Improving Web Performance on the Server

There are many ways to improve the performance of code in ASP.NET applications. Some improvements can be made using techniques that are valid for both ASP.NET applications and desktop applications, such as using threads or Tasks for non-dependent asynchronous operations, but there are also some improvements that relate to the way you write your ASP.NET code, whether it is a WebForm's code-behind, or an ASP.NET MVC Controller. These changes, however small, can help utilize your server better, allowing the application to run faster, and handle more concurrent requests.

Cache Commonly Used Objects

The processing of a request in a web application often requires the use of fetched data, usually from a remote location such as a database, or a web service. These data lookups are expensive operations, often causing latency in response time. Instead of fetching the data for each operation, that data can be pre-fetched once, and stored in-memory, in some sort of a caching mechanism. New requests that will come in after the data was fetched can use the cached data instead of fetching it again from its original source. The caching paradigm is often described as:

1. If the data is already cached, use it.
2. Else:
   - a. Fetch the data.
   - b. Store it in the cache.
   - c. Use it.

Caution Since several requests can access the same cached object at any given time, causing the same object be referenced from multiple threads, it is expected that an update to a cached object will be done responsibly, whether by treating it as immutable (changes to a cached object will require cloning it, making the changes on the new copy, and then updating the cache with the cloned object), or by using locking mechanisms to make sure it is not being updated by other threads.

Many developers use ASP.NET's Application state collection as a sort of caching mechanism, because it provides an in-memory caching store, accessible for all users and sessions. Using the Application collection is quite simple:

```
Application["listOfCountries"] = countries; // Store a value in the collection
countries = (IEnumerable < string>)Application["listOfCountries"]; // Get the value back
```

When using the Application collection, the resources which are stored in memory and accumulated over time can eventually fill the server memory, causing the ASP.NET application to start using paged memory from disk, or even fail due to lack of memory. Therefore, ASP.NET provides a special caching mechanism, which provides some sort of management over cached items, freeing unused items when memory is lacking.

ASP.NET caching, accessible through the `Cache` class, provides an extensive caching mechanism, which in addition to storing resources also allows you to:

- Define an expiration for a cached object, either by specified a `TimeSpan` or a fixed `DateTime`. Once the allotted life time of the cached object expires, the object will automatically be removed from the cache.
- Define priority for cached objects. When there is a memory lack and objects need to be freed, the priority can assist the cache mechanism to decide which objects are less "important."
- Define validity for a cached object by adding dependencies, such as an SQL dependency. For example, if the cached object is a result of an SQL query, an SQL dependency can be set, so changes in the database that affects the results returned from the query will invalidate the cached object.
- Attach callbacks to cached objects, so when an object is being removed from the cache, the callback is invoked. Using callbacks can assist in retrieving updated resource information when expiration or invalidation occurs in the cache.

Adding items to the cache is as simple as adding items to a dictionary:

```
Cache["listOfCountries"] = listOfCountries;
```

When adding an item to the cache using the above code, the cached item will have the default priority of `Normal` and will not use any expiration or dependency checks. For example, to add an item to the cache with a sliding expiration, use the `Insert` method:

```
Cache.Insert("products", productsList,
  Cache.NoAbsoluteExpiration, TimeSpan.FromMinutes(60), dependencies: null);
```

Note The `Cache` class also provides the `Add` method. Unlike the `Insert` method, the `Add` method will throw an exception if the cache already contains an item with the same `key` parameter.

The cache access paradigm, using ASP.NET's `Cache` class is usually implemented as follows:

```
object retrievedObject = null;

retrievedObject = Cache["theKey"];
if (retrievedObject == null) {
  //Lookup the data somewhere (database, web service, etc.)
```

```
    object originalData = null;
    . . .
    //Store the newly retrieved data in the cache
    Cache["theKey"] = originalData;
    retrievedObject = originalData;
  }
  //Use the retrieved object (either from the cache or the one that was just cached)
  . . .
```

You'll notice that the first line of code attempts to retrieve the object from the cache, without first checking whether it exists in the cache. This is because objects can be removed from the cache at any time by other requests or by the cache mechanism itself, so an item can be removed from the cache between checking and retrieval.

Using Asynchronous Pages, Modules, and Controllers

When IIS passes a request to ASP.NET, the request is queued to the thread pool, and a worker thread is assigned to handle the request whether it is a request for a simple HTTP handler, a page within an ASP.NET WebForm application, or a controller within an ASP.NET MVC application.

Since the number of worker threads in the thread pool is limited (defined by the value set for `processModel` ➤ `maxWorkerThreads` section in the web.config), then that implies ASP.NET is also limited by the number of threads, or requests, it can execute simultaneously.

The thread limit is usually high enough to support small-to-medium web applications that need to handle only several dozens of concurrent requests. However, if your web application is required to handle hundreds of requests concurrently, you should keep reading this section.

The limit on the number of concurrently executing requests encourages developers to try to minimize the execution time of requests, but what happens when the execution of a request depends on some other I/O operation, such as calling a web service, or waiting for a database operation to complete? In this case the execution time of the requests is highly dependent on the time it takes to get the information back from the remote process, and during that time the worker thread attached to the request is occupied and cannot be freed for another request.

Eventually, when the number of currently executing requests exceeds the limit of the thread pool, new requests will be placed in a special waiting queue. When the number of queued requests exceeds the limit of the queue, incoming requests will fail, returning an HTTP 503 response ("service unavailable").

Note The limitation of the thread pool and requests queue is defined for ASP.NET applications in the `processModel` section of the web.config file, and is controlled partially by the `processModel` ➤ `autoConfig` attribute.

