Passing Values between Pages

There are so many ways to pass values between webpages that finding the optimal solution in any given scenario can be challenging. Conflicting goals — such as ease of development, usability, security, efficiency, data size, and reliability — can all influence your decision.

1. Application State

HttpApplicationState is a classic ASP object that serves as a great place to store global values or objects. For example, if you've got a fairly static DataSet that is used frequently, you might choose to store it in application state. Retrieving the object is then as simple as one line of code:

```
//C#
DataSet Source = (DataSet)(Application["MyDataSet"]);
```

Storing items in application state is nearly as easy as retrieving them. The main difference is that you need to lock the Application object before storing a value, and unlock it when you're finished. This helps ensure no nasty threading issues muck things up, such as two processes trying to update the value at the same time. Instead, the processes will be queued, if necessary, to avoid deadlocks:

```
//C#
Application.Lock();
Application["MyDataSet"] = MyDataSet;
Application.UnLock();
```

Modifications to objects stored in application state are not persisted automatically back into application state. In other words, if you modify the DataSet after retrieving it from application state, you'll probably want to explicitly overwrite the old DataSet in application state with the new one using the code above.

As you might have guessed by the name, the Application object is in scope only for the current Web application. In other words, if you have two ASP.NET Web sites on your server, Application2 will not be able to read values from the Application object of Application1.

2. Cache Object

There is no debating the utility of the Application object. However, its age is starting to show a bit — and there's a new object in town: HttpCache.

Like the HttpApplicationState object, the HttpCache object is a container useful for storing global variables and objects. At its simplest, the syntax is very similar to using the Application object, although no locking or unlocking is necessary because thread management is built in to the object:

```
//C#
Cache["MyGlobalValue"] = TextBox1.Text; //Store
```

```
string s = Cache["MyGlobalValue"].ToString(); //Retrieve
```

The Cache object has features that the Application object doesn’t. These features are all geared toward increasing scalability.

The Cache object implements more intelligent storage techniques than the Application object. For example, it will automatically remove seldom-used items from the Cache if memory starts to get low. Luckily, it is possible to optionally specify a priority for each item in the cache — so important items will be more likely to stick around. It is also possible to be informed when an item is removed from the cache using the CacheItemRemovedCallback delegate.

The Cache object allows you to modify how objects are stored in the cache, and for how long. For example, you can specify that a cache item expire after a certain amount of time (Sliding Expiration) or at a specific, fixed time (Absolute Expiration).

The following code stores a value in the Cache object just like the previous code; however, the item will expire (and be removed from the cache) after 20 minutes:

```
//C#
Cache.Insert("MyGlobalValue", TextBox1.Text,
  null, System.Web.Caching.Cache.NoAbsoluteExpiration,
  new TimeSpan(0, 20, 0));
```

The Cache object can also expire items in response to other kinds of events. For example, cache items can be dependent upon a specific file. When the file changes, the related cache item is removed. Cache items can also be dependent upon other cache items. Using this technique, when a parent cache item is removed, any related children are also automatically removed. The CacheDependency object is the key to all of this. There is also a SqlCacheDependency object that can remove an item from the cache whenever specific data in a SQL Server database changes.

Like the Application object, the Cache object’s scope is global to the current Web application.

3. Session State

While the Application and Cache objects are great for storing items globally, the Session object specializes in storing user-specific items. The syntax is simple and familiar:

```
//C#
Session["UserName"] = TextBox1.Text; //Store
string s = Session["UserName"].ToString();//Retrieve
```

The Session object applies to the current Web application only, and also applies only to the current user session. In other words, if two users visit a site that implements the above code, their user names will be stored separately and they’ll never see each other’s user names.

The Session object is sure handy, but beware of scalability problems. Like the Application and Cache objects, session items are stored in server memory by default. However, the Session object can consume memory far faster in situations where there are many users and/or many session variables. If you store 10 session items per user and your site gets 100 simultaneous users, there will be a total of 1,000 items in session state. Luckily, modern versions of ASP.NET provide reasonable ways to deal with this issue. For example, it is possible to configure an application (via the web.config file) to store session items in a SQL Server database, or on a specific server dedicated to managing session data. If these don't suit your needs, it is also possible to have session data stored in a custom storage provider of your own design.

