Managing State

Originally, the World Wide Web was designed solely for “stateless” applications. An application could display pages and even request information from the user, but it couldn’t keep track of a particular user from one transaction to the next. Such an application is like a person with no long-term memory. Each interaction begins with not so much as a “Haven’t we met somewhere before?” and ends with an implied “Farewell forever!” Stateless applications aren’t well-suited for on-line commerce since it wouldn’t do to lose a customer’s order between the catalog and billing pages. A remedy had to be found.

Given the ingenuity of software developers, not one but several solutions have been advanced. They fall into two basic categories:

- Storing state information on the client’s machine. With each transaction the client passes the state information back to the server, in effect “reminding” the server of the client’s identity and the state information associated with that client. (This approach includes storing *state in the page* and using *cookies*, as explained later in this chapter.)
- Storing state information on the server. With each transaction, the web application locates the state information associated with a request from a particular client. The state information might be stored in memory, in a file on disk, or in a standard database, depending on the application.

Passing state back to the client with every transaction simplifies the accounting associated with state management but is inefficient and can constrain the design of your site. Storing state on the server, on the other hand, requires sophisticated applications that can keep track of per-session information no matter how many users are accessing the application simultaneously. However, without support from your programming environment, storing state on the server is not an attractive option.

As you’ll see in this chapter, WebObjects lets you easily make use of any of these state-storage solutions. For a given application, state management can be as simple as selecting the management strategy you want to use and identifying the information that you want stored on a per-session basis. The WebObjects framework does the rest no matter how many users will be accessing the application simultaneously.

This chapter discusses the issues involved with storing session state and describes the different mechanisms available for storing and managing state information. Some of the topics covered are:

- When Do You Need to Store State?
- Objects and State
- State Management and the Request-Response Loop

- State Storage Strategies
- Controlling Session State
- Controlling Component State

When Do You Need to Store State?

Web applications that store state information are necessarily somewhat more complex than those that don't. State storage can also raise scalability (such as how much physical storage should an application server have for a given number of simultaneous users) and performance issues . Given these considerations, it's clearly best to avoid storing state.

Applications differ widely in their state storage requirements. At one extreme are simple applications that vend read-only pages (company information, specifications for hardware devices, and so on). These traditional World Wide Web applications don't need to store state information. At the other extreme are commercial applications that let users wheel virtual shopping carts from page to page, selecting items for purchase. These applications must keep track of order information on a per-user basis. Considering that a popular site could have scores of simultaneous sessions, these commercial applications must employ a sophisticated means of handling state for each session. Somewhere between these extremes are applications with simple state storage requirements, such as keeping track of the total number of votes on an issue, the number of visitors to the web site, and so on.

Characteristically, WebObjects takes an object-oriented approach to fulfilling any of these state-storage requirements.

Objects and State

WebObjects defines three classes that manage state in an application—WOApplication, WOSession, and WOComponent. (Note: In Java these classes are known as WebApplication, WebSession, and Component.) An application object handles state associated with the application as a whole, session objects handle state associated with a particular user session within the application, and component objects handle state associated with a particular page or component within a session:

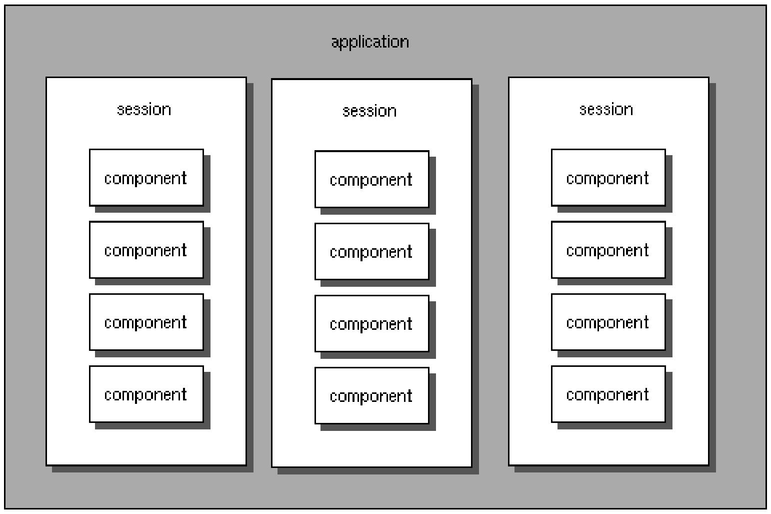


Figure 1. Application, Sessions, and Components

Keep in mind as you read about these classes and about their participation in the request-response loop that the behaviors described are the default ones. Since these classes are public, you are free to change or augment their behaviors either by overriding their methods in scripts, by adding methods through categories, or by creating and using subclasses in their place.

