Optimizing the Network

Even if you write code that runs fast and you have a hosting environment that has a high throughput, still one of the more problematic bottlenecks of web applications is the bandwidth of your clients and the amount of data and number of requests that the client passes through the network. There are several techniques that can help reduce the number of requests and the size of the responses, some of them are easy as configuring IIS, while others require some more attention to in the application's code.

Apply HTTP Caching Headers

One of the ways to conserve bandwidth is to make sure that all the content that is not going to change for some time will be cached in the browser. Static content, such as images, scripts, and CSS files are good candidates for browser cache, but also dynamic content such as .aspx and .ashx files can often be cached if the content is not getting updated often.

Setting Cache Headers for Static Content

Static files are usually sent back to the client with two caching headers:

- **ETag**. This HTTP header is set by IIS to contain a calculated hash, based on the last modification date of the requested content. For static content, such as image files and CSS files, IIS sets the ETag according to the last modification date of the file. When subsequent requests are sent with the previously cached ETag value, IIS calculates the ETag for the requested file, and if it does not match the client's ETag, the requested file is sent back. If the ETags match, an HTTP 304 (Not Modified) response is sent back. For subsequent requests, the value of the cached *ETag* is placed in the *If-None-Match* HTTP header.
- **Last-Modified**. IIS sets this HTTP header to the last modification date of the requested file. This is an additional caching header which provides a backup in case IIS's ETag support is disabled. When a subsequent request containing the last modified date is sent to the server, IIS verifies the last modification time of the file and decides whether to respond with the content of the file, if the modification time has changed, or with an HTTP 304 response. For subsequent requests, the value of the cached *Last-Modified* is placed in the *If-Modified-Since* HTTP header.

These caching headers will ensure that content is not sent back to the client if the client already has the recent version, but it still requires that a request will be sent from the client to the server to verify that the content hasn't changed. If you have static files in your application that you know probably won't change in the next couple of weeks, or even months, such as your company's logo or script files that are not going to be changed until the next version of the application, you may want to set caching headers that will allow the client to cache that content and reuse it without verifying with the server if the content has changed every time that content is requested. This behavior can be achieved by using either the Cache-Control HTTP header with max-age or the Expires HTTP header. The different between max-age and Expires is that max-age sets a sliding expiration value while Expires allows you to set a fixed point in time (date + time) when the content will expire. For example, setting the max-age to 3600 will allow the browser to use the cached content for one hour (3600 seconds = 60 minutes = 1 hour) automatically, without sending requests to the server to validate it. Once the content expires, either due to the sliding window expiration or due to the arrival of the fixed expiration time, it is marked as stale. When a new request is made to a stale content, the browser will send a request to the server asking for newer content.

Tip You can verify no requests are being sent for cached content by using HTTP monitoring tools, such as Fiddler, and inspecting which requests are sent to the server. If you notice a request being sent although it was supposed to be cached, check the response of that request to verify the existence of the max-age / Expires headers.

Using max-age / Expires together with the ETag / Last-Modified ensures that a request that is sent after the content has expired can return with an HTTP 304 response if the content on the server hasn't actually changed. The response in this case will contain a new max-age / Expires HTTP header.

In most browsers, clicking the Refresh button (or pressing F5) will force the browser to refresh the cache by ignoring the max-age / Expires header, and sending requests for cached content even if the content has yet to expire. The requests will still have the If-Modified-Since / If-None-Match headers, if applicable, so that the server can return a 304 response if the content is still up-to-date.

To set max-age, add the following configuration to your web.config file:

```
<system.webServer>
  <staticContent>
  <clientCache cacheControlMode = "UseMaxAge" cacheControlMaxAge = "0:10:00" />
  </staticContent>
</system.webServer>
```

The above configuration will cause all responses sent for static content to have the Cache-Control HTTP header with the max-age attribute set to 600 seconds.

To use the Expires header, change the `clientCache` element configuration as shown in the following sample:

```
<system.webServer>
  <staticContent>
    <clientCache cacheControlMode = "UseExpires" httpExpires = "Wed, 11 Jul 2013 6:00:00 GMT"/>
  </staticContent>
</system.webServer>
```

The above configuration will make all static content expire on July 11, 2013, at 6 AM.

