Tweaking the ASP.NET Environment

In addition to our code, every incoming request and outgoing response has to go through ASP.NET's components. Some of ASP.NET's mechanisms were created to serve the developer's needs, such as the ViewState mechanism, but can affect the overall performance of our application. When fine-tuning ASP.NET applications for performance, it is advisable to change the default behavior of some of these mechanisms, although changing them may sometimes require changing the way your application code is constructed.

Turn Off ASP.NET Tracing and Debugging

ASP.NET Tracing enables developers to view diagnostic information for requested pages, such as execution time and path, session state, and HTTP headers list.

Although tracing is a great feature and provides added value when developing and debugging ASP.NET applications, it has some effects on the overall performance of the application, due to the tracing mechanism and the collection of data which is performed on each request. Therefore, if you have enabled tracing during development, turn it off before deploying your web application to the production environment, by changing the trace settings in the web.config:

```
<configuration>
  <system.web>
  <trace enabled = "false"/>
  </system.web>
</configuration>
```

Note The default value of trace, if not specified otherwise in the web.config, is disabled (`enabled = "false"`), so removing the trace settings from the web.config file will also disable it.

When creating new ASP.NET web applications, one of the things automatically added to the application's web.config file is the `system.web` ➤ `compilation` configuration section with the `debug` attribute set to `true`:

```
<configuration>
  <system.web>
  <compilation debug = "true" targetFramework = "4.5" />
  </system.web>
</configuration>
```

Note This is the default behavior when creating ASP.NET web applications in Visual Studio 2012 or 2010. In prior versions of Visual Studio the default behavior was to set the debug setting to `false`, and when the developer first tried to debug the application, a dialog appeared asking permission to change the setting to `true`.

The issue with this setting is that developers often neglect to change the setting from `true` to `false` when they deploy the application to production, or even do it intentionally to get more detailed exception information. In fact, keeping this setting can lead to several performance problems:

- Scripts that are downloaded using the WebResources.axd handler, for example when using validation controls in pages, will not be cached by the browser. When setting the `debug` flag to `false`, responses from this handler will be returned with caching headers, allowing browsers to cache the response for future visits.
- Requests will not timeout when debug is set to `true`. Although this is very convenient when debugging the code, this behavior is less desired in production environments when such requests can lead to the server not being able to handle other incoming requests, or even cause extensive CPU usage, increased memory consumption and other resource allocation issues.
- Setting the `debug` flag to false will enable ASP.NET to define timeouts for requests according to the `httpRuntime` ➤ `executionTimeout` configuration settings (the default value is 110 seconds).
- JIT optimizations will not be applied to the code when running with `debug = true`. JIT optimizations are one of the most important advantages of .NET and can efficiently improve the performance of your ASP.NET application without requiring you changing your code. Setting `debug` to `false` will allow the JIT compiler to perform its deeds, making your application perform faster and more efficient.
- The compilation process does not use batch compilations when using `debug = true`. Without batch compilation an assembly will be created for each page and user control, causing the web application to load dozens and even hundreds of assemblies during runtime; loading that many assemblies may cause future memory exceptions due to fragmented address space. When the debug mode is set to false, batch compilation is used, generating a single assembly for the user controls, and several assemblies for the pages (pages are grouped to assemblies according to their use of user controls).

Changing this setting is very easy: either remove the `debug` attribute entirely from the configuration, or set it to `false`:

```
<configuration>
```

```
  <system.web>
  <compilation debug = "false" targetFramework = "4.5" />
  </system.web>
</configuration>
```

In case you fear forgetting to change this setting when deploying applications to production servers, you can force *all* ASP.NET applications in a server to ignore the `debug` setting by adding the following configuration in the server's machine.config file:

```
<configuration>
  <system.web>
  <deployment retail = "true"/>
  </system.web>
</configuration>
```

Disable View State

View state is the technique used in ASP.NET Web Forms applications to persist a page's state into the rendered HTML output (ASP.NET MVC applications do not use this mechanism). View state is used to allow ASP.NET to keep the state of a page between postbacks performed by the user. The view state data is stored in the HTML output by serializing the state, encrypting it (not set by default), encoding it to a Base64 string, and storing it in a hidden field. When the user posts back the page, the content is decoded, and then deserialized back to the view state dictionary. Many server controls use the view state to persist their own state, for example storing their property values in the view state.

Although very useful and powerful, this mechanism generates a payload that, when placed in the page as a Base64 string, can increase the size of the response by a magnitude. For example, a page containing a single GridView with paging, bounded to a list of 800 customers, will generate an output HTML 17 KB in size, out of which 6 KB is the view state field—this is because `GridView` controls store their data source in the view state. In addition, using view state requires serializing and deserializing the view state on each request, which adds additional overhead to the processing of the page.

