Chapter No. 6
"Asynchrony in .NET"
In this package, you will find:

A Biography of the author of the book

A preview chapter from the book, Chapter NO.6 "Asynchrony in .NET"

A synopsis of the book’s content

Information on where to buy this book

About the Author

Richard Banks has developed software for the last 20 years for a wide range of industries and development platforms and over the years has filled many roles including Developer, Team Lead, Project Manager, and CIO/CTO. He is a Professional Scrum Trainer, runs the Sydney Alt.Net user group and the Talking Shop Down Under podcast, owns and contributes to a few open source projects, and has spoken at Microsoft Tech.Ed and a number of other events and user groups around Australia. For some strange reason he gets a real kick out of helping development teams to improve and produce great software. If you want to get in touch, his tweet handle is @rbanks54. He blogs at http://www.richard-banks.org/. He currently works as a Principal Consultant for Readify and is a Microsoft Visual Studio ALM MVP.

It might have my name on the front cover but a book is never the work of just one person.

I would firstly like to thank my fantastic wife, Anne, and my two wonderful children, Hannah and Leisel, for giving me the time and space to work on this book. Their support throughout the process has been invaluable and without that I would have never undertaken this book in the first place.

I'd also like to thank the staff of Packt Publishing for the opportunity and help in bringing this together, and my tech reviewers who gave up their spare time reading my scribble and checking that what I wrote actually made sense, instead of being just a delirium fuelled pile of nonsense.

Thank you all!

For More Information:
**Visual Studio 2012 Cookbook**

*Visual Studio 2012 Cookbook* is a set of simple-to-follow recipes that you can use to discover and master the features of the latest version of Microsoft's premier development tool.

While you could try and discover features by clicking around in the menus, it's easy to miss the new features and to see how they can help you. Plus Visual Studio 2012 has so much more to offer than just features that can be accessed via menu entries. The recipes in this book will help you quickly get up to speed with what those features are, how they work, and how you might use them to produce fantastic software in less time than you thought possible.

**What This Book Covers**

*Chapter 1, Discovering Visual Studio 2012,* introduces you to the common IDE features that you can take advantage of, regardless of the language you are developing in or the type of software you are building. Discover the new project types, navigation options, search facilities, and more.

*Chapter 2, Getting Started with Windows Store Applications,* shows you how Visual Studio 2012—the only way you can build the new modern style apps for Windows 8—supports Windows Store app development, how the simulator works, and how to package up an application for submission to the Windows Store.

*Chapter 3, Web Development: ASP.NET, HTML5, CSS, and JavaScript,* brings you up to speed with the wide ranging improvements in web development that Visual Studio 2012 brings to the table. This includes the CSS and JavaScript editing improvements, the new Page Inspector, and the bundling and minification features in ASP.NET.

*Chapter 4, .NET Framework 4.5 Development,* shows you how Visual Studio 2012 provides outstanding support for the .NET Framework 4.5 development and touches on some of the new key features in the framework. You will also be shown how Visual Studio 2012 helps you raise the quality of the code you build using the new Test Explorer and code clone detection features.

*Chapter 5, Debugging Your .NET Application,* steps you through the new and improved debugging capabilities of Visual Studio 2012. These include the new production debugging capability and improved ways of understanding what your parallel and concurrent code is doing.

---

**For More Information:**

Chapter 6, Asynchrony in .NET, takes a deeper look into the support Visual Studio 2012 provides for writing asynchronous code in .NET so that you can make better use of multi-core machines to improve your application's responsiveness and performance. You will see how the async and await keywords make development much simpler and how new libraries such as the TPL DataFlow library can open up new ways of solving concurrency problems.

Chapter 7, Unwrapping C++ Development, gives you an insight into Visual Studio 2012's fresh love for C++ developers, the new language features it supports, and the tooling to make developing C++ applications quicker. You will see how you can mix C++ and XAML to build a Windows Store app UI, how to unit test and analyze your code, and how to diagnose how a single pixel was drawn to screen in DirectX apps.

Chapter 8, Working with Team Foundation Server 2012, guides you through both the Team Foundation Server 2012 and Visual Studio 2012 improvements for team-based development, and agile development in particular. This includes source control, code reviews, gaining feedback from your users, and more.

Appendix, Visual Studio Pot Pourri, is all about the wonderful features of Visual Studio 2012 that didn't really fit anywhere else but that are still of great value. This includes features such as the new SQL Server Developer Tools, the creation of application installers, and how to submit an app to the Windows Store.