If you wish to have different max-age or expiration settings for different content, such as have a fixed expiration for JavaScript files and a 100 day sliding window for images, you can use the location section to apply different configuration to different parts of the application, as shown in the following example:

```
<location path = "Scripts">
  <system.webServer>
   <staticContent>
    <clientCache cacheControlMode = "UseExpires" httpExpires = "Wed, 11 Jul 2013 6:00:00 GMT" /

  </staticContent>
  </system.webServer>
</location>
    <location path = "Images">  <system.webServer>
  <staticContent>
<clientCache cacheControlMode = "UseMaxAge" cacheControlMaxAge = "100.0:00:0" />
```

```
  </staticContent>
  </system.webServer>
</location>
```

Note You must use a fully formatted date and time to set the Expires header. Also, according to the HTTP specifications, the Expires header's date must not exceed a year from the current date.

Setting Cache Headers for Dynamic Content

Static files have modification dates that can be used to verify if a cached content has changed or not. Dynamic content, however, does not have a modification date, because every time a dynamic content is requested it is recreated and its modification date is actually the current date, therefore headers such as ETag and Last-Modified are not relevant when dealing with dynamic content.

Having said that, if you look at the content of a dynamic page, you may find a way to express the modification date of that content, or maybe calculate an ETag for it. For example, if a request is sent to retrieve product information from the database, the product table might hold a last update date column that can be used to set the Last-Modified header. If the database table doesn't have a last update column, you can try calculating an MD5 hash from the entity's fields and set the ETag to the result. When a subsequent request is sent to the server, the server can recalculate the MD5 hash for the entity, and if none of the fields has changed, the ETags will be identical and the server can return an HTTP 304 response.

For example, the following code sets the Last-Modified cache header in a dynamic page to the last update date of a product:

```
Response.Cache.SetLastModified(product.LastUpdateDate);
```

If you don't have the last update date, you can set the ETag to an MD5 hash calculated by the entity's properties, as demonstrated in the following code:

```
Response.Cache.SetCacheability(HttpCacheability.ServerAndPrivate);
//Calculate MD5 hash
System.Security.Cryptography.MD5 md5 = System.Security.Cryptography.MD5.Create();
string contentForEtag = entity.PropertyA + entity.NumericProperty.ToString();
byte[] checksum = md5.ComputeHash(System.Text.Encoding.UTF8.GetBytes(contentForEtag));

//Create an ETag string from the hash.
//ETag strings must be surrounded with double quotes, according to the standard
string etag = "\"" + Convert.ToBase64String(checksum, 0, checksum.Length) + "\"";

Response.Cache.SetETag(etag);
```

Note The default cacheability mode of requests in ASP.NET prevents the use of ETags. To support ETags, we need to change the cacheability mode to `ServerAndPrivate`, allowing content to be cached on the server-side and on the client-side, but not on shared machines, such as proxies.

When receiving a request containing an ETag, you can compare the calculated ETag with the ETag supplied by the browser, and if they match, respond with a 304 response, as shown in the following code:

```
if (Request.Headers["If-None-Match"] == calculatedETag) {
  Response.Clear();
  Response.StatusCode = (int)System.Net.HttpStatusCode.NotModified; Response.End();
}
```

If you have any assumptions on the life span of the dynamic content, you can also apply values to the max-age or Expires headers. For example, if you assume that a discontinued product will not be changed, you can set the page returned for the product to expire in one year, as follow:

```
if (productIsDiscontinued)
  Response.Cache.SetExpires(DateTime.Now.AddYears(1));
```

You can also do the same using the Cache-Control max-age header:

```
if (productIsDiscontinued)
  Response.Cache.SetMaxAge(TimeSpan.FromDays(365));
```

Instead of setting the response's expiration in code, you can specify it in the .aspx file as an output cache directive. For example, if the product information shown in the product page can be cached for 10 minutes (600 seconds) in the client, you can set the product page's output cache directive to the following:

```
<%@ Page . . . %>
<%@ OutputCache Duration = "600" Location = "Client"%>
```

When using the `OutputCache` directive, the specified duration is output to the response's HTTP headers as both max-age and expires (expires is calculated from the current date).

