Managing State

In a WebObjects application, you often have variables whose state you want to maintain for a page, across a user session, or across the entire application. For example, your application might maintain a shopping cart that each user fills with items as he or she progresses through a series of pages and forms.

This chapter describes the different approaches you can use in a WebObjects application to store and manage state. Some of the topics covered in the chapter are:

- Why do you need to store state?
- How WOApplication manages state
- Using global, session, and persistent variables
- Storing state in the application
- Storing state in the page
- Implementing your own state storage
- Setting session timeout

Why Do You Need to Store State?

In a WebObjects application, pages aren't persistent. They're created at the beginning of a transaction, and they disappear at the end. A transaction is defined as a client request coming in and a response (usually an HTML page) going out. The life of a page actually spans two transactions:

1. First, the WOWebScriptComponentController associated with the page generates a response for a given request.

2. The WOWebScriptComponentController then handles the subsequent incoming request (such as a request triggered by a user clicking on a hyperlink).

Between these two occurrences, the WOWebScriptComponentController associated with a page is destroyed and reconstructed. Any variables in your component script that haven't explicitly been made persistent are lost.

For most variables, this automatic destruction isn't a problem. Either the variables' values don't need to be maintained or you can re-initialize them in your script's **awake** method. However, some variables need to live beyond the natural life cycle of a page. Such variables need to be declared as global, session, or persistent. This is discussed in more detail in the section "Variables and State."

When you declare a variable as global, it's maintained by the application across the life of the application. When you declare a variable as session or persistent, its state is restored when the application receives a request and stored before a response is returned.

Storing state is expensive. As users access an application and progress through its pages, the application accumulates state information for all active pages in what can be multiple sessions. This state can grow to be quite large. Consequently, you should use session and persistent variables only where necessary.

How WOApplication Manages State

As with most activities in a WebObjects application, state management is organized around the request-response loop. The exact steps in a cycle of the loop depend on where state is being stored—on the application server (which is the default), or in the page. WebObjects uses an NSDictionary to store state data in the application—in other words, when you store objects in the application, they're kept intact in memory. (The NSDictionary class stores data as key-value pairs; you access a value through its key.) When state is stored in the page, however, state data is copied from the application server and archived into an NSData object whose ASCII representation is then put into a hidden field in the page. (The NSData class provides an object-oriented wrapper for byte buffers.) The corresponding state data is subsequently removed from the server. For a discussion of the advantages and disadvantages of each approach, see the section "Storing State in the Application vs. Storing State in the Page." For more information on NSDictionary and NSData, see the *Foundation Framework Reference*.

Figure 1 illustrates the general sequence of events that occurs when an application receives a request (for example, when a user clicks a hyperlink).

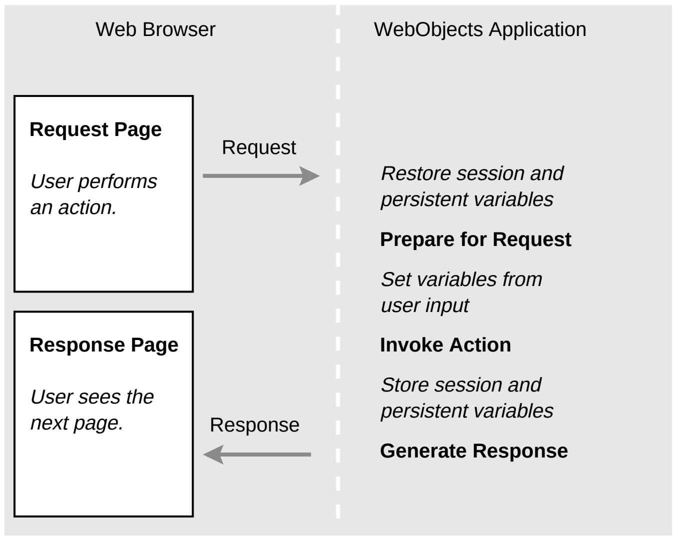

Figure 1. Handling a Request

The following steps describe in more detail how an application manages state when it receives a request:

1. In its **handleRequest:** method, WOApplication asks the request for the stateID in its URL.

   A stateID is a value generated by WOApplication during request handling that's used to store and retrieve state information. A stateID is only present if the application has session or persistent variables—otherwise, the value returned is **nil**.

