Building High Performance Web Applications

James Senior, Microsoft Corporation

Dan Wahlin, Wahlin Consulting

Copyright

Contents

Overview

Many of today's Web applications rely on JavaScript to minimize the number of round trips being made to the server and provide a more desktop-like experience for end users. While adding JavaScript and Ajax functionality into applications provides many benefits, it can also cause additional performance issues unless key development principles are taken into consideration and applied. It goes without saying that having a good understanding of Ajax development techniques and tools is important for building successful, high performance applications.

Although browsers have continued to evolve over the years, the principles that apply to building efficient client-side functionality using JavaScript have remained fairly constant. Having an understanding of how browsers load JavaScript code and perform Document Object Model (DOM) lookups is an important skill for any Ajax developer to learn. Taking advantage of simple development tips such as caching objects returned from document.getElementById lookups, creating function pointers when a function is called multiple times in looping operations, avoiding property access methods whenever possible and using script combining can result in less client-side memory consumption and improve overall Ajax application performance (visit http://channel9.msdn.com/posts/gioker84/Application-Performance-with-IE8 for an excellent set of JavaScript performance tips).

Understanding the rules that apply to JavaScript and Ajax application performance is also important. Steve Souders, one of the pioneers in Ajax performance techniques, provided an important set of rules that should be followed in his book High Performance Web Sites: Essential Knowledge for Front-End Engineers. Key rules discussed include minimizing HTTP requests, using a Content Delivery Network (CDN), adding expiration headers, GZipping components, placing scripts at the bottom of a page and minifying JavaScript files. By following these rules Ajax applications can take advantage of modern browser features and load pages more quickly.

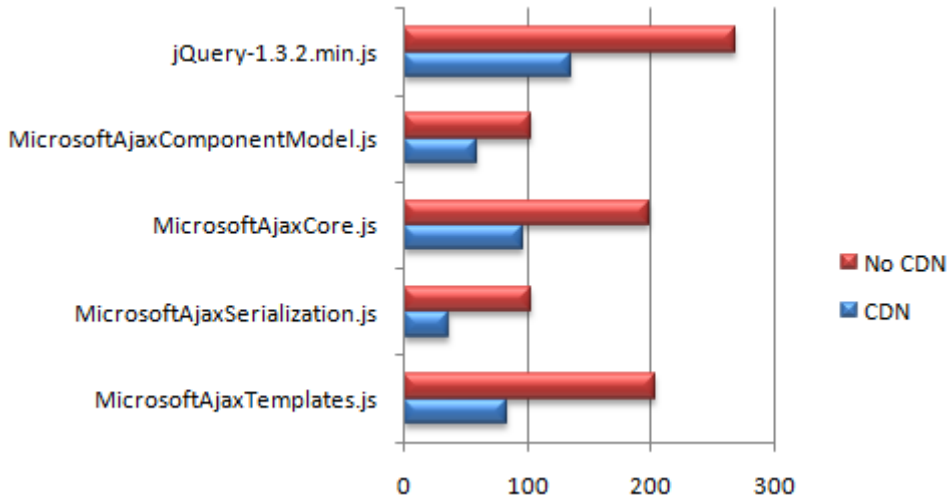
This white paper will focus on the ASP.NET Ajax Library and discuss different development rules, techniques and tools that can be used to enhance Ajax application performance. Topics covered include using the ASP.NET Ajax Content Delivery Network (CDN) to load scripts more quickly and using the ASP.NET Ajax Library's Script Loader to load scripts in parallel or behind the scenes using lazy loading techniques. Different Ajax application performance tools and techniques will also be discussed along with IIS7 compression and cache settings that can be used to enhance Ajax application performance.

The Microsoft Ajax Content Delivery Network (CDN)

Content Delivery Networks (CDNs) have become a popular way to distribute audio, video and image files consumed by applications around the world. A CDN consists of a network of servers each containing copies of the media to be distributed. These servers are placed at strategic network points around the world allowing a client to access media more quickly and with maximum bandwidth.

In addition to using CDNs to stream media files, they're also being used to distribute JavaScript files which provides several key performance benefits to Ajax applications including faster load times and enhanced caching. The Microsoft Ajax CDN is a free service that places "edge cache" servers at strategic locations around the world to allow scripts to be loaded more efficiently by Ajax applications. Instead of making multiple hops to retrieve a script from a hosted server (at times located on a different continent), scripts loaded using the Microsoft Ajax CDN can be loaded quickly and with a minimal number of hops between networks. The performance examples that follow provide an overall view of CDN performance and are intended to show the big picture as opposed to following more rigorous testing practices.

A comparison of download times achieved when hosting ASP.NET Ajax scripts on a single shared host server in the United States versus retrieving the same scripts from the Microsoft Ajax CDN are shown in Table 1.

Location	Load Times (in milliseconds)
USA	

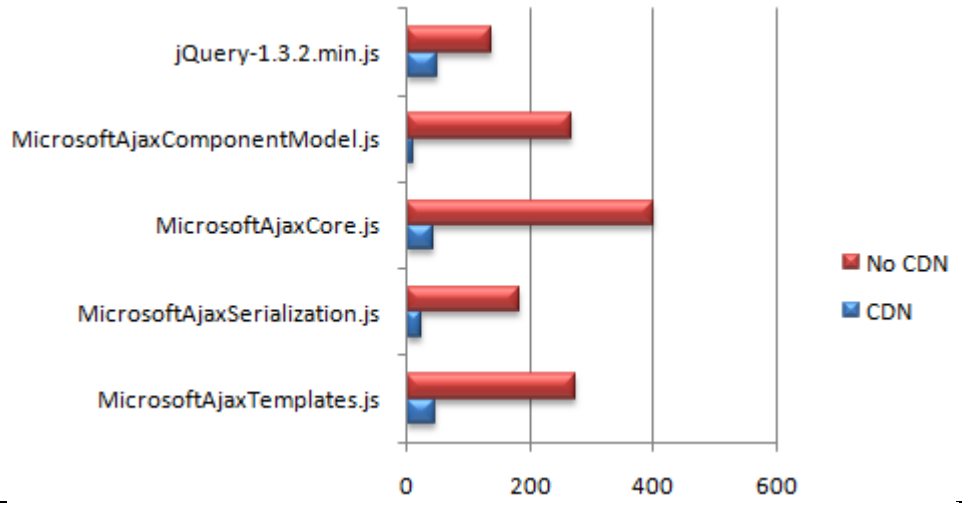
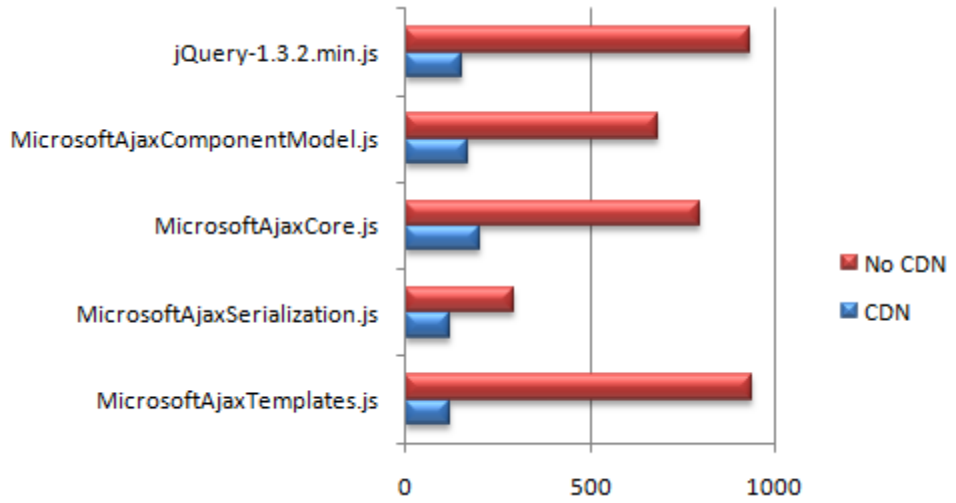
Switzerland	
England	

