Deferred ASP.NET Session State Management

Page Options

Average rating:
9 out of 9

Rate this page

Print this page

E-mail this page

Callum Shillan
Microsoft Consulting Services

Peter Norell
Atos Origin

March 2004

Summary: Describes a mechanism to defer instantiating items held in ASP.NET session state until the point of use. This allows large but infrequently accessed objects to be held in session state with minimal runtime overhead compared to the standard mechanism. (23 printed pages)

Applies To:
Microsoft® Framework versions 1.0 and 1.1
Microsoft® Visual C#®
Microsoft® ASP.NET

Download the source code for this article.

Contents

Introduction

In order to keep information that a user has entered on a Web page from being lost in the round trip from the user's browser to the Web server and back to the browser, some form of state management is needed. The Microsoft® ASP.NET page framework supports and provides mechanisms such as viewstate, cookies, and hidden fields in order to maintain state information at the client.

ASP.NET also has ways in which state information can be maintained on the server, and this is often an important element in creating scaleable websites. Server-based state management removes the need to transfer a potentially large data set to the browser, as the state information is held on the server. Instead of the large data set, a small token is returned and this is used to index into the server's session data store and uniquely identifies the user's information.

ASP.NET provides a flexible framework in which session information can be stored in the Web server's memory, in the memory of an out-of-process server, or held in a Microsoft® SQL Server database. It is a common practice to use SQL Server as the session state store on large websites, as it can preserve the user's state information if the database server must be restarted. The use of SQL Server also improves scalability of the Web application by removing the physical memory constraint implicit in the use of the out-of-process server.

ASP.NET server-based state management provides for the management of both application state and session state. Application state allows you to save and restore values for each active Web application. This is a global storage mechanism accessible from all pages in the Web application and is useful for storing information relevant to and accessible from every page on the website.

Session state allows you to save and restore values for each user session of the Web application. If different users are using the Web application, each will have an individual session instance on the server. Data stored on the server will be available for their use across multiple requests within the same session and ASP.NET looks after the automated purging of stale session-state entries after a timeout period has elapsed.

Overview of the Problem

Every time a user with an active session returns to the website, the whole of their session state information is loaded. This may include large objects not needed on the particular page being accessed. This behavior can waste precious processing cycles as well as unnecessarily consume memory and, therefore, can reduce the performance of the site.

For example, consider a Web application that serves as an online trading hub on which users can register and advertise items for sale. It is easy to imagine a series of pages being used to gather information about items for sale. If sellers were able to upload a number of pictures of the items they were selling, the "item for sale" session data could become quite large. We would want to keep track of this information in session state as the user stepped through our pages. A confirmation page would display all the gathered information and allow the seller to decide whether to proceed and enter the item in the "items for sale" database.

A potential solution to the problem of storing the large images is to keep them in a separate database table and only maintain an index key in the user's session. However, this leaves an operational problem of how often to purge the temporary database? Once an hour, once a day, once a week? And what about a user who has uploaded an image just as the purge job kicks in?

We could add timestamps to the images stored in the temporary database table and only purge items that are considered stale. But now we need to synchronize the ASP.NET session state timeouts and the purging of our stale items. This is so that we avoid the situation in which a user's ASP.NET session is still active but their images have been purged from the temporary database table. All of this introduces operational complexity into our application that is better avoided.

Overview of the Solution

What we really need is a mechanism that ties the isolated storage of large objects in temporary storage to the automated purging of session state items performed by ASP.NET. And we also want to provide an easy and intuitive mechanism for developers to access and manipulate session state information. This will reduce the number of potential bugs in the application, as well as improve the rate at which the developers can code a given page.

The implementation of deferred session state management described in this article meets these requirements by introducing a series of classes to manipulate the user's session data, ensuring that large data items are only loaded when needed, and tying in with the session expiration controlled by ASP.NET.

A **UserSession** class and an **ApplicationPage** class expose the user's session data as a series of properties with associated get/set accessors. This makes it easy for developers to manipulate the session data. The ASP.NET session state database is extended to include a table to hold deferred session state items. The **UserSession** object is then extended to make use of new table that holds the deferred session state items.

The UserSession and the ApplicationPage Objects

The **UserSession** and **ApplicationPage** objects serve as a platform on which the management of deferred session state is built.