For More Information:
Introduction

Microsoft realized that, while most developers understand the benefits of asynchronous code and the improvements it can bring about in their applications, the programming models involved in asynchrony were fairly cumbersome, verbose and in some cases quite difficult to get right. As a result most developers ignored asynchrony unless circumstances forced it upon them. The extra complexity, effort, time, chance for bugs, and difficulty in debugging meant that it simply wasn't worth it for most developers.

To ensure reading and writing asynchronous code is no longer restricted to only the superhuman amongst us, Visual Studio 2012 and .NET 4.5 introduced the async and await keywords for both the C# and Visual Basic languages. Keywords make asynchronous code as easy to read, write, and debug as normal synchronous code.

As you saw in Chapter 5, Debugging Your .NET Application, the debugging experience is greatly improved in multi-threaded asynchronous code, and now with .NET 4.5 the language support is present to make the development of asynchronous code so easy that there is no excuse for not using it.

In this chapter, you'll be looking specifically at the async and await keywords and seeing how Visual Studio 2012 supports them.

For More Information:
Making your code asynchronous

So you've got yourself an application that might be lacking in the performance department. If you're honest, it's probably horribly slow, and yet when you look at the performance counters on the machine it doesn't seem to be doing all that much. What are the odds that your code is doing a slow operation and blocking the execution thread, preventing other code from executing? Pretty high, huh?

It gets even worse in web applications that come under heavy load. Every request thread that gets blocked is a point where other requests can get queued, and before too long you've got yourself a server that is throwing 503 Service Unavailable errors.

Time to take that synchronous code, stick an "a" on the front of it and make your system start to sing.

Just remember that before you make all of your code asynchronous, you should understand where it blocks and where it doesn't. The overhead of threading can actually make your application run slower if you aren't careful. Now with that in mind, let's go and make some asynchronous code.

Getting ready

You will need an Internet connection for this recipe to work since you load data from various RSS feeds and display it.

Ensure you have a connection, then simply start Visual Studio 2012 and you're ready to go.

How to do it...

Perform the following steps:

1. Create a C# Console Application named FeedReader. This application will read the feeds from a number of sites and display them on the console. At the end of the display, the total time required for the feeds to be fetched and displayed will be shown.

2. In the program, classes from a number of different namespaces will be used. To save some time, add the following code to the using statements at the top of Program.cs:

```csharp
using System.Diagnostics;
using System.Net;
using System.Net.Cache;
using System.Xml.Linq;
```

For More Information:

3. Before you implement the main method, you need to create some supporting methods. Add a `ReadFeed()` private method, as shown after the `Main()` method. It creates a web client to read an RSS feed with the cache setting turned off. This will ensure that we always pull data from the Internet and not a local cached copy.

```csharp
private static string ReadFeed(string url)
{
    var client = new WebClient()
    {
        CachePolicy = new RequestCachePolicy(RequestCacheLevel.NoCacheNoStore)
    }
    var contents = client.DownloadString(url);
    return contents;
}
```

4. Add a `PublishedDate()` method below the `ReadFeed()` method. It will convert dates in the feed, that `System.DateTime` doesn't handle, into dates that can be parsed.

```csharp
public static DateTime PublishedDate(XElement item)
{
    var s = (string)item.Element("pubDate");
    s = s.Replace("EST", "-0500");
    s = s.Replace("EDT", "-0400");
    s = s.Replace("CST", "-0600");
    s = s.Replace("CDT", "-0500");
    s = s.Replace("MST", "-0700");
    s = s.Replace("MDT", "-0600");
    s = s.Replace("PST", "-0800");
    s = s.Replace("PDT", "-0700");
    DateTime d;
    if (DateTime.TryParse(s, out d)) return d;
    return DateTime.MinValue;
}
```

5. Now move back into the `Main()` method and create a variable for the list of feeds to read from.

```csharp
static void Main(string[] args)
{
    var feedUrls = new List<string>(){
        "http://massively-joystiq.com/rss.xml",
        "http://feeds2.feedburner.com/alvinashcraft",
        "http://feeds.feedburner.com/ScottHanselman",
    };
}
```
6. Next create a `Stopwatch` so that you can start timing how long the execution takes, and then add the code to load the data from the feeds.

```csharp
var stopwatch = Stopwatch.StartNew();
var feeds = (from url in feedUrls
            select ReadFeed(url))
            .ToArray();
```

