IIS and ASP.NET

Regardless of whether you use vanilla ASP.NET or MVC, you use IIS to serve up your web pages. Properly configured, IIS can certainly help improve your application throughput but, misconfigured, it can be a major bottleneck. Some of these configuration settings are out of the scope of memory management but, to be honest, *all* settings need to be carefully reviewed to ensure that they are not causing problems.

Caching

Caching is a broad and subtle art, and so I'm going to devote a fair amount of attention to it. It is common to try and improve application performance with caching. If a page changes rarely, or is expensive to retrieve or calculate, caching can be an easy way to create a more responsive user experience. While this is a common strategy, it is not without risks. If not properly implemented, this clever performance enhancement can easily create more problems than you were trying to solve, as I'm sure you can imagine.

You can easily cache a page in its entirety:

```
<%@ OutputCache Duration="240" VaryByParam="none" %>
```

Listing 5.1: Simple caching.

This means that the page will not be processed again for 240 seconds, and the exact same page will be returned, unchanged, for every request in that timeframe. This can take some load off the web server and make key pages seem more responsive. We have to balance this against the amount of memory that caching will consume. All of this cached memory will ultimately show up in the memory space for the worker process. If we cache too much data, memory usage will grow out of hand, and the web server may spend more time than it should in garbage collections or recycling the web server to reclaim these resources. This is a delicate balance that must be determined for your own application and server configuration.

You may not always be able to cache an entire page. Indeed, caching the whole page would often not make sense. For example, perhaps caching the entire page would take up too much memory, or perhaps there would be too many copies based on the number of potential versions. A more common practice is to cache only parts of the page. You can pull the parts that can be cached into user controls and cache them, while keeping the rest of the page dynamic.

The caching directive has several parameters, of which two are required:

- the `Duration` parameter sets how many seconds the page cache will be retained
- the `VaryByParam` parameter specifies `QueryString` or `postback` variables on which different caches will be served, but the value " `none` " indicates that this is to be ignored.

Other values can be included and separated with semicolons. For example, the `page;count` value indicates that a different cache should be stored for each encountered combination of variables `page` and `count.` However you handle your caching, you want to limit the number of parameters listed and the number of unique combinations specified, otherwise there will be too many copies of the page or control in cache. Trying to cache the paging output for a grid could easily consume all of your memory. Consider a paging scenario where you had 20 pages with 5 potential values for the count variable. In this scenario, there would be 100 copies of the page or control being cached. As you add parameters, multiply the number of potential values for each parameter together, to get the number of potential copies.

Sometimes, instead of a set duration, we might like to refresh a page or control's cache based upon modifications to SQL data. This requires setting up a SQL Cache Dependency, and enabling SQL Server Service Broker on the database. There are a couple of different ways to set up the Service Broker. We will step through one approach in some detail, but if you do not need these details, feel free to skip to the next section.

To start with, exclusive access to the database is necessary to enable the broker.

```
ALTER DATABASE MyDatabase SET SINGLE_USER WITH ROLLBACK IMMEDIATE
ALTER DATABASE  MyDatabase  SET ENABLE_BROKER
ALTER DATABASE  MyDatabase  SET MULTI_USER
```

Listing 5.2: Setting up access to your SQL database.

We also need to run `aspnet_regsql` to register the database and tables for notification.

```
aspnet_regsql -d  MyDatabase  -E -S . —ed
aspnet_regsql -d  MyDatabase  -E -S . -et -t ReferenceTable
```

Listing 5.3: Registering the database and tables for notification.

`aspnet_regsql` is a handy tool when it comes to creating key databases used by the membership provider for ASP.NET. It can also be used to add or remove options on an existing database. In this case, we will use it to configure cache dependency options. If you want to know a bit more about how to use it, Table 5.1 summarizes some of the key options.

-d	The name of the database to act on
-E	Tells the command to use windows authentication
-S	The name of the server. "." Means to use the current server
-ed	Enables SQL cache dependency
-dd	Disables SQL cache dependency
-et	Enables SQL cache dependency on a table
-dt	Disables SQL cache dependency on a table
-t	Specifies which table to enable caching on
-lt	Lists the tables that have cache dependency enabled

Table 5.1: The key options for `aspnet_regsql.`

Finally, we need to configure the caching element in the config file (Listing 5.4).

```
<configuration>
 <system.web>
  <compilation debug="false" targetFramework="4.0" />
  <caching>
   <sqlCacheDependency enabled = "true" pollTime = "60000" >
    <databases>
     <add name="ReferenceData"
      connectionStringName="ConnectionString"
      pollTime="6000000" />
     </databases>
    </sqlCacheDependency>
   </caching>
 </system.web>
 <connectionStrings>
   <add name="ConnectionString" connectionString="…"/>
 </connectionStrings>
</configuration>
```

Listing 5.4: Making sure the caching element in the `config` file is correctly set up.