The Application Object and Application State

No matter how many client sessions a WebObjects application is serving, it has one and only one application object. Each page (actually each component, as you learned in previous chapters) knows how to access the application object, so this is the logical place to store data that needs to be shared by all components in all sessions of an application.

Application state is typically stored in the application object's instance variables. For example, if you look at **Application.wos** in the CyberWind example that comes with the WebObjects release, you'll see that the application object keeps track of the number of sessions that have been created, the total number of requests for all sessions, how long the application has been running, and other statistics:

```
id sessionCount;
id requestCount;
id upSince;
id activeSessions;
```

```
- init {
    [super init];
    upSince = [NSCalendarDate date];
    [upSince setCalendarFormat:@"%d %b %Y %H:%M:%S"];
    return self;
}

- createSession {
    activeSessions++;
    return [super createSession];
}
```

Components can access this information in a couple of ways. Using "dot notation" you can bind an attribute of a component's dynamic element to the state stored in the application object:

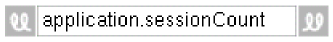


Figure 2. Binding a WOString to a Session Variable

A component can also access application variables through its scripted or compiled code:

```
- isLuckyWinner {
    if ([[self application] sessionCount] == 1000)
        return YES;
    return NO;
}
```

Application state is accessible to any component within the application and, of course, persists for as long as the application is running. If your site runs multiple instances of the same application, application state must be accessible to all instances. In this case, application state might be best stored in a file or database, where application instances could easily access it. This approach is also useful as a safeguard against losing application state (such as the number of visitors to the site) if an application instance crashes.

The Session Object and Session State

A more interesting type of state that web applications can store is the state associated with a user's session. This state might include the selections a user makes from a catalog, the total cost of the selections so far, or the user's billing information. The details of how WebObjects handles session state are discussed in "State Management and the Request-Response Loop" below, but a quick overview will help you understand the scope and duration of session state.

A WebObjects application centralizes session state in objects of the WOSession class (called WebSession, in Java). Each user session has one and only one session object, and a single WebObjects application has as many session objects as it has active user sessions. The session objects segregate data in one session from that in another. There's no way for one session to query or set the data in another. If data needs to be shared across sessions, the application object should be used.

The URLs that make up the requests to a WebObjects application contain an identifier for a particular session within the application. Using this identifier, the application can restore the state corresponding to that session before the request is processed. If the request is that of a user contacting the application for the first time, a new session object is created for that user.

As you can imagine, storing data for each session has the potential of consuming excessive amounts of resources, so WebObjects lets you set a timeout for session objects and lets you terminate them directly.

In summary, session state is only accessible to objects within the same session, and persists only as long as the session object persists.

Component Objects and Component State

In WebObjects, state can also be scoped to a component, such as a dynamically generated page or a reusable component within a page. This state is encapsulated in an object of the WOComponent class (or Component in Java). A component only exists within a session; that is, each session has it own component instances. Component instances are not shared across sessions.

Component state typically includes the data that a page displays, such as a list of choices to present to the user (see **Main.wo** in the CyberWind or DodgeLite examples). Suppose a user requests the page that lists these choices. The component that represents the page needs to initialize itself with the choice data and then return the response page. This completes one transaction. Now suppose the user looks at the list of choices, selects the third car down, and submits a new request. The same component must be present in this second transaction to identify the choice and take the appropriate action. In short, component state often needs to persist from one client-server transaction to the next.

Component state is scoped to a component object, but it only exists within a session. Within a session, a component's state is often set or queried by other components, but a component's state is not visible across user sessions. So, you can think of component state as being a specific type of session state.

Component state persists until the component object is deallocated, which occurs for various reasons, as described later.

State Management and the Request-Response Loop

As you've seen in the previous chapter, "How WebObjects Works," WebObjects manages session state as part of the request-response loop. User-specific state—whether it's associated with an individual component or with the entire session—is kept in a session object that's made available when a request is received and is stored away after the response page is sent. This section takes a closer look at how WebObjects manages session objects.