Turn on IIS Compression

With the exception of multimedia files (sound, images, and videos) and binary files, such as Silverlight and Flash components, most of the content returned from our web server is text-based, such as HTML, CSS, JavaScript, XML, and JSON. By using IIS compression, those textual responses can be shrunk in size, allowing a quicker response with smaller payloads. With the use of IIS compression, responses can be reduced in size up to 50–60 percent of their original size, and sometimes even more than that. IIS supports two types of compression, static and dynamic. To use IIS compression, make sure you first install the static and dynamic compression IIS components.

Static Compression

When using static compression in IIS, compressed content is stored on disk, so for subsequent requests for the resource, the already compressed content will be returned without the need to perform the compression again. By only compressing the content once, you pay with disk space, but reduce CPU usage and latency that is usually the result of using compression.

Static compression is useful for files that don't usually change (therefore "static"), such as CSS files, JavaScript files, but even if the original file changes, IIS will notice the change, and will recompress the updated file.

Note that compression works best for text files (*.htm, *.txt, *.css) and even for binary files, such as Microsoft Office documents (*.doc, *.xsl), but doesn't work that well for files which are already compressed, such as image files (*.jpg, *.png) and compressed Microsoft Office documents (.docx, .xslx).

Dynamic Compression

When using dynamic compression, IIS performs the compression each time a resource is requested, without storing the post-compression content. This means every time the resource is requested it will be compressed before being sent to the client, incurring both CPU usage and some latency due to the compression process. Dynamic compression, therefore, is more suitable for content that changes often, such as ASP.NET pages.

Since dynamic compression increases the CPU usage, it is advisable that you check your CPU utilization after turning on the compression, to verify it does not put too much strain on the CPU.

Configuring Compression

The first thing that needs to be done to use compression is to enable either static compression, dynamic compression, or both. To enable compression in IIS, open the IIS Manager application, select your machine, click the *Compression* option, and select which compression features your wish to use, as shown in Figure 11-3:

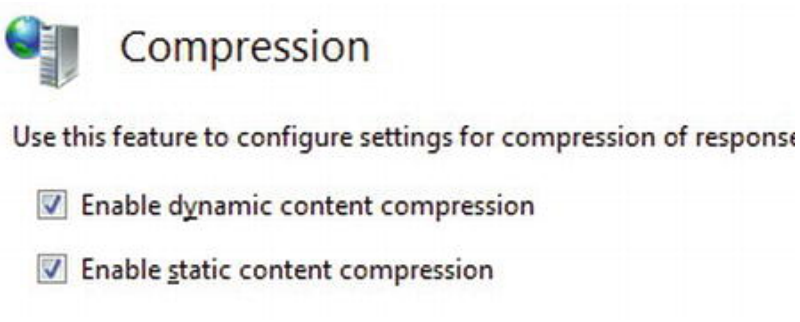

Figure 11-3. Enabling dynamic and static compression in the IIS Manager application

You can also use the *Compression* dialog to set static compression settings, such as the folder where the cached content will be stored, and the minimum file size eligible for compression.

After selecting which compression types are active, you can go ahead and select which MIME types will be statically compressed and which will be dynamically compressed. Unfortunately, IIS doesn't support changing these settings from the IIS Manager application, so you will need to change it manually in the IIS configuration file, applicationHost.config, which is located in the *%windir%\System32\inetsrv\config* folder. Open the file and search for the `< httpCompression >` section, you should already see several MIME types defined for static compression and several other types for dynamic compression. In addition to the already specified MIME types, you can add additional types which you use in your web applications. For example, if you have AJAX calls that return JSON responses, you may want to add dynamic compression support for those responses. The following configuration shows how to dynamic compression support for JSON (existing content was removed for brevity):

```
<httpCompression>
   <dynamicTypes>
     <add mimeType = "application/json; charset = utf-8" enabled = "true" />

   </dynamicTypes>

</httpCompression>
```

Note After adding new MIME types to the list, it is advised that you verify the compression is indeed working by checking the responses with HTTP sniffing tools, such as Fiddler. Compressed responses should have the Content-Encoding HTTP header, and it should be set to either gzip or deflate.