The UserSession Object

The ASP.NET page framework uses the **HttpSessionState** class in the **System.Web.SessionState** namespace to provide access to the current user session. An instance of this class is provided as the intrinsic **Session** object, and the **Item** property is used to get or set individual session values.

The **Session** object and **Item** property allow us to write code such as:

```
this.Session[ "ItemDescription" ] = TextBoxItemDescription.Text;
```

Of course, from a secure coding perspective, the user's input is run through a subsystem (perhaps as simple as a regular expression validator that accepts a restricted character set) to ensure that no malicious code is entered as the preferred name.

In large Web application projects, it is conventional to use an object with associated properties to access information in the user's session state. The reason for this is that it removes the potential for developers to mistype the session item's key and introduce bugs into the Web application.

For example, the session state item above is named "ItemDescription" and without use of a **UserSession** object, it would be too easy for a developer to write the following code on a Web page and so introduce a bug.

```
// Oops!  The item description's key is spelled incorrectly
string itemDescription = this.Session[ "ItemDesc" ];
```

To avoid this, we provide a **UserSession** object that allows the following safer code to be used:

```
string itemDescription = this.UserSession.ItemDescription;
```

The **UserSession** object is quite simple. Its constructor accepts an object of type **HttpSessionState** so that it can access a user's session state, and it provides a series of properties with get/set accessors that merely read and write the appropriate session state item. Further, a **SessionItem** class is created with a series of public properties that forms the keys of the various items that can be stored in session state. This makes it even less error prone when accessing and identifying session items and their keys.

The code for the **SessionItem** and **UserSession** classes is shown below.

```
/// <summary>
/// Used to give a prescribed set of values for session item keys.
/// </summary>
public class SessionItemKey
{
   public const string ITEM_DESCRIPTION = "ItemDescription";

   public SessionItemKey()
   {
   }
}

/// <summary>
/// Used to access properties that constitute the user's session state.
/// </summary>
public class UserSession
{
   // The HTTP session state for the user
   private HttpSessionState userHttpSessionState;

   // Class Constructor
   public UserSession( HttpSessionState httpSessionState )
   {
      // Remember the HTTP session state for this user
      userHttpSessionState = httpSessionState;
   }

   /// <summary>
   /// This is the item's description
   /// </summary>
   public string ItemDescription
   {
      get { return (string)
        userHttpSessionState[SessionItemKey.ITEM_DESCRIPTION]; }
      set { userHttpSessionState[SessionItemKey.ITEM_DESCRIPTION] = value; }
   }
}
```

This basic mechanism can be extended to handle objects of any type, and we can provide appropriate code in the get/set property accessors. Such extensions will be used to implement the management of deferred session state items.

The ApplicationPage Object

But how do we get the **UserSession** object to be accessed from the "this" object in the page's code behind? The answer, again, lies in a standard mechanism used in large Web application projects. It is conventional to change the class that a Web page's code behind derives from. This enables the Web application to provide a series of helper properties and methods that ease the developer burden and provide for a faster and a more bug-free development cycle.

This mechanism is used to expose the **UserSession** object. The code for the **ApplicationPage** class is shown below.

```
/// <summary>
/// All pages in the application will derive from ApplicationPage
/// </summary>
public class ApplicationPage : System.Web.UI.Page
{
   // Used on ALL pages to access the user's session state
   public UserSession userSession;

   public ApplicationPage()
   {
   }
```

```
    /// <summary>
    /// Automatically invoked before the page is displayed
    /// </summary>
    /// <param name="e"></param>
    protected override void OnLoad(EventArgs e)
    {
       // Instantiate a new UserSession object
       userSession = new UserSession( this.Session );
       base.OnLoad (e);
    }
 }
```

The **ApplicationPage** class derives from the **System.Web.UI.Page** class, so we inherit all necessary methods, events, properties, and so on. We override the **OnLoad()** event so that we can instantiate a new **UserSession** object with the intrinsic **Session** object when a page is loaded for display by the ASP.NET framework.