4. Context

One of the lesser known techniques for passing data between pages is the Context object. An instance of the context object is associated with every page instance. Because a page generally only lives on the server for milliseconds (while being executed and rendered), items stored in the Context object are short lived. In many situations this is the most efficient way to store items because they are quickly and automatically purged from memory. In addition to living for the life of a page, the Context object also stays in memory while transferring to another page. The following code stores an item in Page1 and then retrieves the item in Page2, after which the Context object (and all items it contains) are cleared from server memory:

```
//C#
Context.Items["UserName"] = TextBox1.Text; //Page1
Server.Transfer("Page2.aspx"); //Page1
string s = Context.Items["UserName"].ToString(); //Page2
```

Note that Server.Transfer must be used here for this technique to work. Response.Redirect would fail because that makes a round trip to the client, which kills the instance of the Context object. **Figure 1** details some of the side effects you may encounter.

Response.Redirect	Server.Transfer
• Allows redirection to any URL on any web server. • Data must be passed manually via QueryString or one of the other techniques mentioned in this article. • Allows users to refresh & bookmark the page normally • Requires an extra round trip to the client, which is inefficient and can therefore hurt scalability • A classic, time proven technique	• Can only transfer to pages in the same web application • Allows use of the Context object to automatically pass values between pages • The client is never informed the URL has changed, which has several effects: o The browser address bar still (incorrectly) reflects the original page. o Can cause problems when the user tries to refresh or bookmark the page. o Can cause path mismatches when referring to images files, css files, etc. o Can be useful for intentionally masking the true path of a page.

Figure 1: *Response.Redirect and Server.Transfer both allow a new page to be sent to the user, but which one is best depends on which pros and cons are most tolerable in a given situation.*

Besides passing values between pages, the Context object is also useful for a variety of other things. For example, if you're calling a custom object from your page, that custom object can't access the Application or Session objects directly unless they use the Context object to retrieve them:

```
Context.Session("Whatever")…
```

Advanced techniques are also possible, such as directly referring to public properties of the previous page instance.

5. ViewState

The ViewState object is useful for storing objects between postbacks to the same page. It cannot be used for passing values to other pages. The syntax is virtually identical to the Session object:

```
//C#
ViewState["PageValue"] = TextBox1.Text; //Store
string s = ViewState["PageValue"].ToString(); //Retrieve
```

Instead of being stored in server memory (like the previously mentioned techniques), ViewState items are encoded and output into the generated HTML of the page. If you right click on an ASP.NET-generated Web page in Internet Explorer and choose View Source, you'll see an HTML element that looks a lot like this:

```
<input type="hidden" name="__VIEWSTATE" id="__VIEWSTATE"
value="/wEPDwUKMTkwNjc4NTIwMWRkv1e5TcWOq4qnwyDuryos=" />
```

When the page is posted back to the server, ASP.NET grabs this garbled-looking value and decodes it back into its original state. Be careful, though. Although ViewState values are encoded, they are not encrypted. It may be possible for savvy users to decode ViewState values, so you shouldn't store sensitive data in ViewState.

You should also try to avoid storing large amounts of data in ViewState as it eats valuable bandwidth on its way to the client and back. Because of such concerns, it is possible to turn off ViewState for pages where it is not needed or wanted. You should keep this in mind when developing controls, because they may not be able to use ViewState if they are hosted on a page that has ViewState turned off. One solution is to use ASP.NET 2.0 ControlState instead of ViewState. ControlState is similar to ViewState but remains on all the time, so it is useful for control development when you need to store critical information between postbacks. It is recommended that ViewState should still be used in control development for storing non-critical values between postbacks.

6. QueryString

Passing data via QueryString is a classic, time-tested technique. Whenever you see a URL with question marks and ampersands and other strange values tacked on after the page name, you know a "Get" is being performed to pass data along with the URL:

www.somesite.com/pg1.aspx?name=Bob&userid=9&clr=red

The QueryString portion of the URL begins at the question mark (which is only necessary when passing data via QueryString). Each data item consists of a name/value pair. Every data item following the first one must be separated by an ampersand (&). Some characters are not valid in a URL, so they must be encoded (usually using the Server.UrlEncode method):

```
//C#
string s = Server.UrlEncode(TextBox1.Text); //Page1
Response.Redirect("Page2.aspx?UserName=" + s); //Page1
string s2 = Request.QueryString["UserName"]; //Page2
```

Because QueryString values are visible to the user in the address bar of their browser, they are not in the slightest bit secure. Expect curious people to tinker with them — and be sure to put code in place to deal with any resulting errors.

