**Session Helper**

If you are developing ASP.NET web applications, you surely had to store and retrieve some persistent values belonging to each user navigating through your ASP.NET Web Application’s pages, so you naturally have used Session.

Session is nothing but a Dictionary (a collection based on key/value pairs) whose key type is a string and the value type is an object (i.e., any serializable object inheriting from object).

You can access the Session from "everywhere" using the current HTTP context System.Web.HttpContext.Current.Session or from the page’s shortcut reference in System.Web.UI.Page.Session.

Concretely, when you want to store a value in the Session, you may write such a line of code (without taking care of any session manager):

Then, here is the code you may write to retrieve the previously stored value, mostly but not only from another page (or control):

As you see, Session is somewhat raw, and you’ll be quickly annoyed when you want to use advanced features such as categorized values, values belonging to a page or a control, etc.

In these cases, you should (you must) use the Session, but you should have some collision problems, a day or another!

A solution to this problem is to use "scoped and categorized" session values. This is not a feature of the ASP.NET framework, but a session helper that I built and used with success, so I’m pleased to share this tip with you.

## Using the SessionHelper in your project

This helper class is easy to incorporate and use in your project. You can:

1. move the class code in to your solution



1. use the Initia.Web project included in the sample project
2. compile and reference the Initia.Web project’s assembly.

Once you’ve done this, you may use the Initia.Web namespace like this:

Hide   Copy Code

using Initia.Web;

Then, you can start to use the helper class. Let’s say you have a couple of pages, and you want to count the total hits, each page’s hits, and the hits belonging to the click on a button.

First, create a page and go to the Page\_Load event handler. We don’t have to bother about the IsPostBackstate since we want to increment hits at each page load.

Hide   Copy Code

protected void Page\_Load(object sender, EventArgs e)

{

...

}

Here is the code to retrieve the (implicit) global scope "Hits" session value, with the default value set to 0. Then, we store the incremented value:

Hide   Copy Code

...

*// Incremented global session value*

*// (note that it doesn't scope with other "Hits" keys)*

int totalHits = (int)SessionHelper.RetrieveWithDefault("Hits", 0);

SessionHelper.Store("Hits", totalHits + 1);

...

A very close code, using the SessionHelper.Scope.Page scope, can be written to be used in the scope of the current page (without bothering about the query string). Note that we’re using the same key, but there is no collision since we’re using another scope:

Hide   Copy Code

...

*// Incremented current page session value*

*// (note that it doesn't scope with other "Hits" keys)*

int currentPageHits = (int)SessionHelper.RetrieveWithDefault(

SessionHelper.Scope.Page, "Hits", 0);

SessionHelper.Store(SessionHelper.Scope.Page, "Hits", ++currentPageHits);

...

If you want to restrict the scope of some session values to the scope of the current page, including the query string, you may write this piece of code using the SessionHelper.Scope.PageAndQuery scope:

Hide   Copy Code

...

*// Incremented current page session value*

*// (note that it doesn't scope with other "Hits" keys)*

int currentPageQueryHits = (int)SessionHelper.RetrieveWithDefault(

SessionHelper.Scope.PageAndQuery, "Hits", 0);

SessionHelper.Store(SessionHelper.Scope.PageAndQuery,

"Hits", ++currentPageQueryHits);

...

The last example is about categorized session values. This time, we’re going to increment a hits counter each time the user clicks on a button, and we’ll clear the category when the user clicks another button. Just add buttons in your page like this:

Hide   Copy Code

<asp:Button ID="btnAddUserHit" Text="Add hit to user category" runat="server" />

<asp:Button ID="btnClearUserCategory" Text="Clear user category" runat="server" />

Then, add an event handler in the code-behind (you can do it in the ASPX page, if you want):

Hide   Copy Code

protected override void OnInit(EventArgs e)

{

base.OnInit(e);

*// Events handlers*

btnAddUserHit.Click += new EventHandler(btnAddUserHit\_Click);

btnClearUserCategory.Click += new EventHandler(btnClearUserCategory \_Click);

}

And finally, implement the event handlers like the following (UpdateUI is explained below):

Hide   Copy Code

#region Events handlers

private void btnAddUserHit\_Click(object sender, EventArgs e)

{

*// Retrieves, increments then stores*

*// categorized session value (default value is 0)*

int userHits = (int)SessionHelper.RetrieveWithDefault(

"User", "Hits", 0);

