**ASP.NET Performance and Scalability Secrets**

ASP.NET 2.0 has many secrets, which when revealed can give you big performance and scalability boost. For instance, there are secret bottlenecks in Membership and Profile provider which can be solved easily to make authentication and authorization faster. Furthermore, ASP.NET HTTP pipeline can be tweaked to avoid executing unnecessary code that gets hit on each and every request. Not only that, ASP.NET Worker Process can be pushed to its limit to squeeze out every drop of performance out of it. Page fragment output caching on the browser (not on the server) can save significant amount of download time on repeated visits. On demand UI loading can give your site a fast and smooth feeling. Finally, Content Delivery Networks (CDN) and proper use of HTTP Cache headers can make your website screaming fast when implemented properly. In this article, you will learn these techniques that can give your ASP.NET application a big performance and scalability boost and prepare it to perform well under 10 times to 100 times more traffic.

In this article we will discuss the following techniques:

* ASP.NET pipeline optimization
* ASP.NET process configuration optimization
* Things you must do for ASP.NET before going live
* Content Delivery Network
* Caching AJAX calls on browser
* Making best use of Browser Cache
* On demand progressive UI loading for fast smooth experience
* Optimize ASP.NET 2.0 Profile provider
* How to query ASP.NET 2.0 Membership tables without bringing down the site
* Prevent Denial of Service (DOS) attack

The above techniques can be implemented on any ASP.NET website, especially those who use ASP.NET 2.0's Membership and Profile provider.

You can learn a lot more about performance and scalability improvement of ASP.NET and ASP.NET AJAX websites from my book - [Building a Web 2.0 portal using ASP.NET 3.5](http://www.oreilly.com/catalog/9780596510503/).

## ASP.NET Pipeline Optimization

There are several ASP.NET default HttpModules which sit in the request pipeline and intercept each and every request. For example, SessionStateModule intercepts each request, parses the session cookie and then loads the proper session in the HttpContext. Not all of these modules are always necessary. For example, if you aren't using Membership and Profile provider, you don't need FormsAuthentication module. If you aren't using Windows Authentication for your users, you don't need WindowsAuthentication. These modules are just sitting in the pipeline, executing some unnecessary code for each and every request.

The default modules are defined in machine.config file (located in the$WINDOWS$\Microsoft.NET\Framework\$VERSION$\CONFIG directory).

http://www.codeproject.com/images/minus.gif Collapse | [Copy Code](http://www.codeproject.com/Articles/23306/10-ASP-NET-Performance-and-Scalability-Secrets)

<httpModules>

<add name="OutputCache" type="System.Web.Caching.OutputCacheModule" />

<add name="Session" type="System.Web.SessionState.SessionStateModule" />

<add name="WindowsAuthentication"

type="System.Web.Security.WindowsAuthenticationModule" />

<add name="FormsAuthentication"

type="System.Web.Security.FormsAuthenticationModule" />

<add name="PassportAuthentication"

type="System.Web.Security.PassportAuthenticationModule" />

<add name="UrlAuthorization" type="System.Web.Security.UrlAuthorizationModule" />

<add name="FileAuthorization" type="System.Web.Security.FileAuthorizationModule" />

<add name="ErrorHandlerModule" type="System.Web.Mobile.ErrorHandlerModule,

System.Web.Mobile, Version=1.0.5000.0,

Culture=neutral, PublicKeyToken=b03f5f7f11d50a3a" />

</httpModules>

You can remove these default modules from your Web application by adding <remove> nodes in your site'sweb.config. For example:

http://www.codeproject.com/images/minus.gif Collapse | [Copy Code](http://www.codeproject.com/Articles/23306/10-ASP-NET-Performance-and-Scalability-Secrets)

<httpModules>

*<!-- Remove unnecessary Http Modules for faster pipeline* -->

<remove name="Session" />

<remove name="WindowsAuthentication" />

<remove name="PassportAuthentication" />

<remove name="AnonymousIdentification" />

<remove name="UrlAuthorization" />

<remove name="FileAuthorization" />

</httpModules>

The above configuration is suitable for websites that use database based Forms Authentication and do not need any Session support. So, all these modules can safely be removed.

## ASP.NET Process Configuration Optimization

ASP.NET Process Model configuration defines some process level properties like how many number of threads ASP.NET uses, how long it blocks a thread before timing out, how many requests to keep waiting for IO works to complete and so on. The default is in many cases too limiting. Nowadays hardware has become quite cheap and dual core with gigabyte RAM servers have become a very common choice. So, the process model configuration can be tweaked to make ASP.NET process consume more system resources and provide better scalability from each server.

A regular ASP.NET installation will create machine.config with the following configuration:

http://www.codeproject.com/images/minus.gif Collapse | [Copy Code](http://www.codeproject.com/Articles/23306/10-ASP-NET-Performance-and-Scalability-Secrets)

<system.web>

<processModel autoConfig="true" />

You need to tweak this auto configuration and use some specific values for different attributes in order to customize the way ASP.NET worker process works. For example:

http://www.codeproject.com/images/minus.gif Collapse | [Copy Code](http://www.codeproject.com/Articles/23306/10-ASP-NET-Performance-and-Scalability-Secrets)

<processModel

enable="true"

timeout="Infinite"

idleTimeout="Infinite"

shutdownTimeout="00:00:05"

requestLimit="Infinite"

requestQueueLimit="5000"

restartQueueLimit="10"

memoryLimit="60"

webGarden="false"

cpuMask="0xffffffff"

userName="machine"

password="AutoGenerate"

logLevel="Errors"

clientConnectedCheck="00:00:05"

comAuthenticationLevel="Connect"

comImpersonationLevel="Impersonate"

responseDeadlockInterval="00:03:00"

responseRestartDeadlockInterval="00:03:00"

autoConfig="false"

maxWorkerThreads="100"

maxIoThreads="100"

minWorkerThreads="40"

minIoThreads="30"

serverErrorMessageFile=""

pingFrequency="Infinite"

pingTimeout="Infinite"

asyncOption="20"

maxAppDomains="2000"

/>

Here all the values are default values except for the following ones:

* maxWorkerThreads - This is default to 20 per process. On a dual core computer, there'll be 40 threads allocated for ASP.NET. This means at a time ASP.NET can process 40 requests in parallel on a dual core server. I have increased it to 100 in order to give ASP.NET more threads per process. If you have an application which is not that CPU intensive and can easily take in more requests per second, then you can increase this value. Especially if your Web application uses a lot of Web service calls or downloads/uploads a lot of data which does not put pressure on the CPU. When ASP.NET runs out of worker threads, it stops processing more requests that come in. Requests get into a queue and keeps waiting until a worker thread is freed. This generally happens when site starts receiving much more hits than you originally planned. In that case, if you have CPU to spare, increase the worker threads count per process.
* maxIOThreads - This is default to 20 per process. On a dual core computer, there'll be 40 threads allocated for ASP.NET for I/O operations. This means at a time ASP.NET can process 40 I/O requests in parallel on a dual core server. I/O requests can be file read/write, database operations, web service calls, HTTP requests generated from within the Web application and so on. So, you can set this to 100 if your server has enough system resource to do more of these I/O requests. Especially when your Web application downloads/uploads data, calls many external webservices in parallel.
* minWorkerThreads - When a number of free ASP.NET worker threads fall below this number, ASP.NET starts putting incoming requests into a queue. So, you can set this value to a low number in order to increase the number of concurrent requests. However, do not set this to a very low number because Web application code might need to do some background processing and parallel processing for which it will need some free worker threads.
* minIOThreads - Same as minWorkerThreads but this is for the I/O threads. However, you can set this to a lower value than minWorkerThreads because there's no issue of parallel processing in case of I/O threads.
* memoryLimit - Specifies the maximum allowed memory size, as a percentage of total system memory, that the worker process can consume before ASP.NET launches a new process and reassigns existing requests. If you have only your Web application running in a dedicated box and there's no other process that needs RAM, you can set a high value like 80. However, if you have a leaky application that continuously leaks memory, then it's better to set it to a lower value so that the leaky process is recycled pretty soon before it becomes a memory hog and thus keep your site healthy. Especially when you are using COM components and leaking memory. However, this is a temporary solution, you of course need to fix the leak.