First Contact: A New Session

A user first contacts a WebObjects application by directing a browser to open a location with a URL of this type (using the CyberWind example for illustration):

```
http://localhost/cgi-bin/WebObjects/Examples/CyberWind
```

When the WebObjects application receives this request (in the application objects's **handleRequest:** method), the application object searches the request URL for a session identifier. Since this is the first request, the URL doesn't include this identifier and so the application creates a new session by sending itself a **createSession** message.

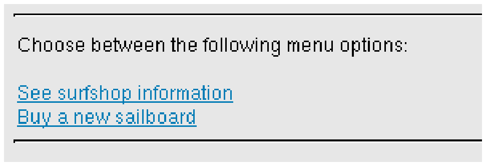
The next step is to find the requested page. Pages are normally accessed through the session object, but since this is a new session it doesn't yet have any pages. So, the application creates a new component for the page called "Main". (Remember, if no page is specified in the request, WebObjects assumes a page name of "Main".)

From this point on, the request processing proceeds as described in the last chapter, with the page taking values from the request, invoking an action method, and finally returning a response page to the application.

The application object saves the response page in the session object and then saves the session object in the application-wide session store (an object of the WOSessionStore class). You'll learn more about the session store in "State Storage Strategies" below, but for now it's sufficient to know that a session store is a repository for the application's session objects when they are not actively engaged in request-response loop processing. Finally, the application returns to the client the HTTP response generated by the response page.

Accessing an Existing Session

The page returned to the user may contain hyperlinks, active images, or submit buttons that let the user make some choice about what happens next. For instance, the CyberWind example gives the user the choice of visiting some on-line surf shops or placing an order.


Choose between the following menu options:

See surfshop information
Buy a new sailboard

Figure 3. Initial Page in a Session

If you view the web browser's source for this HTML page, you'll find that the hyperlinks specify destinations like this:

```
Choose between the following menu options:<BR><BR>
<A href="/cgi-bin/WebObjects/Examples/CyberWind.woa/1933547151826100838039837077512/Main.wo/62793212911/0.0.0/-/ursa">
              See surfshop information</A><BR>
<A href="/cgi-bin/WebObjects/Examples/CyberWind.woa/1933547151826100838039837077512/Main.wo/62793212911/0.1.0/-/ursa">
              Buy a new sailboard</A><BR>
```

Clicking a hyperlink has the effect of submitting one of the URLs above. These URLs encode everything the WebObjects application needs to find the appropriate page within the newly established session. More generally, once a session has been established, an HTTP request to a WebObjects application has this format:

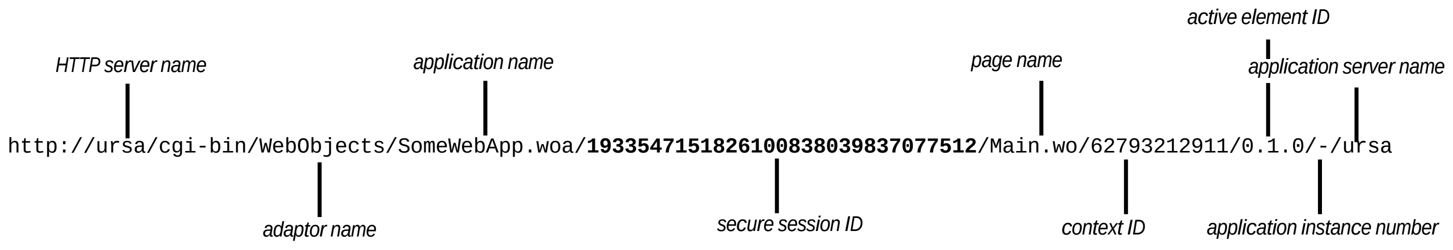

active element ID
HTTP server name
application name
page name
application server name
http://ursa/cgi-bin/WebObjects/SomeWebApp.woa/**1933547151826100838039837077512**/Main.wo/62793212911/0.1.0/-/ursa
adaptor name
secure session ID
context ID
application instance number

Figure 4. Parts of a WebObjects URL

Once a session has been established, URLs to the application contain an embedded session identifier, as you see above. Since sessions are designed to protect the data of one user's transactions from that of another, it's important that session IDs cannot be easily predicted or faked. To this end, WebObjects uses randomly generated 32-digit integers as session IDs. (You can override WOSession's **sessionID** method to implement another security scheme if you'd like.) The URL also specifies the name of the page that should process this request (Main.wo) and provides a context ID to further identify the request-processing page—more on context IDs below.

