

Username: Pralay Patoria **Book:** Pro .NET Performance. No part of any chapter or book may be reproduced or transmitted in any form by any means without the prior written permission for reprints and excerpts from the publisher of the book or chapter. Redistribution or other use that violates the fair use privilege under U.S. copyright laws (see 17 USC107) or that otherwise violates these Terms of Service is strictly prohibited. Violators will be prosecuted to the full extent of U.S. Federal and Massachusetts laws.

Scaling ASP.NET Applications

So you've improved the performance of your web application by incorporating everything you learned here, maybe even applied some other techniques to boost things up, and now your application is working great, and is as optimized as it can be. You put your application to production, everything works great for the first couple of weeks, but then additional users start using the web site, and more new users are joining every day, increasing the number of requests your server has to handle, and suddenly, your server starts to choke. It may begin with requests taking longer than usual, your worker process starts to use more memory and more CPU, and eventually requests give in to timeouts, and HTTP 500 ("Internal Server Error") messages are filling your log files.

What has gone wrong? Should you try to improve the performance of your application again? More users will join and again you'll be in the same situation. Should you increase the machine's memory, or add more CPUs? There's a limit to the amount of scale-up you can do on a single machine. It's time to face the facts—you need to scale out to more servers.

Scaling out web applications is a natural process that has to occur at some point in the life cycle of web applications. One server can hold dozens and even thousands of concurrent users, but it cannot handle that kind of stress for that long. Session state filling up the server's memory, threads getting starved because there are no more threads available, and context-switching that occurs too often will eventually increase the latency and lower the throughput of your single server.

Scaling Out

From an architectural point of view, scaling is not hard: just buy another machine or two (or ten), place the servers behind a load-balancer, and from there on, all should be OK. The problem is that usually it's not that easy.

One of the major issues developers face when having to scale is how to handle *server affinity*. For example, when you use a single web server, the session state of your users is kept in-memory. If you add another server, how will you make those session objects available to it? How will you synchronize the sessions between the servers? Some web developers tend to solve this issue by keeping the state in the server and having an affinity between the client and the server. Once a client connects to one of the servers through the load balancer, the load balancer will, from that point on, keep sending all of that client's requests to the same web server, this is also referred to as a "sticky" session. Using sticky sessions is a workaround, not a solution, since it does not allow you to truly balance the work between the servers. It is easy getting to a situation where one of the servers is handling too many users, whereas other servers are not handling requests at all, because all of their clients already disconnected.

So the real solution for good scalability is not to rely on your machine's memory, whether its regarding the state you keep for your users, or the cache you store in-memory to boost performance. Why does storing cache on a machine become problematic when scaling? Just think what will happen when a user sends a request that causes the cache to be updated: the server that got the request will update its in-memory cache, but other servers won't know of this change, and if they also have a copy of that cached object, it will become stale and cause application-wide inconsistencies. One of the ways you might choose to solve this issue is by synchronizing cached objects between servers. Although possible, this solution adds another layer of complexity to the architecture of your web application, not to mention the amount of chatter your servers are going to have between them.

ASP.NET Scaling Mechanisms

Scaling out to multiple servers requires an out-of-process state management. ASP.NET has two built-in out-of-process mechanisms for state management:

- **State Service.** The state service is a Windows service that provides state management for multiple machines. This service is installed automatically when installing the .NET framework, but is turned off by default. You can simply choose which machine will run the state service, and configure all other machines to use it. Although state service enables several machines to use the same session store, it does not provide persistency, so if something happens to the machine hosting the service, the entire session state of your web farm will be lost.
- **SQL Server.** ASP.NET supports storing the session state in SQL Server. Storing state in SQL Server gives the same sharing abilities as that of the state service, but it also enables the persistency of the state, so even if your web servers fail, and if the SQL Server fails, the state information can be restored.

For caching, the solution in most cases is to use a distributed cache mechanism, such as Microsoft's AppFabric Cache, NCache, or Memcached which is an open-source distributed cache. With distributed cache, you can take several servers and combine their memory into one distributed memory, which is used as a cache store for your cached data. Distributed caches offer abstraction of location so you don't need to know where each piece of data is located, notification services so you can know when something has changed, and high-availability to make sure that even if one of the cache servers fails, the data is not lost.


Some distributed caches, such as AppFabric Cache and Memcached also have custom Session State and Cache providers for ASP.NET.

Scaling Out Pitfalls

Although not relevant to performance, it is a good place to mention some other issues you should be aware of when scaling out your web application. Certain parts in web applications require the use of a special security keys to generate unique identifiers that prevent tampering and spoofing the web application. For example, a unique key is used when creating Forms Authentication cookies, and when encrypting view state data. By default, the security keys for a web application are generated each time the application pool starts. For a single server this might not be a problem, but when you scale out your server into multiple servers this will pose a problem, since each server will have its own unique key. Consider the following scenario: a client sends a request to server A and gets back a cookie signed with server A's unique key, then the client sends a new request to server B with the cookie it received before. Since server B has a different unique key, the validation of the cookie's content will fail and an error will be returned.

You can control how these keys are generated in ASP.NET by configuring the `machineKey` section in your web.config. When scaling out web applications to multiple servers, you need to configure the machine key to use the same pre-generated key in all the servers by hard-coding it into the application's configuration.

Another issue relating to scaling out and the use of unique keys is the ability to encrypt sections in web.config files. Sensitive information located in the web.config file is often encrypted when the application is deployed to production servers. For example, the `connectionString` section can be encrypted to prevent the username and password to the database from being discovered. Instead of encrypting the web.config file separately in each server after deployment, making the deployment process tedious, you can generate one encrypted web.config file and deploy it to all the servers. To be able to do that, all you need to do is to create an RSA key container and import it once in all the web servers.

 **Note** To learn more about generating machine keys and applying them to the application's configuration, consult the Microsoft Knowledge Base document KB312906 (<http://support.microsoft.com/?id=312906>). For more information on generating the RSA key container, read the "Importing and Exporting Protected Configuration RSA Key Containers" MSDN article (<http://msdn.microsoft.com/library/yxw286t2>).