Table 1. Comparing load times for scripts hosted in a shared host environment to scripts hosted on Microsoft Ajax CDN.

The results from Table 1 with a CDN value of "No CDN" were run against a web page located on a shared host server in the United States that retrieves scripts from that same server. The results shown with a value of "CDN" were run against a page on the same shared host server but the scripts were retrieved from the Microsoft CDN. The retrieval times shown in Table 1 will certainly change depending upon where the tests are run, when they're run, the type of computer used, available bandwidth and the shared host server that is called. However, even with those caveats it's clear to see the performance benefits that the CDN can offer.

The amount of time it takes to retrieve a script can be affected by many different factors including the number of network hops that are required to get to the script. An increase in network hops doesn't necessarily indicate a slower route since some networks hide internal hops but it does give an overall indication of how messages are traveling to and from a client. Table 2 shows how the network hops compare for calling a shared host server versus getting the scripts from the Microsoft Ajax CDN.

Location	CDN	Network Hops
United States	No	10
	Yes	6
Switzerland	No	13
	Yes	11
England	No	13
	Yes	11

Table 2. Network hops required to retrieve scripts used in an JavaScript application.

The time it takes between network hops can also be factored into the overall performance equation. Figure 1 shows a simple tracert command run against a shared host server and the ASP.NET Ajax CDN. You can see that each segment of the CDN performs quite well.

Tracing the route to a Shared Host Server

```
  1     1 ms     1 ms     1 ms  192.168.1.1
  2     *        *        *     Request timed out.
  3    12 ms    11 ms    10 ms  ip68-2-1-109.ph.ph.cox.net [68.2.1.109]
  4    23 ms    13 ms    62 ms  70.169.73.53
  5    28 ms    23 ms    23 ms  langbbr01-ae0.r2.la.cox.net [68.1.0.232]
  6    83 ms    82 ms    84 ms  66.192.245.58
  7    87 ms   111 ms    84 ms  v11.cd2.peak10.net [66.129.64.9]
  8    81 ms    84 ms    83 ms  66.129.112.188
  9    86 ms    84 ms    84 ms  206.72.122.189
 10    87 ms    84 ms    90 ms  66.129.67.130
```

Tracing the route to the ASP.NET Ajax CDN

```
  1     1 ms     1 ms     1 ms  192.168.1.1
  2     *        *        *     Request timed out.
  3    11 ms    11 ms    12 ms  ip68-2-1-109.ph.ph.cox.net [68.2.1.109]
  4    15 ms    24 ms    13 ms  70.169.73.45
  5    24 ms    52 ms    39 ms  langbbrj02-as0.r2.la.cox.net [68.1.1.231]
  6    30 ms    26 ms    24 ms  98.174.31.233
```

Figure 1. Comparing network hop speed when calling a shared host server and the Microsoft Ajax CDN

The time it takes to load scripts can be further enhanced through caching. When multiple Ajax applications reference Microsoft Ajax CDN files the scripts that are loaded will be cached in the client browser and re-used. For example, if a client accesses an application on the contoso.com domain that uses the Microsoft Ajax CDN to load scripts, all of the scripts will be cached locally and re-used when the client hits the xyz.com domain that also relies on the CDN. Every application that takes advantage of the Microsoft Ajax CDN will see this caching benefit without any extra effort on the developer's part. Different Microsoft web sites that receive millions of visits will also start to use the CDN which means that many browsers around the world will have the scripts served by the Microsoft Ajax CDN already cached.

Loading ASP.NET Ajax Scripts

The Microsoft Ajax CDN hosts several different types of scripts at the http://ajax.microsoft.com domain including ASP.NET Ajax scripts and jQuery scripts. Accessing Microsoft Ajax CDN scripts can be accomplished by including a standard script tag in a page that references Ajax.microsoft.com:

```
<script
  src="http://ajax.microsoft.com/ajax/beta/0910/MicrosoftAjaxTemplates.js"
  type="text/javascript">
</script>
```

The 0910 segment of the script tag's URI defines the version of the script that should be loaded allowing multiple versions to be hosted over time by the CDN. This means that as new versions of scripts are released the previous versions will be kept intact so that applications don't break. All of the scripts supported by the ASP.NET Ajax script library are available on the CDN including key scripts such as MicrosoftAjax.js, MicrosoftAjaxDataContext.js, MicrosoftAjaxTemplates.js, MicrosoftAjaxWebForms.js and Start.js to name a few.

Microsoft Ajax CDN scripts can also be loaded automatically by the ScriptManager control in ASP.NET 4. ASP.NET 4 adds a new property to the ScriptManager named EnableCdn that can be used to load scripts from the Microsoft Ajax CDN instead of from the application's host server in order to leverage performance and caching benefits:

```
<asp:ScriptManager
  id="sm"
  runat="server"
  EnableCdn="true" />
```

Debug versions of the scripts are also available on the Microsoft Ajax CDN which is useful when developers need to view the code in a more legible format, step through it in Visual Studio or debug scripts using Internet Explorer 8. A debug version of a script can be loaded by adding "debug" to the end of the script name:

```
<script
  src="http://ajax.microsoft.com/ajax/beta/0909/MicrosoftAjaxTemplates.debug.js"
  type="text/javascript"></script>
```

Loading jQuery Scripts

The Microsoft Ajax CDN also allows jQuery scripts to be loaded including the standard jQuery library as well as the jQuery Validation Library:

```
<script src="http://ajax.microsoft.com/ajax/jquery/jquery-1.3.2.min.js"
  type="text/javascript"></script>

<script
  src="http://ajax.microsoft.com/ajax/jquery.validate/1.5.5/jquery.validate.min.js"
```

```
type="text/javascript"></script>
```

Several jQuery script formats are available including the full version, minified version, vsdoc version (used for intellisense in Visual Studio) and the minified vsdoc version:

- jquery-1.3.2.js
- jquery-1.3.2.min.js
- jquery-1.3.2-vsdoc.js
- jquery-1.3.2.min-vsdoc.js

Available jQuery Validation Library script formats include:

- jquery.validate.js
- jquery.validate.min.js
- jquery.validate-vsdoc.js

By using the Microsoft Ajax CDN scripts can be loaded more quickly and efficiently with a minimal number of network hops. Applications using CDN scripts can also take advantage of browser caching across domains to avoid unnecessary loading of scripts. Additional CDN benefits include a reduction in the number of files that have to be deployed for an application as well as a simplified way to handle script versions as an application evolves over time. Visit http://www.asp.net/ajaxlibrary/cdn.ashx for additional details on the Microsoft Ajax CDN.