7. You need to parse the feed so you can extract something to show on screen. Add the following code in the `Main()` method, to do so:

```csharp
var items = from feed in feeds
            from channel in XElement.Parse(feed).Elements("channel")
            from item in channel.Elements("item").Take(1)
            let date = PublishedDate(item)
            orderby date descending
            select new
            {
                Title = (string)channel.Element("title"),
                Link = (string)channel.Element("link"),
                PostTitle = (string)item.Element("title"),
                PostLink = (string)item.Element("link"),
                Date = date
            };
```

8. Now, tie a bow around it and finish the program. Complete the `Main()` method by adding the following code to display an item from each feed on the console and show the total time it took to process all feeds:

```csharp
foreach (var item in items)
{
    Console.WriteLine("Title: {} [{}]", item.Title, item.Link);
    Console.WriteLine("  Post: {} [{}]", item.PostTitle, item.PostLink);
    Console.WriteLine("  Date: "+ item.Date);
    Console.WriteLine("--------");
}

Console.WriteLine("Total Time: " + stopwatch.Elapsed);
Console.ReadKey();
```

9. Compile the program and check that it runs. Don't panic if the console takes a little while to show some text, you've got some slow code running here. When it does eventually complete you should see output similar to the following screenshot:

For More Information:
10. Ok, so it's not the fastest code in the world. Time to introduce the await and async keywords and see if you can't speed this thing up.

First, locate the ReadFeed() method and change the return type from `string` to `Task<string>`.

11. You will then need to return a `Task<string>` object from the method, but you can't just cast the `contents` variable to that type. Fortunately, the `WebClient` class includes a task based version of `DownloadString` called `DownloadStringTaskAsync` that returns a `Task<string>`. Perfect for your needs. Change the code to use `client.DownloadStringTaskAsync(url)`.

```csharp
private static Task<string> ReadFeed(string url)
{
    var client = new WebClient()
    {
        CachePolicy = new RequestCachePolicy(RequestCacheLevel.NoCacheNoStore)
    }
    var contents = client.DownloadStringTaskAsync(url);
    return contents;
}
```

12. Navigate back up to the `Main()` method and you will see a problem with the `Parse()` method in the LINQ statement. The root cause is that the `feeds` variable is now an array of the `Task<string>` objects, and not `string` objects.
13. Change the code where `feeds` is assigned to wrap the LINQ statement in a Task. `Task.WhenAll()` call instead of using `.ToArray()`. The `Task.WhenAll` method creates a task that waits until all of the inner tasks returned by the enclosed LINQ statement are complete. The `await` keyword tells the compiler that the task should be executed asynchronously and the result assigned to the `feeds` variable.

```csharp
var feeds = await Task.WhenAll(from url in feedUrls
                                select ReadFeed(url));
```

14. There is still a problem. The compiler is now complaining about the `await` keyword not being valid. Any method where the `await` keyword is used must have the `async` keyword in its declaration. Go to the declaration of the `Main()` method and add the `async` keyword as shown in the following screenshot:

```csharp
static async void Main(string[] args)
{
}
```

15. Compile the application. You will get an error indicating the `Main` method can’t be made asynchronous as it is the program entry point.

16. This is easy enough to work around. Simply rename the `Main()` method to `ProcessFeedsAsync()` and insert a new `Main()` method above it, using the code shown in the following screenshot. Also, remove the `ReadKey()` method from the end of the `ProcessFeedsAsync()` method so that you are not prompted for user input twice.

```csharp
static void Main(string[] args)
{
    Console.WriteLine("starting...");
    ProcessFeedsAsync();
    Console.WriteLine("finished...");
    Console.ReadKey();
}

static async void ProcessFeedsAsync()
{
    var feedUrls = new List<string>(){
        ------------------
        Console.WriteLine("Total Time: " + stopwatch.Elapsed);
        Console.ReadKey(); // Delete this line
    }
```
17. Compile and run the program. You should see output somewhat similar to the following screenshot and the elapsed time should be shorter than before:

![Screenshot of output](image)

**How it works...**

As you've seen, Visual Studio offers enough warnings and errors through IntelliSense to make conversion of synchronous code to asynchronous reasonably straightforward, as long as you make changes in small, incremental steps. Large scale changes of code, regardless of what those changes may be, are always difficult and error prone, especially if you lack unit tests or other mechanisms to verify your changes haven't broken any functionality.

