

A quick guide to ASP.NET Asynchronous Server Calls

by Leslie Drewery

Keynote

As ASP.NET development has become easier and more powerful, developers are creating bigger applications that take full use of the server and have more business logic than ever before. Thinking that they have written a web application, they assume it must be scalable by default.

Developers that believe the statement “web application - it must be scalable by default” are in for scalability problems. There are a few very easy solutions to this problem.

- Increase the thread count in the thread pool (not advisable).
- Add more CPU's as the default threads apply per CPU, (IIS 5 – 25 threads per CPU, IIS 6 – 50 threads per CPU).
- Add asynchronous server calls to your web applications and avoid all the aggravation of trying to justify server changes and add unparalleled scalability.

As this topic of asynchronous server calls is extremely under-documented and it requires very little effort to implement this technology, I have put together this guide.

Code Samples can be downloaded from <http://www.richplum.co.uk/downloads>.

Understanding asynchronous server calls and their affect on the threadpool.

When a request is made to the IIS Server, a thread is allocated to handle each call until completion. This decreases the thread count in the pool, on IIS 5 with default thread pool settings, it can only handle 25 requests simultaneously and further requests will be queued until a thread is returned to the pool, adding to slow server response even for simple requests when the 26th or greater request is made.

After we implement the asynchronous calls, the execution alters slightly to overcome the thread count issues, allowing the server to carry on processing incoming requests. This is done by passing the caller to a new thread outside the thread pool and returning the original thread back to the pool ready for a new call. The newly allocated thread is managed by call-back methods and state is managed by the IAsyncResult interface. The downside to this is that, if you allocate numerous threads to a resource-thirsty bit of business logic, you are still going to push CPU usage up causing the server to slow down.

IAsyncResult Interface

The `IAsyncResult` interface references an object that stores state information for an asynchronous operation, and provides a synchronization object to allow threads to be signalled when the operation completes and passes state information. This interface is used by all asynchronous calls from file handling to server calls.

Implementing asynchronous server calls

This implementation can be used for both User Controls as well as pages allowing developers to optimise their server calls, lowering their thread pool requirements.

Implementation of asynchronous server calls only requires the following four easy steps.

Step 1: Add the `Async="true"` attribute to the page reference in your aspx/ascx page to enable asynchronous calls.

```
<%@ Page Async="true"
```

Step 2: Create a method to handle the beginning of the operation; this method is used to initialise variables and general preparation. The thing to note is that the `IAsyncResult` is returned from this method, so if you create your own `AsyncResult` object all custom properties and variables must be initialised and assigned before the method is completed.

```
public IAsyncResult BeginAsyncOperation(object sender, EventArgs e, AsyncCallback cb, object
    state)
{
    _request = WebRequest.Create("http://www.amazon.co.uk");
    return _request.BeginGetResponse(cb, state);
}
```

Step 3 : Create the finally method for the callback: all rendering and data preparation to be passed back to the client must be done in this method. The thing to note is that the AsyncResult Object created in step 2 is passed as a parameter into this method and is called as part of the page's prerendering.

In my example I assign the data to a literal control on the screen which will display the home page for Amazon in the master page.

```
public IAsyncResult BeginAsyncOperation(object sender, EventArgs e, AsyncCallback cb, object
    state)
{
    _request = WebRequest.Create("http://www.amazon.co.uk");
    return _request.BeginGetResponse(cb, state);
}
```

Step 4: The AddOnPreRenderCompleteAsync is the key to an asynchronous call, it registers the BeginAsyncOperation and EndAsyncOperation methods for the Asynchronous call. This method will need to be called from the Page_Load method.

```
protected void Page_Load(object sender, EventArgs e)
{
    //assign the Begin and End Operation Callback methods.
    AddOnPreRenderCompleteAsync(
        new BeginEventHandler(BeginAsyncOperation),
        new EndEventHandler(EndAsyncOperation));
}
```

About the Author



Leslie Drewery GG (general gopher) is one of the developers for CompuFile Limited,

He develops in Delphi and C# covering win32 through to ASP.NET and recently DirectShow using DirectX and not much of a writer☺.

He also has a keen interest in PC based Digital TV systems using .NET.

He can be reached via his website <http://www.firedtv.com> or email leslie@firedtv.com