All that needs to be changed on the code behind of the Web page is to derive the class from **ApplicationPage** instead of **System.Web.UI.Page**, as shown below.

```
 /// <summary>
 /// Summary description for WebForm1.
 /// </summary>
 public class WebForm1 : ApplicationPage
 {
    private void Page_Load(object sender, System.EventArgs e)
    {
       // Put user code to initialize the page here
    }

    // Web Form Designer code omitted for brevity Ã‚Â…
 }
```

The developer now has access to the properties exposed by the **UserSession** object and can safely access and manipulate the session items used within the Web application without introducing trivial item-naming bugs.

The combination of the **UserSession** object and the **ApplicationPage** object serves as the underlying platform from which you can build the management of deferred session state items.

ASP Session State Database Additions

You now need to extend the ASP.NET session state database so that you can store deferred items in it and tie in with the automated purging of stale session state data.

The InstallSqlState.sql and InstallPersistSqlState.sql scripts are installed by default at *systemroot*\Microsoft.NET\Framework\versionNumber where *systemroot* is the Microsoft® Windows® install directory and *versionNumber* is the Microsoft® .NET version number. Both scripts create a database called ASPState that is used by the ASP.NET session state infrastructure, and they include several tables as well as a number of stored procedures.

The InstallSqlState.sql script adds these tables and stored procedures to the TempDB database and consequently loses session data if the computer running this database is restarted. The InstallPersistSqlState.sql script adds these items to the ASPState database and allows session data to be retained when the computer running this database is restarted.

In order to extend this database to support deferred session state items, we need to add one table and two stored procedures: ASPStateDeferredData, usp_WriteDeferredData , and usp_ReadDeferredData.

ASPStateDeferredData

This is the table in which deferred session items will be held. The SQL script to create the ASPStateDeferredData is shown below.

```
 CREATE TABLE [dbo].[ASPStateDeferredData] (
    [SessionId] [char] (32) NOT NULL ,
    [KeyID] [char] (48) NOT NULL ,
    [SessionItemLong] [image] NULL
 )
```

This table allows you to read and write session items with a given KeyID for a user's session. The SQL Server image data type is used for binary data and you use it to hold the deferred session item in binary serialized form. In order to improve access performance, create two indexes for the SessionId and KeyID columns.

One of our goals was to avoid having to write batch procedures to remove stale items in the deferred session state table and to also avoid having to use timestamps to make sure we didn't purge active data. Given that the ASP.NET framework has already implemented this functionality for the ASPStateTempSessions table, you can make use of this by implementing a delete trigger. Then, whenever the ASP.NET framework decides that session information should be deleted, the relevant rows in your ASPStateDeferredData table will also be deleted.

To do this, first create a foreign key constraint that connects the ASPStateDeferredData table's SessionId field to the one in the ASPStateTempSessions table and then add the cascaded delete. This is shown in the following script.

```
ALTER TABLE [dbo].[ASPStateDeferredData] ADD
   CONSTRAINT [FK_DeferredData_ASPStateTempSessions] FOREIGN KEY
   (
      [SessionId]
   ) REFERENCES [dbo].[ASPStateTempSessions] (
      [SessionId]
   ) ON DELETE CASCADE
```

usp_WriteDeferredData Stored Procedure

The usp_WriteDeferredData stored procedure will be used to write items into the ASPStateDeferrredData table and is shown below.

```
ALTER PROCEDURE dbo.usp_WriteDeferredData (
   @SessionID tSessionId,
   @KeyID char(48),
   @SessionItem tSessionItemLong )
AS
DECLARE @TempSessionID tSessionId
   -- Test to see if there is an existing entry in the deferred items table
   IF EXISTS (   SELECT SessionID FROM ASPState..ASPStateDeferredData
            WHERE SessionID = @SessionID
            AND KeyID = @KeyID)

      BEGIN -- There is an existing entry, so update it

         UPDATE ASPState..ASPStateDeferredData
            SET SessionItemLong = @SessionItem
            WHERE SessionID = @SessionID
            AND KeyID = @KeyID

      END
   ELSE
      BEGIN -- There is not an existing entry, so insert one

         INSERT INTO ASPState..ASPStateDeferredData (SessionID, KeyID, SessionItemLong)
            VALUES(@SessionID, @KeyID, @SessionItem)

      END
```

The usp_WriteDeferredData stored procedure accepts three input parameters: the **SessionID**, the **KeyID**, and **SessionItem**. If there is an existing entry for this **SessionItem** in the ASPStateDeferredData table, it is updated with the new **SessionItem** value. If there was not an existing entry, the **SessionItem** value is inserted into the ASPStateDeferredData table.