IIS Compression and Client Applications

In order for IIS to compress outgoing responses, it needs to know that the client application can handle compressed responses. Therefore, when a client application sends a request to the server, it needs to add the `Accept-Encoding` HTTP header and set it to either gzip or deflate.

Most known browsers add this header automatically, so when using a browser with a web application or a Silverlight application, IIS will respond with compressed content. However, in .NET applications, when sending HTTP requests with the `HttpWebRequest` type, the Accept-Encoding header is not automatically added, and you will need to add it manually. Furthermore, `HttpWebRequest` will not try to decompress responses unless it is set to expect compressed responses. For example, if you are using an `HttpWebRequest` object, you will need to add the following code to be able to receive and decompress compressed responses:

```
var request = (HttpWebRequest)HttpWebRequest.Create(uri);
request.Headers.Add(HttpRequestHeader.AcceptEncoding, "gzip,deflate");

request.AutomaticDecompression = DecompressionMethods.GZip | DecompressionMethods.Deflate;
```

Other HTTP communication objects, such as an ASMX web service proxy or a `WebClient` object, also support IIS compression, but need to be manually configured to send the header and decompress the response. As for HTTP-based WCF services, prior to WCF 4, .NET clients using either a service reference or a channel factory did not support IIS compression. As of WCF 4, IIS compression is supported automatically, both for sending the header and decompressing the response.

Minification and Bundling

When working with web applications, you often work with pages that use several JavaScript and CSS files. When a page has several links to external resources, loading the page in a browser becomes a lengthy operation, since the user will often need to wait until the page and all of its related styles and scripts are downloaded and parsed. When working with external resources we face two problems:

1. The number of requests the browser needs to send and wait for the response. The more requests there are, the more time it will take the browser to send all requests, since browsers are limited by the number of concurrent connections they can have to a single server (for example, in IE 9 the number of concurrent requests per server is 6).
2. The size of the response, which affects the overall time it takes the browser to download all the responses. The larger the responses, the more time it will take for the browser to download the response. This may also affect the browser's ability to begin sending new requests if hitting the maximum number of concurrent requests.

To resolve this issue we need a technique that will both enable us to lower the size of responses and reduce the number of requests (and therefore responses). With ASP.NET MVC 4 and in ASP.NET 4.5, this technique is now built in to the framework and is called "bundling and minification."

Bundling refers to the ability to bundle a set of files into one URL which when requested, returns all the files concatenated as one response, and minification refers to the reduction of size of a style or script file by removing whitespaces, and in the case of script files, renaming variables and functions so they use less characters, therefore taking less space.

The use of minification together with compression can significantly reduce the size of responses. For example, the size of the jQuery 1.6.2 script file before minification is 240 kb. After compression the file size is approximately 68 kb. The minified version of the original file is 93 kb, a bit bigger than the compressed version, but after applying compression to the minified file, the size comes down to only 33 kb, about 14 percent of the original file size.

To create a minified bundle, first install the Microsoft.AspNet.Web.Optimization NuGet package, and add a reference to the `System.Web.Optimization` assembly. Once added, you can use the `BundleTable` static class to create new bundles for scripts and styles. The bundling should be set prior to loading pages, therefore you should place the bundling code in the *global.asax*, in the `Application_Start` method. For example, the following code creates a bundle named *MyScripts* (accessible from the virtual *bundles* folder) with three script files which will automatically be minified:

```
protected void Application_Start() {
  Bundle myScriptsBundle = new ScriptBundle("~/bundles/MyScripts").Include(
    "~/Scripts/myCustomJsFunctions.js",
    "~/Scripts/thirdPartyFunctions.js",
    "~/Scripts/myNewJsTypes.js");

  BundleTable.Bundles.Add(myScriptsBundle);
  BundleTable.EnableOptimizations = true;
}
```

Note By default, bundling and minification only work when the web application's compilation mode is set to release. To enable bundling even when in debug mode, we set `EnableOptimizations` to `true`.