When a user bookmarks a URL into their browser favorites, the full URL (including any QueryString values) is saved and used again the next time the user chooses it from their favorites. This could be a good thing or a bad thing, depending on what values are involved. It can be quite useful for a user to click on a favorite link and resume right where they left off with all relevant data immediately available (since the data was in the QueryString). It can also be a pain to users when a URL they bookmarked doesn't work anymore, simply because it contains stale QueryString data. Keep this in mind when developing with QueryStrings so you can give your users the best possible experience.

Because the QueryString can only contain text characters it is only useful for storing simple data types, and is therefore less flexible than previously mentioned techniques (which can store virtually any kind of object). Also keep in mind that browsers impose size limits for URLs. Although the limit varies from browser to browser, expect to run into problems if a URL (including its QueryString) reaches about 2,000 characters. Users don't like QueryStrings that are that long because they are ugly, confusing, and cumbersome to type in manually; try to limit usage of QueryStrings to situations that truly benefit from them.

7. Cookies

Cookies are a small bit of text (no more than 4096 bytes) that are sent from the server and stored on the client. All relevant cookies are automatically transferred back and forth between the client and server on each page request, so server-side code can use them as needed. The syntax for basic use is simple:

```
//C#
Response.Cookies["myval"].Value = TextBox1.Text; //Store
Response.Cookies["myval"].Value; //Retrieve
```

Cookies can be customized in many ways, such as automatic expiration, storing multiple values in a cookie, and limiting cookie scope to specific domains and folders.

Because cookies are stored on the user's hard drive, it is possible for users to tamper with them. Therefore, sensitive data should not be stored in cookies. It's also possible (and fairly common) for users to turn off cookie support in their browser as they've gotten a rather bad reputation for invading privacy. (Because no errors are thrown in such a situation, the only way to detect this condition is to try to set a cookie and see if it's still there after a postback.) Browsers also limit cookie usage in a variety of ways in an attempt to deal with privacy abuse. Because of these reasons I suggest avoiding cookies most of the time — they simply aren't reliable.

8. Post

In the days before ASP.NET, posting data was as common as the QueryString (aka “Get”) method for passing data between pages. However, when ASP.NET 1.x came along it was difficult for a page to post data to another page. Instead, the ASP.NET 1.x model ordained that pages should post back only to themselves. ASP.NET 2.0 has freed us from this limitation. The Button control now has a PostbackUrl property that can be used to specify that the form should be posted to a different page. By setting the PostbackUrl property of a button on Page1, the value of a page 1 textbox can be retrieved in page 2 with this code:

```
string s = Request.Form["TextBox1"].ToString(); //C#
```

You can also use hidden fields to pass around data the same way. Data in hidden fields isn’t visible to the user unless they view the page source. Data that is posted to the server (whether in a hidden field or not) is susceptible to tampering; therefore, sensitive data should stay on the server using one of the previously mentioned techniques.

Conclusion

There are other techniques for passing data between pages (see **Figure 2**). One approach is to store data in a database between page requests. I haven’t bothered to include sample code for this because the Internet is full of sample ADO.NET code that reads and writes to databases.

Technique	Scope	Memory Consumption	Security
Application	Web application	Medium	Very secure
Cache	Web application	Medium	Very secure
Session	User Session	Potentially High (Configurable)	Very secure
Context	Between 2 pages	Low	Very secure
ViewState	1 page (stores between postbacks)	Potentially High	Medium
QueryString	Between 2 pages	Low	Low
Cookies	Per user, per computer, potentially infinite expiration	Low	Low
Post	Between 2 pages	Medium/Low	Low
Database	Customizable	Medium/High	Very secure

Figure 2: *There are a variety of techniques for passing values between pages. Which one is best for a particular situation depends on your needs.*

You should now have a reasonably deep understanding of the various techniques for passing data between pages. As you can see, there is no single technique that is best in all cases. The method that is best for a given situation depends on many variables, such as scope, data size, security, usability, and scalability. Now that you know the details you can pick whichever technique best meets your requirements.

Reference:

http://steveorr.net/articles/passdata.aspx