Using the session ID, the application can retrieve the corresponding session object from the session store, thus maintaining an association between incoming requests and the sessions they belong to:

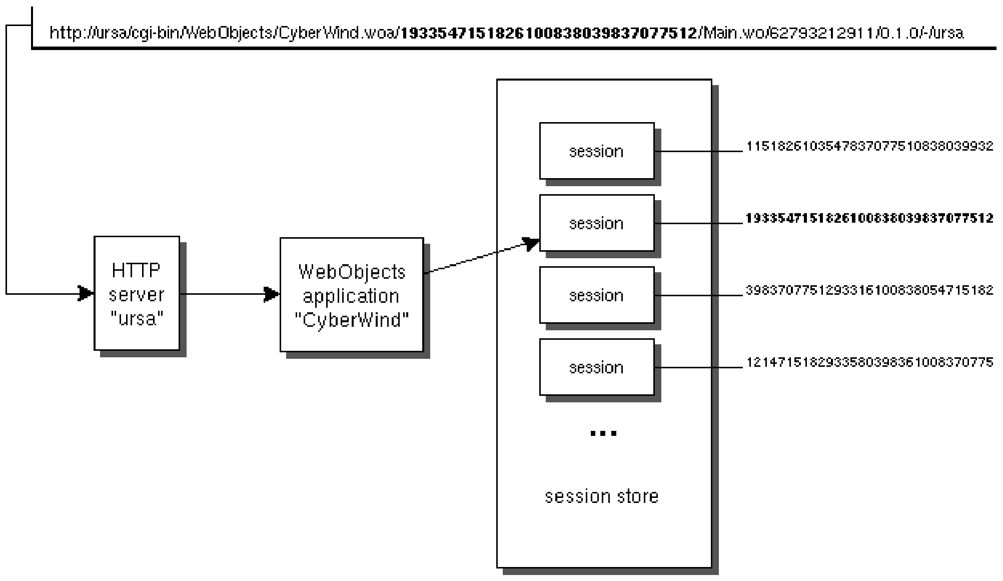


Figure 5. Associating a Request with a Session Object

Next, the session object must locate the page that will process this request. The page name is part of the URL, but the name alone is not enough—this is where the context ID comes in. A context ID is needed to identify a page *as it existed at the end of a particular transaction*. An example will help clarify the need for a context identifier.

Imagine you're accessing a WebObjects application that lets you subscribe to various publications. You navigate from the site's home page to the order page where you select a publication, and then you go to the customer information page and fill in your address. After submitting this information, you navigate back to the home page. Next, you decide to enter a subscription for a friend. You follow the process a second time, selecting a different publication and entering your friend's address.

At this point, within a single session with the subscriptions application, you've accessed the same pages twice, entering different information each time. Let's say that you now realize that you made a mistake in your own address, so you backtrack to that page, change the address, and resubmit the information. It's important that the new address information is submitted to the customer information page as it existed during the first order so that the revised information can be associated with the right publication order.

WebObjects associates a different context ID (again, a randomly generated integer—to maintain security) with each transaction that occurs between a client browser and the WebObjects application. A request to a session includes both the name of request page and a context ID so the session object can locate, from its cache of page instances, the appropriate one to handle the request.

State Storage Strategies

WebObjects gives you the option of storing state in various ways:

- **In the server**. State is maintained in memory within a WebObjects application.
- **In the page**. State is embedded in the HTML page that's returned to the user.
- **In cookies**. State is embedded in name-value pairs ("cookies") in the HTTP header and passed between the client and server. Like "state-in-the-page", cookies store state on the client.
- **In custom stores**. State is stored using a mechanism of your own design.

By default, WebObjects uses the first approach, storing state in the server. Before examining how to use these different strategies, let's take a look at some of their advantages and disadvantages.

Comparison of Storage Options

These options are discussed in more detail in later sections, but seeing an overall comparison might save you time in deciding which options to explore.

Table 1: Comparing Storage Schemes

feature	State in server	State in page	State in cookies	Custom storage
Simplicity	Simplest approach; WebObject's default.	Relatively simple, but can involve design changes to application.	Relatively simple.	More complex.
Security	Secure since state is on server and accessed by encrypted session IDs.	Since data is passed to client, opens possibility that data could be modified by user.	Since data is passed to client, opens possibility that data could be modified by user.	If stored on server, can be as secure or more secure than state-in-server.