To use the bundle which was created, add the following script line to the page:

```
<% = Scripts.Render("~/bundles/MyScripts") %>
```

When the page is rendered, the above line will be replaced with a `< script >` tag that points to the bundle, for example the above line may translate to the following HTML:

```
<script src = "/bundles/MyScript?v = XGaE5OlO_bpMLuETD5_XmgfU5dchi8G0SSBExK294I41"
  type = "text/javascript" > </script>
```

By default, the bundle and minification framework sets the response to expire after one year, so the bundle will remain in the browser's cache and served from the cache. To prevent bundles from becoming stale, each bundle has a token which is placed in the URL's query string. If any of the files are removed from the bundle, if new files are added, or if the bundled files are changed, the token will change, and the next request to the page will generate a different URL with a different token, making the browser request for the new bundle.

In a similar manner, we can create a bundle for CSS files:

```
Bundle myStylesBundle = new StyleBundle("~/bundles/MyStyles")
  .Include("~/Styles/defaultStyle.css",
       "~/Styles/extensions.css",
       "~/Styles/someMoreStyles.js");

BundleTable.Bundles.Add(myStylesBundle);
```

And use the bundle in a page:

```
<% = Styles.Render("~/bundles/MyStyles") %>
```

Which will render a `< link >` element:

```
<link href = "/bundles/MyStyles?v = ji3nO1pdg6VLv3CVUWntxgZNf1zRciWDbm4YfW-y0RI1"
  rel = "stylesheet" type = "text/css" />
```

The bundling and minification framework also supports custom transformations, allowing the creation of specialized transform classes, for example to create your own custom minification for JavaScript files.

Use Content Delivery Networks (CDNs)

One of the performance issues related to web applications is the latency involved with accessing resources over the network. End-users who use the same local network as the web server usually have good response time, but once your web application goes global and users from all over the world access it over the Internet, distant users, such as users from other continents, will probably suffer from longer latencies and slower bandwidth due to network problems.

One solution to the location problem is to spread multiple instances of the web server in different locations, geographically dispersed, so end-users will always be geographically close to one of the servers. Of course this creates a whole management issue since you will need to replicate and synchronize servers all the time, and possibly instruct each end-user to use a different URL according to where they are in the world.

That is where Content Delivery Networks (CDNs) come in. A CDN is a collection of web servers, placed in different locations around the globe, allowing end-users to always be close to your web application's content. When using CDNs, you actually use the same address of the CDN all over the world, but the local DNS in your area translates that address to the actual CDN server which is closest to your location. Various Internet companies, such as Microsoft, Amazon, and Akamai have their own CDNs which you can use for a certain fee.

The following scenario describes the common use of a CDN:

1. You set up your CDN and point it to where the original content is.
2. The first time an end-user accesses the content through the CDN, the local CDN server connects to your web server, pulls the content from it, caches it in the CDN server, and returns the content to the end-user.
3. For subsequent requests, the CDN server returns the cached content without contacting your web server, allowing for quicker response times, and possibly faster bandwidth.

Note In addition to serving your end-users faster, the use of a CDN also reduces the number of requests for static content your web server needs to handle, allowing it to dedicate most of its resources for handling dynamic content.

To achieve the first step, you will need to choose the CDN provider you wish to use, and configure your CDN as instructed by its provider. Once you have the address of the CDN, simply change the links for your static content (images, styles, scripts) to point to the CDN address. For example, if you upload your static content to Windows Azure blob storage and register your content with Windows Azure CDN, you can change the URLs in your pages to point to the CDN like so:

```
<link href = "http://az18253.vo.msecnd.net/static/Content/Site.css
" rel = "stylesheet" type = "text/css" />
```

For debugging purposes you can replace the static URL with a variable that will allow you to control whether to use a local address or the CDN's address. For example, the following Razor code constructs the URL by prefixing it with the CDN address provided by the `CdnUrl` application setting from the *web.config* file:

```
@using System.Web.Configuration
<script src = "@WebConfigurationManager.AppSettings["CdnUrl"]/Scripts/jquery-1.6.2.js"
  type = "text/javascript" > </script>
```

When debugging your web application, change the `CdnUrl` to an empty string to get the content from your local web server.