usp_ReadDeferredData Stored Procedure

The usp_ReadDeferredData stored procedure is used to read items from the ASPStateDeferrredData table and is shown below.

```
ALTER PROCEDURE dbo.usp_ReadDeferredData (
   @SessionID tSessionId,
   @KeyID char(48) )
AS

DECLARE @textptr AS tTextPtr
DECLARE @length AS INT

   -- Get the deferred session item from the deferred data
   SELECT
      @textptr = TEXTPTR(SessionItemLong),
      @length = DATALENGTH(SessionItemLong)
   FROM AspState..ASPStateDeferredData
   WHERE SessionID = @SessionID AND KeyID = @KeyID

   IF @length IS NOT NULL
BEGIN
   READTEXT ASPState..ASPStateDeferredData.SessionItemLong @textptr 0 @length
END
   RETURN 0
```

The usp_ReadDeferredData stored procedure attempts to select the session item from the ASPStateDeferredData table for a given **SessionID** and **KeyID**. If one is found, the Transact-SQL READTEXT is used to read the item.

Extending the UserSession Object for Deferred Data

We now have the essential framework in place to implement deferred session state management: We have a **UserSession** object used to hold user session information, and a table capable of holding deferred session data. All that is needed is to extend the **UserSession** object to make use of the deferred data table.

We need to extend the **UserSession** object in two ways: Implement a Deferred Session Manager with methods to read/write items to/from the deferred session table, and add new property items with get/set accessors that use the Deferred Session Manager.

Deferred Session Manager

The Deferred Session Manager is a useful object to create. It gives the ability to pass it within your application and thus give other objects the ability to access the deferred storage items. You can use this to good effect by creating a **DeferredBitmaps** object used to store the optional images. Before we discuss this advanced use of the Deferred Session Manager, we will describe its implementation and show it being used to read/write a compulsory image of the sale item.

The Deferred Session Manager exposes two public methods: **ReadDeferredItem()** and **WriteDeferredItem()**.

Deferred session manager constructor

The Deferred Session Manager is going to be used for all access to the deferred storage and it is accessed as a property from the **UserSession** object. In this way, development is kept simple: all properties, methods, and objects associated with the management of user session are accessed from the **UserSession** object.

Items in the deferred session storage need to have access to the user's **sessionID** so that the delete trigger will be correctly invoked when ASP.NET purges stale session items. Unfortunately, the **sessionID** value used by the ASPStateTempSessions table is not simply the **Session.SessionID** property. In order to guarantee unique entries when the table is used by more than one ASP.NET Web application, an application ID is appended to the session ID to create an extended **sessionID** value.

As the Deferred Session Manager will need to use a similarly extended session ID value, we use a private property, called **extendedSessionID**. This appends the application ID value to the **Session.SessionID** value.

The code for the constructor and for assigning the private **extendedSessionID** property is given below.

```
public class DeferredSessionManager
{

   // The user's session
   private HttpSessionState  userSession;

   // The user's Session ID postfoxed with the hex formatted applicationID
   private string extendedSessionID;

   /// <summary>
   /// Constructor for the Deferred Session Manager
   /// </summary>
   /// <param name="userSession">The user's session state</param>
   public DeferredSessionManager( HttpSessionState userSession )
   {
      // Remember the user's session object and the application ID
      this.userSession = userSession;

      // Remember the hex formatted application ID
      extendedSessionID = userSession.SessionID + ApplicationID();
   }
}
```

The **ApplicationID()** routine is simply a private method that invokes the **TempGetAppID** stored procedure and returns the application ID as a string in hexadecimal format. As it is such a trivial routine, it is not discussed in this article. However, a full implementation is given in the code download.