The `DownloadStringTaskAsync()` method shows off an important convention to be aware of in the .NET 4.5 Framework design. There is a naming convention to help you to locate the asynchronous versions of methods, where methods that are asynchronous all have an "Async" suffix on their names. In situations where an asynchronous method exists from previous framework versions the newer, task-based, asynchronous methods are named with the "TaskAsync" suffix instead.

In step 16, the `ReadKey()` method was added to stop the main method from completing immediately and terminating the program before any output was returned. In the console window you can see that the `starting` and `finishing` messages are displayed before any of the feed details appear. This occurs because the `ProcessFeedsAsync` method was being executed asynchronously on a separate thread, while the `Main()` method was still being executed on the main application thread. This is exactly what we would expect from non-blocking, asynchronous code.
There's more...

It's possible to overdo it. Every piece of asynchronous code comes with a certain amount of overhead. There is a CPU cost to context switching and a higher memory footprint needed for maintaining memory state for each thread, and if you have too many threads you can actually reduce the performance of your application.

The design guideline for the Windows Runtime libraries in Windows 8 was that any method that was likely to take more than 50 ms to complete was to be made asynchronous; there was a minimum duration used as a way of determining when it made sense to go asynchronous. 50 ms is probably a good final target for your methods as well, but before you go and improve all the methods in your application, start by determining which of your current methods are the slowest. These should be what you target first. Start by improving only methods that take more than 500 ms to complete and resolve those first, before targeting the faster methods.

Whenever determining the appropriate balance between synchronous and asynchronous code, you should be doing performance and load testing on your application to determine what the current performance profile is, and what effect your changes will have on it. Because each and every application is different, finding the right mix can be an art. As a tip, identify the slowest areas of your application and target them first. As you improve performance, keep an eye on how much time it costs you to make your code asynchronous versus the improvement you are seeing in the overall application performance.

Asynchrony and Windows Runtime

When developing the Windows Runtime for Windows 8, Microsoft followed a design guideline where any synchronous method that might take longer than 50 ms to complete was to be removed and replaced with an asynchronous version. The goal behind this design decision is to dramatically improve the chances of developers building applications that feel smooth and fluid by not blocking threads on framework calls.

In this recipe you're going to load the RSS feed details again, just as you did in the Making your code asynchronous recipe, though this time you're going to be creating a Windows Store application.

There are a few differences between a Windows Store application and a console one, including differences in the classes available. For example, the WebClient class doesn't exist in WinRT so you'll be using the HttpClient class instead.

For variety, you will be writing this code using Visual Basic.

Getting ready

Ensure you are running Windows 8 and then launch Visual Studio 2012.

For More Information:
How to do it...

Perform the following steps:

1. Create a new project by selecting Visual Basic | Windows Store | Blank App (XAML) and name it FeedReader.

2. Add a class named Post to the application using the following code. This class will hold the details of each post from the RSS feed that we will show on screen.

   ```
   Public Class Post
   Public Property Title As String
   Public Property Link As String
   Public Property PostTitle As String
   Public Property PostLink As String
   Public Property PostDate As DateTime
   End Class
   ```

3. Open MainPage.xaml and add the following XAML to the <Grid /> element to define the markup of how the results should appear. The layout consists of a button to start the feed loading and a ListBox element in which the results are displayed. You also have a TextBlock element in which you'll post the time it takes to read the feeds.

   ```
   <Grid Background="{StaticResource ApplicationPageBackgroundThemeBrush}">
     <Button Name="LoadFeeds" Margin="116,60,0,0" VerticalAlignment="Top">
       Load Feeds
     </Button>
     <TextBlock Name="TimeTaken" HorizontalAlignment="Left"
                Height="36" Margin="257,60,0,0" TextWrapping="Wrap"
                VerticalAlignment="Top" Width="360" FontSize="32">
       Waiting for click...
     </TextBlock>
     <ListBox Height="450" HorizontalAlignment="Left"
              Margin="116,140,0,0" Name="PostsListBox"
              VerticalAlignment="Top" Width="500">
       <ListBox.ItemTemplate>
         <DataTemplate>
           <StackPanel Orientation="Vertical" Height="110">
             <TextBlock Text="{(Binding Title)}" />
             <TextBlock Text="{(Binding PostTitle)}" />
             <TextBlock Text="{(Binding PostLink)}" />
             <TextBlock Text="{(Binding PostDate)}" />
           </StackPanel>
         </DataTemplate>
       </ListBox.ItemTemplate>
     </ListBox>
   </Grid>
   ```