Besides the processModel, there's another very important section with the system.net where you can specify the maximum number of outbound requests that can be made to a single IP.

http://www.codeproject.com/images/minus.gif Collapse | [Copy Code](http://www.codeproject.com/Articles/23306/10-ASP-NET-Performance-and-Scalability-Secrets)

<system.net>

<connectionManagement>

<add address="\*" maxconnection="100" />

</connectionManagement>

</system.net>

Default is 2, which is just too low. This means you cannot make more than 2 simultaneous connections to an IP from your Web application. Sites that fetch external content a lot suffer from congestion due to the default setting. Here I have set it to 100. If your Web applications make a lot of calls to a specific server, you can consider setting an even higher value.

You can learn more about these configuration settings from "[Improving ASP.NET Performance](http://msdn2.microsoft.com/en-us/library/ms998549.aspx)".

## Things You Must Do for ASP.NET Before Going Live

You should do some tweaking on your web.config if you are using ASP.NET 2.0 Membership Provider before you go live on your production server:

* Add applicationname attribute in Profile Provider. If you do not add a specific name here, Profile provider will use a GUID. So, on your local machine you will have one GUID and on the production server you will have another GUID. If you copy your local DB to the production server, you won't be able to reuse the records available in your local DB and ASP.NET will create a new application on production. Here's where you need to add it:

http://www.codeproject.com/images/minus.gif Collapse | [Copy Code](http://www.codeproject.com/Articles/23306/10-ASP-NET-Performance-and-Scalability-Secrets)

<profile enabled="true">

<providers>

<clear />

<add name="..." type="System.Web.Profile.SqlProfileProvider"

connectionStringName="..." applicationName="YourApplicationName"

description="..." />

</providers>

* Profile provider will automatically save the profile whenever a page request completes. So, this might result in an unnecessary UPDATE on your DB which has significant performance penalty. So, turn off automatic save and do it explicitly from your code using Profile.Save();

http://www.codeproject.com/images/minus.gif Collapse | [Copy Code](http://www.codeproject.com/Articles/23306/10-ASP-NET-Performance-and-Scalability-Secrets)

<profile enabled="true" automaticSaveEnabled="false" >

* Role Manager always queries the database in order to get the user roles. This has significant performance penalty. You can avoid this by letting the Role Manager cache role information on a cookie. But this will work for users who do not have a lot of roles assigned to them which exceeds the 2 KB limit of Cookie. But it's not a common scenario. So, you can safely store role info on a cookie and save one DB roundtrip on every request to\*.aspx and \*.asmx.

http://www.codeproject.com/images/minus.gif Collapse | [Copy Code](http://www.codeproject.com/Articles/23306/10-ASP-NET-Performance-and-Scalability-Secrets)

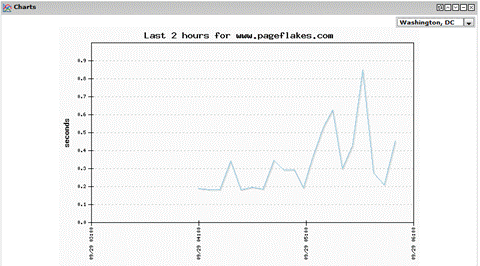
<roleManager enabled="true" cacheRolesInCookie="true" >

The above three settings are must haves for high volume websites.

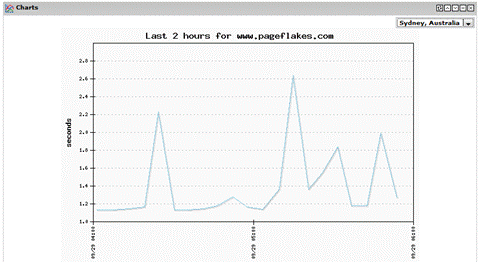
## Content Delivery Network

Every request from a browser goes to your server traveling through the Internet backbones that span the world. The number of countries, continents, oceans a request has to go through to reach your server, the slower it is. For example, if you have your servers in USA and someone from Australia is browsing your site, each request is literary crossing the planet from one end to the other in order to reach your server and then come back again to the browser. If your site has large number of static files like images, CSS, JavaScript; sending request for each of them and downloading them across the world takes a significant amount of time. If you could setup a server in Australia and redirect users to your Australian server, then each request would take fraction of the time it takes to reach USA. Not only the network latency will be lower but also the data transfer rate will be faster and thus static content will download a lot faster. This will give significant performance improvement on the user’s end if your website is rich in static content. Moreover, ISPs provide far greater speed for country wide network compared to the Internet because each country generally has handful of connectivity to the Internet backbone that is shared by all ISPs within the country. As a result, users having 4 MBPS broadband connection will get the full 4 MBPS speed from servers that are within the same country. But they will get as low as 512 KBPS from servers which are outside the country. Thus having a server in the same country significantly improves site download speed and responsiveness.

Besides improving site load speed, CDN also offloads traffic from your Web servers. As it deals with static cacheable content, your webservers rarely get hits for those content. Thus hits going to your webservers drop significantly and webservers free up more resource to process dynamic requests. Your webservers also save a lot of IIS log space because IIS need not log requests for the static contents. If you have a lot of graphics, CSS and JavaScript on your website, you can save gigabytes of IIS logs every day.

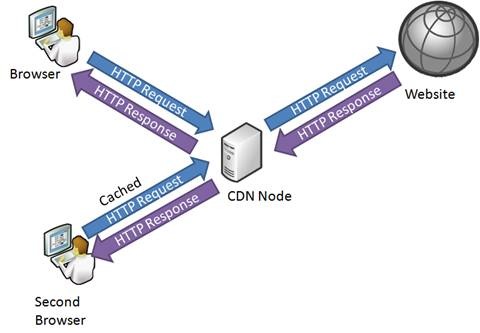


Above figure shows average response time for [www.pageflakes.com](http://www.pageflakes.com/) from Washington, DC, where the servers are in Dallas, Texas. The average response time is around 0.4 seconds. This response includes server side execution time as well. Generally it takes around 0.3 to 0.35 seconds to execute the page on the server. So, time spent on the network is around 0.05 seconds or 50ms. This is a really fast connectivity as there are only 4 to 6 hops to reach Dallas from Washington DC.



This figure shows average response time from Sydney, Australia. The average response time is 1.5 seconds which is significantly higher than that of Washington DC. It’s almost 4 times compared to USA. There's almost 1.2 seconds overhead on network only. Moreover, there are around 17 to 23 hops from Sydney to Dallas. So, the site downloads at least 4 times slower in Australia than it is from anywhere in USA.