Table 1: Comparing Storage Schemes

feature	State in server	State in page	State in cookies	Custom storage
Scalability	Can consume lots of memory. Also, can't use round-robin request handling once state is established in a particular application instance.	More scalable since any application instance can handle a request (because state is bundled with each request). Applications don't grow when new sessions are added.	Not very scalable. Cookie specification limits capacity to 4K bytes per cookie, but some browsers have further limitations.	Depends on design of storage. If filesystem or database used for storage, can scale to accommodate almost any need.
Reliability	Least reliable since if the server crashes, state is lost.	More reliable since a server crash doesn't affect state stored on client.	More reliable since a server crash doesn't affect state stored on client.	Can be extremely reliable if state is stored in server file system or database.
Other		Performance can suffer if lots of data is passed back and forth between client and server. State can become out of sync, especially when using frames (see below).	Client can refuse to accept cookies.	

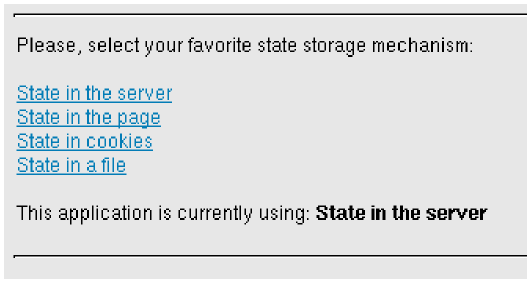
If you know you want to use WebObjects' default server-side storage mechanism, read the "State in the Server" section below and then you can skip to "Controlling Session State" for information about managing the memory requirements of your application. If you want to examine the other storage options in more detail, continue with "A Closer Look at Storage Strategies".

A Closer Look at Storage Strategies

The SessionStores example application that accompanies this documentation demonstrates various ways to store state. The code excerpts used in the following sections come from the SessionStores example, so please refer to the example itself for more details about the implementation.

Note: The SessionStores example was designed to illustrate WebObjects' support for various state storage strategies and so lets you switch between strategies while the application is running. This is not a design you should emulate in your applications—changing storage strategies mid-session can cause errors. For example, imagine an application that stores state in the page during the first half of a session and stores state in cookies for the second. Now suppose that the user backtracks from a page in the second half to one in the first and resubmits the page. The application's strategy and the actual storage mechanism won't match, and state will be lost.

The SessionStores application presents the user with a choice of storage strategies:

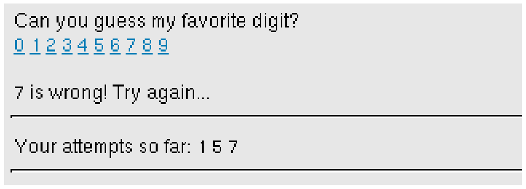

Please, select your favorite state storage mechanism:

State in the server
State in the page
State in cookies
State in a file

This application is currently using: **State in the server**

Figure 6. SessionStores: Storage Choices

Once an initial choice has been made, the application plays a guessing game with the user:


Can you guess my favorite digit?
0 1 2 3 4 5 6 7 8 9

7 is wrong! Try again...

Your attempts so far: 1 5 7

Figure 7. SessionStores: Guessing Game

As you can see, the application keeps track of a user's previous guesses within a session—this, in part, is the state that must be stored from transaction to transaction.

This application switches between storage strategies through the facilities of the WOApplication and WOSessionStore classes. WOApplication declares the **setSessionStore:** method that lets you switch between strategies, and WOSessionStore declares the following methods to create specific types of session stores:

- serverSessionStore
- pageSessionStore
- cookieSessionStoreWithDistributionDomain:secure:

In the SessionStores example, the **setStateStorageStrategy** method demonstrates how these methods work together to set the application's storage type (from

StoreSwitch.wos). When the user makes a choice from the strategy list, the **setStateStorageStrategy** method is invoked and sets the desired strategy:

```
- setStateStorageStrategy {
    id sessionStore;
    id strategyIndex;

    ...
    // Code to determine the value of strategyIndex
    // which indicates which choice the user has made
    ...

    // Set the state storage strategy
    if ( strategyIndex == 0 ) {
        sessionStore = [WOSessionStore serverSessionStore];
    } else if ( strategyIndex == 1 ) {
        sessionStore = [WOSessionStore pageSessionStore];
    } else if ( strategyIndex == 2 ) {
        sessionStore = [WOSessionStore
            cookieSessionStoreWithDistributionDomain:@”” secure:NO];
    } else if ( strategyIndex == 3 ) {
        // Use a custom session store
        sessionStore = [[[FileSessionStore alloc] init] autorelease];
    }
    [[self application] setSessionStore:sessionStore];
    ...

    return [[self application] pageWithName:@"Pages/Guess"];
}
```

(Notice too that this application lets the user choose to store state in the file system using the custom FileSessionStore class. We’ll examine this approach in “Custom State Storage Options” below.)