SessionHelper.Store("User", "Hits", userHits + 1);

*// Update user interface*

UpdateUI();

}

private void btnClearGlobalCategory\_Click(object sender, EventArgs e)

{

SessionHelper.ClearCategory(SessionHelper.Scope.Global, "User");

*// Update user interface*

UpdateUI();

}

#endregion

As "bonus", here is a simple method updating the user interface according to the session values (just add a textbox with a tbUserHits ID in your ASPX page):

Hide   Copy Code

#region User interface

/// *<span class="code-SummaryComment"><summary></span>*

/// *Update user interface*

/// *<span class="code-SummaryComment"></summary></span>*

public void UpdateUI()

{

*// Show categorized session value (default value is 0)*

tbUserHits.Text = SessionHelper.RetrieveWithDefault("User",

"Hits", 0).ToString();

}

#endregion

**Please check below attachment for example project on Session helper**

****

**Cache Helper**

 You’ll notice the class uses generics to allow for some, umm, generic functionality.

public static class CacheHelper

{

/// <summary>

/// Insert value into the cache using

/// appropriate name/value pairs

/// </summary>

/// <typeparam name="T">Type of cached item</typeparam>

/// <param name="o">Item to be cached</param>

/// <param name="key">Name of item</param>

public static void Add<T>(T o, string key) where T : class

{

// NOTE: Apply expiration parameters as you see fit.

// In this example, I want an absolute

// timeout so changes will always be reflected

// at that time. Hence, the NoSlidingExpiration.

HttpContext.Current.Cache.Insert(

key,

o,

null,

DateTime.Now.AddMinutes(

ConfigurationHelper.CacheExpirationMinutes),

System.Web.Caching.Cache.NoSlidingExpiration);

}

/// <summary>

/// Remove item from cache

/// </summary>

/// <param name="key">Name of cached item</param>

public static void Clear(string key)

{

HttpContext.Current.Cache.Remove(key);

}

/// <summary>

/// Check for item in cache

/// </summary>

/// <param name="key">Name of cached item</param>

/// <returns></returns>

public static bool Exists(string key)

{

return HttpContext.Current.Cache[key] != null;

}

/// <summary>

/// Retrieve cached item

/// </summary>

/// <typeparam name="T">Type of cached item</typeparam>

/// <param name="key">Name of cached item</param>

/// <returns>Cached item as type</returns>

public static T Get<T>(string key) where T : class

{

try

{

return (T) HttpContext.Current.Cache[key];

}

catch

{

return null;

}

}

}

And here is a relatively standard sample usage of the library.

public override List<Employee> GetEmployeeList()

{

string key = ConfigurationHelper.CacheKeyEmployeeList;

List<Employee> employees = CacheHelper.Get<List<Employee>>(key);

if (employees == null)

{

employees = instance.GetEmployeeList();

CacheHelper.Add(employees, key);

}

return employees;

}

Notice how I’m grabbing the cached value, storing it in a local variable and then checking if it is equal to null rather than using the CacheHelper.Exists() method.  If I used the CacheHelper.Exists() method, the cached object could expire between the time I check its existence and the time I get its value through the CacheHelper.Get() method.  Therefore, the above approach is the safest strategy to use when retrieving cached values.  CacheHelper.Exists() should really only be used for quick existence checks which are unrelated to the fetch.

But if you want to use the code CORRECTLY there’s a catch.  Did you notice the “class” constraint on the CacheHelper.Get() and CacheHelper.Add() methods?   I did this because you can’t always return null from a generic method.  If the return type were always a reference type it would be fine, but comparing a non-nullable value type to null would throw a runtime exception or would always evaluate to false.  Therefore, I’ve constrainted CacheHelper which limits its functionality but unsure safe use of the cache.  If you’re feeling dangerous, you’re welcome to remove the constraints.

Please check below attachment for reference



**Error Handling**

When errors occur in an ASP.NET application, they either get handled or propagates unhandled to higher scopes. When an unhandled exception propagates, the user may be redirected to an error page using different ASP.NET configuration settings. However, such a redirection may be prevented in the first place by handling the exceptions that get thrown. Error handling in ASP.NET therefore, may be divided into two separate logics:

* Redirecting the user to an error page when errors go unhandled.
* Handling exceptions when they get thrown.