Tip The payload created by using view state is usually not noticeable by clients that access a web server in their own local area network. This is because LANs are usually very fast and able to transfer very large pages in a matter of milliseconds (an optimal 1Gb LAN can reach a throughput of ~40–100 MB/s, depending on hardware). However, the payload of the view state is most notable when using slow wide area networks, such as the Internet.

If you do not require the use of the view state, it is advisable to disable it. View state can be disabled for the entire application by disabling it in the web.config file:

```
<system.web>
  <pages enableViewState = "false"/>
</system.web>
```

If you don't wish to disable view state for the entire application, it can also be disabled for a single page and all of its controls:

```
<%@ PageEnableViewState = "false". . . %>
```

You can also disable view state per control:

```
<asp:GridView ID = "gdvCustomers" runat = "server" DataSourceID = "mySqlDataSource"
  AllowPaging = "True"EnableViewState = "false"/>
```

Prior to ASP.NET 4, disabling the view state in the page made it impossible to re-enable it for specific controls in the page. As of ASP.NET 4, a new approach was added to allow disabling view state on a page, but re-enabling it for specific controls. This is achieved in ASP.NET 4 by using the `ViewStateMode` property. For example, the following code disables the view state for the entire page, excluding the GridView control:

```
<%@ Page EnableViewState = "true"ViewStateMode = "Disabled". . . %>

<asp:GridView ID = "gdvCustomers" runat = "server" DataSourceID = "mySqlDataSource"
  AllowPaging = "True"ViewStateMode = "Enabled"/>
```

Caution Disabling the view state by setting the `EnableViewState` to `false` will override any setting done to the `ViewStateMode`. Therefore, if you wish to use the `ViewStateMode`, make sure `EnableViewState` is set to `true` or omitted (the default value is `true`).

Server-Side Output Cache

Although ASP.NET pages are considered dynamic in content, you often end up in scenarios where the dynamic content of a page does not necessarily change over time. For example, a page can receive the ID of a product and return an HTML content describing that product. The page itself is dynamic, because it can return different HTML content for different products, but the product page for a specific product does not change that often, at least not until the product details themselves change in the database.

Continuing our product example, to prevent our page from requerying the database every time a product is requested, we may want to cache that product information in a local cache so we can access it faster, but still we will need to render the HTML page every time. Instead of caching the data we need, ASP.NET offers a different caching mechanism, the ASP.NET Output Cache, which caches the outputted HTML itself.

By using output cache, ASP.NET can cache the rendered HTML so subsequent requests will automatically receive the rendered HTML without needing to execute our page's code. Output cache is supported in ASP.NET Web Forms for caching pages, as well as in ASP.NET MVC for caching controller actions.

For example, the following code uses output cache to cache the view returned by an ASP.NET MVC controller's action for 30 seconds:

```
public class ProductController : Controller {
  [OutputCache(Duration = 30)]
  public ActionResult Index() {
  return View();
  }
}
```

If the index action in the above example received an ID parameter and returned a view displaying specific product information, we would need to cache several versions of the output, according to the different IDs the action receives. Output cache, therefore, supports not just a single caching of the output, but also supports caching different outputs of the same action according to the parameters passed to that action. The following code shows how to alter the action to cache outputs according to an ID parameter passed to the method:

```
public class ProductController : Controller {
  [OutputCache(Duration = 30, VaryByParam = "id")]
  public ActionResult Index(int id) {
  //Retrieve the matching product and set the model accordingly . . .
  return View();
  }
}
```

Note In addition to varying by query string parameters, output cache can also vary the cached output by the request's HTTP headers, such as the Accept-Encoding and Accept-Language header. For example, if your action returns content in different languages according to the Accept-Language HTTP header, you can set the output cache to vary by that header, creating a different cached version of the output for each requested language.

If you have the same caching settings for different pages or actions, you can create a caching profile, and use that profile instead of repeating the caching settings over and over again. Caching profiles are created in the web.config, under the `system.web` ➤ `caching` section. For example, the following configuration declares a caching profile which we want to use in several pages:

```
<system.web>
  <caching>
    <outputCacheSettings>
      <outputCacheProfiles>
         <add name = "CacheFor30Seconds" duration = "30" varyByParam = "id"/>
      </outputCacheProfiles>
    </outputCacheSettings>
  </caching>
</system.web>
```

Now the profile can be used for our `Index` action, instead of repeating the duration and parameter:

```
public class ProductController : Controller {
  [OutputCache(CacheProfile = "CacheFor30Seconds")]
  public ActionResult Index(int id) {
  //Retrieve the matching product and set the model
  . . .
  return View();
  }
}
```

We can also use the same caching profile in an ASP.NET web form, by using the `OutputCache` directive:

```
<%@ OutputCache CacheProfile = "CacheEntityFor30Seconds" %>
```

Note By default, the ASP.NET output cache mechanism keeps the cached content in the server's memory. As of ASP.NET 4, you can create your own output cache provider to be used instead of the default one. For example, you can write your own custom provider which stores the output cache to disk.