In modern web applications, where I/O operations are an inevitable part of our system's design (calling web services, querying databases, reading from network file storages, etc.), this behavior often leads to having many running threads waiting on I/O and only several threads actually performing CPU consuming tasks. This often leads to a low utilization of the server's CPU, which cannot be used by other requests since there are no more free threads for the incoming requests.

In web applications where many requests start by fetching data from web services or databases, it is common to see low CPU utilization even with high user load. You can use performance counters to check the CPU utilization of your web application, by checking the `Processor\% CPU Utilization` counter in conjunction with the `ASP.NET Applications\Requests/Sec` and `ASP.NET\Requests Queued` counters.

If some of your requests are executing lengthy I/O operations, then there is no need to hold the worker thread until completion. With ASP.NET you can write asynchronous pages, controllers, handlers, and modules, which enable you to return worker threads back to the thread pool while your code is waiting for an I/O operation to complete, and once completed, to grab a worker thread from the pool to complete the execution of the request. From the end-user's point of view, the page will still seem to take some time to load, since the server is holding the request until the processing is complete and the response is ready to be sent back.

By changing I/O-bound requests to use asynchronous processing instead of synchronous processing, you can increase the number of worker threads available for CPU-intensive requests, enabling your server to better utilize its CPU(s) and prevent requests from being queued.

Creating an Asynchronous Page

If you have an ASP.NET Web Forms application, and you wish to create an async page, first you will need to mark the page as async:

```
<%@ Page Async = "true" . . .
```

Once marked as async, create a new `PageAsyncTask` object and pass it the delegates for the begin, end, and timeout methods. After creating the `PageAsyncTask` object, call the `Page.RegisterAsyncTask` method to start the asynchronous operation.

The following code shows how to start a lengthy SQL query using the `PageAsyncTask`:

```
public partial class MyAsyncPage : System.Web.UI.Page {
  private SqlConnection _sqlConnection;
  private SqlCommand _sqlCommand;
  private SqlDataReader _sqlReader;

  IAsyncResult BeginAsyncOp(object sender, EventArgs e, AsyncCallback cb, object state) {
  //This part of the code will execute in the original worker thread,
  //so do not perform any lengthy operations in this method
  _sqlCommand = CreateSqlCommand(_sqlConnection);
  return _sqlCommand.BeginExecuteReader(cb, state);
  }
  void EndAsyncOp(IAsyncResult asyncResult) {
  _sqlReader = _sqlCommand.EndExecuteReader(asyncResult);
  //Read the data and build the page's content
  . . .
  }
 void TimeoutAsyncOp(IAsyncResult asyncResult) {
  _sqlReader = _sqlCommand.EndExecuteReader(asyncResult);
  //Read the data and build the page’s content
  . . .
  }

  public override void Dispose() {
  if (_sqlConnection ! = null) {
  _sqlConnection.Close();
  }
  base.Dispose();
  }
  protected void btnClick_Click(object sender, EventArgs e) {
  PageAsyncTask task = new PageAsyncTask(
  new BeginEventHandler(BeginAsyncOp),
  new EndEventHandler(EndAsyncOp),
  new EndEventHandler(TimeoutAsyncOp),
  state:null);
  RegisterAsyncTask(task);
  }
}
```

Another way of creating async pages is by using completion events, such as the ones created when using web services or WCF services-generated proxies:

```
public partial class MyAsyncPage2 : System.Web.UI.Page {
  protected void btnGetData_Click(object sender, EventArgs e) {
  Services.MyService serviceProxy = new Services.MyService();
  //Attach to the service’s xxCompleted event
  serviceProxy.GetDataCompleted + = new
  Services.GetDataCompletedEventHandler(GetData_Completed);
  //Use the Async service call which executes on an I/O thread
  serviceProxy.GetDataAsync();
  }
  void GetData_Completed (object sender, Services. GetDataCompletedEventArgs e) {
  //Extract the result from the event args and build the page’s content
  }
}
```

In the above example the page is also marked as `Async`, as the first example, but there is no need to create the `PageAsyncTask` object, since the page automatically receives notification when the `xxAsync` method is called, and after the `xxCompleted` event is fired.

Note When setting the page to async, ASP.NET changes the page to implement the `IHttpAsyncHandler` instead of the synchronous `IHttpHandler`. If you wish to create your own asynchronous generic HTTP handler, create a generic HTTP handler class which implements the `IHttpAsyncHandler` interface.

Creating an Asynchronous Controller

Controller classes in ASP.NET MVC can also be created as asynchronous controllers, if they perform lengthy I/O operations. To create an asynchronous controller you will need to perform these steps:

1. Create a controller class that inherits from the `AsyncController` type.
2. Implement a set of action methods for each async operation according to the following convention, where *xx* is the name of the action: `xxAsync` and `xxCompleted`.
3. In the `xxAsync` method, call the `AsyncManager.OutstandingOperations.Increment` method with the number of asynchronous operations you are about to perform.
4. In the code which executes during the return of the async operation, call the `AsyncManager.OutstandingOperations.Decrement` method to notify the operation has completed.

For example, the following code shows a controller with an asynchronous action named *Index*, which calls a service that returns the data for the view:

```
public class MyController : AsyncController {
  public void IndexAsync() {
  //Notify the AsyncManager there is going to be only one Async operation
  AsyncManager.OutstandingOperations.Increment();
  MyService serviceProxy = new MyService();
  //Register to the completed event
  serviceProxy.GetDataCompleted + = (sender, e) = > {
  AsyncManager.Parameters["result"] = e.Value;
  AsyncManager.OutstandingOperations.Decrement();
  };
  serviceProxy.GetHeadlinesAsync();
  }
  public ActionResult IndexCompleted(MyData result) {
  return View("Index", new MyViewModel { TheData = result });
  }
}
```