The ASP.NET Ajax Library Script Loader

Minimizing the number of scripts required to load a JavaScript application is a common technique used to speed up the load time of a page so that end users aren't waiting for pages to render. Additional scripts used in a page can be retrieved as needed behind the scenes and executed only when required as opposed to retrieving and executing them when the page first loads. While this sounds like a straightforward process it can become rather complex when scripts depend upon other scripts loading at the proper time and in the proper order.

Today's modern browsers are capable of loading multiple scripts (more than two from the same domain) in parallel to enhance performance unlike browsers of old that could only handle loading two scripts in parallel at a time without resorting to additional coding tricks such as dynamic script attachment (creating <script> blocks programmatically), using document.write statements or iframes. It goes without saying that loading scripts in parallel (see Figure 2) can have a significant impact on how fast an application loads since the serial bottleneck is eliminated (see Figure 3).

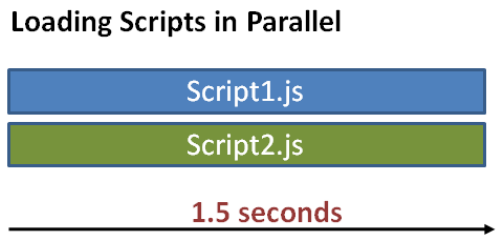


Figure 2. Loading scripts in parallel can cut down the overall time it takes to load an application but can add additional complexities.

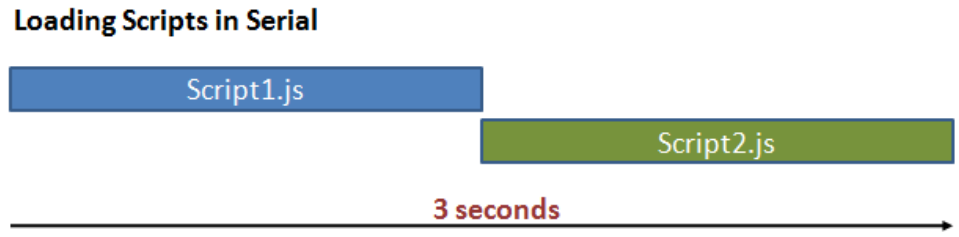


Figure 3. Loading scripts in serial results in a bottle-neck that can slow down the time it takes to start an application.

While parallel script loading is a welcome performance enhancement it can result in timing issues that ultimately cause problems in applications. For example, if Foo.js loads in parallel with another script it's dependent upon named FooDependency.js, problems can arise when Foo.js loads first and tries to execute. Several solutions exist to alleviate this problem such as combining scripts into a single file or handling each script's onreadystatechange event to know when it's been downloaded.

Each of these solutions can get the job done but having a more flexible, re-useable and readily available Script Loader framework that works cross-browser is certainly preferable. The ASP.NET Ajax Library

provides a robust Script Loader that can help minimize the number of scripts required to start an application, handle loading other required scripts and their dependencies in parallel and simplify the timing issues associated with parallel script loading. All of this functionality is packaged into a script named Start.js which weighs in at a lightweight 11KB.

Table 3 compares the performance of loading a page with 5 scripts defined using the <script> element to a page that loads the same 5 scripts using the Script Loader (note that the Script Loader uses the Start.js script making a total of 6 scripts in the page).

Script Loading Technique	Total Scripts Loaded	Time (in seconds)
<script> element	5	1.6 sec
Script Loader	6	1.1 sec

Table 3. Comparing page load times where traditional <script> elements are used versus the Script Loader.

Although only a few scripts were loaded for this test you can see that the page using the Script Loader loaded ½ second faster. As additional scripts are added the load time difference between the two pages will be even greater. The first page (load time of 1.6 seconds) defined scripts in the following way:

```
<script type="text/javascript"
    src="Scripts/MicrosoftAjax/MicrosoftAjaxCore.js"></script>
<script type="text/javascript"
    src="Scripts/MicrosoftAjax/MicrosoftAjaxComponentModel.js"></script>
<script type="text/javascript"
    src="Scripts/MicrosoftAjax/MicrosoftAjaxSerialization.js"></script>
<script type="text/javascript"
    src="Scripts/MicrosoftAjax/MicrosoftAjaxTemplates.js"></script>
<script type="text/javascript" src="Scripts/jQuery/jquery-1.3.2.min.js" ></script>
```

The second page (load time of 1.1 seconds) loaded the same scripts using the Script Loader:

```
<script type="text/javascript" src="Scripts/MicrosoftAjax/start.js"></script>
<script type="text/javascript">
    Sys.require([Sys.scripts.Core, Sys.scripts.ComponentModel,
                 Sys.scripts.Serialization, Sys.scripts.Templates,
                 Sys.scripts.jQuery]);
</script>
```

The next section provides details and examples of how the Script Loader can be used to load ASP.NET Ajax Library scripts, jQuery scripts and custom scripts.

Using the ASP.NET Ajax Script Loader

The Start.js Script Loader provides out-of-the-box support for loading ASP.NET Ajax Library or jQuery scripts as well as custom scripts needed by an application. It can be loaded from a local server or from the ASP.NET Ajax CDN:

```
<script type="text/javascript" src="Scripts/MicrosoftAjax/start.js"></script>

<script type="text/javascript"
  src="http://ajax.microsoft.com/ajax/beta/0910/Start.js"></script>
```

When Start.js is loaded from a local path it looks for required ASP.NET Ajax Library scripts that are stored locally as well. When the CDN version of Start.js is used it retrieves ASP.NET Ajax Library scripts and jQuery scripts from the CDN allowing network performance and caching benefits to be leveraged.

Once Start.js is defined in a page it can be used to load ASP.NET Ajax Library scripts, jQuery scripts and even custom scripts used by an application. Required application scripts can then be loaded using Sys.require which accepts three parameters including an array containing components and scripts to load, the callback function to execute once the scripts are ready and a user context object that can be used to pass state information. An example of passing an array of the required components and scripts to Sys.require is shown next:

```
<script type="text/javascript">
    Sys.require([Sys.components.dataView, Sys.scripts.jQuery]);

    Sys.onReady(function() {
        //Use dataView object
    });
</script>
```

The Script Loader will retrieve the required scripts in parallel and their associated objects and functions can be used once onReady is invoked. Alternatively, Sys.require provides a second parameter that can be used to define what actions should be taken once the required scripts are loaded and the DOM is ready:

```
<script type="text/javascript">
    Sys.require([Sys.components.dataView, Sys.scripts.jQuery], function() {
        //Use dataView object
    });
</script>
```

Table 4 shows the built-in Sys.scripts collection members as well as the scripts they're associated with in the ASP.NET Ajax Library while Figure 4 shows ASP.NET Ajax Library script relationships.

