**Quickstart: Calling asynchronous APIs in C# or Visual Basic**

The Windows Runtime includes many asynchronous APIs to ensure that your app remains responsive when it does work that might take an extended amount of time. This means that your app doesn't need to wait for large operations to complete before continuing execution. For example, an app that downloads info from the Internet might spend several seconds waiting for the info to arrive. If you use a synchronous method to retrieve the info, the app is blocked until the method returns. The app won't respond to user interaction and because it seems non-responsive, the user might become frustrated. By providing asynchronous APIs, the Windows Runtime helps to ensure that your app stays responsive to the user when it's performing long operations.

Most of the asynchronous APIs in the Windows Runtime don't have synchronous counterparts, so you need to be sure to understand how to use the asynchronous APIs with C# or Visual Basic in your Windows Store app. Here we show how to call asynchronous APIs of the Windows Runtime.

**Roadmap:** How does this topic relate to others? See:

* [Roadmap for Windows Store apps using C# or Visual Basic](http://msdn.microsoft.com/en-us/library/windows/apps/br229583.aspx)
* [Roadmap for Windows Store apps using C++](http://msdn.microsoft.com/en-us/library/windows/apps/hh700360.aspx)

**Prerequisites**

See [Create your first Windows Store app using C# or Visual Basic](http://msdn.microsoft.com/en-us/library/windows/apps/hh974581.aspx).

**Using asynchronous APIs**

By convention, asynchronous methods are given names that end in "Async". You typically call asynchronous APIs in response to a user's action, such as when the user clicks a button. Calling an asynchronous method in an event handler is one of the simplest ways of using asynchronous APIs. Here we use the **await** operator as an example.

Suppose that you have an app that lists the titles of blog posts from a certain location. The app has a [**Button**](http://msdn.microsoft.com/en-us/library/windows/apps/windows.ui.xaml.controls.button.aspx) that the user clicks to get the titles. The titles are displayed in a [**TextBlock**](http://msdn.microsoft.com/en-us/library/windows/apps/windows.ui.xaml.controls.textblock.aspx). When the user clicks the button, it is important that the app remains responsive while it waits for the info from the blog's website. To ensure this responsiveness, the Windows Runtime provides an asynchronous method, [**SyndicationClient.RetrieveFeedAsync**](http://msdn.microsoft.com/en-us/library/windows/apps/windows.web.syndication.syndicationclient.retrievefeedasync.aspx), to download the feed.

The example here gets the lists of blog posts from a blog by calling the asynchronous method, [**SyndicationClient.RetrieveFeedAsync**](http://msdn.microsoft.com/en-us/library/windows/apps/windows.web.syndication.syndicationclient.retrievefeedasync.aspx). This is the simplest way to call an asynchronous method.

C#

// Put the keyword, async on the declaration of the event handler.

private async void Button\_Click\_1(object sender, RoutedEventArgs e)

{

Windows.Web.Syndication.SyndicationClient client = new SyndicationClient();

// Force the SyndicationClient to download the information.

client.BypassCacheOnRetrieve = true;

Uri feedUri

= new Uri("http://windowsteamblog.com/windows/b/windowsexperience/atom.aspx");

try

{

// Call SyndicationClient RetrieveFeedAsync to download the list of blog posts.

SyndicationFeed feed = await client.RetrieveFeedAsync(feedUri);

// The rest of this method executes after await RetrieveFeedAsync completes.

rssOutput.Text = feed.Title.Text + Environment.NewLine;

foreach (SyndicationItem item in feed.Items)

{

rssOutput.Text += item.Title.Text + ", " +

item.PublishedDate.ToString() + Environment.NewLine;

}

}

catch (Exception ex)

{

// Log Error.

rssOutput.Text =

"I'm sorry, but I couldn't load the page," +

" possibly due to network problems." +

"Here's the error message I received: "

+ ex.ToString();