A content delivery network (CDN) is a system of computers networked together across the Internet. The computers cooperate transparently to deliver content (especially large media content) to end users. CDN nodes (cluster of servers in a specific location) are deployed in multiple locations, often over multiple backbones. These nodes cooperate with each other to serve requests for content by end users. They also transparently move content behind the scenes to optimize the delivery process. CDN serves requests by intelligently choosing the nearest server. It looks for the fastest connectivity between your computer to a nearest node that has the content you are looking for. The number of nodes in different countries and the number of redundant backbone connectivity a CDN has measures its strength. Some of the most popular CDNs are Akamai, Limelight, EdgeCast. Akamai is used by large companies like Microsoft, Yahoo, AOL. It’s a comparatively expensive solution. However, Akamai has the best performance throughout the world because they have servers in almost every prominent city in the world. However, Akamai is very expensive and they only accept a customer who can spend a minimum of 5K on CDN per month. For smaller companies, Edgecast is a more affordable solution.



This figure shows CDN Nodes that are closest to the browser intercepts traffic and serves response. If it does not have the response in cache, it fetches it from origin server using a faster route and much more optimized connectivity than the browser’s ISP can provide. If the content is already cached, then it’s served directly from the node and no request goes to the origin server.

There are generally two types of CDNs. One is where you upload content to CDN's servers via FTP and you get a subdomain in their domain like dropthings.somecdn.net. You change all the URL of static content throughout your site to download content from the CDN domain instead of the relative URL to your own domain. So, a URL like /logo.gif will be renamed to http://dropthings.somecdn.net/logo.gif. This is easy to configure but has maintenance problems. You will have to keep CDN’s store synchronized with the files all the time. Deployment becomes complicated because you need to update both your website and the CDN store at the same time. Example of such a CDN (which is very cheap) is [Cachefly](http://www.cachefly.com/).

A more convenient approach is to store static content on your own site but use domain aliasing. You can store your content in a subdomain that points to your own domain like static.dropthings.com. Then you use CNAME to map that subdomain to a CDN’s nameserver like cache.somecdn.net. When a browser tries to resolve static.dropthigns.com, the DNS lookup request goes to the CDN nameserver. The nameserver then returns IP of a CDN node which is closest to you and can give you the best download performance. The browser then sends requests for files to that CDN node. When CDN node sees the request, it checks whether it has the content already cached. If it is cached, it delivers the content directly from its local store. If not, it makes a request to your server and then looks at the cache header generated in response. Based on the cache header it decides how long to cache the response in its own cache. In the meantime, the browser does not wait for CDN node to get content and return to it. CDN does an interesting trick on the Internet backbone to actually route requests to the origin server so that the browser gets the response directly served from origin server while CDN is updating its cache. Sometimes CDN act as a proxy, intercepting each request and then fetching uncached content from the origin using a faster route and optimized connectivity to the origin server. Example of such CDN is [Edgecast](http://edgecast.com/).

## Caching AJAX Calls on Browser

Browsers can cache images, JavaScript, CSS files on a user's hard drive, and it can also cache XML HTTP calls if the call is a HTTP GET. The cache is based on the URL. If it's the same URL, and it's cached on the computer, then the response is loaded from the cache, not from the server when it is requested again. Basically, the browser can cache any HTTP GET call and return cached data based on the URL. If you make an XML HTTP call as HTTP GET and the server returns some special header which informs the browser to cache the response, on future calls, the response will be immediately returned from the cache and thus saves the delay of network roundtrip and download time.

At Pageflakes, we cache the user's state so that when the user visits again the following day, the user gets a cached page which loads instantly from the browser cache, not from the server. Thus the second time load becomes very fast. We also cache several small parts of the page which appears on user's actions. When the user performs the same action again, a cached result is loaded immediately from the local cache and thus saves the network roundtrip time. The user gets a fast loading site and a very responsive site. The perceived speed increases dramatically.

The idea is to make HTTP GET calls while making Atlas Web service calls and return some specific HTTP Response headers that tell the browser to cache the response for some specific duration. If you return the Expires header during the response, the browser will cache the XML HTTP response. There are two headers that you need to return with the response that instruct the browser to cache the response:

http://www.codeproject.com/images/minus.gif Collapse | [Copy Code](http://www.codeproject.com/Articles/23306/10-ASP-NET-Performance-and-Scalability-Secrets)

HTTP/1.1 200 OK

Expires: Fri, 1 Jan 2030

Cache-Control: public

This instructs the browser to cache the response till Jan 2030. As long as you make the same XML HTTP call with the same parameters, you will get cached response from the computer and no call will go to the origin server. There are more advanced ways to get further control over response caching. For example, here is a header that instructs the browser to cache for 60 seconds but contact the server and get a fresh response after 60 seconds. It will also prevent proxies from returning cached response when the browser local cache expires after 60 seconds.

http://www.codeproject.com/images/minus.gif Collapse | [Copy Code](http://www.codeproject.com/Articles/23306/10-ASP-NET-Performance-and-Scalability-Secrets)

HTTP/1.1 200 OK

Cache-Control: private, must-revalidate, proxy-revalidate, max-age=60

Let's try to produce such response headers from an ASP.NET Web service call:

http://www.codeproject.com/images/minus.gif Collapse | [Copy Code](http://www.codeproject.com/Articles/23306/10-ASP-NET-Performance-and-Scalability-Secrets)

[WebMethod][ScriptMethod(UseHttpGet=true)]

public string CachedGet()

{

TimeSpan cacheDuration = TimeSpan.FromMinutes(1);

Context.Response.Cache.SetCacheability(HttpCacheability.Public);

Context.Response.Cache.SetExpires(DateTime.Now.Add(cacheDuration));

Context.Response.Cache.SetMaxAge(cacheDuration);

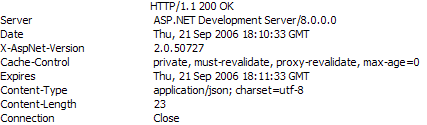
Context.Response.Cache.AppendCacheExtension(

"must-revalidate, proxy-revalidate");

return DateTime.Now.ToString();

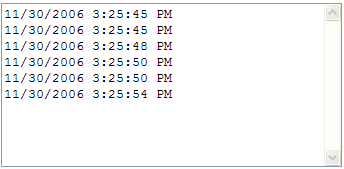
}

This will result in the following response headers:



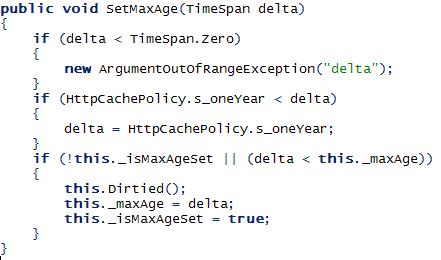
The Expires header is set properly. But the problem is with the Cache-control. It is showing that max-age is set to 0 which will prevent the browser from doing any kind of caching. If you seriously want to prevent caching, you should emit such a cache-control header. Looks like exactly the opposite thing happened.

The output is as usual incorrect, and not cached:



There's a bug in ASP.NET 2.0 that you cannot change the max-age header. As max-age is set to 0, ASP.NET 2.0 sets the Cache-control to private because max-age = 0 means no cache is needed. So, there's no way you can make ASP.NET 2.0 return proper headers which cache the response. This is due to the ASP.NET AJAX Framework that intercepts calls to Webservices and incorrectly sets the max-age to 0 by default before executing a request.

