ASP.NET Caching: Techniques and Best Practices

Steven A. Smith
ASPAlliance.com

August 2003

Applies to:
Microsoft® ASP.NET

Summary: ASP.NET provides three primary forms of caching: page level output caching, user control level output caching (or fragment caching), and the Cache API. Output caching and fragment caching have the advantage of being incredibly simple to implement, and are sufficient in many cases. The cache API provides additional flexibility (quite a lot, in fact), and can be used to take advantage of caching throughout every layer of an application.

Download CacheDemos.msi.

Contents

Of the many features available in ASP.NET, caching support is far and away my favorite, and for good reason. It has the greatest potential impact on an application's performance, out of all the other features in ASP.NET, and it is one of the things that allows ASP.NET developers to accept the additional overhead of building sites using rather heavy controls like DataGrids without fear that performance will suffer too much. In order to see the most benefit from caching in your applications, you should consider ways of implementing caching at all levels of your program.

Steve's Caching Tips

Cache Early; Cache Often

Implement caching at every layer of your application. Add caching support to the data layer, the business logic layer, and the UI or output layer. Memory is cheap—by implementing caching in an intelligent fashion throughout the application, you can achieve great performance gains.

Caching Hides Many Sins

Caching can be a good way to get "good enough" performance without requiring a lot of time and analysis. Again, memory is cheap, so if you can get the performance you need by caching the output for 30 seconds instead of spending a day or a week trying to optimize your code or database, do the caching solution (assuming 30-second old data is ok) and move on. Caching is one of those things where 20% of the work provides 80% of the benefit, so it should be one of the first things you try to improve performance. Eventually, poor design will probably catch up to you, so of course you should try to design your applications correctly. But if you just need to get good enough performance today, caching can be an excellent, buying you time to refactor your application at a later date when you have the time to do so.

Page Level Output Caching

The simplest form of caching, output caching simply keeps a copy of the HTML that was sent in response to a request in memory. Subsequent requests are then sent the cached output until the cache expires, resulting in potentially very large performance gains (depending on how much effort was required to create the original page output—sending cached output is always very fast and fairly constant).

Implementation

To implement page output caching, simply add an OutputCache directive to the page.

```
<%@ OutputCache Duration="60" VaryByParam="*" %>
```

This directive, as with other page directives should appear at the top of the ASPX page, before any output. It supports five attributes (or parameters), two of which are required.

Duration	Required. Time, in seconds, the page should be cached. Must be a positive integer.
Location	Specifies where the output should be cached. If specified, must be one of: Any, Client, Downstream, None, Server or ServerAndClient.
VaryByParam	Required. The names of the variables in the Request, which should result in, separate cache entries. "none" can be used to specify no variation. "*" can be used to create new cache entries for every different set of variables. Separate variables with ";".
VaryByHeader	Varies cache entries based on variations in a specified header.
VaryByCustom	Allows custom variations to be specified in the global.asax (for example, "Browser").

Most situations can be handled with a combination of the required Duration and VaryByParam options. For instance, if you have a product catalog that allows the user to view pages of the catalog based on a categoryID and a page variable, you could cache it for some period of time (an hour would probably be acceptable unless the products change all the time, so a duration of 3600 seconds) with a VaryByParam of "categoryID;page". This would create separate cache entries for every page of the catalog for each category. Each entry would persist for one hour from its first request.

VaryByHeader and VaryByCustom are primarily used to allow customization of the page's look or content based on the client that is accessing them. Perhaps the same URL renders output for both browsers and mobile phone clients, and needs to cache separate versions based on this. Or perhaps the page is optimized for IE but needs to be able to degrade gracefully for Netscape or Opera (instead of just breaking). This last example is common enough that we'll show an example of how to do it:

Example: VaryByCustom to support browser customization
In order to enable separate cache entries for each browser, VaryByCustom can be set to a value of "browser". This functionality is built into the caching module, and will insert separate cached versions of the page for each browser name and major version.

```
<%@ OutputCache Duration="60" VaryByParam="None" VaryByCustom="browser"  %>
```

Fragment Caching, User Control Output Caching

Often, caching an entire page is not feasible, because certain parts of the page are customized for the user. However, there may be other parts of the page that are common to the entire application. These are perfect candidates for caching, using fragment caching and user controls. Menus and other layout elements, especially ones that are dynamically generated from a data source, should be cached with this technique. If need be, the cached controls can be configured to vary based on the changes to its controls (or other properties) or any of the other variations supported by page level output caching. Hundreds of pages using the same controls can also share the cached entries for those controls, rather than keeping separate cached versions for each page.