   A stateID is an NSString that has the format *sessionIdentifier.key.extension* —for example, 12.124563322.234. WebObjects doesn't require this format (in case you want to implement your own stateID scheme). However, it may be useful to understand why WebObjects takes this approach.

   The *sessionIdentifier* uniquely identifies each user session. To prevent users from accessing the pages in another user's session, the stateID also includes a key field. The *key* is a randomly generated number; you can't access the pages in a session without the proper key. The key also ensures that if an application crashes and restarts (thereby renumbering the sessions), an existing session is not in danger of having its stateID duplicated. The *extension* is incremented with every transaction, which makes the URL for each transaction unique. This prevents the situation in which a user might be iterating on a single page, but the page is never refreshed since it's being cached by the browser.

2. WOApplication checks to see if there is an NSData object embedded in the request. If there is, it means that state is stored in the page (instead of in the application).

3. WOApplication restores the state using the method **restoreToStateWithID:** if the state is in the application, or **restoreToStateWithID:data:** if the state is in the page.

4. The WOApplication object finds or creates an object to represent the *request page*—the page from which the request was made.

5. The WOApplication object sends the request page object a **prepareForRequest:inContext:** message which stores user input in objects.

6. The WOApplication object sends the request page object an **invokeActionForRequest:inContext:** message. This method invokes an action method if one has been triggered, and determines which object to use to generate the response. This is described in more detail in the chapter "Inside the Request-Response Loop."

7. WOApplication gets a new stateID. If no stateID existed before, an entirely new one is created. If there is an existing stateID, then the last part of the stateID (the extension) is incremented—this ensures that every transaction has a unique stateID. The first two fields of the stateID stay the same.

   The **stateID** method also snapshots the session and persistent variables and stores their state (whether in the application or in the page).

8. WOApplication checks to see if state should be stored in the response page—that is, if the page contains a WOStateStorage element (this is described in more detail in the section "How to Store State in the Page").

   If so, WOApplication invokes **stateDataForID:**, which takes the state stored in the application, archives it into an NSData object, and returns the NSData object. The WOStateStorage element then puts an ASCII representation of this NSData object into a hidden field in the page.

9. The WOApplication object sends a **generateResponse:inContext:** message to the object representing the response page. The method **generateResponse:inContext:** generates HTML for the page.

   State is frozen before a response is generated (it was frozen in Step 7, when the stateID method snapshotted session and persistent variable values and stored them). If you modify session or persistent variables during a response, those changes will be lost. The stored state will be used to restore state across your session the next time the application handles a request.

10. If you're storing state in the page, WOApplication removes the corresponding state from the server using **removeStateDataForID:**.

11. WOApplication performs garbage collection. Specifically, it removes the state for sessions that have timed out. This doesn't happen for every transaction—it's time-based.

Variables and State

To store a variable's state in a WebObjects application, you declare it as global, session, or persistent.

The following table summarizes the different types of variables and their scope. For a comprehensive discussion of variables and scope, see the chapter "Using WebScript."

Variable Type	**Where It's Declared**	**How You Declare It**	**Where It's Visible**	**How Long It Lives**
Local	Inside a method in either an application or a component script	id myVar;	Only inside the method in which it's declared	For the duration of the method
Transaction	Outside a method in a component script	id myVar;	Inside the script in which it's declared	For the duration of a transaction, which is defined as a request coming in and a response (usually an HTML page) going out
Persistent	**In WebScript:** Outside a method in a component script	**In WebScript:** persistent id myVar;	**In WebScript:** Inside the script in which it's declared	For the duration of a session
	In Objective-C: In a class that's subclassed from WOComponentController, declare a persistent variable just as you would any instance variable. Then implement the **persistentKeys** method. This method returns an array of the key names of the instance variables you want to make persistent. Persistent variables you declare in Objective-C are only visible within the class in which they're declared.			
Session	**In WebScript:** Outside a method in an application script	**In WebScript:** session id myVar;	**In WebScript:** In the application script. Component scripts can access session variables by messaging the application. Every session has its own version of a session variable.	For the duration of the session
	In Objective-C: In a class that's subclassed from WOWebScriptApplication, declare a session variable just as you would any instance variable. Then implement the **sessionKeys** method. This method returns an array of the key names of the instance variables for which you want to store state across a session. Session variables you declare in Objective-C are only visible within the class in which they're declared, but they can be accessed by messaging the application object.			
Global	Outside a method in an application script	id myVar;	In the application script. Component scripts can access global variables by messaging the application. Every session sees global variables with the same value.	For the duration of the application

Note: A persistent variable is identical to a session variable except that it's scoped to the page in which it is declared.