ReadDeferredItem()

The **ReadDeferredItem** method simply invokes the **usp_ReadDeferredData** stored procedure and is shown below.

```
/// <summary>
/// Used to read items that have been placed in deferred session storage
/// </summary>
/// <param name="deferredItemKey">The key of the item to retrieve</param>
/// <returns>The retrieved item</returns>
public object ReadDeferredItem( string deferredItemKey )
```

```
{
   object returnValue = null;
   SqlCommand cmdReadDeferredItem = null;

   try
   {
      // The connection to the ASPState Database
      SqlConnection aspStateConnection =
        new SqlConnection(
        ConfigurationSettings.AppSettings[ "ASPStateDatabaseConnection" ] );

      // The command to invoke the stored procedure
      // to get the item from the deferred session state table
      cmdReadDeferredItem = new SqlCommand();
      cmdReadDeferredItem.CommandText = "usp_ReadDeferredData";
      cmdReadDeferredItem.CommandType = CommandType.StoredProcedure;
      cmdReadDeferredItem.Connection = aspStateConnection;

      // The "@sessionID" input paramter is a
      ..user defined data type that resolves to char(32)
      SqlParameter sessionID =
        new SqlParameter( "@SessionID",
        this.extendedSessionID);
      sessionID.SqlDbType = SqlDbType.Char;
      sessionID.Size = 32;

      // The "@keyID" parameter is a SQL defined data type of char(48)
      SqlParameter keyID = new SqlParameter( "@KeyID", deferredItemKey );
      keyID.SqlDbType = SqlDbType.Char;
      keyID.Size = 48;

      // Append the parameters to the stored procedure command
      cmdReadDeferredItem.Parameters.Add( sessionID );
      cmdReadDeferredItem.Parameters.Add( keyID );

      // Open the connection and execute the command
      cmdReadDeferredItem.Connection.Open();

      SqlDataReader dataReaderDeferredItem =
        cmdReadDeferredItem.ExecuteReader();

      // If we got a response
      if ( dataReaderDeferredItem != null )
      {
         // Try to read the first record
         dataReaderDeferredItem.Read();

         // Get the object to be deserialised
         // from the SessionItemLong parameter
         byte[] objectToDeserialize =
           (byte[]) dataReaderDeferredItem[ "SessionItemLong" ];

         // Return the deserialised item
         returnValue = DeserializeItem( objectToDeserialize );
      }

   }
   catch( SqlException eSQL )
   {
      // Report errors
      throw new Exception(
        string.Format( "SQL Exception {0:F0} - {1}",
        eSQL.Number, eSQL.Message ), eSQL.InnerException);
   }
   catch( Exception e )
   {
      // Report errors
      throw new Exception( e.Message, e.InnerException);
   }
   finally
   {
      // Close the connection
      cmdReadDeferredItem.Connection.Close();
   }

   // Return the return value
   return returnValue;
}
```

This code first gets a connection to the ASPState database and builds a SQL **Command** object to invoke the **usp_ReadDeferredData** stored procedure. It then adds the extended **sessionID** and **keyID** parameters to the SQL **Command** to uniquely identify the object to return. The stored procedure is then invoked and the returned object is accessed from the **SessionItemLong** parameter. This parameter is a serialized representation of the object, and it is deserialised before it is returned.

WriteDeferredItem()

The **WriteDeferredItem** method simply invokes the **usp_WriteDeferredData** stored procedure and is shown below.

```
/// <summary>
/// Write a single deferred session state item
/// </summary>
/// <param name="deferredItem">The item to write</param>
/// <param name="deferredItemKey">The item's key</param>
public void WriteDeferredItem( object deferredItem, string deferredItemKey )
{
   // Only write non-null items
   if ( deferredItem != null )
   {
      SqlCommand cmdWriteDeferredItem = null;

      try
      {
         // The connection to the ASPState Database
         SqlConnection aspStateConnection =
         new SqlConnection(
         ConfigurationSettings.AppSettings[ "ASPStateDatabaseConnection" ] );

         // The command to invoke the stored procedure
         // to insert the item into the deferred session state table
         cmdWriteDeferredItem = new SqlCommand();
         cmdWriteDeferredItem.CommandText = "usp_WriteDeferredData";
         cmdWriteDeferredItem.CommandType = CommandType.StoredProcedure;
         cmdWriteDeferredItem.Connection = aspStateConnection;

         // The "@sessionID" input paramter is a
         // user defined data type that resolves to char(32)
         SqlParameter sessionID =
           new SqlParameter( "@SessionID",
           this.extendedSessionID );

         // The "@keyID" parameter is a
         // user defined data types that resolves to char(48)
         SqlParameter keyID = new SqlParameter( "@KeyID", deferredItemKey );

         // The "@itemLong" parameter is a
         // user defined data types that resolves to image(16)
         byte[] serializedItem = SerializeItem( deferredItem );
         SqlParameter sessionItem =
           new SqlParameter( "@SessionItem", serializedItem );

         // Append the parameters to the stored procedure command
         cmdWriteDeferredItem.Parameters.Add( sessionID );
         cmdWriteDeferredItem.Parameters.Add( keyID );
         cmdWriteDeferredItem.Parameters.Add( sessionItem );

         // Open the connection and execute the command
         cmdWriteDeferredItem.Connection.Open();
         cmdWriteDeferredItem.ExecuteScalar();
      }
      catch( SqlException eSQL )
      {
         // Report errors
         throw new Exception(
           string.Format( "SQL Exception {0:F0} - {1}",
           eSQL.Number, eSQL.Message ), eSQL.InnerException);
      }
      catch( Exception e )
      {
         // Report errors
         throw new Exception( e.Message, e.InnerException);
      }
      finally
      {
         // Close the connection
         cmdWriteDeferredItem.Connection.Close();
      }
   }
}
```