## Redirecting the user to an error page

There are two different scopes where we could specify which page the user should be redirected to, when errors go unhandled:

* Page level (applies to errors that happen within a single page).
* Application level (applies to errors that happen anywhere in the application).

### **Page Level**

Use the errorPage attribute in the webform.

This attribute defines the page the user should be redirected to when an unhandled exception occurs in that specific page. For example,

Hide   Copy Code

<%@ Page language="c#" Codebehind="WebForm1.aspx.cs"

AutoEventWireup="false" Inherits="WebTest.WebForm1"

**errorPage="/WebTest/ErrorPages/PageError.html"**%>

The errorPage attribute maps to the Page.ErrorPage property, and hence may be set programmatically. The value may optionally include query string parameters. If no parameters are added, ASP.NET would automatically add one with the name aspxerrorpath. This parameter would hold the value of the relative URL to this page, so that the error page would be able to determine which page caused the error.

If a value is specified in this attribute (or property) and an unhandled exception occurs in the page, the Pageclass would automatically perform a redirect to the specified page. If a value is not specified, the exception is assumed to be unhandled, wrapped in a new HttpUnhandledException and then thrown, propagating it to the next higher level.

### **Application Level**

Use the customErrors section in *web.config*.

This section lets you specify the error page to which the user should be redirected to when an unhandled exception propagates in the application level. This section specifies error pages for both default errors as well as the HTTP status code errors.

Hide   Copy Code

<customErrors mode="On" defaultRedirect="/WebTest/ErrorPages/AppError.html">

<error statusCode="404" redirect="/WebTest/ErrorPages/404.html" />

</customErrors>

The mode attribute specifies whether to show user-defined custom error pages or ASP.NET error pages. Three values are supported for this attribute:

* RemoteOnly - Custom error pages are shown for all remote users. ASP.NET error pages with rich error information are displayed only for local users.
* On - Custom error pages are always shown, unless one is not specified. When a custom error page is not defined, an ASP.NET error page will be displayed which describes how to enable remote viewing of errors.
* Off - Custom error pages are not shown. Instead, ASP.NET error pages will be displayed always, which will have rich error information.

It's a bad idea to give users more information than what is required. ASP.NET error pages describe technical details that shouldn't be exposed. Ideally, the mode attribute thus should not be set to Off.

The defaultRedirect attribute specifies the path to a generic error page. This page would typically have a link to let the user go back to the home page or perform the request once again.

Each error element defines a redirect specific to a particular HTTP status code. For example, if the error is a 404 (File Not Found), then you could set the error page as *FileNotFound.htm*. You could add as many error elements in the customErrors section as required, each of which specifies a status code and the corresponding error page path. If ASP.NET can’t find any specific error element corresponding to a status code, it would use the value specified in the defaultRedirect attribute.

#### Notes

* The settings specified in the page level (errorPage attribute) would override those specified in the customErrors section. The reason is because errors in the page would be handled by the Page class first, which might thus prevent the exception from being propagated to the application level. It’s only when the Page class fails to handle the exception that the values set in customErrors come into scope.
* All these settings mentioned above apply only for requests that are made for ASP.NET files. More specifically, these settings would work only for requests for files with extensions that are mapped to the aspnet\_isapi. For example, if you request for an ASP or JPG file (extensions that are not mapped to aspnet\_isapi) which does not exist, then these settings won’t work, and the standard error page specified in IIS would be displayed. To modify this behavior, either map the required extensions to aspnet\_isapi or modify the custom error pages specified in IIS.

## Handling exceptions

There are different levels where you could handle exceptions.

* Locally (method level), where exceptions could be thrown.
* Page level by handling the Page.Error event.
* Application level by handling the HttpApplication.Error event.
* HTTP Module level by handling the HttpApplication.Error event.

### **Local error handling**

Wrap code that might throw exceptions in a try-catch-finally block.

If you can recover from the exception, then handle it in the catch block. If the exception cannot be recovered from locally, let the exception propagate to higher levels by throwing it. If the exception cannot be recovered from locally, but additional information can be provided, then wrap the exception with the new information and throw the new exception. This method is used when you use custom exceptions. Place the clean up code in the finally block.

Find more information on exception handling best practices available in MSDN.

**Note**: The more exceptions you catch and throw, the slower your application would run. This is more significant in web applications.

### **Page Level**

Attach a handler to the Page.Error event. In C#, you will have to write the event wire up code yourself in the Page\_Load method.

When an exception goes unhandled in a page, the Error event of the Page class gets triggered.