Time for a hack. After decompiling the code of the HttpCachePolicy class (Context.Response.Cache object's class), I found the following code:



Somehow, this.\_maxAge is getting set to 0 and the check: "if (!this.\_isMaxAgeSet || (delta < this.\_maxAge))" is preventing it from getting set to a bigger value. Due to this problem, we need to bypass theSetMaxAge function and set the value of the \_maxAge field directly, using Reflection.

http://www.codeproject.com/images/minus.gif Collapse | [Copy Code](http://www.codeproject.com/Articles/23306/10-ASP-NET-Performance-and-Scalability-Secrets)

[WebMethod][ScriptMethod(UseHttpGet=true)]

public string CachedGet2()

{

TimeSpan cacheDuration = TimeSpan.FromMinutes(1);

FieldInfo maxAge = Context.Response.Cache.GetType().GetField("\_maxAge",

BindingFlags.Instance|BindingFlags.NonPublic);

maxAge.SetValue(Context.Response.Cache, cacheDuration);

Context.Response.Cache.SetCacheability(HttpCacheability.Public);

Context.Response.Cache.SetExpires(DateTime.Now.Add(cacheDuration));

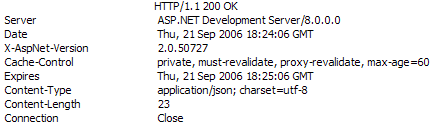
Context.Response.Cache.AppendCacheExtension(

"must-revalidate, proxy-revalidate");

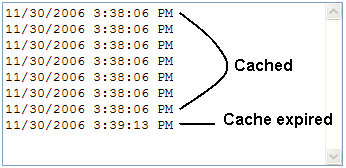
return DateTime.Now.ToString();

}

This will return the following headers:



Now max-age is set to 60 and thus the browser will cache the response for 60 seconds. If you make the same call again within 60 seconds, it will return the same response. Here's a test output which shows the date time returned from the server:



After 1 minute, the cache expires and the browser makes a call to the server again. The client side code is like this:

http://www.codeproject.com/images/minus.gif Collapse | [Copy Code](http://www.codeproject.com/Articles/23306/10-ASP-NET-Performance-and-Scalability-Secrets)

function testCache()

{

TestService.CachedGet(function(result)

{

debug.trace(result);

});

}

There's another problem to solve. In web.config, you will see ASP.NET Ajax will add:

http://www.codeproject.com/images/minus.gif Collapse | [Copy Code](http://www.codeproject.com/Articles/23306/10-ASP-NET-Performance-and-Scalability-Secrets)

<system.web>

<trust level="Medium"/>

This prevents us from setting \_maxAge field of Response object because it requires Reflection. So, you will have to remove this trust level or put Full.

http://www.codeproject.com/images/minus.gif Collapse | [Copy Code](http://www.codeproject.com/Articles/23306/10-ASP-NET-Performance-and-Scalability-Secrets)

<system.web>

<trust level="Full"/>

## Making Best Use of Browser Cache

### Use URLs Consistently

Browsers cache content based on the URL. When the URL changes, the browser fetches a new version from origin server. URL can be changed by changing the query string parameters. For example, /default.aspx is cached on the browser. If you request /default.aspx?123 it will fetch new content from the server. Response from the new URL can also be cached in the browser if you return proper caching headers. In that case, changing the query parameter to something else like /default.aspx?456 will return new content from the server. So, you need to make sure you use URL consistently everywhere when you want to get cached response. From homepage, if you have requested a file with URL /welcome.gif, make sure from another page you request the same file using the same URL. One common mistake is to sometimes omit the “www” subdomain from the URL. [www.pageflakes.com/default.aspx](http://www.pageflakes.com/default.aspx) is not same as[pageflakes.com/default.aspx](http://www.pageflakes.com/default.aspx). Both will be cached separately.

### Cache Static Content for Longer Period

Static files can be cached for longer periods like one month. If you are thinking that you should cache for couple of days so that when you change the file, users will pick it up sooner, you're mistaken. If you update a file which was cached by Expires header, new users will immediately get the new file while old users will see the old content until it expires on their browser. So, as long as you are using Expires header to cache static files, you should use as high a value as possible.

For example, if you have set Expires header to cache a file for three days, one user will get the file today and store it in cache for next three days. Another user will get the file tomorrow and cache it for three days after tomorrow. If you change the file on the day after tomorrow, the first user will see it on fourth day and the second user will see it on fifth day. So, different users will see different versions of the file. As a result, it does not help setting a lower value assuming all users will pick up the latest file soon. You will have to change the URL of the file in order to ensure everyone gets the exact same file immediately.

You can setup Expires header from static files from IIS Manager. You'll learn how to do it in a later section.

### ****Use Cache Friendly Folder Structure****

Store cached content under a common folder. For example, store all images of your site under the /static folder instead of storing images separately under different subfolders. This will help you use consistent URL throughout the site because from anywhere you can use /static/images/somefile.gif. Later on, we will learn it’s easier to move to a Content Delivery Network when you have static cacheable files under a common root folder.

### ****Reuse Common Graphics Files****

Sometimes we put common graphics files under several virtual directories so that we can write smaller paths. For example, say you have indicator.gif in root folder, some subfolders and under CSS folder. You did it because you need not worry about paths from different places and you could just use the file name as relative URL. This does not help in caching. Each copy of the file is cached in the browser separately. So, you should collect all graphics files in the whole solution and put them under the same root static folder after eliminating duplicates and use the same URL from all the pages and CSS files.

### ****Change File Name When You Want to Expire Cache****

When you want a static file to be changed, don't just update the file because it’s already cached in the user’s browser. You need to change the file name and update all references everywhere so that browser downloads the new file. You can also store the file names in database or configuration files and use data binding to generate the URL dynamically. This way you can change the URL from one place and have the whole site receive the change immediately.

### Use a Version Number While Accessing Static Files

If you do not want to clutter your static folder with multiple copies of the same file, you can use query string to differentiate versions of same file. For example, a GIF can be accessed with a dummy query string like/static/images/indicator.gif?v=1. When you change the indicator.gif, you can overwrite the same file and then update all references to the file to /static/images/indicator.gif?v=2. This way you can keep changing the same file again and again and just update the references to access the graphics using the new version number.

### Store Cacheable Files in a Different Domain

It’s always a good idea to put static contents in a different domain. First of all, the browser can open other two concurrent connections to download the static files. Another benefit is that you don't need to send the cookies to the static files. When you put the static files on the same domain as your Web application, browser sends all the ASP.NET cookies and all other cookies that your Web application is producing. This makes the request headers unnecessarily large and waste bandwidth. You don't need to send these cookies to access the static files. So, if you put the static files in a different domain, those cookies will not be sent. For example, put your static files in[www.staticcontent.com](http://www.staticcontent.com/) domain while your website is running on [www.dropthings.com](http://www.dropthings.com/). The other domain does not need to be a completely different Web site. It can just be an alias and share the same Web application path.

### ****SSL is Not Cached, so Minimize SSL Use****

Any content that is served over SSL is not cached. So, you need to put static content outside SSL. Moreover, you should try limiting SSL to only secure pages like Login page or Payment page. Rest of the site should be outside SSL over regular HTTP. SSL encrypts request and response and thus puts extra load on the server. Encrypted content is also larger than the original content and thus takes more bandwidth.