Sys.scripts Value	Script	Description
AdoNet	MicrosoftAjaxAdoNet.js	Allows client-side code to interact with ADO.NET Data Services.
ApplicationServices	MicrosoftAjaxApplicationServices.js	Provides access to ASP.NET profile, authentication and role services.
ComponentModel	MicrosoftAjaxComponentModel.js	Provides behavior and component functionality.
Core	MicrosoftAjaxCore.js	Base script that contains core ASP.NET Ajax Library functionality and JavaScript language enhancements.
DataContext	MicrosoftAjaxDataContext.js	Contains the DataContext and AdoNetDataContext client-side classes.
Globalization	MicrosoftAjaxGlobalization.js	Contains language globalization functionality for dates, times and numbers.
History	MicrosoftAjaxHistory.js	Provides browser history functionality for Ajax applications.
jQuery	jquery-1.3.2.min.js	Minified jQuery library script.
jQueryValidate	jquery.validate.min.js	Minified jQuery Validate script used to validate form elements.
Network	MicrosoftAjaxNetwork.js	Contains core networking features used to communicate with remote sites and services.
Serialization	MicrosoftAjaxSerialization.js	Provides object serialization and deserialization functionality.
Templates	MicrosoftAjaxTemplates.js	Provides client-side template rendering functionality.
WebServices	MicrosoftAjaxWebServices.js	Contains a web service proxy class used to call web services.

Table 4. Sys.scripts members that can be used to load required scripts using the ASP.NET Ajax Library's Script Loader.

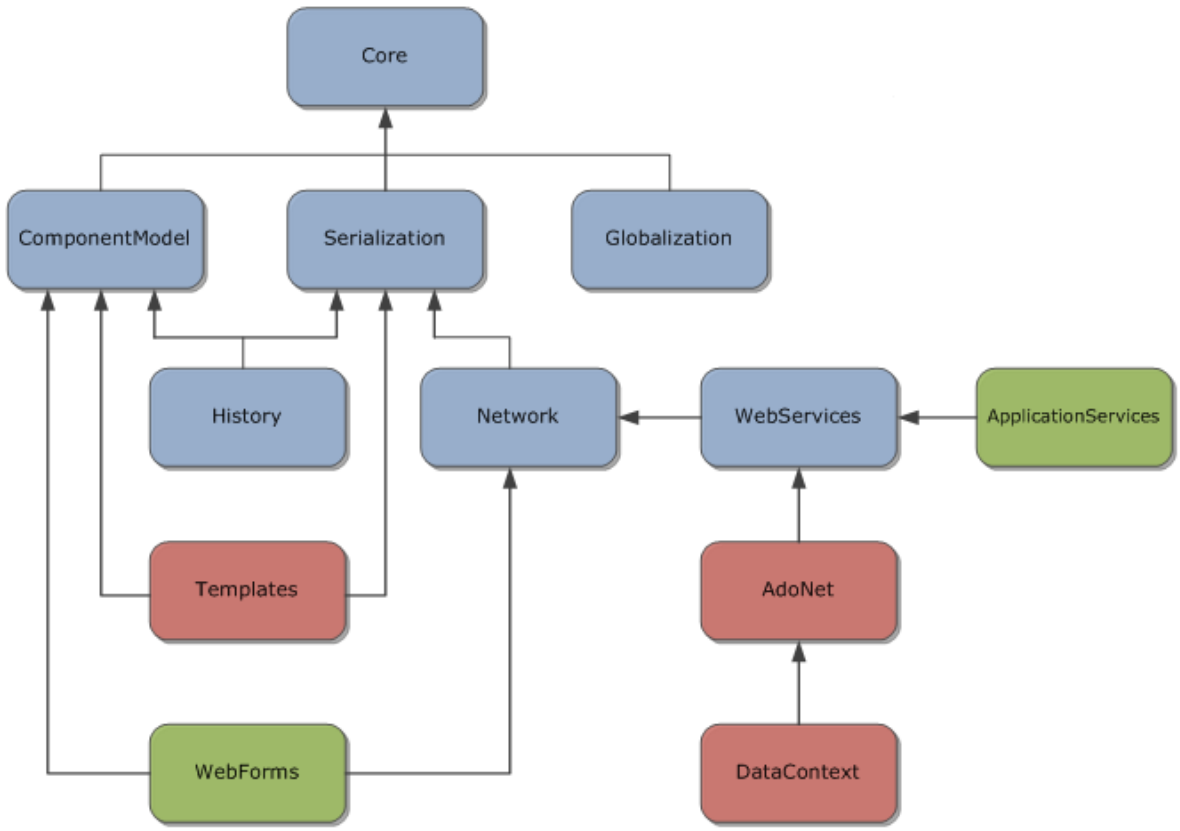


Figure 4. ASP.NET Ajax Library scripts and their dependencies.

The Script Loader automatically handles loading any required scripts as well as dependencies and has a built-in mechanism to prevent scripts from being loaded more than once as well. If a value of Sys.require([Sys.scripts.DataContext]) is passed the Script Loader will handle loading 4 other scripts behind the scenes in addition to MicrosoftAjaxDataContext.js including MicrosoftAjaxComponentModel.js, MicrosoftAjaxSerialization.js, MicrosoftAjaxWebServices.js and MicrosoftAjaxAdoNet.js. This reduces the amount of code required to get a page running and minimizes the complexities associated with managing parallel loading of script files and their dependencies. If the scripts were also needed by another script they'll only be loaded once.

Sys.require allows components used by an application and their script dependencies to be loaded as well. Built-in values include adoNetDataContext, adoNetServiceProxy, dataContext and dataView. Passing a value of Sys.components.dataView to Sys.require will cause scripts that support the DataView object to be loaded automatically. Scripts that are loaded when the DataView component is referenced can be viewed using tools such as the Internet Explorer Developer Tools (see Figure 5).

http://localhost:49571/WebSite/Scripts/MicrosoftAjax
MicrosoftAjaxComponentModel.debug.js
MicrosoftAjaxCore.debug.js
MicrosoftAjaxGlobalization.debug.js
MicrosoftAjaxSerialization.debug.js
MicrosoftAjaxTemplates.debug.js

Figure 5. Scripts loaded when a DataView component is loaded using the Script Loader.