}

}

There are a couple of important things about this example. First, the line, SyndicationFeed feed = await client.RetrieveFeedAsync(feedUri) uses the **await** operator with the call to the asynchronous method, [**RetrieveFeedAsync**](http://msdn.microsoft.com/en-us/library/windows/apps/windows.web.syndication.syndicationclient.retrievefeedasync.aspx). You can think of the **await** operator as telling the compiler that you are calling an asynchronous method, which causes the compiler to do some extra work so you don’t have to. Next, the declaration of the event handler includes the keyword **async**. You must include this keyword in the method declaration of any method in which you use the **await** operator. There's nothing about the [**Button.Click**](http://msdn.microsoft.com/en-us/library/windows/apps/windows.ui.xaml.controls.primitives.buttonbase.click.aspx) event handler signature that required the **async** keyword; **async** was added deliberately so that the compiler will permit calling asynchronous methods from the body of that handler. For more info about handling input events and running awaitable methods within them, see [Events and routed events overview](http://msdn.microsoft.com/en-us/library/windows/apps/hh758286.aspx).

In this topic, we won't go into a lot of the details of what the compiler does with the **await** operator, but let's examine what your app does so that it is asynchronous and responsive. Consider what happens when you use synchronous code. For example, suppose that there is a method called SyndicationClient.RetrieveFeed that is synchronous. (There is no such method, but imagine that there is.) If your app included the line SyndicationFeed feed = client.RetrieveFeed(feedUri), instead of SyndicationFeed feed = await client.RetrieveFeedAsync(feedUri), execution of the app stops until the return value of RetrieveFeed is available. And while your app waits for the method to complete, it can't respond to any other events, such another [**Click**](http://msdn.microsoft.com/en-us/library/windows/apps/windows.ui.xaml.controls.primitives.buttonbase.click.aspx) event. That is, your app would be blocked until RetrieveFeed returns.

But if you call client.RetrieveFeedAsync, the method initiates the retrieval and immediately returns. When you use **await** with [**RetrieveFeedAsync**](http://msdn.microsoft.com/en-us/library/windows/apps/windows.web.syndication.syndicationclient.retrievefeedasync.aspx), the app temporarily exits the event handler. Then it can process other events while **RetrieveFeedAsync** executes asynchronously. This keeps the app responsive to the user. When **RetrieveFeedAsync** completes and the [**SyndicationFeed**](http://msdn.microsoft.com/en-us/library/windows/apps/windows.web.syndication.syndicationfeed.aspx) is available, the app essentially reenters the event handler where it left off, after SyndicationFeed feed = await client.RetrieveFeedAsync(feedUri), and finishes the rest of the method.

The nice thing about using the **await** operator is that the code doesn't look much different from how the code looks if you used the imaginary RetrieveFeed method. There are ways to write asynchronous code in C# or Visual Basic without the **await** operator, but the resulting code tends to emphasize the mechanics of executing asynchronously. This makes asynchronous code hard to write, hard to understand, and hard to maintain. By using the **await** operator, you get the benefits of an asynchronous app without making your code complex.

**Return types and results of asynchronous APIs**

If you followed the link to [**RetrieveFeedAsync**](http://msdn.microsoft.com/en-us/library/windows/apps/windows.web.syndication.syndicationclient.retrievefeedasync.aspx), you might have noticed that the return type of **RetrieveFeedAsync** is not a [**SyndicationFeed**](http://msdn.microsoft.com/en-us/library/windows/apps/windows.web.syndication.syndicationfeed.aspx). Instead, the return type is IAsyncOperationWithProgress<SyndicationFeed, RetrievalProgress>. Viewed from the raw syntax, an asynchronous API returns an object that contains the result within it. While it is common, and sometimes useful, to think of an asynchronous method as being awaitable, the **await** operator actually operates on the method’s return value, not on the method. When you apply the **await** operator, you get the result of the asynchronous methods. Specifically, you get the result of calling **GetResult** on the object you see in the raw syntax. In the example, the **SyndicationFeed** is the result of **RetrieveFeedAsync**.