### HTTP POST Requests are Never Cached

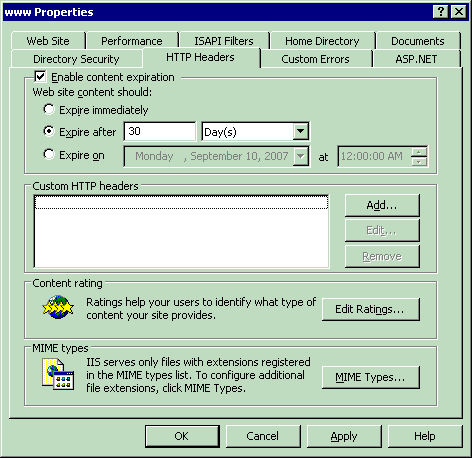
Cache only happens for HTTP GET requests. HTTP POST requests are never cached. So, any kind of AJAX call you want to make cacheable needs to be HTTP GET enabled.

### Generate Content-Length Response Header

When you are dynamically serving content via Web service calls or HTTP handlers, make sure you emit Content-Length header. Browsers have several optimizations for downloading contents faster when it knows how many bytes to download from the response by looking at the Content-Length header. Browsers can use persisted connections more effectively when this header is present. This saves the browser from opening a new connection for each request. When there’s no Content-Length header, browser doesn't know how many bytes it’s going to receive from the server and thus keeps the connection open as long as it gets bytes delivered from the server until the connection closes. So, you miss the benefit of Persisted Connections that can greatly reduce download time of several small files like CS, JavaScripts, and images.

### How to Configure Static Content Caching in IIS

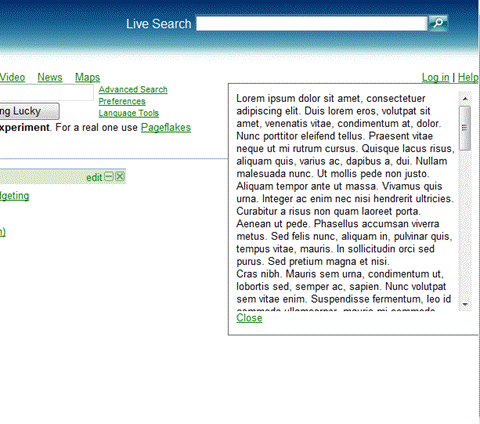
In IIS Manager, Web site properties dialog box has “HTTP Headers” tab where you can define Expires header for all requests that IIS handles. There you can define whether to expire content immediately or expire after certain number of days or on a specific date. The second option (Expire after) uses sliding expiration, not absolute expiration. This is very useful because it works per request. Whenever someone requests a static file, IIS will calculate the expiration date based on the number of days/months from the Expire after.

[](http://omar.mvps.org/images/Makingbestuseofcacheforbestsiteperforman_15143/clip_image001.gif)

For dynamic pages that are served by ASP.NET, a handler can modify the Expires header and override IIS default setting.

## On Demand Progressive UI Loading for Fast Smooth Experience

AJAX websites are all about loading as many features as possible into the browser without having any postback. If you look at the Start Pages like [Pageflakes](http://www.pageflakes.com/), it's only one single page that gives you all the features of the whole application with zero postback. A quick and dirty approach for doing this is to deliver every possible HTML snippet inside hidden divs during page load and then make those divs visible when needed. But this makes first time loading way too slow and browser gives sluggish performance as too much stuff is there to process on the DOM. So, a better approach is to load the HTML snippet and necessary JavaScript on-demand. In my [dropthings](http://www.dropthings.com/) project, I have shown an example how this is done.



When you click on the "help" link, it loads the content of the help dynamically. This HTML is not delivered as part of the default.aspx that renders the first page. Thus the giant HTML and graphics related to the help section has no effect on site load performance. It is only loaded when user clicks the "help" link. Moreover, it gets cached on the browser and thus loads only once. When user clicks the "help" link again, it's served directly from the browser cache, instead of fetching from the origin server again.

The principle is making an XMLHTTP call to an \*.aspx page, get the response HTML, put that response HTML inside a container DIV, make that DIV visible.

AJAX Framework has a Sys.Net.WebRequest class which you can use to make regular HTTP calls. You can define HTTP method, URI, headers and the body of the call. It’s kind of a low level function for making direct calls via XMLHTTP. Once you construct a Web request, you can execute it using Sys.Net.XMLHttpExecutor.

http://www.codeproject.com/images/minus.gif Collapse | [Copy Code](http://www.codeproject.com/Articles/23306/10-ASP-NET-Performance-and-Scalability-Secrets)

function showHelp()

{

var request = new Sys.Net.WebRequest();

request.set\_httpVerb("GET");

request.set\_url('help.aspx');

request.add\_completed( function( executor )

{

if (executor.get\_responseAvailable())

{

var helpDiv = $get('HelpDiv');

var helpLink = $get('HelpLink');

var helpLinkBounds = Sys.UI.DomElement.getBounds(helpLink);

helpDiv.style.top = (helpLinkBounds.y + helpLinkBounds.height) + "px";

var content = executor.get\_responseData();

helpDiv.innerHTML = content;

helpDiv.style.display = "block";

}

});

var executor = new Sys.Net.XMLHttpExecutor();

request.set\_executor(executor);

executor.executeRequest();

}

The example shows how the help section is loaded by hitting help.aspx and injecting its response inside theHelpDiv. The response can be cached by the output cache directive set on help.aspx. So, next time when the user clicks on the link, the UI pops up immediately. The help.aspx file has no <html> block, only the content that is set inside the DIV.

http://www.codeproject.com/images/minus.gif Collapse | [Copy Code](http://www.codeproject.com/Articles/23306/10-ASP-NET-Performance-and-Scalability-Secrets)

<%@ Page Language="C#" AutoEventWireup="true" CodeFile="Help.aspx.cs"

Inherits="Help" %>

<%@ OutputCache Location="ServerAndClient" Duration="604800" VaryByParam="none" %>

<div class="helpContent">

<div id="lipsum">

<p>

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Duis lorem

eros, volutpat sit amet, venenatis vitae, condimentum at, dolor. Nunc

porttitor eleifend tellus. Praesent vitae neque ut mi rutrum cursus.

Using this approach, you can break the UI into smaller \*.aspxfiles. Although these \*.aspxfiles cannot have JavaScript or stylesheet blocks, they can contain large amount of HTML that you need to show on the UI on-demand. Thus you can keep initial download to absolute minimum just for loading the basic stuff. When the user explores new features on the site, load those areas incrementally.

## Optimize ASP.NET 2.0 Profile Provider

Do you know there are two important stored procedures in ASP.NET 2.0 Profile Provider that you can optimize significantly? If you use them without doing the necessary optimization, your servers will sink taking your business down with you during heavy load. Here's a story:

During March, [Pageflakes](http://www.pageflakes.com/) was shown on MIX 2006. We were having a glamorous time back then. We were right on[Showcase of Atlas Web site](http://atlas.asp.net/default.aspx?tabid=47&subtabid=472) as the first company. Number of visits per day were rising sky high. One day we noticed, the database server was no more. We restarted the server, brought it back, again it died within an hour. After doing a lot of postmortem analysis on the remains of the server's body parts, we found that it was having 100% CPU and super high IO usage. The hard drives were over heated and turned themselves off in order to save themselves. This was quite surprising to us because we thought we were very intelligent back then and we used to profile every single Web service function. So, we went through hundreds of megabytes of logs hoping to find which webservice function was taking the time. We suspected one. It was the first function that loads a user's page setup. We broke it up into smaller parts in order to see which part is taking most of the time.

http://www.codeproject.com/images/minus.gif Collapse | [Copy Code](http://www.codeproject.com/Articles/23306/10-ASP-NET-Performance-and-Scalability-Secrets)

private GetPageflake(string source, string pageID, string userUniqueName)

{

if( Profile.IsAnonymous )

{

using (new TimedLog(Profile.UserName,"GetPageflake"))

{

You see, the entire function body is timed. If you want to learn how this timing works, I will explain it in a new article. We also timed smaller parts which we suspected were taking the most resource. But we could find not a single place in our code which was taking any significant time. Our codebase is always super optimized (after all, you know who is reviewing it, me).