Loading Custom Scripts

The ASP.NET Ajax Library Script Loader really stands out when it comes to loading custom scripts and associated dependencies used by an application. It can be used to load both debug and release versions of scripts and will automatically ensure that dependencies are loaded at the proper time. To load a custom script the Sys.loader.defineScripts function can be called. An example of passing release and debug URLs along with dependencies to Sys.loader.defineScripts is shown next:

```
Sys.loader.defineScripts({
    releaseUrl: "../Scripts/ACT/{0}.js",
    debugUrl: "../Scripts/ACT/{0}.js"
},
    [
        { name: "ACTWatermark",
            executionDependencies: ["ACTExtenderBase"],
            behaviors: ["AjaxControlToolkit.Watermark"],
            isLoaded: !!(window.AjaxControlToolkit && AjaxControlToolkit.Watermark)
        },
        { name: "ACTExtenderBase",
            executionDependencies: ["ACTCommon"],
            isLoaded: !!(window.AjaxControlToolkit && AjaxControlToolkit.ControlBase)
        },
        { name: "ACTCommon",
            executionDependencies: ["ComponentModel", "Globalization"],
            isLoaded: !!(window.AjaxControlToolkit &&
              AjaxControlToolkit.TextBoxWrapper)
        }
    ]
);
```

This example loads the ACTWatermark.js (a client-side component available in the Ajax Control Toolkit), ACTExtenderBase.js and ACTCommon.js scripts in the proper order. The releaseUrl and debugUrl path used to load each script is defined first followed by a list of custom scripts that must be loaded. Each script definition provides a name that is plugged into the releaseUrl or debugUrl properties as appropriate, an array of dependencies, behaviors available in the script (intended for controls), that should be used as well as a check to ensure that the script has been loaded (which is also used to prevent the script from loading more than once).

The final script named ACTCommon shows how flexible the Script Loader is when it comes to loading additional ASP.NET Ajax Library (or even jQuery) dependencies. The executionDependencies property for ACTCommon defines that MicrosoftAjaxComponentModel.js and MicrosoftAjax.Globalization.js should be loaded before ACTCommon can be used at runtime.

The different scripts can be used by including the path to the JavaScript definition file (ACTRegisterExtended.js in the example that follows) in the page and then referencing the necessary components. The Watermark component defined in the ACTWatermark script is used in this example.

```
<script type="text/javascript"
   src="http://ajax.microsoft.com/ajax/beta/0910/Start.js"></script>
<script src="../Scripts/ACT/ACTRegisterExtended.js"
   type="text/javascript"></script>

<script type="text/javascript">
    Sys.require([Sys.components.watermark, Sys.components.dataView,
      Sys.scripts.jQuery], function() {
         //Scripts loaded and Watermark and DataView are available to use
    });
</script>
```

Script Combining

In addition to loading scripts and their dependencies, the Script Loader can also take advantage of script combining which can reduce the number of network calls made to the server – another performance benefit that is important. Instead of loading all ASP.NET Ajax Library scripts individually, a single script can be loaded where appropriate. For example, the following Sys.require syntax will result in MicrosoftAjax.js being retrieved from the server instead of 7 individual scripts:

```
Sys.require([
   Sys.scripts.Core,
   Sys.scripts.ComponentModel,
   Sys.scripts.Network,
   Sys.scripts.Serialization,
   Sys.scripts.History,
   Sys.scripts.Globalization,
   Sys.scripts.WebServices
]);
```

How is it possible for the Script Loader to know when to combine scripts? First of all, the combined script must be available on the server. The Script Loader is a client-side component and doesn't have the ability to perform any server-side script combining as a result. However, by defining what scripts are inside of a "combined" script, the Script Loader can take advantage of script combining. The previous code sample retrieves MicrosoftAjax.js from the server due to a "contains" definition within Start.js. When the Script Loader sees all of the scripts defined in "contains" are needed it automatically leverages script combining. It's important to note that the same technique can also be used for custom scripts.

```
name: "MicrosoftAjax",
releaseUrl: "%/MicrosoftAjax.js",
debugUrl: "%/MicrosoftAjax.debug.js",
codependencies: null,
contains: ["Core", "ComponentModel", "History", "Serialization", "Network",
           "WebServices", "Globalization"]
```

Adding multiple Sys.require calls into a page will load scripts individually and eliminate any script combining benefits:

```
Sys.require([Sys.scripts.Core]);
Sys.require([Sys.scripts.ComponentModel]);
Sys.require([Sys.scripts.Network]);
Sys.require([Sys.scripts.Serialization]);
Sys.require([Sys.scripts.History]);
Sys.require([Sys.scripts.Globalization]);
Sys.require([Sys.scripts.WebServices]);
```

Using the Script Loader when Debugging

It's common for developers to use debug versions of the ASP.NET Ajax Library scripts and custom scripts to enable intellisense and a better debugging experience in Visual Studio. While using debug scripts isn't recommended for production applications due to their size and impact on performance, having the ability to easily switch between release scripts and debug scripts is useful in a variety of situations.

The ASP.NET Ajax Library provides a quick and simple way to switch between release and debug versions of scripts by using the Sys.debug property. Rather than changing the names of the scripts that are passed to the Script Loader, Sys.debug can be set to a value of true causing the Script Loader to automatically load scripts ending with "debug.js".

```
<script type="text/javascript"
   src="http://ajax.microsoft.com/ajax/beta/0910/Start.js"></script>

<script type="text/javascript">

    Sys.debug = true;

    Sys.require([Sys.components.dataView,Sys.scripts.jQuery], function() {
         //Scripts loaded and Watermark and DataView are available to use
    });
</script>
```

Figure 6 shows the scripts that are loaded by the Script Loader when Sys.debug is set to true while Figure 7 shows the scripts loaded when debugging is turned off.

MicrosoftAjaxComponentModel.debug.js
MicrosoftAjaxCore.debug.js
MicrosoftAjaxSerialization.debug.js
MicrosoftAjaxTemplates.debug.js
Start.js

http://ajax.microsoft.com/ajax/jquery
jquery-1.3.2.js

Figure 6. Setting Sys.debug to true causes the ASP.NET Ajax Library Script Loader to automatically load debug scripts without changing any other code.

MicrosoftAjaxComponentModel.js
MicrosoftAjaxCore.js
MicrosoftAjaxSerialization.js
MicrosoftAjaxTemplates.js
Start.js

http://ajax.microsoft.com/ajax/jquery
jquery-1.3.2.min.js

Figure 7. Scripts loaded when Sys.debug is set to false (the default).

This works for the ASP.NET Ajax Library's scripts, custom scripts and even jQuery scripts as long as the location of the debug scripts is properly defined. Start.js contains the following code to define release and debug script paths for ASP.NET Ajax Library scripts. The % character is used to define that the scripts are relative to the location of Start.js.

```
loader.defineScripts({
    releaseUrl: "%/MicrosoftAjax{0}.js",
    debugUrl: "%/MicrosoftAjax{0}.debug.js",
    executionDependencies: ["Core"]
}
```

When using jQuery, the minified version of jQuery is used as the release version of the script whereas the formatted version is used for the debug version in Start.js:

```
loader.defineScripts(null, [
 { name: "jQuery",
   releaseUrl: ajaxPath + "jquery/jquery-1.3.2.min.js",
   debugUrl: ajaxPath + "jquery/jquery-1.3.2.js",
   isLoaded: !!window.jQuery
 }
```

As with web.config debug settings, it's important to switch the Sys.debug setting to false (the default) before moving an application to production so that the application achieves the best performance possible. Forgetting to switch this setting will result in larger JavaScript files being loaded which will of course slow down the loading of the application.