Storing State in the Application vs. Storing State in the Page

By default, WebObjects stores state on the application server. However, it also provides a mechanism for storing state in the page. This section discusses the advantages and disadvantages of each approach. Note that for a given page, you can't mix approaches, though you can use different approaches for different pages in an application.

Advantages of State in the Application

The advantages of storing state in the application are as follows:

- Simplicity

  Because state in the application is the default in WebObjects, you don't have to do any extra work to achieve it. All you have to do is declare session and persistent variables, and their state is automatically stored in the application.

- Security

  When you store state in the page, it's conceivable that the data could be modified by users. This isn't a problem when you store state in the application.

Disadvantages of State in the Application

The disadvantages of storing state in the application are as follows:

- Scalability (limited to one application)

  You can't use round-robin to service requests for a given session since every request from that session has to go back to the same application on the same server.

- Scalability (your application can consume a lot of memory)

  When you store state in the application, your application grows with each request. Your state and thereby your application can consume a lot of memory. This isn't the case when you store state in the page.

- Reliability

  If your server crashes, you lose your state. This doesn't happen when you store state in the page—your server could crash and reboot, and as long as the user's browser is still running, the state is maintained in the application's pages.

Advantages to State in the Page

The advantages to storing state in the page are as follows:

- Scalability (not limited to one application)

  When you store state in the application, every request for a given session has to go back to the same application on the same server. This isn't the case when you store state in the page—you can use a round-robin approach that performs load balancing across multiple versions of your application running on multiple servers.

- Scalability (application doesn't grow)

  When you store state in the page, your application doesn't grow with each new session and request.

- Reliability

  When you store state on the page, your state isn't dependent on the application. The server can crash and reboot, but as long as a user's browser is still running, the state continues to be stored in the pages of the application.

Disadvantages to State in the Page

The disadvantages to state in the page are as follows:

- If your application has a lot of state, this means that a lot of data has to be passed from the user's browser to the application with every request.

  This could degrade your application's performance.

- Security

  WebObjects doesn't encrypt the data stored in a page (though you can implement your own encryption). Users could conceivably access and modify the data stored in a page, which wouldn't be a problem with state stored in the application.

- Frames

  Storing state in the page is a problem if the "pages" in question are frames. Your state can quickly get out of sync. For example, suppose you have a mail application with two frames. One of the frames shows a list of messages with one message selected, and the other frame shows the text of the selected message. If you delete the message in the top frame, the state of the bottom frame isn't updated (unless you implement your own solution). If you're using frames, it's preferable to store state in the application.

- The only way to pass your state input to the application is by submitting a form.

  State in the page is stored in hidden fields. The contents of hidden fields can only be passed in a URL when you submit a form. This constrains your user interface—you have to use a button or an active image to perform an action in a page; otherwise your state information will be treated as if it doesn't exist. Note that you must use a button or active image; even if a hyperlink is placed inside of a form, clicking it won't have the effect of submitting the form.

- Need to implement archiving for custom objects

  When state is stored in the page, objects are archived into an NSData object. If you have custom classes for which you need to store state, they must know how to archive and unarchive themselves (see the section "Storing State for Custom Objects" for more information). This isn't required for storing state in the application. This restriction only applies to custom classes—most of the Foundation classes you'll use already implement archiving and unarchiving.

- If a user restores state from an old page, it overwrites a session's current state.

  For example, suppose your application has a shopping cart object that users fill with items. A user might have put flowers in it on one page, and then put a t-shirt in it on a later page. If the user backtracks to the flower page and changes the order, the t-shirt is no longer included in the current state.

How to Store State in the Page

By default, when you use session and persistent variables, their state is stored in the application. However, you can also choose to store state in the page. To store state in the page:

1. In the HTML template associated with the page, add a WebObject that represents the state you want to store:

   ```
   <WEBOBJECT NAME="FORM">
         <WEBOBJECT NAME = "STATE"></WEBOBJECT>
         <WEBOBJECT NAME = "NAME_FIELD"></WEBOBJECT>
         <P>
         <WEBOBJECT NAME = "SUBMIT_BUTTON"></WEBOBJECT><P>
   </WEBOBJECT>
   ```

   Notice that this WebObject is declared inside a form This is required since the state will be stored in a hidden field, and in HTML, hidden fields have to be inside forms.