For More Information:
4. Next, navigate to the code behind file MainPage.xaml.vb, and add a couple of imports statements that you will need for later:

```csharp
Imports System.Net.Http
```

5. Now add some initial code to define the RSS feeds to use and a collection to hold the Post objects.

```csharp
Public NotInheritable Class MainPage
    Public Property Posts As List(Of Post)
    Dim feedUrls As New List(Of String)
    Public Sub New()
        InitializeComponent()
        feedUrls = New List(Of String) From {
            "http://massively.joystiq.com/rss.xml",
            "http://feeds2.feedburner.com/alyvishcraft",
            "http://feeds.feedburner.com/ScottHanselman"
        }
        Posts = New List(Of Post)
    End Sub
End Class
```

6. Add the PublishedDate() helper method to the class after the New() method.

```csharp
Public Function PublishedDate(item As XElement) As DateTime
    Dim s As String = CType(item.Element("pubDate"), String)
    s = s.Replace("EST", "-0500")
    s = s.Replace("EDT", "-0400")
    s = s.Replace("CST", "-0600")
    s = s.Replace("CDT", "-0500")
    s = s.Replace("MST", "-0700")
    s = s.Replace("MDT", "-0600")
    s = s.Replace("PST", "-0800")
    s = s.Replace("PDT", "-0700")
    Dim d As DateTime
    If DateTime.TryParse(s, d) Then
        Return d
    End If
    Return DateTime.MinValue
End Function
```

For More Information:
7. Add the `ReadFeed()` helper method below the `PublishedDate()` method using the following code:

```vbnet
Private Async Function ReadFeed(url As String) As Task(Of String)
    Dim client As New HttpClient
    Dim cacheControl As New CacheControlHeaderValue With {
        .NoCache = True,
        .NoStore = True
    }
    client.DefaultRequestHeaders.CacheControl = cacheControl
    client.MaxResponseContentBufferSize = Integer.MaxValue
    Dim response As HttpResponseMessage = Await client.GetAsync(url)
    Dim content As String = Await response.Content.ReadAsStringAsync()
    Dim _posts = From channel In XElement.Parse(content).Elements("channel")
                From item In channel.Elements("item").Take(1)
                Let _date = PublishedDate(item)
                Order By _date Descending
                Select New Post With {
                    .Title = CType(channel.Element("title"), String),
                    .Link = CType(channel.Element("link"), String),
                    .PostTitle = CType(item.Element("title"), String),
                    .PostLink = CType(item.Element("link"), String),
                    .PublishedDate = _date
                }
    Dim post = _posts.First
    Posts.Add(post)
    Return ""
End Function
```

8. It's now time to add some functionality to the button that loads the feeds. Write a handler for the `LoadFeeds` button's click event using the following code:

```vbnet
Private Async Sub LoadFeeds_Click(sender As Object, e As RoutedEventArgs)
    Handles LoadFeeds.Click
    Dim _stopwatch = Stopwatch.StartNew
    Await Task.WhenAll(From url In feedUrls Select ReadFeed(url))
    Dim _timespan As TimeSpan = _stopwatch.Elapsed
    TimeTaken.Text = _timespan.ToString
    PostslistBox.ItemsSource = Posts
End Sub
```
9. Compile and run the program. When the UI appears click on the **Load Feeds** button, wait a few seconds and you should see the results of your work appear as in the following screenshot:

![Screenshot of Load Feeds button](image)

**How it works...**

In step 8 you added a `LoadFeeds.Click` event handler. The important thing to note about this method is that it is an **async** method and that `await` is used with the `Task.WhenAll` method. When the application runs and you click on the button, the click event fires the event handler, which in turn starts the background processing that reads the feeds. While the application is waiting for that background process to complete, control is returned to the main application for any other work that needs to be done, ensuring you do not block the application while waiting for the feeds to be retrieved. When the feed retrieval completes, execution returns back to the click event handler, which then updates the UI with the results.

In step 7 the `ReadFeed()` method looks similar to what you used in the console application in the **Making your code asynchronous** recipe, however you will now see that you are using the `HttpClient` class instead of the `WebClient` class as it isn't available in the Windows Runtime. The `HttpClient` class also requires different code to set up the cache control values and you have to specify the response buffer size, otherwise you can get runtime exceptions on long feeds.