Using the Script Loader's Lazy Loading Feature to increase performance

Many JavaScript applications rely on features that aren't required to be available when a page initially loads. For example, an application may have a print button that a user can click to print the page. The script that is executed as the print button is clicked doesn't need to be loaded when the page first loads since it won't be used until the end user interacts with the page. Loading the script after the page has loaded and rendered makes more sense in this type of scenario.

Loading scripts later in the page life-cycle is referred to as "lazy loading" and can have a significant impact on the start-up time of a page and allow the user to interact with the application more quickly. Although lazy loading of scripts was possible in the past using different JavaScript frameworks or custom coding techniques, the ASP.NET Ajax Library's Script Loader makes it extremely straightforward to lazy load scripts in an application to enhance performance.

The Script Loader provides built-in support for lazy loading through Sys.require. Previous examples have shown how Sys.require can be used to load scripts and their dependencies immediately as a page loads. In cases where scripts don't have to be loaded upfront as a page loads, Sys.require can also be used to lazy load scripts as users click a button or perform another type of action. The following code sample demonstrates how the Script Loader can use lazy loading to retrieve additional scripts as a user interacts with a page. The key part of the code sample is shown in the Print function.

```
<script src="../Scripts/MicrosoftAjax/start.js"
   type="text/javascript"></script>
<script src="../Scripts/CustomScripts/RegisterPrintScripts.js"
   type="text/javascript"></script>

<script type="text/javascript">
    function Print() {
        Sys.require([Sys.scripts.printScript]);

        Sys.onReady(function() {
            //Handle print operation
        });
    }
</script>

<input type="button" onclick="Print()" value="Print Page"/>
```

As the page loads only Start.js and RegisterPrintScripts.js will be retrieved from the server causing the page to load quickly. As the Print Page button is clicked the Print function is called which uses Sys.require to lazy load a printing script and any associated scripts behind the scenes. There will be a slight delay while the print script and its dependencies are loaded but it's a onetime delay since the scripts will be cached for any future visits to the page.

By placing Sys.require in the Print function the custom Print.js script and required ASP.NET Ajax Library scripts will be lazy loaded behind the scenes by the Script Loader. Once they're available, Sys.onReady will be called and the code that uses the print scripts can be executed. It's important to note that Sys.onReady will only be called once the scripts defined in the Print function are loaded. It won't be called when the page initially loads in this case since it is placed inside of Print.

JavaScript Application Performance Tools

There are many features that Ajax application developers can take advantage of to enhance the performance of an Ajax application such as using the ASP.NET Ajax CDN and ASP.NET Ajax Library Script Loader features. In addition to using ASP.NET Ajax Library features and following recommended coding practices, several different tools can also be used to enhance and analyze Ajax application performance. Key performance tools discussed in this section include the Download Time Optimizer (Doloto), the ASP.NET Ajax Minifier, the Internet Explorer 8 JavaScript Profiler and Internet Information Services (IIS) 7 GZip compression feature.

The Download Time Optimizer (Doloto)

As more and more applications embrace JavaScript and Ajax features and client-side rendering of data, the amount of code that has to be loaded by the browser in order for an application to run tends to increase. Although the ASP.NET Ajax Library provides powerful features such as the Script Loader to enable faster loading of pages and lazy loading of scripts, tools such as the Download Time Optimizer (Doloto) can play an important role in optimizing how code is loaded. The Doloto website (http://research.microsoft.com/en-us/projects/doloto) provides the following explanation about the tool:

"Doloto is a system that analyzes application workloads and automatically performs code splitting of existing large Web 2.0 applications. After being processed by Doloto, an application will initially transfer only the portion of code necessary for application initialization. The rest of the application's code is replaced by short stubs -- their actual function code is transferred lazily in the background or, at the latest, on-demand on first execution."

Doloto goes through three main phases including a training phase which creates an "access profile" defining how code is being used, a rewriting phase that optimizes scripts for on-demand code loading and a prefetch phase that performs background loading of scripts as the application is running.

Optimizations are done by analyzing how a Web application's code is used and then performing code splitting to "cluster" JavaScript functions together to create the "access profile" mentioned previously.

To use Doloto you'll need to start the program, change the browser's proxy to use http://localhost:8888, click Doloto's Profile button and then browse to the site that Doloto should inspect using the browser. From there Doloto uses proxy-based code profiling to create an "access profile" for the site and organizes JavaScript functions into optimized "clusters" (see Figure 8). Once the clusters are created the estimated size savings can be displayed and the existing website scripts can be rewritten into an optimized format that takes advantage of background loading.

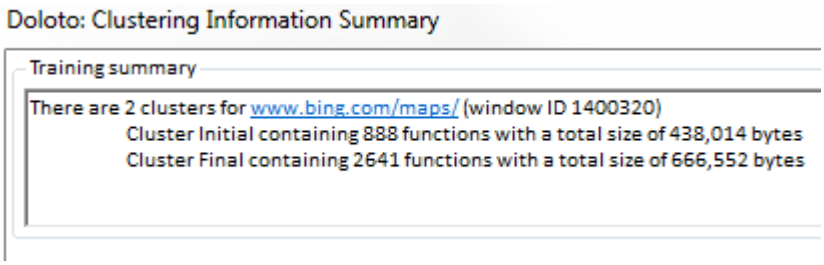


Figure 8. Clusters created by Doloto for http://www.bing.com/maps

Running Doloto against several large Ajax sites including Live.com (now Bing.com) and Google Spreadsheets generates a 20-40% reduction in the size of scripts loaded and decreases the amount of time it takes before an end user can start using the application. Figure 9 shows the percentage decrease in download size after running Doloto against several sites.

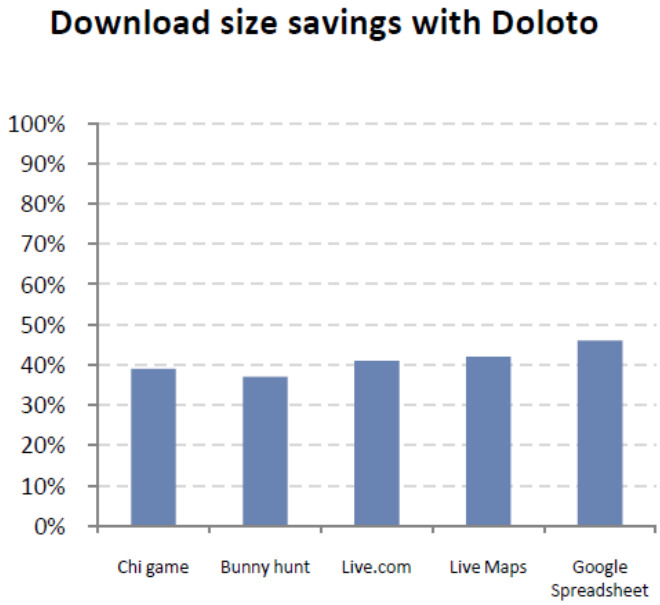


Figure 9. Doloto minimizes the number of scripts that have to be loaded upfront to run an Ajax application. This graph shows the percentage decrease in download size after Doloto optimizations.