Normally, an application chooses just one storage strategy for the duration of its execution and so establishes that strategy in the **init** method of the **Application.wos** file.

State in the Server

Storing state in memory on the application server is WebObjects default behavior. As you can see from the SessionStores example, the server session store object is accessible from the WOSessionStore class:

```
id sessionStore = [WOSessionStore serverSessionStore];
[[self application] setSessionStore:sessionStore];
```

In later sections, we'll look at the principal ways of controlling the amount of memory that this state storage mechanism consumes:

- Setting session timeouts (see "Managing Session Resources")
- Setting the size of the page cache (see "Server-Side Page Caching")
- Page uniquing by implement **pageWithName:** in the session object (see "pageWithName: and Page Caching")

One consequence of storing state in memory should be emphasized: Once state for a session is established in a particular application instance, all subsequent requests in that session must return to that instance. WebObjects handles this automatically by including the application instance number in the URL, as illustrated in Figure 2: "Parts of a WebObjects URL".

However, at a popular web site it may be desirable to have multiple application instances (perhaps running on different physical machines) service incoming requests. As long as the application is stateless—or the state is stored outside the application (as with any of the other techniques described here)—the WebObjects adaptor can route requests to any available application. However, if session state is stored in memory, a request must return to the application that stores that state. (See *Serving WebObjects* for more information about running multiple application instances.)

State in the Page

The HTML specification defines an input element of type "hidden" that's commonly used to pass state information back and forth between the client and server. The hidden field simply contains text that is not displayed in the user's browser. For example using state in the page, the HTML source for the Guess page of the SessionStores example would look something like this:

```
<FORM METHOD=Post ACTION=someAction>
    Can you guess my favorite digit?<BR>
    <SELECT NAME="guesses">
        <OPTION>1
        <OPTION>2
        ...
        <OPTION>9
    </SELECT>
    <INPUT TYPE=”hidden” NAME=”hiddenState” VALUE=”previousGuesses”>
    <INPUT TYPE=”submit” VALUE="Guess">
</FORM>
```

The hidden field carries the record of the user’s previous guesses back and forth between the client and server for the duration of the game.

Through its page session store and WOStateStorage dynamic element, WebObjects makes it simple to use the page state storage mechanism, as you’ll soon see. However, there are some limitations inherent in storing state in the page, as we can deduce from the code excerpt above:

- Since state is stored in an input element—which according to the HTML specification must exist within a form element—you must structure your application around forms. If you want session state to be available at any point in the application, each page of the application must have a form, and that form must contain a hidden field (or in the case of WebObjects, a WOStateStorage element, as discussed later).
- Each page carries a record of the state existing at the time of its creation, so backtracking can make the page state and the actual state disagree. For example, if the user make five guesses in the SessionStores example, backtracks two pages, and submits another guess, the application will claim that four guesses were made when the actual number is six.
- 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 a page has multiple forms, you must include the page state data in each form. If a form lacking this data is submitted, the application will no longer have the state information it needs.

A WebObjects application can store state in the page by establishing the page session store as the application’s state storage mechanism and by structuring its pages so that they each contain an HTML form and a WOStateStorage element.

You generally set the session storage type in the **init** method of the application script (**Application.wos**):

```
- init {
    [super init];
    [self setSessionStore:[WOSessionStore pageSessionStore]];
    return self;
}
```

Next, you must add a form to each page of the application and place a WOStateStorage object within the form. For example, the HTML template of a page might look like this:

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

The declarations file declares that the State element is a WOStateStorage dynamic element:

```
STATE: WOStateStorage{ size = 500 };
```

(With WebObjects Builder, you embed a WOStateStorage element in a page by dragging a Custom element from the Palettes panel into the component window and then using the Inspector panel to specify that the type of the element is “WOStateStorage”.)