---

**For More Information:**

Since this is a Windows Store app and you are coding against WinRT and the .NET Framework 4.5 Windows Store app profile you cannot produce a synchronous version of the application. The synchronous API calls that you might have used with a console or WPF application simply aren't available.

This makes the `await` and `async` keywords critical for Windows Store apps. Get used to them, know them, love them; even send them thank you cards! Without these keywords, developing asynchronous applications that meet modern design guidelines would be so much harder to do and so much more fragile and difficult to debug. These two little keywords make asynchronous programming very, very simple.

### See also
- The *Making your code asynchronous* recipe

### Asynchrony and web applications

Web applications don't need to be asynchronous, do they? IIS gives each request its own thread so people don't need to worry about it, right? Even if one request goes slow, all the others will still be processed quickly so it's not really a problem, right?

It's surprising how many times this is said by developers, often the same ones who have slow sites even though they have small user loads and few requests per second. If you want a responsive, scalable web application that supports hundreds or thousands of users per server, you need to make the best use of the hardware you are on and you must consider the problems that are caused by blocking threads.

IIS has limits on the number of requests and I/O threads it uses. Blocking any of these threads means IIS is forced to wait until the thread is released before another request can be processed. When there are no threads available to process requests (because of blocking or high-server load), requests start to queue up and, over time, that queue can grow until it reaches its maximum size, at which point the dreaded 503 Service Unavailable message will start showing to your site's visitors. Not really what you want.

High-server load due to a large volume of visitors is not something you can control. What is in your control, however, is your ability to write code that doesn't block threads and allows IIS to scale and process more requests than would have been possible otherwise.

Once again, you'll use the feed reader scenario, but for simplicity you'll just make the network calls to retrieve the RSS feeds and then display the time it took to do so.

For More Information:

Asynchrony in .NET

We're not going to cover load testing the site in this recipe since it's a feature that's been around in Visual Studio for quite some time now. See http://msdn.microsoft.com/en-us/library/dd293540.aspx for more information on the load testing features in Visual Studio.

Getting ready

Simply start Visual Studio 2012 and you're ready to go.

How to do it...

Create an asynchronous web application by following these steps:

1. Start a new ASP.NET Empty Web Application project using C# and give it the default name.
2. Add a new Web Form item to the project, leaving it with the default name, which should be the very creative WebForm1.
3. In WebForm1.aspx add async="true" to the end of the page directive. This tells ASP.NET to allow the page lifecycle events prior to the PreRender event to execute asynchronous tasks.

```xml
<% Page Language="C#" AutoEventWireup="true" 
   CodeBehind="WebForm1.aspx.cs" 
   Inherits="WebApplication10.WebForm1" 
   Async="true" %>
```

4. Further down in the page body add an id attribute to the <div> element and a runat="server" attribute so that you can place the timing results in it when the page executes.

```html
<form id="form1" runat="server">
  <div id="timeTaken" runat="server">
  
  </div>
</form>
```

5. Now navigate to the WebForm1.aspx.cs code behind file and add some supporting using statements as follows:

```csharp
using System.Diagnostics;
using System.Threading.Tasks;
using System.Net;
using System.Net.Cache;
```

For More Information:
6. Next add the supporting `ReadFeed()` method to read a single RSS feed.

```csharp
private async static Task<string> ReadFeed(string url)
{
    var client = new WebClient()
    {
        CachePolicy = new RequestCachePolicy(RequestCacheLevel.NoCacheNoStore)
    }
    return await client.DownloadStringTaskAsync(url);
}
```

7. Now that you have the `ReadFeed()` method implemented, you should implement the `Page_Init()` method to read all the feed information during page startup. Because you want the page to load asynchronously you will need to register a `PageAsyncTask` object. This lets ASP.NET know that you are performing an asynchronous operation, which is important since page lifecycle events themselves are not asynchronous and without them the page would render before your tasks were complete.

```csharp
private TimeSpan duration;

protected void Page_Init(object sender, EventArgs e)
{
    var feedUrls = new List<string>()
    {
        "http://massively.joystiq.com/rss.xml",
        "http://feeds2.feedburner.com/alvinashcraft",
        "http://feeds.feedburner.com/ScottHanselman"
    };
    RegisterAsyncTask(new PageAsyncTask(async (ct) =>
    {
        var stopwatch = Stopwatch.StartNew();
        var feeds = await Task.WhenAll(
            from url in feedUrls select ReadFeed(url));
        foreach (var feed in feeds)
        {
            Debug.WriteLine(feed.Length);
        }
        duration = stopwatch.Elapsed;
        timeTaken.InnerText = duration.ToString();
    }));
}
```