When you use an asynchronous method, you can get info about its result by looking at the return type. All asynchronous APIs in the Windows Runtime return one of the following types:

* [**IAsyncOperation<TResult>**](http://msdn.microsoft.com/en-us/library/windows/apps/br206598.aspx)
* [**IAsyncOperationWithProgress<TResult, TProgress>**](http://msdn.microsoft.com/en-us/library/windows/apps/br206594.aspx)
* [**IAsyncAction**](http://msdn.microsoft.com/en-us/library/windows/apps/windows.foundation.iasyncaction.aspx)
* [**IAsyncActionWithProgress<TProgress>**](http://msdn.microsoft.com/en-us/library/windows/apps/br206581.aspx)

The result type of an asynchronous method is the same as the TResult type parameter. Types without a TResult don't have a result. You can think of the result as being **void**. In Visual Basic, a [Sub](http://msdn.microsoft.com/en-us/library/windows/apps/831f9wka.aspx) procedure is equivalent to a method with a **void** return type

The table here gives examples of asynchronous methods and lists the return type and result type of each.

|  |  |  |
| --- | --- | --- |
| **Asynchronous method** | **Return type** | **Result type** |
| [**SyndicationClient.RetrieveFeedAsync**](http://msdn.microsoft.com/en-us/library/windows/apps/windows.web.syndication.syndicationclient.retrievefeedasync.aspx) | [**IAsyncOperationWithProgress<SyndicationFeed, RetrievalProgress>**](http://msdn.microsoft.com/en-us/library/windows/apps/br206594.aspx) | [**SyndicationFeed**](http://msdn.microsoft.com/en-us/library/windows/apps/windows.web.syndication.syndicationfeed.aspx) |
| [**FileOpenPicker.PickSingleFileAsync**](http://msdn.microsoft.com/en-us/library/windows/apps/br207852.aspx) | [**IAsyncOperation<StorageFile>**](http://msdn.microsoft.com/en-us/library/windows/apps/br206598.aspx) | [**StorageFile**](http://msdn.microsoft.com/en-us/library/windows/apps/windows.storage.storagefile.aspx) |
| [**XmlDocument.SaveToFileAsync**](http://msdn.microsoft.com/en-us/library/windows/apps/windows.data.xml.dom.xmldocument.savetofileasync.aspx) | [**IAsyncAction**](http://msdn.microsoft.com/en-us/library/windows/apps/windows.foundation.iasyncaction.aspx) | **void** |
| [**InkStrokeContainer.LoadAsync**](http://msdn.microsoft.com/en-us/library/windows/apps/windows.ui.input.inking.inkstrokecontainer.loadasync.aspx) | [**IAsyncActionWithProgress<UInt64>**](http://msdn.microsoft.com/en-us/library/windows/apps/br206581.aspx) | **void** |
| [**DataReader.LoadAsync**](http://msdn.microsoft.com/en-us/library/windows/apps/windows.storage.streams.datareader.loadasync.aspx) | [**DataReaderLoadOperation**](http://msdn.microsoft.com/en-us/library/windows/apps/windows.storage.streams.datareaderloadoperation.aspx), a custom results class that implements **IAsyncOperation<UInt32>** | [**UInt32**](http://msdn.microsoft.com/en-us/library/windows/apps/system.uint32.aspx) |

Asynchronous methods that are defined in [**.NET for Windows Store apps**](http://msdn.microsoft.com/en-us/library/windows/apps/br230232.aspx) have the return type [**Task**](http://msdn.microsoft.com/en-us/library/windows/apps/system.threading.tasks.task.aspx) or [**Task<TResult>**](http://msdn.microsoft.com/en-us/library/windows/apps/dd321424.aspx). Methods that return **Task** are similar to the asynchronous methods in the Windows Runtime that return [**IAsyncAction**](http://msdn.microsoft.com/en-us/library/windows/apps/windows.foundation.iasyncaction.aspx). In each case, the result of the asynchronous method is **void**. The return type **Task<TResult>** is similar to [**IAsyncOperation<TResult>**](http://msdn.microsoft.com/en-us/library/windows/apps/br206598.aspx) in that the result of the asynchronous method when running the task is the same type as the TResult type parameter. For more info about using **.NET for Windows Store apps** and tasks, see [.NET for Windows Store apps overview](http://msdn.microsoft.com/en-us/library/windows/apps/br230302.aspx).