Once everything is configured, we can now include the `SQLDependency` in the `OutputCache` directive (Listing 5.5).

```
<%@ OutputCache Duration="600" VaryByParam="none"
SqlDependency="ReferenceData:ReferenceTable"%>
```

Listing 5.5: Caching based on a database dependency.

This configuration will cause the page or control to be refreshed whenever the data in the `ReferenceTable` is updated.

Cache limits

We also have various cache limit configuration settings that we need to be aware of. These limits are intended to help ensure that caching does not create memory problems but, if misconfigured, these limits could undermine your caching strategy.

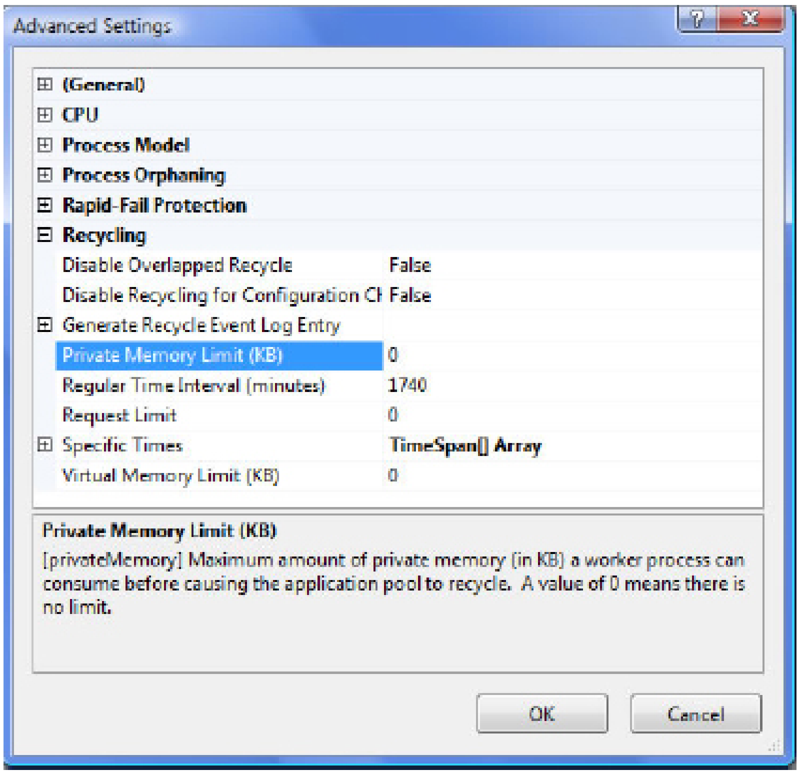
The cache element in the `web.config` file defines the following attributes to help manage the size of your caches. When any of these limits are exceeded, the data in the cache will be trimmed.

privateBytesLimit	This is the maximum number of bytes that the host process is permitted to consume before the cache is trimmed. This is not the memory dedicated to the cache. This is the total memory usage by the w3wp process (in IIS7)
percentage PhysicalMemory UsedLimit	This is the maximum percentage of RAM that the machine can use before the cache is trimmed. With 4GB of RAM and this attribute set to 80%, the cache would be trimmed if Task Manager reported over 3.2gb (i.e. 80% of 4GB) was in use. This is not the percentage memory used by ASP.NET or caching; this is the memory used machine-wide.

Table 5.2: Attributes in the cache element of the ASP.NET `web.config` file.

IIS7 provides another configuration point that may complicate matters from a caching perspective. The Private Memory Limit is an advanced property of the App Pool, which defines the maximum number of private bytes a worker process can consume before it is recycled. Among other things, this will clear the cache and reclaim all of the memory used by the worker process. If you set this limit lower than `privateBytesLimit` in `web.config`, you'll always get a recycle before a cache trim. You will probably always want to set this value higher than the `privateBytesLimit` so that trimming of the cache can have a chance to solve memory pressure before recycling the worker process.

