

# Asynchronous Programming For Windows Phone 8

Windows Phone 8 brings a lot of new features to developers and behind the scenes there are also some major improvements. The .NET framework 4.5 and the new C# compiler come with several major new features. Networked applications had to rely - in the past - on callbacks, but with the new asynchronous programming capabilities on the windows phone platform, the compiler can make some magic happen and increase productivity.

## What is asynchronous Programming?



In a console application, with the full .Net framework, to download data from an URL with a WebClient you can do

```
var client = new WebClient();
// Download data.
var textData = client.DownloadString("http://www.myurl.com/myFile.txt");

// Write values.
Debug.WriteLine(textData);
```

In graphical applications - unlike console applications where the whole execution flow can be driven from a single thread, including all I/O - , the execution flow often ends up being event driven. That means you start your logic after an event (click on a button or application loaded) and instead of waiting for something to complete you have to write more event handling to do something after the completion of your action.

With Windows Phone, only the asynchronous version of all I/O operations are available and it forces you to write code like this to accomplish the same task:

```
...
var client = new WebClient();

client.DownloadStringCompleted += new DownloadStringCompletedEventHandler(loadHTMLCallback);
client.DownloadStringAsync(new Uri("http://www.myurl.com/myFile.txt"));
...

public void loadHTMLCallback(Object sender, DownloadStringCompletedEventArgs e)
{
    var textData = (string)e.Result;
    // Do cool stuff with result
    Debug.WriteLine(textData);
}
```

## What is the difference?

Obviously the asynchronous case did increase the complexity and number of lines quite dramatically. We already jumped from one line to 6 to achieve the same result. In the first case, the execution thread has to wait all the required time to download the URL (remember network can be slow) and it holds its resources while it waits. In the second example with the asynchronous case, all the resources can be released after `DownloadStringAsync` and the thread calling this method can continue its execution. The thread that runs the UI code is generally the dispatcher thread and you should let it continue processing other interactions, otherwise your UI will freeze until you finish your method.

## Asynchronous programming in C# 5.0

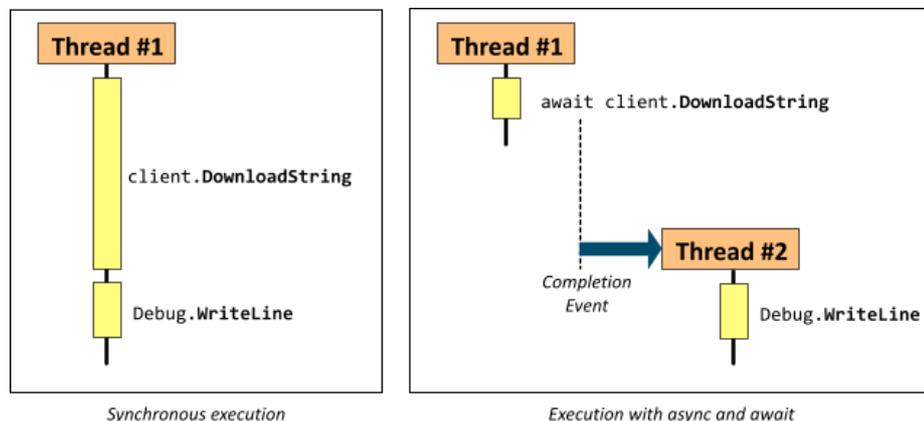
Asynchronous programming for .Net adds two new keywords: `async` and `await`. Inside an asynchronous context (in general this is simply within another asynchronous method), you can call an asynchronous method in a synchronous way by simply prefixing it with `await`. Behind the scenes, instead of holding the execution thread to wait for the result (in most cases a network call), the compiler will create callbacks to process the results (and the caller's thread resources will be released). Essentially `await` is blocking, but only in the current function - from the perspective of the calling function life continues on until it too is forced to await something.

UI events have already an asynchronous context so all you need to do is add `async` to your event handlers to make your methods asynchronous and call other asynchronous events. So in this initial example, if `DownloadString` was asynchronous (I will show later how this can be done), we could write

```
var client = new WebClient();
// Download data.
var textData = await client.DownloadString("http://www.myurl.com/myFile.txt");

// Write values.
Debug.WriteLine(textData);
```

and it would be the exact equivalent of the version with callbacks.



In the synchronous version, the thread #1 has to wait and download the URL (remember network can be slow). In the second example with the asynchronous case, the caller from Thread #1 (calling this method in an asynchronous context) can continue its execution right before `await client.DownloadString`.

In the most common cases, the asynchronous methods are UI events and the caller is the dispatcher. In the first case, the UI would be frozen waiting for the download to complete, while in the second case the Thread #1 can continue executing other UI events, the rest of the code is executed by a new thread (Thread #2) once the download has completed.

In C#, the actual signature of `DownloadString` has to return a `Task<string>` instead of a string, the compiler with the `await` are doing the rest of the magic for you.