When you run the application, WebObjects stores session state in the HTML page at the location specified by the WOStateStorage element. What happens is this: When WebObjects generates a page to return to the user, it packages the session state by archiving the session object—and consequently, all the component objects that it contains—into an NSData object. The NSData object is then asked for its ASCII representation, which is written into the HTML page as hidden fields.

WOStateStorage’s **size** attributes specifies the maximum size of each of these hidden fields (500 bytes in the example above). WebObjects writes as many hidden fields as necessary to accommodate the state data. The **size** attribute is provided since browsers differ in the amount of text that they allow within a single hidden field. Most browsers have no problem with the default value of 1000 bytes.

When the user submits the HTML page to the server, the process is reversed. The application's page session store restores the session state by recombining the ASCII data it finds in the hidden fields into the original ASCII archive, converting the ASCII archive to its binary, NSData, representation, and then unarchiving the session object and its contents from the NSData object. (See the class specification for NSArchiver in the *Foundation Framework Reference* for more information on archiving.)

One consequence of storing state in the page is that only objects that know how to archive themselves can be stored. For scripted objects, WebObjects provides a default archiving implementation that will archive data stored in the object's instance variables. For compiled objects, on the other hand, you have to implement the archiving methods yourself, as described in "Storing State for Custom Objects".

State in Cookies

A "cookie" is another way that a web application can store state information in the client machine. Instead of being part of the HTML page as with the state-in-the-page mechanism, a cookie is passed as part of the HTTP header information. The syntax for the cookie header line is:

```
Set-Cookie: NAME=VALUE; expires=DATE; domain=DOMAIN_NAME; path=PATH; secure
```

The NAME=VALUE association is the only required field. It holds the cookie's data and the name by which it can be accessed. The other fields are optional and set limitations on when the data will be passed from the client back to the server:

expires	The date after which the cookie is no longer valid. Once a cookie expires, the client will no longer return it to the server, and client is free to delete it.
domain	The Internet domain name for which the cookie is valid. For example, if the specified domain is **apple.com** for a given cookie, that cookie will be returned along with a request to any host whose domain ends with **apple.com** (for example, **www.apple.com**)—assuming the URL is within the directories specified by **path**.
path	The directories within a given domain for which this cookie is valid. For example, if a cookie has a domain of **www.apple.com** and a path of **/devDoc**, the client will return the cookie to the server for any request that begins with **http://www.apple.com/devDoc...**
secure	Specifies that the cookie can only be passed using a secure communications channel, such as SHTTP.

See **http://www.netscape.com/newsref/std/cookie_spec.html** for a complete description of cookies.

WebObjects makes it simple to use cookies as a state storage mechanism. As you might expect, you generally set the application's session storage type in the **init** method of the application script:

```
- init {
    [super init];
    [self setSessionStore:
        [WOSessionStore cookieSessionStoreWithDomain:@”” secure:NO]];
    return self;
}
```

In this case, we set the domain to the empty string so that cookies that this application sends to the user are valid for all domains. We also turn off the requirement for a secure communications channel. Note that the cookie store API doesn't allow for a path argument. WebObjects automatically restricts the path so that cookies that an application produces are only valid within the application directory. For example, if you set the SessionStores application to use a cookie session store, the client only returns a cookie if the request URLs have this prefix:

```
/cgi-bin/WebObjects/Examples/SessionStores.woa/
```

Once the cookie session store has been established as the application's state storage mechanism, WebObjects does the rest. Just as with storing state in the page, WebObjects packages the session state by archiving the session object (and all the component objects that it contains) into an NSData object. The NSData object is then asked for its ASCII representation. WebObjects pairs this data with names it generates and creates the Set-Cookie headers of the response page.

The process is reversed when a user submits a request containing cookies. The ASCII archive from the Set-Cookie headers is converted to its binary, NSData, representation. The session object and the components it contains are then unarchived from the NSData object, thus restoring the session state.

One of the big advantage of using cookies over state in the page is that cookie storage does not require your application to be designed around forms. As you recall, storing state in the page implied using hidden field elements, which must be located in HTML forms. Cookies, however, are stored in the HTTP header so are independent of the HTML elements in the page. With a cookie session store you could, for example, let users navigate from page to by using hyperlinks rather than by submitting forms. In addition (and for similar reasons), storing state in cookies works better with frames than does storing state in the page.

You should be aware, however, that the cookie mechanism has a size restriction that limits its usefulness. Currently, cookie data is passed from the HTTP server to the WebObjects application either through environment variables that typically are limited to 8K bytes or through a server's own API that in some cases is even more restrictive. We recommend that cookie state data (that is the ASCII representation of the state data) be kept to 2K bytes or less. Given these limitations, cookies can be best used for such things as storing keys used to fetch information from a database.

