code
}
```

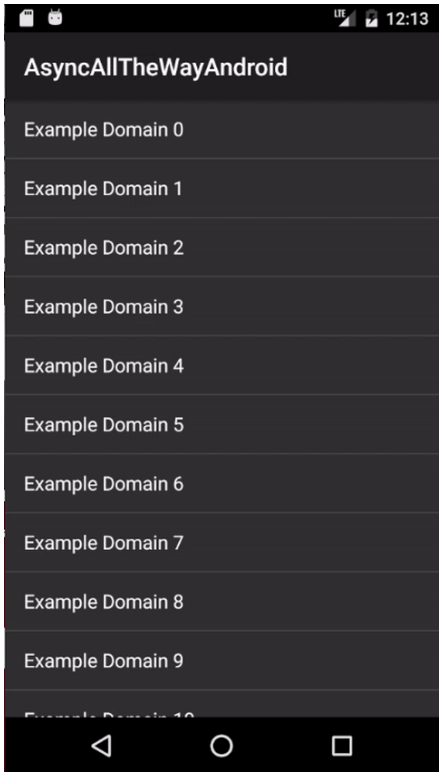
Calling `Wait()` on the async method causes the calling thread to pause until the async method completes. If this is the UI thread, as would be the case here, then your UI will hang while the async task runs. This isn't good, especially in an `ArrayAdapter` that is supplying the data for the rows of a `ListView`. The user will not be able to interact with the list view until the data for all of the rows has been downloaded, and scrolling will likely be jerky and/or completely non-responsive, which isn't a good user experience. There's also a `Result` property you can call on the async task. This would be used if your async task was returning data by

using `Task<T>` as the return type of the async method. This would also cause the calling thread to wait for the result of the async task:

```
public override View GetView(int position, View convertView, ViewGroup parent)
{
    view.Text = IntermediateMethod().Result;
    // more code
}

async Task<string> IntermediateMethod()
{
     return await MyMethodAsync(); // MyMethodAsync also returns Task<string> in th
}
```

In fact doing the above may cause your UI to hang completely and for the `ListView` never to be populated, which is a non-starter. It may also just be jerky:



In general, you should avoid using `Wait()` and `Result`, especially on the UI thread. In the iOS and Android sample projects linked at the end of this blog, you can look in `ViewControllerJerky` and `MainActivityJerky` respectively to see this behavior. Those files are not set to compile in the sample projects.

Using Async All the Way

So how do I get "async all the way" in this scenario?

One way around the above problems is to revert to the old TPL upon which `async / await` is based. You're going to use TPL directly, but only once to start the chain of async method calls (and to start a new thread right away). Somewhere down the line the TPL will be used directly again, as you need to use TPL to start a new thread. You can't start a new thread using only the `async / await` keywords, so some method down the chain will have to launch the new thread with TPL (or another mechanism). The async method that launches a new thread will be a framework method, like a .NET `HttpClient` async method in many, if not most, cases. If not using async framework methods, then some method of yours down the chain will have to launch a new thread and return `Task` or `Task<T>`.

Let's start with an example using `GetView` in an Android project (though the same concept will work for any platform, i.e. Xamarin.iOS, Xamarin.Forms, etc.) Let's say I have a `ListView` that I want to populate with text downloaded from the web dynamically (more likely one would download the whole list of strings first and then populate the list rows with the already downloaded content, but I'm downloading the strings row by row here for demonstration purposes, plus there are occasions where one may want to do it this way anyway). I certainly don't want to block the UI thread waiting for the multiple downloads; rather, I would like the user to be able to start working with the ListView, scroll around, and have the text appear in each `ListView` cell as the text gets downloaded. I also want to make sure that if a cell scrolls out of view, that when it is reused it will cancel loading the text that is in the process of being downloaded and start loading new text for that row instead. We do this with TPL and cancellation tokens. Comments in the code should explain what's being done.