Meanwhile, users were shouting, management was screaming, support staff was complaining on the phone. Developers were furiously sweating and blood vessels on their forehead were coming out. Nothing special, just a typical situation we have couple of times every month.

Now you must be shouting, "You could have used SQL Profiler, you idiot!" We were using SQL Server workgroup edition. It does not have SQL Profiler. So, we had to hack our way through to get it running on a server somehow. Don't ask how. After running the SQL Profiler, boy, were we surprised! The name of the honorable SP which was giving us so much pain was the great stored procedure dbo.aspnet\_Profile\_GetProfiles!

We used (and still use) Profile provider extensively.

Here's the SP:

http://www.codeproject.com/images/minus.gif Collapse | [Copy Code](http://www.codeproject.com/Articles/23306/10-ASP-NET-Performance-and-Scalability-Secrets)

CREATE PROCEDURE [dbo].[aspnet\_Profile\_GetProfiles]

@ApplicationName nvarchar(256),

@ProfileAuthOptions int,

@PageIndex int,

@PageSize int,

@UserNameToMatch nvarchar(256) = NULL,

@InactiveSinceDate datetime = NULL

AS

BEGIN

DECLARE @ApplicationId uniqueidentifier

SELECT @ApplicationId = NULL

SELECT @ApplicationId = ApplicationId

FROM aspnet\_Applications

WHERE LOWER(@ApplicationName)

= LoweredApplicationName

IF (@ApplicationId IS NULL)

RETURN

*-- Set the page bounds*

DECLARE @PageLowerBound int

DECLARE @PageUpperBound int

DECLARE @TotalRecords int

SET @PageLowerBound = @PageSize \* @PageIndex

SET @PageUpperBound = @PageSize - 1 + @PageLowerBound

*-- Create a temp table TO store the select results*

CREATE TABLE #PageIndexForUsers

(

IndexId int IDENTITY (0, 1) NOT NULL,

UserId uniqueidentifier

)

*-- Insert into our temp table*

INSERT INTO #PageIndexForUsers (UserId)

SELECT u.UserId

FROM dbo.aspnet\_Users

u, dbo.aspnet\_Profile p

WHERE ApplicationId = @ApplicationId

AND u.UserId = p.UserId

AND (@InactiveSinceDate

IS NULL OR LastActivityDate

<= @InactiveSinceDate)

AND (

(@ProfileAuthOptions = 2)

OR (@ProfileAuthOptions = 0

AND IsAnonymous = 1)

OR (@ProfileAuthOptions = 1

AND IsAnonymous = 0)

)

AND (@UserNameToMatch

IS NULL OR LoweredUserName

LIKE LOWER(@UserNameToMatch))

ORDER BY UserName

SELECT u.UserName, u.IsAnonymous, u.LastActivityDate,

p.LastUpdatedDate, DATALENGTH(p.PropertyNames)

+ DATALENGTH(p.PropertyValuesString)

+ DATALENGTH(p.PropertyValuesBinary)

FROM dbo.aspnet\_Users

u, dbo.aspnet\_Profile p, #PageIndexForUsers i

WHERE

u.UserId = p.UserId

AND p.UserId = i.UserId

AND i.IndexId >= @PageLowerBound

AND i.IndexId <= @PageUpperBound

DROP TABLE #PageIndexForUsers

END

END

First it looks up for ApplicationID.

http://www.codeproject.com/images/minus.gif Collapse | [Copy Code](http://www.codeproject.com/Articles/23306/10-ASP-NET-Performance-and-Scalability-Secrets)

DECLARE @ApplicationId uniqueidentifier

SELECT @ApplicationId = NULL

SELECT @ApplicationId = ApplicationId FROM aspnet\_Applications

WHERE LOWER(@ApplicationName) = LoweredApplicationName

IF (@ApplicationId IS NULL)

RETURN

Then it creates a temporary table (it should use table data type) in order to store profiles of users.

http://www.codeproject.com/images/minus.gif Collapse | [Copy Code](http://www.codeproject.com/Articles/23306/10-ASP-NET-Performance-and-Scalability-Secrets)

*-- Create a temp table TO store the select results*

CREATE TABLE #PageIndexForUsers

(

IndexId int IDENTITY (0, 1) NOT NULL,

UserId uniqueidentifier

)

*-- Insert into our temp table*

INSERT INTO #PageIndexForUsers (UserId)

If it gets called very frequently, there will be too high IO due to the temporary table creation. It also runs through two very big tables - aspnet\_Users and aspnet\_Profile. The SP is written in such a way that if one user has multiple profiles, it will return all profiles of the user. But normally we store one profile per user. So, there's no need for creating a temporary table. Moreover, there's no need for doing LIKE LOWER(@UserNameToMatch). We always called with a full user name which we can match directly using the equal operator.

So, we opened up the stored proc and did a open heart bypass surgery like this:

http://www.codeproject.com/images/minus.gif Collapse | [Copy Code](http://www.codeproject.com/Articles/23306/10-ASP-NET-Performance-and-Scalability-Secrets)

IF @UserNameToMatch IS NOT NULL

BEGIN

SELECT u.UserName, u.IsAnonymous, u.LastActivityDate, p.LastUpdatedDate,

DATALENGTH(p.PropertyNames)

+ DATALENGTH(p.PropertyValuesString) + DATALENGTH(p.PropertyValuesBinary)

FROM dbo.aspnet\_Users u

INNER JOIN dbo.aspnet\_Profile p ON u.UserId = p.UserId

WHERE u.LoweredUserName = LOWER(@UserNameToMatch)

SELECT @@ROWCOUNT

END

ELSE

BEGIN *-- Do the original bad things*

It ran fine locally. Now it was time to run it on the server. This is an important SP which is used by the ASP.NET 2.0 Profile Provider, heart of ASP.NET Framework. If we do something wrong here, we might not be able to see the problem immediately, but may be after one month we will realize users profile is mixed up and there's no way to get it back. So, it was a pretty hard decision to run this on a live production server directly without doing enough testing. We did not have time to do enough testing anyway. We are already down. So, we all gathered, said our prayers and hit the "Execute" button on SQL Server Management Studio.

The SP ran fine. On the server we noticed from 100% CPU usage it came down to 30% CPU usage. IO usage also came down to 40%.

We went live again!