Typically, the first action you would perform in this handler would be to obtain the exception thrown, by using the Server.GetLastError method. This method would return a reference to the last Exception object that was thrown.

After you get the Exception object, you will want to redirect the user to an error page. We could make ASP.NET do the redirection by using the errorPage attribute of the Page (design time) or by using the Page.ErrorPageproperty (runtime). Obviously, the choice here would be to programmatically set the value using the Page.ErrorPage property in the event handler.

Hide   Copy Code

private void WebForm1\_Error(object sender, EventArgs e)

{

*// Get the last exception thrown*

Exception ex = Server.GetLastError();

*// Do something with the exception like logging etc.*

*// Set the error page*

this.ErrorPage = "/ErrorHandling/ErrorPages/BaseError.html";

}

If you do not specify an error page, the exception gets wrapped inside an HttpUnhandledException object and propagates. If you don’t want the exception to be wrapped, then simply throw the last exception, which would force immediate propagation escaping any intervention. However, this would prevent ASP.NET from redirecting the user to a page specific page either. In other words, if you are going to throw the last error (or any exception for that matter), setting the error page will have no effect.

Hide   Copy Code

private void BasePage\_Error(object sender, EventArgs e)

{

*// Get the last exception thrown*

Exception ex = Server.GetLastError();

*// Do something with the exception like logging etc.*

*// The statement below has no significance - it's as good as commented*

this.ErrorPage = "/ErrorHandling/ErrorPages/BaseError.html";

*// Throw the error to prevent wrapping*

throw ex;

}

To reduce redundant code, you could define a base web form page which defines the Page.Error event handler and then wire up code in the constructor, and then make all your Web Form pages derive from this base page. This would save you the effort of writing the error handler in each web form.

### **Application Level**

Attach an event handler to the Application.Error event.

When an unhandled exception leaves a page, it gets propagated to the application level, which would trigger this event.

There are two things you would want to do in an application error handler.

* Get the last exception thrown using Server.GetLastError.
* Clear the error using Server.ClearError, to inform ASP.NET that you have handled the error.

If you don’t clear the error, the exception would propagate. However, since there isn't any higher scope where the exception could be caught, ASP.NET is forced to handle it. The way ASP.NET handles the exception depends upon the settings specified in the customErrors section we saw before. If no settings are defined, ASP.NET would use the defaults and display the infamous 'yellow' error page.

### **HTTP Module Level**

Instead of handling application errors in *global.asax*, exceptions may also be handled by attaching an HTTP Module which would have a handler attached to the Application.Error event. This method would be triggered before the corresponding application handler would be invoked. Such an implementation would be beneficial if you have multiple projects with the same global error handling implementation. In such a scenario, you could create a module and attach it to each web application you have.

All the points we saw in the Page and Application handlers apply to the Module handler as well.

## Important Notes

### **Prevent infinite recursion**

If an error occurs in the error handling code, an infinite recursive loop would result, which would soon drag your server down. The reason why this happens is because the new exception would trigger the error event once again which would in turn redirect control to the handler, which would cause yet another exception to be thrown, making an infinite loop.

This might also happen if the error page itself throws an exception. To counter this possibility, making error pages static is a good idea.

Errors may also happen while attempting to redirect to an error page using Server.Transfer or Response.Redirect maybe due to an invalid path. To tackle this scenario, we could wrap the redirection code in a try-catch block. If the redirection fails, then we have nothing more to do other than setting the response code and completing the response, using the Response.StatusCode property and the HttpApplication.CompleteResponse method. This would then be handled by the settings specified in the customErrors section.

### **Parser Errors**

Parser errors are caused due to invalid tags (or similar reasons) in an aspx page. These errors are usually of type HttpParseException. Such errors will not be caught by the Page level handler as page parsing happens before ASP.NET creates the assembly for the aspx page. In other words, parser errors are thrown while ASP.NET reads the aspx file and tries to create its assembly, and hence is way before the corresponding type is created. Thus, such errors will have to be handled in the application scope.

### **Exception logging and response time**

Users need to get responses as quick as possible. Implementation wise, this means that when errors happen, error recovery processes should be quick and users should be redirected or informed of the error as soon as possible. If exceptions are going to be logged to a file or other mediums, then it could take time which would lead to a slow response. Making exception logging an asynchronous process would be a good idea in this respect.

**Please check below attachment for ur reference**