Implementation

Fragment caching uses the same syntax as page level output caching, but applied to a user control (.ascx file) instead of to a web form (.aspx file). All of the attributes supported by the OutputCache directive on a web form are also supported for user controls except for the Location attribute. User controls also support an OutputCache attribute called VaryByControl, which will vary the caching of the user control depending on the value of a member of that control (typically a control on the page, such as a DropDownList). If VaryByControl is specified, VaryByParam may be omitted. Finally, by default each user control on each page is cached separately. However, if a user control does not vary between pages in an application and is named the same across all such pages, the Shared="true" parameter can be applied to it, which will cause the cached version(s) of the user control to be used by all pages referencing that control.

Examples

```
<%@ OutputCache Duration="60" VaryByParam="*" %>
```

This would cache the user control for 60 seconds, and would create a separate cache entry for every variation of querystring and for every page this control is placed on.

```
<%@ OutputCache Duration="60" VaryByParam="none"
  VaryByControl="CategoryDropDownList" %>
```

This would cache the user control for 60 seconds, and would create a separate cache entry for each different value of the CategoryDropDownList control, and for each page this control is placed on.

```
<%@ OutputCache Duration="60" VaryByParam="none" VaryByCustom="browser"
  Shared="true %>
```

Finally, this would cache the user control for 60 seconds, and would create one cache entry for each browser name and major version. The cache entries for each browser would then be shared by all pages referencing this user control (as long as all pages refer to the control with the same ID).

Caching API, Using the Cache Object

Page and user control level output caching can be a quick and easy way to boost performance of your site, but the real flexibility and power of caching in ASP.NET is exposed via the Cache object. Using the Cache object, you can store any serializeable data object, and control how that cache entry expires based on a combination of one or more dependencies. These dependencies can include time elapsed since the item was cached, time elapsed since the item was last accessed, changes to files and/or folders, changes to other cached items, or (with a little work) changes to particular tables in a database.

Storing Data in the Cache

The simplest way to store data in the Cache is simply to assign it, using a key, just like a HashTable or Dictionary object:

Cache["key"] = "value";

This will store the item in the cache without any dependencies, so it will not expire unless the cache engine removes it in order to make room for additional cached data. To include specific cache dependencies, the Add() or Insert() method is used. Each of these has several overloads. The only difference between Add() and Insert() is that Add() returns a reference to the cached object, while has no return value (void in C#, Sub in VB).

Examples

```
Cache.Insert("key", myXMLFileData, new
  System.Web.Caching.CacheDependency(Server.MapPath("users.xml")));
```

This would insert xml data from a file into the cache, eliminating the need to read from the file on subsequent requests. The CacheDependency will ensure that when the file changes, the cache will immediately expire, allowing the latest data to be pulled from the file and re-cached. An array of filenames can also be specified if the cached data depends on several files.

```
Cache.Insert("dependentkey", myDependentData, new
  System.Web.Caching.CacheDependency(new string[] {}, new string[]
  {"key"}));
```

This example would insert a second piece of data that depended on the existence of the first piece (with a key value of "key"). If no key existed in the cache called "key", or if the item associated with that key expires or is updated, then the cache entry for "dependentkey" would expire.

```
Cache.Insert("key", myTimeSensitiveData, null,
  DateTime.Now.AddMinutes(1), TimeSpan.Zero);
```

Absolute Expiration: This example will cache the time sensitive data for one minute, at which point the cache will expire. Note that absolute expiration and sliding expiration (below) cannot be used together.

```
Cache.Insert("key", myFrequentlyAccessedData, null,
  System.Web.Caching.Cache.NoAbsoluteExpiration,
  TimeSpan.FromMinutes(1));
```

Sliding Expiration: This example will cache some frequently used data. The data will remain in the cache until one minute passes without anything referencing it. Note that sliding expiration and absolute expiration cannot be used together.