Caching can be a great way to make an application more responsive. Used appropriately, caching can solve performance problems, but abusing caching (by not carefully tracking what is being cached) can create a memory hog in even a well-behaved application. When things go wrong, the cache limits can control the damage and prevent your application from running out of memory.



Disable Overlapped Recycle	False
Disable Recycling for Configuration Ch	False
Generate Recycle Event Log Entry	
Private Memory Limit (KB)	0
Regular Time Interval (minutes)	1740
Request Limit	0
Specific Times	TimeSpan[] Array
Virtual Memory Limit (KB)	0

Private Memory Limit (KB)
[privateMemory] Maximum amount of private memory (in KB) a worker process can consume before causing the application pool to recycle. A value of 0 means there is no limit.

Figure 5.1: App Pool configuration for the Private Memory Limit.

Debug

Inside the `web.config`, there is a critical setting (see Listing 5.6). This is the default setting, but it has an unfortunate side effect from a performance perspective.

```
<compilation debug="true">
```

Listing 5.6: Turning on debugging in `web.config.`

This setting will inject debug symbols into the compiled assemblies. It also disables batch compilation, which means that each page will be compiled into separate assemblies. More specifically, each `aspx, ascx`, and `asax` will get its own assembly. In many web applications this could easily mean having dozens of assemblies and, as well as being unwieldy, these extra assemblies can also lead to memory fragmentation. Remember that each assembly has its associated metadata, and when you have assemblies with only one class in them, you can easily have more metadata than application data.

As I mentioned a moment ago, each of these assemblies will be compiled with debug symbols, resulting in poorer performance, and meaning that the GC will not work as effectively as in a release build. Essentially, the GC will be less aggressive in reclaiming memory when debug symbols are included. Since the debug symbols are included, the GC needs to be prepared that a debugger could be attached, and many of the rules for identifying unreachable references may not be applicable. With a debugger attached, a lot more objects may be reachable.

So, there is an overhead for each assembly, and since each assembly is compiled in debug mode there is *additional* overhead for the debugging details. But that's not all. Finally, each assembly gets its own block of memory for its metadata, meaning that there will be space between the assemblies. With a large number of assemblies, that means we'll quickly start to see fragmentation within the Virtual Address Space, making it increasingly harder to find large enough spaces to store the managed heaps, which then runs the risk of causing out of memory exceptions.

Even if you have debug set to `false`, you may still end up with multiple assemblies. “Why?” I hear you ask. This is because even with debug set to `false`, edits to a page will result in recompiling that one page into a separate assembly. Fortunately, the framework is prepared for this deployment strategy, and provides the `numRecompiles BeforeAppRestart` configuration setting.

Thus, if you need to update individual pages frequently, you can set the `numRecompiles BeforeAppRestart` attribute in the compilation section of the `web.config` to force an App restart after a certain number of recompiles. This can at least limit the potential damage. Just to give you a quiet sense of foreboding, the default value of this attribute is 15, meaning that you could have 15 extra assemblies before the application recompiles.

As always, test with different values to see what works best with your application. You have to balance the number of assemblies created with how often you are forcing an application restart. Of course, you need to rethink your deployment strategy if it relies on editing deployed pages individually.

StringBuilder

Every control is built using a `StringBuilder.` The `Render` method accepts an `HTMLTextWriter`, which is backed by a `StringBuilder`, and the contents of the `StringBuilder` are ultimately written back to the client as the HTML that is displayed. The `StringBuilder` is a good choice for string concatenation and offers much better performance than a simple string.

Much like we saw in the previous chapter with the `List`, a `StringBuilder` grows to accommodate the string being built. Internally, the `StringBuilder` uses a string to hold the string being built, and uses an `EnsureCapacity` method to verify that this internal string will hold the string that is being built. This method has the same potential problem that we saw with `List;` namely, that the internal string will double in size every time it has to grow. This will result in lots of intermediate strings being created, which will all be garbage collected, but will also cause your application to spend more time garbage collecting than it should, as a result. When you create the `StringBuilder`, you can explicitly initialize the size of the internal buffer and, as we saw in Chapter 4, *Common Memory Problems*, this can substantially reduce the memory pressure in your application.

Unfortunately, you don't have the option to explicitly create the `StringBuilder.` However, you can limit the potential problems by reducing the number of controls you implement, as well as the amount of HTML needed to render these controls.