2. Add a corresponding declaration to the declarations file associated with the page:

   ```
   STATE: WOStateStorage{ };
   ```

When you run your application, your state is stored as the ASCII representation of an NSData object in the location you specified in your HTML template.

Note: If your page has multiple forms and you're storing state in the page, you should include a WOStateStorage element in each form. If you fail to do this, when you submit a form that doesn't have a WOStateStorage element in it, all of your state will be lost.

Storing State for Custom Objects

When state is stored in the page, objects are archived into an NSData object. If you have custom classes for which you need to store state, they must know how to archive and unarchive themselves. To achieve this, your custom classes must conform to the NSCoding protocol and implement its **encodeWithCoder:** and **initWithCoder:** methods. **encodeWithCoder:** instructs an object to encode its instance variables to the coder provided; an object can receive this method any number of times. **initWithCoder:** instructs an object to initialize itself from data in the coder provided; as such, it replaces any other initialization method and is only sent once per object.

Note: Most of the Foundation classes already conform to the NSCoding protocol. This section only applies to the custom classes you write yourself.

For example, the DodgeDemo ShoppingCart class in the WebObjects examples includes the following implementations for **encodeWithCoder:** and **initWithCoder:**.

```
- (void)encodeWithCoder:(NSCoder *)coder {
  [super encodeWithCoder:coder];
  [coder encodeObject:carID];
  [coder encodeObject:colorID];
  [coder encodeObject:colorPicture];
  [coder encodeObject:packagesIDs];
  [coder encodeObject:downPayment];
  [coder encodeObject:leaseTerm];
}

- initWithCoder:(NSCoder *)coder {
  self = [super initWithCoder:coder];
  carID = [[coder decodeObject] retain];
  colorID = [[coder decodeObject] retain];
  colorPicture = [[coder decodeObject] retain];
  packagesIDs = [[coder decodeObject] retain];
  downPayment = [[coder decodeObject] retain];
  leaseTerm = [[coder decodeObject] retain];
  car = nil;
  return self;
}
```

Note: If you're developing WebObjects applications on Windows NT, your code shouldn't invoke `[super encodeWithCoder:coder]`. This is because in the version of the Foundation Framework running on Windows NT, NSObject doesn't conform to the NSCoding protocol.

For more information on archiving, see the NSCoding, NSCoder, NSArchiver, and NSUnarchiver class specifications in the *Foundation Framework Reference*.

Implementing Your Own State Storage

WebObjects provides direct support for storing state in the application and in the page. However, you can also implement your own state storage—for example, you might want to use a database or Netscape "cookies" to store state. Cookies are used to store state in the client. They have all of the advantages of WebObjects' "state in the page" solution, and some additional ones, such as working with hyperlinks and frames.

You can either implement a simple custom solution (for example, your application could maintain a session state instance variable and restore it in **awake**), or you can base your approach on WOApplication's state handling API.

Example: FaultTolerantApplication

The following example, FaultTolerantApplication, shows one approach to implementing a custom storage solution. FaultTolerantApplication is a subclass of WOWebScriptApplication that archives state in the file system. Its name derives from the fact that because it archives state in the file system, no more than the last interaction in a session can ever be lost.

FaultTolerantApplication includes the following methods:

- **stateID** (overridden from WOApplication)

  This method invokes `[super stateID]`, which stores the state in the application and returns a new stateID. The method then invokes the WOApplication method **stateDataForID:**, which takes the state in the application, archives it into an NSData object, and returns the NSData object. Finally, the method stores the NSData object in the file system and removes the state from the server.

- **restoreToStateWithID:** (overridden from WOApplication)

  This method retrieves the NSData object that the **stateID** method stored in the file system. It then invokes `[super restoreToStateWithID:aStateID data:stateData]`, which restores the session's state from the NSData object.

- **storeIDFromStateID:**

  This method constructs a storeID out of the first two fields of the stateID, and returns it. This method is invoked from **stateFilePathForStateID:**, which uses the returned storeID to construct a path for the file used to store the state.

- **stateFilePathForStateID:**

  Given a stateID, this method returns a file path based on the stateID. This method is invoked by both **stateID** and **restoreToStateWithID:**, which use it respectively to create and find the file holding the application's state.

The header (**.h**) and implementation (**.m**) files for FaultTolerantApplication are listed below.