**Handling errors**

When you use the **await** operator to retrieve your results from an asynchronous method, you can use a **try/catch** block to handle errors that occur in asynchronous methods, just as you do for synchronous methods. The previous example wraps the **RetrieveFeedAsync** method and **await** operation in a **try/catch** block to handle errors when an exception is thrown.

When asynchronous methods call other asynchronous methods, any asynchronous method that results in an exception will be propagated to the outer methods. This means that you can put a **try/catch** block on the outer-most method to catch errors for the nested asynchronous methods. Again, this is similar to how you catch exceptions for synchronous methods. However, you can't use **await** in the **catch** block.

**Summary and next steps**

The pattern of calling an asynchronous method that we show here is the simplest one to use when you call asynchronous APIs in an event handler. You can also use this pattern when you call an asynchronous method in an overridden method that returns **void** or a **Sub** in Visual Basic.

As you encounter asynchronous methods in the Windows Runtime, it is important to remember:

* By convention, asynchronous methods are given names that end in "Async".
* Any method that uses the **await** operator must have its declaration marked with the **async** keyword.
* When an app finds the **await** operator, the app remains responsive to user interaction while the asynchronous method executes.
* An asynchronous method returns an object that contains its result. In most cases, the result from the return value is what's useful, not the return value itself. You can find the result type by looking at the return type.
* Using asynchronous APIs and **async** patterns is often a way to improve the performance of your app. For more info on the performance benefits of using asynchronous APIs, see [Keep the UI thread responsive](http://msdn.microsoft.com/en-us/library/windows/apps/hh994635.aspx); that topic specifically discusses implementing logic as a [**Task**](http://msdn.microsoft.com/en-us/library/windows/apps/system.threading.tasks.task.aspx) so that it's asynchronous.

The example in this topic outputs text that looks like this.

Windows Experience Blog

PC Snapshot: Sony VAIO Y, 8/9/2011 10:26:56 AM -07:00

Tech Tuesday Live Twitter #Chat: Too Much Tech #win7tech, 8/8/2011 12:48:26 PM -07:00

Windows 7 themes: what’s new and what’s popular!, 8/4/2011 11:56:28 AM -07:00

PC Snapshot: Toshiba Satellite A665 3D, 8/2/2011 8:59:15 AM -07:00

Time for new school supplies? Find back-to-school deals on Windows 7 PCs and Office 2010, 8/1/2011 2:14:40 PM -07:00

Best PCs for blogging (or working) on the go, 8/1/2011 10:08:14 AM -07:00

Tech Tuesday – Blogging Tips and Tricks–#win7tech, 8/1/2011 9:35:54 AM -07:00

PC Snapshot: Lenovo IdeaPad U460, 7/29/2011 9:23:05 AM -07:00

GIVEAWAY: Survive BlogHer with a Sony VAIO SA and a Samsung Focus, 7/28/2011 7:27:14 AM -07:00

3 Ways to Stay Cool This Summer, 7/26/2011 4:58:23 PM -07:00

Getting RAW support in Photo Gallery & Windows 7 (…and a contest!), 7/26/2011 10:40:51 AM -07:00

Tech Tuesdays Live Twitter Chats: Photography Tips, Tricks and Essentials, 7/25/2011 12:33:06 PM -07:00

3 Tips to Go Green With Your PC, 7/22/2011 9:19:43 AM -07:00

How to: Buy a Green PC, 7/22/2011 9:13:22 AM -07:00

Windows 7 themes: the distinctive artwork of Cheng Ling, 7/20/2011 9:53:07 AM -07:00