Pre-Compiling ASP.NET Applications

When compiling an ASP.NET Web application project, a single assembly is created to hold all application's code. However, web pages (.aspx) and user controls (.ascx) are not compiled, and deployed as-is to the server. The first time the web application starts (upon first request), ASP.NET dynamically compiles the web pages and user controls, and places the compiled files in the ASP.NET Temporary Files folder. This dynamic compilation increases the response time of first requests, causing users to experience a slow-loading web site.

To resolve this issue, web applications can be pre-compiled, including all code, pages, and user controls, by using the ASP.NET compilation tool

(Aspnet_compiler.exe). Running the ASP.NET compilation tool in production servers can reduce the delay users experience on first requests. To run the tool, follow these steps:

1. Open a command prompt in your production server.
2. Navigate to the *%windir%\Microsoft.NET* folder
3. Navigate to either the *Framework* or *Framework64* folder, according to the whether the web application's application pool is configured to support 32-bit applications or not (for 32-bit operating systems, the *Framework* folder is the only option).
4. Navigate to the framework version's folder, according to the .NET framework version used by the application pool (*v2.0.50727* or *v4.0.30319*).
5. Enter the following command to start the compilation (replace *WebApplicationName* with the virtual path of your application):
6. `Aspnet_compiler.exe -v /WebApplicationName`

Fine-Tuning the ASP.NET Process Model

When a call is made to an ASP.NET application, ASP.NET uses a worker thread to handle the request. Sometimes, the code in our application can itself create a new thread, for example when calling out to a service, thus reducing the number of free threads in the thread pool.

To prevent exhaustion of the thread pool, ASP.NET automatically performs several adjustments to the thread pool, and applies several limitations on the number of requests that can execute at any given by. These settings are controlled from three main configuration sections—the `system.web` ➤ `processModel` section, the `system.web` ➤ `httpRuntime` section, and the `system.net` ➤ `connectionManagement` section.

Note The `httpRuntime` and `connectionManagement` sections can be set from within the application's web.config file. The `processModel` section, however, can only be changed in the *machine.config* file.

The `processModel` section controls thread pool limitations such as minimum and maximum number of worker threads, while the `httpRuntime` section defines limitations related to available threads, such as the minimum number of available threads that must exist in order to keep processing incoming requests. The `connectionManagement` section controls the maximum number of outgoing HTTP connections per address.

All of the settings have default values, however, since some of these values are set a bit low, ASP.NET includes another setting, the `autoConfig` setting, which tweaks some of the settings to achieve optimal performance. This setting, which is part of the `processModel` configuration section exists since ASP.NET 2.0, and is automatically set to true.

The `autoConfig` setting controls the following settings (the default values below were obtained from the Microsoft Knowledge Base article KB821268 at `http://support.microsoft.com/?id=821268`):

- `processModel` ➤ `maxWorkerThreads`. Changes the maximum amount of worker threads in the thread pool from 20× the number of cores to 100× the number of cores.
- `processModel` ➤ `maxIoThreads`. Changes the maximum amount of I/O threads in the thread pool from 20× the number of cores to 100× the number of cores.
- `httpRuntime` ➤ `minFreeThreads`. Changes the minimum number of available threads that are required to allow the execution of new requests from 8 to 88× the number of cores.
- `httpRuntime` ➤ `minLocalFreeThreads`. Changes the minimum number of available threads that are required to allow the execution of new local requests (from the local host) from 4 to 76× the number of cores.
- `connectionManagement` ➤ `maxConnections`. Changes the maximum number of concurrent connections from 10 to 12× the number of cores.

Although the above defaults were set in order to achieve optimized performance, there may be cases when you will need to change them, in order to achieve better performance, depending on the scenario you encounter with your web application. For example, if your application calls out to services, you may need to increase the number of maximum concurrent connections to allow more requests to connect to backend services at the same time. The following configuration shows how to increase the number of maximum connection:

```
<configuration>
  <system.net>
  <connectionManagement>
  <add address = "*" maxconnection = "200" />
  </connectionManagement>
  </system.net>
</configuration>
```

In other scenarios, for example when web applications tend to get many requests upon starting, or have a sudden burst of requests, you may need to change the minimum number of worker threads in the thread pool (the value you specify is multiplied at runtime by the number of cores on the machine). To perform this change, apply the following configuration in the machine.config file:

```
<configuration>
```

```
  <system.web>
  <processModel autoConfig = "true" minWorkerThreads = "10"/>
  </system.web>
</configuration>
```

Before you rush to increase the size of minimum and maximum threads, consider the side effects this change may have on your application: if you allow too many requests to run concurrently, this may lead to excessive CPU usage and high memory consumption, which can eventually crash your web application. Therefore, after changing these settings, you must perform load test to verify the machine can sustain that many requests.