This code first gets a connection to the ASPState database and builds a SQL **Command** object to invoke the **usp_WriteDeferredData** stored procedure. It then appends the extended **sessionID** and **keyID** parameters to the SQL **Command** that uniquely identify the object to write. The item to be written to the deferred storage is serialized and appended to the SQL **Command** as the **sessionItem** parameter. The stored procedure is then invoked and the item is written to the deferred storage.

Storing the Deferred Session Manager in Application State

It is worth holding the Deferred Session Manager in application cache. This facilitates the object being created a minimal number of times, because it is not constructed every time a user accesses deferred session data. And by holding it in application cache, it can be scavenged should the Web server come under memory pressure.

As the Deferred Session Manager will be accessed as a property from the **UserSession** object, the following code is added to the **UserSession** object.

```
private const string DEFERRED_SESSION_MANAGER_KEY = "DeferredSessionManager";
private DeferredSessionManager _deferredSessionManager = null;
private DeferredSessionManager deferredSessionManager
{
   get
   {
      // Try to get it from the Application Cache
      _deferredSessionManager =
        (DeferredSessionManager) this.applicationCache.Get(
         DEFERRED_SESSION_MANAGER_KEY );

      // Add it in, if needed
      if ( _deferredSessionManager == null )
      {
         _deferredSessionManager =
           new DeferredSessionManager( userHttpSessionState );
         this.applicationCache[ DEFERRED_SESSION_MANAGER_KEY ] =
           _deferredSessionManager;
      }

      return ( _deferredSessionManager );

   }
}
```

Invoking the Deferred Session Manager

We now have almost all the pieces in place in order to implement management of deferred session state. We have a **UserSession** class that provides abstracted access to the various items that need to be stored in a user's session. We have a deferred storage database table that can be used to store large or infrequently accessed objects. Lastly, we have a Deferred Session Manager that can read/write items to/from the deferred storage database table.

All that remains is to extend the **UserSession** class to make use of the Deferred Session Manager.

The deferred items cache

The obvious place to invoke the Deferred Session Manager is in the get/set accessors for deferred session state items; the get accessor would use the Deferred Session Manager's **ReadDeferredItem()** method and the set accessor would use the **WriteDeferredItem()** method.

Unfortunately, this simple implementation could undermine the performance and scalability of the Web application, as there will be database access every time a deferred item is accessed.

Imagine an object held in deferred storage that exposed a series of properties and methods. Every time a developer accessed a property or invoked a method, the get accessor for the object would use the Deferred Session Manager's **ReadDeferredItem()** method to invoke the **usp_ReadDeferredItem** stored procedure and the object from the deferred storage. Clearly, the developer should declare a local copy of the object and access the properties and invoke the methods from that. However, we cannot guarantee that our developers will be so canny, and we must ensure our deferred session infrastructure will not undermine performance.

The solution to this problem is to place deferred items in a private cache controlled by the **UserSession** object. This will allow us to get the deferred item from the database on the first access and return the cached item on further access while the set accessor simply inserts the item into the cache. Actually writing the cache to deferred storage is postponed until just before the Web page is rendered prior to transmission to the client.