FaultTolerantApplication.h

```
#import <WebObjects/WOWebScriptApplication.h>

@interface FaultTolerantApplication:WOWebScriptApplication

@end
```

FaultTolerantApplication.m

```
#import "FaultTolerantApplication.h"

@implementation FaultTolerantApplication

- (NSString *)storeIDFromStateID:(NSString *)aStateID
{
  NSArray *stateIDComponents = [aStateID componentsSeparatedByString:@"."];
  NSString *storeID = nil;

  if (!stateIDComponents)
     storeID = nil;
  else if ( ([stateIDComponents count] != 3) )
     [NSException raise:NSInvalidArgumentException
       format:@"Invalid state ID: %@.", aStateID];
  else
     storeID = [NSString stringWithFormat:@"%@.%@",
       [stateIDComponents objectAtIndex:0],
       [stateIDComponents objectAtIndex:1]];
  return storeID;
}

- (NSString *)stateFilePathForStateID:(NSString *)aStateID
{
  NSString *storeID = [self storeIDFromStateID:aStateID];
  NSString *stateDirectory = [self pathForResource:@"State" ofType:@""];
  NSString *stateFilePath = [NSString stringWithFormat:@"%@/%@",
     stateDirectory,  storeID];
  return stateFilePath;
}

- (void)restoreToStateWithID:(NSString *)aStateID
{
  NSString *stateFilePath;
  NSData *stateData;

  // Get the path for the state archive file and read the state NSData
  stateFilePath = [self stateFilePathForStateID:aStateID];
  stateData = [[[NSData alloc] initWithContentsOfFile:stateFilePath] autorelease];

  // Restore the stateData as the state for the current session
  [super restoreToStateWithID:aStateID data:stateData];
}

- (NSString *)stateID
{
  NSString *newStateID;
  NSData *stateData;
  NSString *stateFilePath;

  // Ask the application to capture the state for the session (snapshot all
  // the persistent and session keys for all the active pages in the session),
  // and to return a new stateID (stateID for the current interaction in the
  // session).
  newStateID = [super stateID];

  // Now that the session state has been prepared, ask the application for it
  // (in an NSData form).
  stateData = [self stateDataForID:newStateID];

  // Write the session state to the appropriate file
  stateFilePath = [self stateFilePathForStateID:newStateID];
  [stateData writeToFile:stateFilePath atomically:NO];

  // Make sure that the application does not keep its own copy of the state
  [self terminateSession];
  return newStateID;
}
@end
```

Setting Session TimeOut

WOApplication includes two methods for explicitly collecting and removing all of a session's state from the application:

- **terminateSession**

  This method clears the state for a given session. You might choose to invoke this method from an action that represents the end of a user's interaction with an application, for example:

  ```
  - registerMe
  {
     //... process data...
     [WOApp terminateSession];
  }
  ```

- **setSessionTimeOut:**

  This method sets the amount of time a session can be inactive before it's timed out. By default sessions don't time out. For example, you could include the following in your application script's **awake** method:

```
- awake
{
  // Time out sessions that have been inactive for more than 2 minutes
  [WOApp setSessionTimeOut:120];
}
```

You can use these methods to control the size of your process and thereby improve your application's performance. For more information, see the class specification for WOApplication.

Summary

Why Do I Need to Store State?

You need to store state because pages aren't persistent in a WebObjects application. Variables that aren't session or persistent only live for the duration of a transaction, which is defined as a client request coming in and a response page going out.

What Are My Options for Storing State?

You can:

- Accept the default WebObjects behavior of storing state in the application server
- Store state in the page
- Implement your own state storage solution by subclassing WOWebScriptApplication

What Are the Basic Differences Between Storing State in the Application and Storing State in the Page?

When you store state in the application, objects are stored intact in memory. When you store state in the page, objects are archived into an NSData object, and the ASCII representation of the NSData object is put in the dynamically generated HTML page.

What Do I Need to Do to Store State?

You need to declare variables for which you want to store state as either session variables (in the application script) or as persistent variables (in a component script).

When you store state in the application, you don't have to do any extra work beyond declaring variables as session or persistent. When you store state in the page, you have to put a WOStateStorage element inside of a form in your page's HTML template. Also, all of the objects for which you're storing state in the page must be able to archive and unarchive themselves—this isn't a requirement for state in the application.

Can I Mix Approaches?

You can use different approaches (state in the application or state in the page) for different pages in your application. However, you can't mix approaches on a single page.