Microsoft Ajax Minifier

The ASP.NET Ajax Library scripts come in two formats including a formatted version used for debugging purposes and a minified version used for production applications. The minified version of the ASP.NET Ajax Library scripts provides a crunched file (as opposed to a compressed file) that enhances performance in production applications. Microsoft has been using an internal tool to crunch scripts that is now available publicly called the Microsoft Ajax Minifier. This tool removes unnecessary content from a JavaScript file to reduce its size and as a result allows scripts to be downloaded faster by applications.

The Microsoft Ajax Minifier tool can be used directly from the command line and offers a wide range of functionality through various command-line arguments. After installing the program an *Microsoft Ajax Minifier Command Prompt* will appear in Windows programs menu that can be used to run commands. To minify a script the following command can be run:

```
ajaxmin input.js –o output.js
```

This command performs standard crunching which can be strengthened even more using the /H command line switch:

```
ajaxmin input.js –o output.js /H
```

There are a variety of other command-line switches included such as /M for adding code for Macintosh Safari quirks, /Z to terminate a crunched stream with a semi-colon and /Eo to define the output file encoding scheme.

In addition to the command-line version, Microsoft Ajax Minifier also includes an assembly that can be used in custom .NET programs as well as an MSBuild task that can automatically create minified versions of scripts as Visual Studio project are built. The tool's help documentation includes a step-by-step guide for using the assembly and MSBuild task in applications.

An example of using the Microsoft Ajax Minifier to crunch a script is shown next. The first example displays the formatted version of a script while the second code sample displays the minified version. All of the code in the minified version is on a single line in the output file that is generated.

Formatted Script:

```
Sys.require([Sys.components.imageView, Sys.scripts.jQuery]);

Sys.onReady(function() {
    var control = Sys.get("$images");
    if (control) {
       Sys.Observer.setValue(control, "data", gallery);
```

```
    }
    else {
        $("#images").imageView({ data: gallery });
    }
});
```

Minified Script:

```
Sys.require([Sys.components.imageView,Sys.scripts.jQuery]);Sys.onReady(function(){var
control=Sys.get("$images");if(control)Sys.Observer.setValue(control,"data",gallery);el
se $("#images").imageView({data:gallery})})
```

Internet Explorer JavaScript Profiler

Writing optimal code that performs well across multiple browsers it hard to do without being able to take a detailed look into what the code is doing. Fortunately, Internet Explorer 8 introduces a Developer Tools feature that allows HTML source code to be viewed and inspected, CSS to be modified and JavaScript code to be viewed, debugged and profiled. The JavaScript Profiler tool in the IE8 Developer Tools provides a detailed analysis of how functions are being called within an application, the time the functions take as well as the number of times they're called.

The IE8 JavaScript Profiler can be accessed by selecting Tools → Developer Tools from the menu or by pressing F12. It can be opened in a stand-alone window mode or within the current browser window. Once opened, the profiler can be accessed by selecting the Profiler tab. To profile an existing application click the Start Profiling button and navigate to the page that should be profiled. Once the page loads and any necessary testing actions have been performed, click the Stop Profiling button to view the results. Figure 10 shows an example of the IE8 JavaScript Profiler in action.

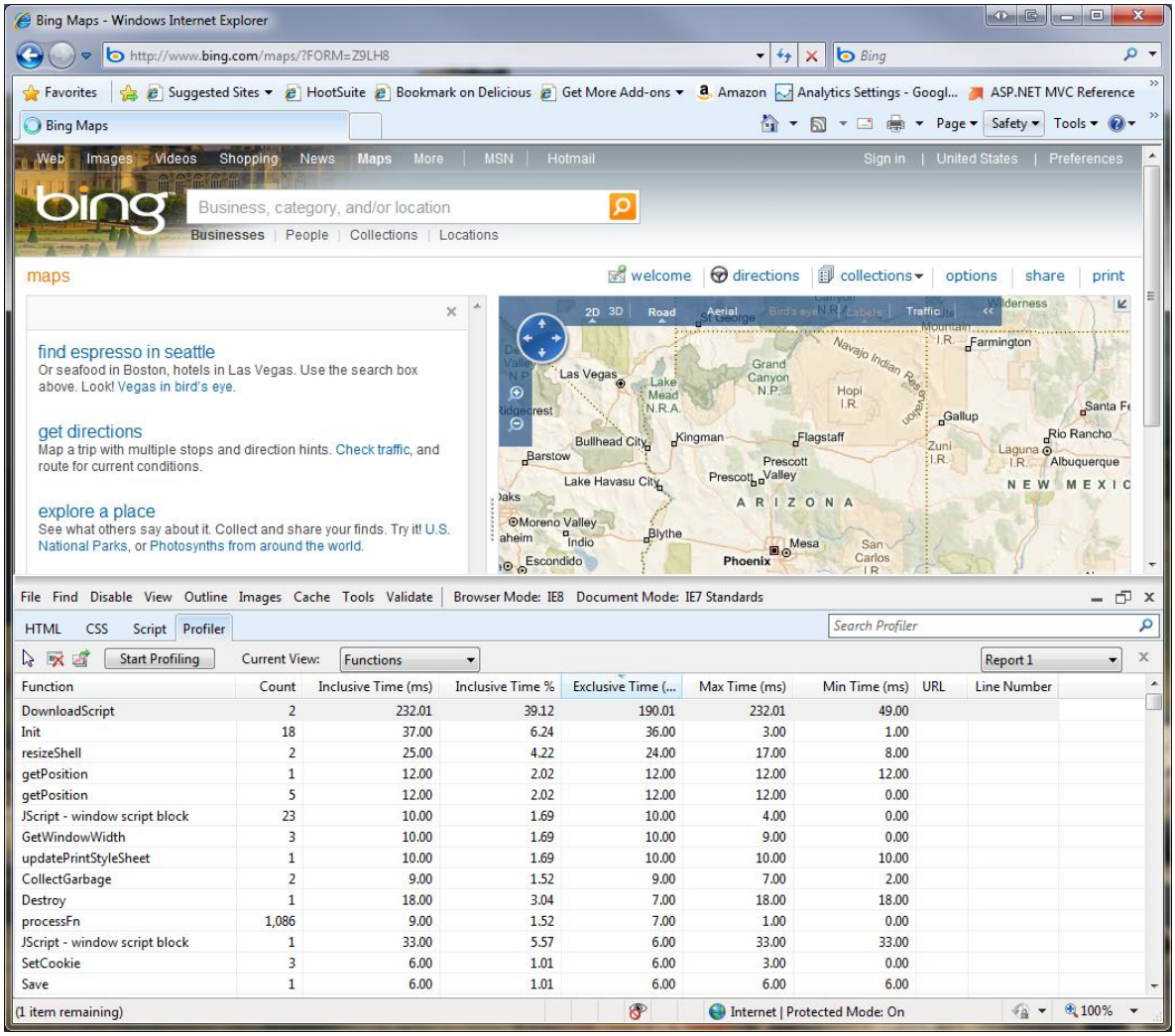


Figure 10. Using the JavaScript Profiler to view calls made on the http://www.bing.com/maps site.