After the declaration of the private **userHttpSessionState**, we include the following code to declare a private hashtable that will be used as the cache of deferrable session state items.

```
   // Used as a cache for deferrable items.
   private Hashtable deferredItemsCache = new Hashtable();
```

Writing the deferred items cache to deferred storage

Reading items from the deferred items cache is as simple as accessing any item from a hashtable, but you need a routine to write all the deferred items to the deferred storage at the last possible moment, as described above.

Fortunately, writing the cache is simple to implement. All we need to do is cycle through all keys in the deferred items cache and invoke the Deferred Session Manger's **WriteDeferredItem ()** method. The code for this is exposed as a public method of the **UserSession** class as shown below.

```
/// <summary>
/// Persist all deferred session state items
/// The items must be persisted one by one in order
/// that they can be individually read back in when accessed
/// </summary>
public void WriteDeferredItems()
```

```
{
   // Loop through all keys in the deferred items cache
   foreach ( string key in deferredItemsCache.Keys )
   {
      // Write out the deferred item
      WriteDeferredItem( deferredItemsCache[key], key );
   }
}
```

The correct place to invoke this routine is from the **OnUnload()** event handler of the **ApplicationPage** class. This will ensure that the cache is written to the ASPStateDeferrredData table at the last possible moment, as it will be invoked before the instance of the Web page is unloaded from the server's memory.

The following code, then, is added to the **ApplicationPage** class.

```
/// <summary>
/// Before the page is unloaded from memory,
/// write the deferred cached items out
/// </summary>
/// <param name="e"></param>
protected override void OnUnload(EventArgs e)
{
   if ( IsPostBack )
   {
      this.userSession.WriteCachedDeferredItems();
   }
   base.OnUnload (e);
}
```

Note that we make a performance improvement by only writing the cached items after a postback to the Web page. The assumption here is that HTTP GETs do not have deferred session items updated. If this is not the case in your Web application, you will need to remove the **IsPostBack** test.

The UserSession's CompulsoryImage Property

We can now make use of the deferred session code infrastructure and define an item that will be stored in deferred storage.

In this simple instance, we define a compulsory image that the user must enter when they submit an item for sale. The code for this is shown below.

```
/// <summary>
/// Used to hold the item's compulsory image in deferred session
/// </summary>
public Bitmap CompulsoryImage
{
   get
   {
      // If the compulsory image is not in the deferred session cache
      if ( deferredItemsCache.ContainsKey(
        SessionItemKey.COMPULSORY_IMAGE ) == false )
      {
         // Use the deferred session manager to read it in to the cache
         deferredItemsCache[ SessionItemKey.COMPULSORY_IMAGE ] =
           deferredSessionManager.ReadDeferredItem(
           SessionItemKey.COMPULSORY_IMAGE );
      }
      // Return the compulsory image from the deferred session cache
      return ( (Bitmap) deferredItemsCache[
        SessionItemKey.COMPULSORY_IMAGE ]);
   }
   set
   {
      // Save the item in the deferred session cache
      deferredItemsCache[ SessionItemKey.COMPULSORY_IMAGE ] = value;
   }
}
```

The get accessor first checks to see whether the "COMPULSORY_IMAGE" item is already in the deferred items cache by using the Hashtable's **ContainsKey()** method. If this fails, the Deferred Session Manager is used to read the item from the deferred storage, and it is put into the cache for subsequent access.

The set accessor merely updates the "COMPULSORY_IMAGE" item in the deferred items cache, as actually writing the deferred items cache has been postponed until the last possible moment—just before the Web page is unloaded from server memory.

The UserSession's OptionalImages Property

The **OptionalImages** property is used to store a number of additional images of the item being advertised. It is tempting to merely implement **OptionalImages** as a hashtable, but this would mean the whole hashtable of all the images would be read every time we wanted to access an optional image—and this undermines our goal of deferring session item access of large or rarely accessed objects until it is really needed.

Our solution is to implement a **DeferredBitmaps** class with an indexer. This will enable instances of the class to be indexed in the same way as arrays. We pass the **UserSession**'s deferred items cache into the **DeferredBitmaps** class constructor, and use the cache to store and retrieve the optional images. This way, when the cache is written to deferred storage, all the optional images will also be written and we take advantage of the existing code to serialize the items and invoke the **usp_WriteDeferredItem** stored procedure.

