exigo

Developing for a Load Balanced Enterprise Environment

v 1.0

Asp.net offers a variety of choices and options when building your web solution. If you plan to deploy your solution to an environment which features high availability, load balancing, SSL off-loaders, and web gardens, there are a few design considerations you'll need to be aware of.

The following outlines a few best practices to ensure your site functions as intended once you deploy to your enterprise environment.

TURN OFF SESSION STATE

The built in "Session" object in asp.net provides a quick and convenient place to store user session info. The default implementation for session state is "InProc" which stores the data in local memory cache. Load balanced environments typically have your site running on two or more machines. In addition web gardens are typically used on each machine. A web garden is a mechanism where multiple processes are created to service your site on the same machine. This allows for a round-robin approach to better handle workload. Each request that comes in for your application will be first round-robin routed to one of the machines in the web farm, and then on the individual machine your request will be round-robin routed to one of the process in your web garden. The "InProc" session state is local to one instance on one machine. There is no guarantee that a second request from the same user will be routed to both the same server and the same web garden instance. This generates the effect of the user losing his session prematurely, often on the very next request.

Ensure session objects are not enabled in your application placing the following in your web.config under <system.web>:

```
<sessionState mode="Off" />
```

Where should you store session state? This will depend on the type of state you are storing.

For login identity information, it is best stored using an encrypted cookie. With a cookie, every request to your site will transmit the same contents of data regardless of which server or web garden instance it hits. You simply decrypt the cookie and establish the identity of the user.

Often the size of data needed to be stored exceeds the limit of what a cookie can hold. An example of this might be a shopping cart checkout session which visits multiple pages and needs to keep track of the transaction on each page. In this scenario you would typically use a single server side storage which is linked to a unique temporary key. This unique key can then be stored in the cookie.

In Exigo, there are two API methods which expose a simple set and get for centrally stored session data:

http://api.exigo.com/3.0/ExigoApi.asmx?op=SetSession

http://api.exigo.com/3.0/ExigoApi.asmx?op=GetSession

However any centrally accessible repository such as a database or other API methods can be used.

You'll also want to think through security implications. Make sure the session data invalidates if the transaction is complete or a particular time period has passed. This is especially important if you expose the unique session key in the URL of your pages. Think through how your site behaves if another user visits the history on the same computer or what would happen if a checkout link was emailed to a different user. A simple random value found both in a cookie as well as the stored session state should solve the above issue, assuming the cookie has a finite life.

DETECT SECURE CONNECTION

For login areas, shopping cart checkout sections and other sensitive pages in your solution, you'll want to enforce SSL. However because the SSL certificate is handled at the load balancer level your application will always report running in non-SSL mode even if the end user is currently navigating to the https version of your pages.

A typical check you might have could look like this:

```
if (!Request.IsSecureConnection)
Response.Redirect("https://" + Request.Url.Host + Request.RawUrl);
```

However this would create an endless loop. As each request back to your page would report that it is “not secure” and each response would be a redirect command to the secure version of your page.

To help detect pages running in secure mode, the load balancer will add an http header called “SSL”.

The following could be called from your Page_Load function. It tests for a server side http header called “SSL”. It also bypasses the redirect for local development and has a measure to prevent endless loops:

```
  void Page_Load(object sender, EventArgs e)
  {
if (!EnsureSecurePage()) return;
  }

  bool EnsureSecurePage()
  {
       if (Request.IsSecureConnection)               return true; //local ssl cert
       if (Request.Headers["SSL"]!=null)             return true; //ssl off-loader
       if (Request.QueryString["pl"]!=null)          return true; //prevent endless loop
       if (Request.IsLocal)                          return true; //allow local testing
       Response.Redirect("https://" + Request.Url.Host +
            Request.RawUrl + (Request.RawUrl.Contains("?") ? "&" : "?") + "pl=1"); return
       false;
  }
```

TURN OFF DEBUG MODE

Finally, make sure your deployed site is not still in debug mode. While not a requirement, your site will perform better in the live environment. You'll find this in the compilation statement on your web.config. The debug attribute should either be set to false or removed altogether. One tip is to set debug=”true” on the web.config for your local machine found at c:\windows\microsoft.net\framework64\v4.0.xxxx\config\web.config. You would then set your website version to not include the debug statement. This way local development always has the benefit of debug environment but production machines do not accidentally run in debug mode. Your local web.config's compilation should look like:

```
<compilation targetFramework="4.0" />
```

Substitute 4.0 with the framework version you are targeting.

ALWAYS BUILD

In Asp.net you have the option to create a “Web Site” or a “Web Application”. With a Web Application you are forced to compile before you deploy each time. Web Sites on the other hand, allow you to deploy files where the server compiles your source code on the fly. The enterprise environment can support either type. However if you are deploying “Web Site”, as a best practice, always “Build” before deployment. The “Build” will not actually generate any dlls of your site but will catch any code which might not compile. The way the load balancers work is that if any site starts to return HTTP 500 errors, which can happen with a compile error, it will take that machine temporarily out of the farm rotation. If all machines start to return that error, it will take them all offline and your site will appear down until the next re-check cycle.

If you always “Build”, you’ll be less likely to upload code which does not compile and you ensure you always have good uptime in your live environment.

CONCLUSION

It’s fairly simple to generate enterprise-ready solutions. By taking the above into consideration when designing your site, you’ll be ready to deploy into an environment engineered for high-availability and rapid scalability to a large number of concurrent users.

Please direct any questions to davidt@exigo.com