Here's another SP that gets called on every page load and webservice call on our site because we use Profile provider extensively.

http://www.codeproject.com/images/minus.gif Collapse | [Copy Code](http://www.codeproject.com/Articles/23306/10-ASP-NET-Performance-and-Scalability-Secrets)

CREATE PROCEDURE [dbo].[aspnet\_Profile\_GetProperties]

@ApplicationName nvarchar(256),

@UserName nvarchar(256),

@CurrentTimeUtc datetime

AS

BEGIN

DECLARE @ApplicationId uniqueidentifier

SELECT @ApplicationId = NULL

SELECT @ApplicationId = ApplicationId

FROM dbo.aspnet\_Applications

WHERE LOWER(@ApplicationName) = LoweredApplicationName

IF (@ApplicationId IS NULL)

RETURN

DECLARE @UserId uniqueidentifier

SELECT @UserId = NULL

SELECT @UserId = UserId

FROM dbo.aspnet\_Users

WHERE ApplicationId = @ApplicationId

AND LoweredUserName =

LOWER(@UserName)

IF (@UserId IS NULL)

RETURN

SELECT TOP 1 PropertyNames, PropertyValuesString, PropertyValuesBinary

FROM dbo.aspnet\_Profile

WHERE UserId = @UserId

IF (@@ROWCOUNT > 0)

BEGIN

UPDATE dbo.aspnet\_Users

SET LastActivityDate=@CurrentTimeUtc

WHERE UserId = @UserId

END

END

When you run the SP, see the statistics:

http://www.codeproject.com/images/minus.gif Collapse | [Copy Code](http://www.codeproject.com/Articles/23306/10-ASP-NET-Performance-and-Scalability-Secrets)

Table 'aspnet\_Applications'. Scan count 1, logical reads 2, physical reads 0,

read-ahead reads 0, lob logical reads 0, lob physical

reads 0, lob read-ahead reads 0.

(1 row(s) affected)

Table 'aspnet\_Users'. Scan count 1, logical reads 4, physical reads 0,

read-ahead reads 0, lob logical reads 0, lob physical

reads 0, lob read-ahead reads 0.

(1 row(s) affected)

(1 row(s) affected)

Table 'aspnet\_Profile'. Scan count 0, logical reads 3, physical reads 0,

read-ahead reads 0, lob logical reads 0, lob physical

reads 0, lob read-ahead reads 0.

(1 row(s) affected)

Table 'aspnet\_Users'. Scan count 0, logical reads 27, physical reads 0,

read-ahead reads 0, lob logical reads 0, lob physical

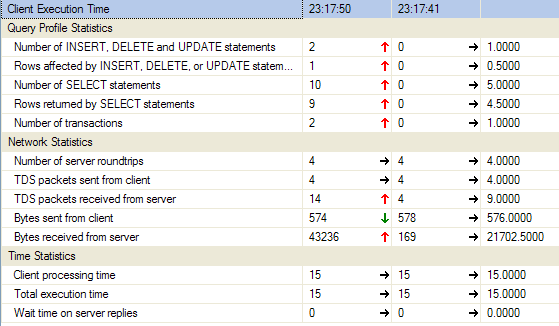
reads 0, lob read-ahead reads 0.

(1 row(s) affected)

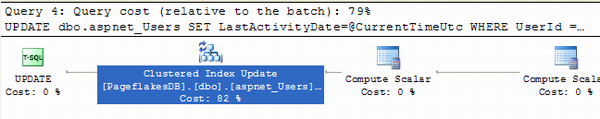
(1 row(s) affected)

This stored proc populates the Profile object with all the custom properties whenever Profile object is accessed for the first time within a request.

First it does a SELECT on aspnet\_application to find out the application ID from application name. You can easily replace this with a hard coded application ID inside the SP and save some effort. Normally we run only one application on our production server. So, there's no need to lookup application ID on every single call. This is one quick optimization to do. However, from client statistics, you can see where's the real performance bottleneck:



Then look at the last block where aspnet\_users table is updated with LastActivityDate. This is the most expensive one.



This is done in order to ensure Profile provider remembers when was the last time a user's profile was accessed. We do not need to do this on every single page load and Web service call where Profile object is accessed. Maybe we can do it when user first logs in and logs out. In our case, a lot of Web service is called while user is on the page. There's only one page anyway. So, we can easily remove this in order to save a costly update on the giant aspnet\_userstable on every single Web service call.

## How to Query ASP.NET 2.0 Membership Tables Without Bringing Down the Site

Such queries will happily run on your development environment:

http://www.codeproject.com/images/minus.gif Collapse | [Copy Code](http://www.codeproject.com/Articles/23306/10-ASP-NET-Performance-and-Scalability-Secrets)

Select \* from aspnet\_users where UserName = 'blabla'

Or you can get some user's profile without any problem using:

http://www.codeproject.com/images/minus.gif Collapse | [Copy Code](http://www.codeproject.com/Articles/23306/10-ASP-NET-Performance-and-Scalability-Secrets)

Select \* from aspnet\_profile where userID = '…...'

Even you can nicely update a user's email in aspnet\_membership table like this:

http://www.codeproject.com/images/minus.gif Collapse | [Copy Code](http://www.codeproject.com/Articles/23306/10-ASP-NET-Performance-and-Scalability-Secrets)

Update aspnet\_membership

SET Email = 'newemailaddress@somewhere.com'

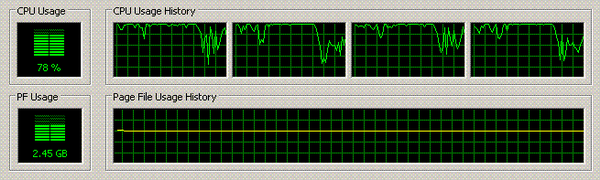
Where Email = '…'

But when you have a giant database on your production server, running any of these will bring your server down. The reason is, although these queries look like very obvious ones that you will be using frequently, none of these are part of any index. So, all of the above results in "Table Scan" (worst case for any query) on millions of rows on respective tables.

Here's what happened to us. We used such fields like UserName, Email, UserID, IsAnonymous etc. on lots of marketing reports at [Pageflakes](http://www.pageflakes.com/). These are some reports which only marketing team use, no one else. Now, the site runs fine but several times a day marketing team and users used to call us and scream "Site is slow!", "Users are reporting extreme slow performance!", "Some pages are getting timed out!" etc. Usually when they call us, we tell them "Hold on, checking right now" and we check the site thoroughly. We use SQL profiler to see what's going wrong. But we cannot find any problem anywhere. Profiler shows queries running file. CPU load is within parameters. Site runs nice and smooth. We tell them on the phone, "We can't see any problem, what's wrong?"

So, why can't we see any slowness when we try to investigate the problem but the site becomes really slow several times throughout the day when we are not investigating?

Marketing team sometimes run analysis reports that use queries like the above several times per day. Whenever they run any of those queries, as the fields are not part of any index, it makes server IO go super high and CPU also goes super high - something like this:



We have SCSI drives which have 15000 RPM, very expensive, very fast. CPU is Dual core Dual Xeon 64bit. Both are very powerful hardware of their kind. Still these queries bring us down due to huge database size.

But this never happens when marketing team calls us and we keep them on the phone and try to find out what's wrong. Because when they are calling us and talking to us, **they are not running any of the reports which bring the servers down**. They are working somewhere else on the site, mostly trying to do the same things complaining users are doing.