However, as the optional images are in the deferred items cache, we may need to populate it when an optional image is accessed for the first time. And this is why we implemented the Deferred Session Manager as a class: we can pass it to the DeferredBitmaps constructor and then invoke the **ReadDeferredItem()** method to read items from deferred storage when needed.

The **DeferredBitmaps** object constructor is shown below.

```
/// <summary>
/// Summary description for DeferredBitmaps.
/// </summary>
public class DeferredBitmaps
{
   // We use the Deferred Session Manager
   // to read items from deferred session storage
   DeferredSessionManager deferredSessionManager;

   // This is where the bitmaps will be stored - in the deferred items cache
   private Hashtable deferredItemsCache;

   /// <summary>
   /// Constructor for storing a number of bitmaps in deferred storage
   /// </summary>
   /// <param name="deferredSessionManager">
   /// The Deferred Session Manager </param>
   /// <param name="deferredItemsCache">The cache of deferred items</param>
   public DeferredBitmaps(
     DeferredSessionManager deferredSessionManager,
     Hashtable deferredItemsCache )
   {
      // Remember the deferred items cache and the Deferred Session Manager
      this.deferredItemsCache = deferredItemsCache;
      this.deferredSessionManager = deferredSessionManager;
   }
}
```

OptionalImages indexer

The indexer to get/set individual optional items accepts a key of type string and this is used to index into the hashtable that is the deferred items cache. This key is prefixed with a fixed string to ensure that entries into the deferred items cache use unique keys.

```
/// <summary>
/// Indexer to get/set an individual bitmap
/// </summary>
public Bitmap this[string bitmapKey]
{
   get
   {
      // Build the bitmap's key
      string imageKey = SessionItemKey.OPTIONAL_IMAGE + bitmapKey;

      // See if the bitmap is in the deferred items cache
      if ( deferredItemsCache.ContainsKey( imageKey ) == false )
      {
         // The bitmap wasn't in the cache,
         // so read it in via the Deferred Session Manager
         deferredItemsCache[ imageKey ] =
           deferredSessionManager.ReadDeferredItem( imageKey );
      }
      // Return the requested item
      return (Bitmap) deferredItemsCache[ imageKey ];
   }
   set
   {
      // Save the bitmap in the deferred items cache
      deferredItemsCache[ SessionItemKey.OPTIONAL_IMAGE + bitmapKey ] = value;
   }
}
```

The get accessor uses the **Hashtable's ContainsKey()** method to see if there is an entry in deferred items cache. If there isn't an entry, we use the Deferred Session Manager to read in the deferred item and place it in the cache.

The set accessor merely places the image in the deferred items cache. As with other items put into the cache, writing it to deferred session storage is postponed until the last possible moment when the whole of the cache is written out.

Conclusion

This article has shown how to implement a generic mechanism that enables the reading and writing of selected session state items to be deferred. This mechanism is most effective when large but rarely used items need to be stored in session state.

The mechanism has taken advantage of existing capabilities within the ASP.NET framework by extending the ASP.NET session state database such that items stored in the deferred session state table are automatically deleted at the same time as items in the standard session state table.

The mechanism has also taken advantage of best practice procedures when creating Web applications by extending the conventionally used **ApplicationPage** and **UserSession** classes. Lastly, consideration has been given to improve performance through the use of a cache for items held in the deferred session store.

Related Books

ASP. NET: Tips, Tutorials, & Code

Microsoft ASP.NET Coding Strategies with the Microsoft ASP.NET Team

Essential ASP.NET with Examples in C#

Programming Microsoft ASP.NET

About the author

Callum Shillan is a Principal Consultant working for Microsoft in the UK, specializing in Internet Business. He's been working with C# and ASP.NET on large Internet websites for the last few years. Callum can be reached at callums@microsoft.com.

Peter Norell is a designer working for Atos Origin in the UK. Peter has been doing object-oriented design and development for the last 8 years. He has, for the last year, been lead designer for a large scale C# and ASP.NET project.

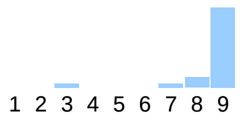
Print E-Mail

How would you rate the quality of this content?

	1	2	3	4	5	6	7	8	9	
Poor	○	○	○	○	○	○	○	○	○	Outstanding

Tell us why you rated the content this way. (optional)

Average rating:
9 out of 9



20 people have rated thi