For More Information:
8. Finally, add code to the `Page_PreRender()` method so that the duration of the entire page lifecycle, inclusive of the RSS reading, can be seen in the debug console in Visual Studio.

```csharp
protected void Page_PreRender(object sender, EventArgs e)
{
    Debug.WriteLine("Duration: {0}", duration);
}
```

9. Press F5 to start debugging the application. After a few seconds the page load should complete and render a screen similar to the following screenshot:

10. Leaving the page open, switch back to Visual Studio which should still be in debug mode. Look at the contents of the **Output** window and the **Debug** messages in particular. As shown in the following screenshot, you should see that the debug message from the `PreRender` event is displayed before the four numbers, showing the size of data pulled from the RSS feeds.

    The duration shows as zero because the `Page_Init` method has completed, but `PageAsyncTask` you registered has not yet executed by the time the `PreRender` method is called.

---

**How it works...**

It's important to keep in mind that with ASP.NET Web Forms the page methods are executed synchronously, even if you put the `async` keyword on the method declarations. You must use `RegisterAsyncTask`, just as you needed to in previous .NET versions.

---

**For More Information:**

Because of the `async` keyword, the registering of tasks is now simply a matter of including a lambda in the code. You don't need to follow the old style of asynchronous programming anymore and you don't have to write any begin and end methods for the framework to call.

You will also notice that the page itself still took a while to load. The asynchronous approach you used allows the web server as a whole to scale and process more requests concurrently. It doesn't magically make those slow network calls to the RSS feeds any faster, so be prepared to think of other ways to improve your user interface to indicate to your users that something is happening and to just be patient.

**There's more...**

You may be thinking "what about ASP.NET MVC 4?". Well, in ASP.NET MVC 4, things are even simpler.

Your controller still inherits from the `AsyncController` class, however, instead of having to write method pairs for the beginning and ending of an asynchronous operation, you simply have to create a controller method that returns a `Task<T>`.

For example:

```csharp
public async Task<ActionResult> Index()
{
    await LongRunningMethod();
    return View();
}
```

This is much better than how asynchronous controllers worked in previous versions of ASP.NET MVC. It's now so easy!

**See also**

- The *Making your code asynchronous* recipe

---

**Actors and the TPL Dataflow Library**

With Visual Studio 2010 and .NET 4.0 we were given the *Task Parallel Library (TPL)*, which allowed us to process a known set of data or operations over multiple threads using constructs like the `Parallel.For` loop.

Coinciding with the release of Visual Studio 2012, Microsoft has now given us the ability to take any data we like and process it in chunks through a series of steps, where each step can be processed independently of the others. This library is called the *TPL Dataflow Library*.

---

**For More Information:**

Asynchrony in .NET

An interesting thing to note about this library is that it was originally included as part of .NET Framework in the pre-release versions, but the team moved it to a NuGet distribution model so that changes and updates to the package could be made outside of the normal .NET lifecycle. A similar approach has been taken with the Managed Extensibility Framework (MEF) for web and Windows Store apps. This change to the distribution model shows a willingness from Microsoft to change their practices so that they can be more responsive to developer needs.

From a terminology perspective, the processing steps are called **Actors** because they "act" on the data they are presented with and the series of steps performed are typically referred to as a **pipeline**.

A fairly common example of this is in image processing where a set of images needs to be converted in some way, such as adding sepia tones, ensuring all images are in portrait mode, or doing facial recognition. Another scenario might be taking streaming data, such as sensor feeds, and processing that to determine actions to take.

This recipe will show you how the library works. However, to keep things short, we won't do any fancy image processing. Instead, we'll just take some keyboard input and display it back on the screen after having converted it to uppercase and Base64 encoding it.

In order to do this we will use an **ActionBlock** object and a **TransformBlock** object. An **ActionBlock** object takes a piece of data passed to it and does something with it, that is, it performs an action using it, and a **TransformBlock** object takes a piece of data and changes it in some way.

In this recipe, you will use a **TransformBlock** object to convert characters to uppercase and encode them before passing them to an **ActionBlock** object to display them on screen.

### Getting ready

Simply start Visual Studio 2012 and you're ready to go.