Let's look at the indexes:

Table: aspnet\_users

* Clustered Index = ApplicationID, LoweredUserName
* NonClustered Index = ApplicationID, LastActivityDate
* Primary Key = UserID

Table: aspnet\_membership

* Clustered Index = ApplicationID, LoweredEmail
* NonClustered = UserID

Table: aspnet\_Profile

* Clustered Index = UserID

Most of the indexes have ApplicationID in it. Unless you put ApplicationID='…' in the WHERE clause, it's not going to use any of the indexes. As a result, all the queries will suffer from Table Scan. Just put ApplicationID in the where clause (Find your ApplicationID from aspnet\_Application table) and all the queries will become blazingly fast.

**DO NOT use**Email **or**UserName **fields in**WHERE **clause. They are not part of the index instead**LoweredUserName **and**LoweredEmail **fields are in conjunction with**ApplicationID **field. All queries must have**ApplicationID **in the**WHERE **clause.**

Our Admin site which contains several of such reports and each contains lots of such queries on aspnet\_users,aspnet\_membership and aspnet\_Profile tables. As a result, whenever marketing team tried to generated reports, they took all the power of the CPU and HDD and the rest of the site became very slow and sometimes non-responsive.

Make sure you always cross check all your queries WHERE and JOIN clauses with index configurations. Otherwise you are doomed for sure when you go live.

## Prevent Denial of Service (DOS) Attack

Web services are the most attractive target for hackers because even a pre-school hacker can bring down a server by repeatedly calling a Web service which does expensive work. Ajax Start Pages like [Pageflakes](http://www.pageflakes.com/) are the best target for such DOS attack because if you just visit the homepage repeatedly without preserving cookie, every hit is producing a brand new user, new page setup, new widgets and what not. The first visit experience is the most expensive one. Nonetheless, it’s the easiest one to exploit and bring down the site. You can try this yourself. Just write a simple code like this:

http://www.codeproject.com/images/minus.gif Collapse | [Copy Code](http://www.codeproject.com/Articles/23306/10-ASP-NET-Performance-and-Scalability-Secrets)

for( int i = 0; i < 100000; i ++ )

{

WebClient client = new WebClient();

client.DownloadString("http://www.pageflakes.com/default.aspx");

}

To your great surprise, you will notice that, after a couple of calls, you don't get a valid response. It’s not that you have succeeded in bringing down the server. It’s that your requests are being rejected. You are happy that you no longer get any service, thus you achieve Denial of Service (for yourself). We are happy to Deny You of Service (DYOS).

The trick I have in my sleeve is an inexpensive way to remember how many requests are coming from a particular IP. When the number of request exceeds the threshold, deny further request for some duration. The idea is to remember caller’s IP in ASP.NET Cache and maintain a count of request per IP. When the count exceeds a predefined limit, reject further request for some specific duration like 10 mins. After 10 mins, again allow requests from that IP.

I have a class named ActionValidator which maintains a count of specific actions like First Visit, Revisit, Asynchronous postbacks, Add New widget, Add New Page etc. It checks whether the count for such specific action for a specific IP exceeds the threshold value or not.

http://www.codeproject.com/images/minus.gif Collapse | [Copy Code](http://www.codeproject.com/Articles/23306/10-ASP-NET-Performance-and-Scalability-Secrets)

public static class ActionValidator

{

private const int DURATION = 10; *// 10 min period*

public enum ActionTypeEnum

{

FirstVisit = 100, *// The most expensive one, choose the value wisely.*

ReVisit = 1000, *// Welcome to revisit as many times as user likes*

Postback = 5000, *// Not must of a problem for us*

AddNewWidget = 100,

AddNewPage = 100,

}

The enumeration contains the type of actions to check for and their threshold value for a specific duration – 10 mins.

A static method named IsValid does the check. It returns true if the request limit is not passed, false if the request needs to be denied. Once you get false, you can call Request.End() and prevent ASP.NET from proceeding further. You can also switch to a page which shows “Congratulations! You have succeeded in Denial of Service Attack.”

http://www.codeproject.com/images/minus.gif Collapse | [Copy Code](http://www.codeproject.com/Articles/23306/10-ASP-NET-Performance-and-Scalability-Secrets)

public static bool IsValid( ActionTypeEnum actionType )

{

HttpContext context = HttpContext.Current;

if( context.Request.Browser.Crawler ) return false;

string key = actionType.ToString() + context.Request.UserHostAddress;

var hit = (HitInfo)(context.Cache[key] ?? new HitInfo());

if( hit.Hits > (int)actionType ) return false;

else hit.Hits ++;

if( hit.Hits == 1 )

context.Cache.Add(key, hit, null, DateTime.Now.AddMinutes(DURATION),

System.Web.Caching.Cache.NoSlidingExpiration,

System.Web.Caching.CacheItemPriority.Normal, null);

return true;

}

The cache key is built with a combination of action type and client IP address. First it checks if there’s any entry for the action and the client IP in cache or not. If not, start the count and remember the count for the IP in cache for the specific duration. The absolute expiration on cache item ensures that after the duration the cache item will be cleared and the count will restart. When there’s already an entry in the cache, get the last hit count, and check if the limit is exceeded or not. If not exceeded, increase the counter. There is no need to store the updated value in the cache again by doing: Cache[url]=hit; because the hit object is by reference and changing it means it gets changed in the cache as well. In fact, if you do put it again in the cache, the cache expiration counter will restart and fail the logic of restarting count after specific duration.

The usage is very simple, on the default.aspx:

http://www.codeproject.com/images/minus.gif Collapse | [Copy Code](http://www.codeproject.com/Articles/23306/10-ASP-NET-Performance-and-Scalability-Secrets)

protected override void OnInit(EventArgs e)

{

base.OnInit(e);

*// Check if revisit is valid or not*

if( !base.IsPostBack )

{

*// Block cookie less visit attempts*

if( Profile.IsFirstVisit )

{

if( !ActionValidator.IsValid(ActionValidator.ActionTypeEnum.FirstVisit))

Response.End();

}

else

{

if( !ActionValidator.IsValid(ActionValidator.ActionTypeEnum.ReVisit) )

Response.End();

}

}

else

{

*// Limit number of postbacks*

if( !ActionValidator.IsValid(ActionValidator.ActionTypeEnum.Postback) )

Response.End();

}

}

Here I am checking specific scenario like first Visit, re-visit, postbacks etc.

Of course you can put in some Cisco firewall and prevent DOS attack. You will get a guarantee from your hosting provider that their entire network is immune to DOS and DDOS (Distributed DOS) attacks. What they guarantee is network level attack like TCP SYN attacks or malformed packet floods etc. There is no way they can analyze the packet and find out a particular IP is trying to load the site too many times without supporting cookie or trying to add too many widgets. These are called application level DOS attack which hardware cannot prevent. It must be implemented in your own code.

There are very few websites out there which take such precaution for application level DOS attacks. Thus it’s quite easy to make servers go mad by writing a simple loop and hitting expensive pages or Web services continuously from your home broadband connection. I hope this small but effective class will help you prevent DOS attack in your own Web applications.

## Conclusion

You have learned several tricks to push ASP.NET to its limit in order to deliver faster performance out of the same hardware configuration. You have also learned some handy AJAX techniques to make your site load and feel faster. Finally, you have learned how to defend against large number of hits and spread static content over Content Delivery Networks to withstand large traffic demands. All these techniques can make your site load faster, feel smoother and serve much higher traffic at lower cost. You can learn a lot more about performance and scalability improvement of ASP.NET and ASP.NET AJAX websites from my book - [Building a Web 2.0 portal using ASP.NET 3.5](http://www.oreilly.com/catalog/9780596510503/).