Once the JavaScript Profiler data loads, information about each function call can be viewed and analyzed. For example, the data in Figure 10 shows that the Init function was called 18 times and that the code spent 6.24% of the total time in this function and its children. By looking at the different columns it's easy to see where potential optimizations can be made in code and where bottlenecks may exist that need to be modified to provide better performance. The JavaScript Profiler is capable of generating multiple reports and data produced by tool can be exported as a CSV file for further analysis. By using the JavaScript Profiler a more in-depth look into the flow of JavaScript calls can be generated which provides an excellent way to make code adjustments and increase application performance.

Internet Information Server 7 Compression and Caching Options

Internet Information Services (IIS) provides a built-in compression feature allowing scripts and pages to be compressed and delivered to browsers more efficiently. Compression can be configured globally across the server using HTTP compression or on a more granular basis using URL compression. HTTP compression is a global setting (httpCompression element) defined in a file named applicationHost.config while URL compression can be configured at the web.config level (urlCompression element) and applies to a specific site, application or folder.

To maximize an Ajax application's performance it's recommended that GZip compression be applied to content such as CSS, JavaScript and HTML files. Doing this compresses the data allowing it to travel over the wire more quickly without requiring extra coding on the developer's part. IIS 7 defines the dynamic and static compression settings in a file named applicationHost.config and allows these settings to be turned on or off through the Internet Information Services Manager as shown in Figure 11.

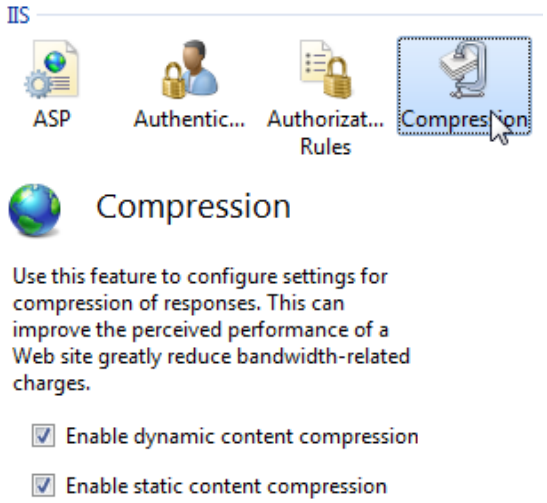


Figure 11. Accessing IIS7 compression settings using the Internet Information Services Manager.

Different types of files can be compressed using IIS7 such as text, JavaScript, XAML and ATOM feeds as shown next:

```
<httpCompression directory="%SystemDrive%\inetpub\temp\IIS
  Temporary Compressed Files">
    <scheme name="gzip" dll="%Windir%\system32\inetsrv\gzip.dll" />
    <dynamicTypes>
        <add mimeType="text/*" enabled="true" />
        <add mimeType="message/*" enabled="true" />
        <add mimeType="application/x-javascript" enabled="true" />
        <add mimeType="*/*" enabled="false" />
    </dynamicTypes>
    <staticTypes>
```

```
            <add mimeType="text/*" enabled="true" />
            <add mimeType="message/*" enabled="true" />
            <add mimeType="application/x-javascript" enabled="true" />
            <add mimeType="application/atom+xml" enabled="true" />
            <add mimeType="application/xaml+xml" enabled="true" />
            <add mimeType="*/*" enabled="false" />
        </staticTypes>
    </httpCompression>
```

In cases where JavaScript files aren't being compressed by IIS7 as expected (tools such as http://www.fiddlertool.com can be used to inspect the Content-Type response header sent from the server), a MIME type may be the culprit. IIS7 uses the application/x-javascript MIME type by default as shown in the previous XML code but this can be changed for any application using web.config if needed. An example of mapping .js files from the application/x-javascript MIME type to application/javascript in web.config is shown next:

```
    <system.webServer>
        <staticContent>
            <remove fileExtension=".js" />
            <mimeMap fileExtension=".js" mimeType="text/javascript" />
        </staticContent>
    </system.webServer>
```

IIS7 can also be configured to compress files based upon how often they're hit. For example, if a page isn't requested more than once in a given time period the file may not be compressed. The serverRuntime element's frequentHitThreshold can be set to a value of 1 to force immediate compression:

```
    <serverRuntime frequentHitThreshold="1" />
```

In addition to enabling JavaScript files to be compressed, IIS7 can also be used to set a far future expires header for static HTML, image, CSS and JavaScript files. By setting a far future expires header the browser doesn't make a network call to the server to check if a file has changed (until the file is stale) allowing the item to be served directly from the browser cache. This type of header can be set in web.config using the following XML syntax:

```
    <system.webServer>
        <staticContent>
             <clientCache httpExpires="Wed, 1 June 2010 00:00:00 GMT"
                cacheControlMode="UseExpires" />
        </staticContent>
    </system.webServer>
```

The httpExpires attribute works fine but requires an explicit date and time to be entered which would have to be changed after the date passes. IIS7 also supports "max-age" which accepts the maximum age (in seconds) of the cache control value. The following configuration section will tell the client to cache content for 100 days.

```
<system.webServer>
    <staticContent>
         <clientCache cacheControlMaxAge="100.00:00:00"
            cacheControlMode="UseMaxAge" />
    </staticContent>
</system.webServer>
```

Conclusion

Microsoft has made many key investments in the JavaScript/Ajax application space that enable developers to build rich, high performance and cross-browser web applications. By using services such as the ASP.NET Ajax CDN and leveraging the ASP.NET Ajax Library's Script Loader, scripts can be loaded more efficiently and take advantage of modern browser capabilities. The ASP.NET Ajax Library's support for the jQuery library in the CDN allows developers to use it directly with the ASP.NET Ajax Library scripts and leverage the strengths of both.

JavaScript application performance can be fine-tuned using tools such as Doloto which analyzes and optimizes the way JavaScript functions are called, the ASP.NET Ajax Minifier which crunches scripts and the IE8 Developer Tools which provide multiple benefits including script debugging and in-depth analysis of how JavaScript code runs in a page. IIS7 can also be used to enhance Ajax application performance through JavaScript file compression. Configuration changes available in IIS7 allow developers to control what files are compressed and set a far future expires header that can minimize the number of calls a browser makes to the server for Ajax application files.

To download the latest version of the ASP.NET Ajax Library visit http://ajax.codeplex.com.

For full documentation on the ASP.NET Ajax Library visit: http://www.asp.net/ajaxlibrary.