### How to do it...

Create a DataFlow powered application using the following steps:

1. Create a new application targeting .NET Framework 4.5 by selecting **Visual C#** | **Console Application** and name it **DataFlow**.
2. Using NuGet, add the TPL Dataflow library to the project.
3. Open Program.cs and at the top of the file add the following using statements:

```csharp
using System.Threading;
using System.Threading.Tasks;
using System.Threading.Tasks.Dataflow;
```

4. In the Main() method of Program.cs add the following code to define the ActionBlock. The method in the ActionBlock object displays a String on the console and has a Sleep method call in it to simulate long running work. This gives you a way to slow down processing and force data to be queued between steps in the pipeline.

```csharp
var slowDisplay = new ActionBlock<string>(async s =>
{
    await Task.Run(() => Thread.Sleep(1000));
    Console.WriteLine(s);
}, new ExecutionDataflowBlockOptions { MaxDegreeOfParallelism = 4 });
```

5. Next, add the code for TransformBlock. The TransformBlock object will take a char as input and return an uppercase base64 encoded string. The TransformBlock object is also linked to the ActionBlock object to create a two-step pipeline.

```csharp
var transformer = new TransformBlock<char, string>(c =>
{
    var upper = c.ToString().ToUpperInvariant();
    var bytes = ASCIIEncoding.ASCII.GetBytes(upper);
    var output = Convert.ToBase64String(bytes);
    return output;
});
transformer.LinkTo(slowDisplay);

For More Information:
6. Now add code to take input from the console and pass it to the first step of the pipeline (the TransformBlock object in this case). You also need to close and flush the pipeline when you hit Enter so that you can exit the program.

```csharp
bool keepGoing = true;
while (keepGoing)
{
    var key = Console.ReadKey();
    if (key.Key == ConsoleKey.Enter)
    {
        keepGoing = false;
        transformer.Complete();
        Console.WriteLine("waiting for the queue to flush");
        transformer.Completion.Wait();
        slowDisplay.Complete();
        slowDisplay.Completion.Wait();
        Console.WriteLine("press any key");
        Console.ReadKey();
        break;
    }
    transformer.Post(key.KeyChar);
}
```

7. Run the program. When the console window appears, just randomly press characters, and when you are done hit Enter. You should see an output similar to the following. Note how the encoded strings appear in batches up to four, though this may be one or two if you have a CPU with less than four cores.

For More Information:
So what just happened here?

Firstly, you defined two actors. The first being the ActionBlock object that takes a string and displays it on screen and a second, the TransformBlock, that takes a character as input and returns an encoded string as output. You then linked the TransformBlock object to the ActionBlock object to create the pipeline for the data to flow through.

Then you took data that was streaming to you (the console key presses) and passed each key press to the pipeline as soon as it arrived. This continued until the user hit Enter at which point the Complete() method is used to tell the actors that they should expect no more data. Once the queues flush, the user is prompted to hit a key to close the program.

If you fail to flush the queues you will lose the data that is still in them when the program completes. I'm not sure about you, but I find that losing data tends to upset people at times and I prefer not having those "please explain" conversations with people.

Now when you ran the program the TransformBlock object did its work very quickly and passed its output to the ActionBlock. The interesting thing to note is that even though the data was queuing up to be processed by the ActionBlock object, the amount of code you had to write to do that was zero! The TPL Dataflow library takes care of all the difficult plumbing code, thread management, and the communication of data between actors, as well as determining how many actors it can run at once.

You may also be wondering what happens in less straightforward scenarios, such as when you want to conditionally pass data or messages to the next actor. Fortunately, the TPL Dataflow Library is quite powerful and you've only scratched the surface in this recipe. For example, the LinkTo() method has a predicate parameter that you can use to filter the messages and decide which actors should do what.

You could also batch up data for processing in the later steps by adding data to a buffer using the BufferBlock object and only passing buffered data to subsequent pipeline steps when the buffer is full. There are lot of possibilities! Feel free to go and explore what the library has to offer!

The eagle eyed amongst you may also have noticed that the lambda function used by the ActionBlock object featured the async keyword. This was done so that the action block doesn't itself block execution of the program when performing the long-running task and prevent any more input from being processed.

For More Information:
Asynchrony in .NET

See also

- The Making your code asynchronous recipe
- The Debugging parallel code recipe in Chapter 5, Debugging Your .NET Application

For More Information:
Where to buy this book


Free shipping to the US, UK, Europe and selected Asian countries. For more information, please read our shipping policy.

Alternatively, you can buy the book from Amazon, BN.com, Computer Manuals and most internet book retailers.