More Options

In addition to the dependencies described above, we can also specify the priority of the item (from low to high to NotRemovable, defined in the System.Web.Caching.CacheItemPriority enumeration) and a **CacheItemRemovedCallback** function to call when the item expires from the cache. Most of the time, the default priority will suffice—let the caching engine do what it's good at and handle the cache's memory management. The CacheItemRemovedCallback option allows for some interesting possibilities, but in practice it too is rarely used. However, to demonstrate the technique, I'll provide an example of its usage:

CacheItemRemovedCallback example

```
System.Web.Caching.CacheItemRemovedCallback callback = new System.Web.Caching.CacheItemRemovedCallback (OnRemove);
Cache.Insert("key",myFile,null,
   System.Web.Caching.Cache.NoAbsoluteExpiration,
   TimeSpan.Zero,
   System.Web.Caching.CacheItemPriority.Default, callback);
 . . .
public static void OnRemove(string key,
   object cacheItem,
   System.Web.Caching.CacheItemRemovedReason reason)
   {
      AppendLog("The cached value with key '" + key +
            "' was removed from the cache.  Reason: " +
            reason.ToString());
}
```

This will log the reason for the expiration of data from the cache using whatever logic is defined in the `AppendLog()` method (not included here—see Writing Entries to Event Logs. Logging items as they are removed from the cache and noting the reason for removal might allow you to determine if you are using the cache effectively, or if you might need to increase the memory on your server. Note that the callback is a static (Shared in VB) method, which is recommended since otherwise an instance of the class holding the callback function will be kept in memory to support the callback (which isn't necessary for a static/Shared method).

One potential use for this feature would be to refresh cached data in the background so that users never need to wait for the data to be populated, but the data is kept relatively fresh. Unfortunately in practice, this doesn't work very well with the current version of the caching API, because the callback doesn't fire or complete execution prior to the cached item being removed from the cache. Thus, a user will frequently make a request that will try to access the cached value, find that it is null, and be forced to wait for it to repopulate. In a future version of ASP.NET, I would like to see an additional callback, which might be called CachedItemExpiredButNotRemovedCallback, which if defined must complete execution before the cached item is removed.

Cached Data Reference Pattern

Whenever an attempt is made to access data from the cache, it should be with the assumption that the data might not be there any more. Thus, the following pattern should be universally applied to your access of cached data. In this case, we're going to assume the object that has been cached is a DataTable.

```
public DataTable GetCustomers(bool BypassCache)
{
   string cacheKey = "CustomersDataTable";
   object cacheItem = Cache[cacheKey] as DataTable;
   if((BypassCache) || (cacheItem == null))
   {
      cacheItem = GetCustomersFromDataSource();
      Cache.Insert(cacheKey, cacheItem, null,
      DateTime.Now.AddSeconds(GetCacheSecondsFromConfig(cacheKey),
      TimeSpan.Zero);
   }
   return (DataTable)cacheItem;
}
```

There are several points I'd like to make about this pattern:

- Values, like cacheKey, cacheItem and the cache duration, are defined once and only once.
- The cache can be bypassed as needed—for example, after registering a new customer and redirecting to a list of customers, it would probably be best to bypass the cache and repopulate it with the latest data, which would include the newly inserted customer.
- Cache is only accessed once. This has performance benefits and ensures that NullReferenceExceptions don't occur because the item was present the first time it was checked but had expired before the second check.
- The pattern uses strong type checking. The "as" operator in C# will attempt to cast an object to a type and will simply return null if it fails or if the object is null.
- The duration is stored in a configuration file. All cache dependencies, whether file-based, time-based, or otherwise, should ideally be stored in a configuration file so that changes can be made and performance measured easily. I also recommend that a default cache duration be specified, and that the GetCacheSecondsFromConfig() method uses the default if no duration is specified for the cacheKey being used.

The associated code sample is a helper class that will handle all of the above but allow cached data to be accessed with one or two lines of code. Download CacheDemos.msi.

Summary

Caching can provide huge performance benefits to applications, and should therefore be considered when an application is being designed as well as when it is being performance tested. I have yet to encounter an application that could not benefit from caching in some capacity, though certainly some applications are better suited than others. A solid understanding of the caching options available in ASP.NET is an important skill for any ASP.NET developer to master.

Steven A. Smith, Microsoft ASP.NET MVP, is president and owner of ASPAlliance.com. He is also the owner and head instructor for ASPSmith Ltd, a .NET-focused training company. He has authored two books, the ASP.NET Developer's Cookbook and ASP.NET By Example, as well as articles in MSDN® and AspNetPRO magazines. Steve speaks at several conferences each year and is a member of the INETA speaker's bureau. Steve has a Master's degree in Business Administration and a Bachelor of Science degree in Computer Science Engineering. Steve can be reached at ssmith@aspalliance.com.

Related Links

Ultimate Caching: Output and Fragment Options

ASP.NET Application Cache Viewer

Effective Cache Expirations

ASP.NET Application Cache and SQL Server; Invalidating a Cached Item

@OutputCache

Output Caching in ASP.NET