```
public override View GetView(int position, View convertView, ViewGroup parent)
{
    // We will need a CancellationTokenSource so we can cancel the async call
    // if the view moves back on screen while text is already being loaded.
    // Without this, if a view is loading some text, but the view moves off and
    // back on screen, the new load may take less time than the old load and
    // then the old load will overwrite the new text load and the wrong data
    // will be displayed. So we will cancel any async task on a recycled view
    // before loading the new text.

    CancellationTokenSource cts;

    // re-use an existing view, if one is available
    View view = convertView; // re-use an existing view, if one is available

    // Otherwise create a new one
    if (view == null) {
        view = context.LayoutInflater.Inflate(Android.Resource.Layout.SimpleListIte
    }
    else
```

```
    {
        // If view exists, cancel any pending async text loading for this view
        // by calling cts.Cancel();
        var wrapper = view.Tag.JavaCast<Wrapper<CancellationTokenSource>>();
        cts = wrapper.Data;

        // If cancellation has not already been requested, cancel the async task
        if (!cts.IsCancellationRequested)
        {
           cts.Cancel();
        }
    }

    TextView textView = view.FindViewById<TextView>(Android.Resource.Id.Text1);
    textView.Text = "placeholder";

    // Create new CancellationTokenSource for this view's async call
    cts = new CancellationTokenSource();

    // Add it to the Tag property of the view wrapped in a Java.Lang.Object
    view.Tag = new Wrapper<CancellationTokenSource> { Data = cts };

    // Get the cancellation token to pass into the async method
    var ct = cts.Token;

    Task.Run(async () => {
        try
        {
            textView.Text = await GetTextAsync(position, ct);
        }
        catch (System.OperationCanceledException ex)
        {
            Console.WriteLine($"Text load cancelled: {ex.Message}");
        }
        catch (Exception ex)
        {
            Console.WriteLine(ex.Message);
        }
    }, ct);

    return view;
}
```

In a nutshell, the above method checks to see if this is a reused cell and, if so, we cancel the existing async text download if still incomplete. It then loads placeholder text into the cell,

launches the async task to download the correct text for the row, and returns the view with placeholder text right away, thereby populating the `ListView`. This keeps the UI responsive and shows something in the cell while the launched task does its work of getting the correct text from the web. As the text gets downloaded, you'll see the placeholders change to the downloaded text one-by-one (not necessarily in order due to differing download times). I added a random delay to the async task to simulate this behavior since I'm making such a simple, quick request.

Here's the implementation of `GetTextAsync(...)`:

```
async Task<string> GetTextAsync(int position, CancellationToken ct)
{
    // Check to see if task was cancelled; if so throw cancelled exception.
    // Good to check at several points, including just prior to returning the strin
    ct.ThrowIfCancellationRequested();

    // to simulate a task that takes variable amount of time
    await Task.Delay(rand.Next(100,500));
    ct.ThrowIfCancellationRequested();
    if (client == null)
        client = new HttpClient();
    string response = await client.GetStringAsync("http://example.com");
    string stringToDisplayInList = response.Substring(41, 14) + " " + position.ToSt
    ct.ThrowIfCancellationRequested();
    return stringToDisplayInList;
}
```

Note that I can decorate the lambda passed into `Task.Run()` with the `async` keyword, thus allowing me to await the call to my async method, and thereby achieving "async all the way." No more Jerky ListView!

SmoothListView

See it in action

If you want to see the above in action for Xamarin.iOS, Xamarin.Android, and Xamarin.Forms, check it out on my GitHub repo. The iOS version is very similar to the above, the only difference being in how I attach the `CancellationTokenSource` to the cell since there is no `Tag` property as there is in an Android `View`. Xamarin.Forms, however, does not have a direct equivalent to `GetView` or `GetCell` that I'm aware of, so I simulate the same behavior by launching an async task from the main `App` class constructor to get the text for each row.

Happy async coding!

Written on March 13, 2017 by
Jon Goldberger

Keep reading

All Xamarin posts

Simplified App Signing with Secure Files in Visual Studio Team Services

James Montemagno
November 6, 2017

Publish to Azure from Visual Studio for Mac

Cody Beyer
November 2, 2017

Five-Star Apps with Mobile Center Test

Mayur Tendulkar
October 20, 2017