With any real world network service, you will have a complex workflow to handle (see [Toodledo syncing](#) for an example) and you are probably looking at writing at least 10 callback methods which does really add to the complexity of code.

## Converting existing methods into to C# asynchronous methods

`TaskCompletionSource<T>` is the easiest way to convert existing code into asynchronous. All you need to do is call `TrySetResult`, `TrySetException` and `TrySetCanceled` within the callback. The following example shows how we could convert the `DownloadStringAsync` into an asynchronous equivalent.

```
public static Task<string> DownloadString(Uri url)
{
    var tcs = new TaskCompletionSource<string>();
    var wc = new WebClient();
    wc.DownloadStringCompleted += (s,e) =>
    {
        if (e.Error != null) tcs.TrySetException(e.Error);
        else if (e.Cancelled) tcs.TrySetCanceled();
        else tcs.TrySetResult(e.Result);
    };
    wc.DownloadStringAsync(url);
    return tcs.Task;
}
```

## Asynchronous code for Windows Phone 7

`async` and `await` are only available out of the box and fully featured with Visual Studio 2012 and Windows Phone 8. If you haven't yet upgraded to VS 2010 and need the Asynchronous features, the [Visual Studio Async CTP](#) adds them to the compiler and visual studio. For Visual Studio 2012 and Windows Phone 7 there is the [Microsoft.Bcl.Async nuget package](#) that lets you use the same keywords with .NET 4.

## The example

The solution attached to this article demonstrates these concepts with a real service: Toodledo. Toodledo is a neat service that manages to do lists with many nice features, its REST API is very similar to many other services. We will demonstrate how to use the asynchronous features on the following scenario:

1. Lookup the toodledo user id using an email
2. Obtain a session token using the user id
3. List the tasks for the user id with the session token

More details on the REST API and the service can be found on <http://api.toodledo.com>

## Toodledo with Windows Phone 7

Instead of writing all the REST calls with the plain `WebClient`, we will use a nice REST library that does all the deserialization work called [RESTSharp](#). With `RESTSharp`, the code for getting the toodledo tasks would look a little like

```
...
var client = new RestClient("http://api.toodledo.com/2");
var request = new RestRequest("account/lookup.php", Method.GET);
client.ExecuteAsync(request, result =>
{
    if (resp.StatusCode == System.Net.HttpStatusCode.OK)
        getTokenId(resp.Value);
});
```

```

...
public void getTokenId(string userAccount)
{
    var client = new RestClient("http://api.toodledo.com/2");
    var request = new RestRequest("account/token.php", Method.GET);
    client.ExecuteAsync(request, result =>
    {
        if (resp.StatusCode == System.Net.HttpStatusCode.OK)
            loadTaskList(resp.Value);
    });
}

```

## Toodledo with asynchronous methods

I am cheating a bit here because for each method the actual REST calls are in the method body but this still demonstrates the potential:

```

var username = await getUsername("email@eail.com")
var token = await getToken(username);
var tasks = await getTasks(token);
...
public async Task<ToodledoToken> getToken(string userId)
{
    var request = new RestRequest("account/token.php", Method.GET);
    return (await ExecuteAsync<ToodledoToken>(request));
}
...

```

## Adding more magic to the application

In a typical application, in addition to these request, you need to manage status message and progress indicators so you need to interleave some GUI code that will run on the dispatcher.

- Get the status and progress bar out of the way

In this example, I used the reactive framework to handle the progress bar and the status text,. You can learn more about it on ... and ...

- Wrap existing asynchronous methods to use the new keywords

Currently many libraries (like REST clients) are not updated yet to work with the new asynchronous framework so it is necessary to transform the callback based methods into the new world that uses Task<T> as return type. Here is how I used the TaskCompletionSource to do wrap a RestRequest into an asynchronous method:

```

private async Task<T> ExecuteAsync<T>(RestRequest request)
{
    var tcs = new TaskCompletionSource<T>();
    _client.ExecuteAsync(request, resp =>
    {
        var value = JsonConvert.DeserializeObject<T>(resp.Content);
        if (value.ErrorCode > 0)
        {
            var ex = new ToodledoException(value.ErrorCode, value.ErrorDesc);
            tcs.SetException(ex);
        }
        else
            tcs.SetResult(value);
    });
    return await tcs.Task;
}

```

## Learn more about Asynchronous Programming

### Official Microsoft documentation

- [Asynchronous Programming with Async and Await](#)
- [Async Performance: Understanding the Costs of Async and Await](#)
- [Control Flow in Async Programs \(C# and Visual Basic\)](#)

### I/O and more

Asynchronous features are not limited to I/O, in [this other article](#) you can learn how to use asynchronous code to simplify UI workflows.

## Download the solution

The zipped solution for this example can be downloaded [Media:WPTodo.zip](#)

Thanks to Hamish Willee for his constructive comments and feedback.