Custom State Storage Options

WebObjects provides direct support for storing state in the application, in the page, and in cookies. In addition, you can implement your own state storage—for example, you might want to store state in a file or database. The SessionStores application provides an example of a state storage mechanism that uses the filesystem. Let's take a look at how it's done.

In WebObjects, an application saves and restores sessions by sending the session store object these messages:

- saveSession:
- restoreSession

This is the minimum interface that a custom session store must present to the application object. In the SessionStores example, the custom storage class FileSessionStore presents this interface:

```
@interface FileSessionStore:NSObject  {
    id archiveDirectory;
}
- init;
- archiveFileForSessionID:aSessionID;
- archiveForSessionID:aSessionID;
- restoreSession;
- saveSession:aSession;
@end
```

These methods have the following implementation:

```
@implementation FileSessionStore

- init {
    self = [super init];
    archiveDirectory = [WOApp pathForResourceNamed:@"SessionArchives" ofType:nil];
    return self;
}

- archiveFileForSessionID:aSessionID {
    return [NSString stringWithFormat:@"%@/%@", archiveDirectory, aSessionID];
}

- archiveForSessionID:aSessionID {
    id archiveFile = [self archiveFileForSessionID:aSessionID];
    return [NSData dataWithContentsOfFile:archiveFile];
}

- restoreSession {

    id request = [[WOApp context] request];
    id archivedSession;
    id restoredSession;

    // Allow requests in this session to go to any application instance.
    [[WOApp context] setDistributionEnabled:YES];

    // Get archived session (as an NSData object)
    archivedSession = [self archiveForSessionID:[request sessionID]];

    // Unarchive session
    restoredSession = [NSUnarchiver unarchiveObjectWithData:archivedSession];

    return restoredSession;
}

- saveSession:aSession {
    id request = [[WOApp context] request];

    // Store data corresponding to session only if necessary.
    if (![aSession isTerminating] && ![request isFromClientComponent]) {
        id sessionData = [NSArchiver archivedDataWithRootObject:aSession];
        id sessionFilePath = [self archiveFileForSessionID:[aSession sessionID]];

        [sessionData writeToFile:sessionFilePath atomically:YES];
    }
}

@end
```

As you can see, when the FileSessionStore receives a **saveSession:** message, it checks to see if the session object needs to be archived, and if so, it asks NSArchiver to create a binary archive of the session object and all of the components it contains. It then invokes its own **archiveFileForSessionID:** to determine the path for the archive file. Finally, it writes the data to the file. Notice that the session data is written to a file whose name is the session ID itself.

FileSessionStore **restoreSession** is responsible for restoring the state for a particular session. An interesting point in the **restoreSession** method implementation is the **setDistributionEnabled:** message to the application object. By enabling distribution, you let any instance of the application process handle a request. (See *Serving WebObjects* for information on using multiple application instances as a means of load balancing.) More specifically, if distribution is enabled, the application instance number is not appended to the response URL. Since session state is store in the file system and not in the application's memory, it's possible for any application instance to handle any request.

Storing State for Custom Objects

When state is stored in the server, the objects that hold state are kept intact in memory between transactions. In contrast, when state is stored in the page, in cookies, or in the filesystem, objects are asked to archive themselves before being put into storage. The objects that are part of the WebObjects and Foundation frameworks know how to archive themselves, so require no effort on your part. But if your application has custom classes that need to store state, these classes must know how to archive and unarchive themselves. How you implement archiving for custom classes depends on whether your application makes use of the Enterprise Objects framework and its EOEditingContext class.

Using EOEditingContext to Archive Custom Objects

In an Enterprise Objects application, an EOEditingContext manages a graph of enterprise objects which represent records fetched from a database. You send messages to the EOEditingContext to fetch objects from the database, insert or delete objects, and save the data from the changed objects back to the database. (See the *Enterprise Objects Framework Developer's Guide* for more information.)

In WebObjects, applications that use the Enterprise Objects framework must enlist the help of the EOEditingContext to archive enterprise objects. The primary reason is so that the EOEditingContext can keep track, from one transaction to the next, of the objects it is designed to manage. But using an

EOEditingContext for archiving also benefits your application in these other ways:

- During archiving, an EOEditingContext stores only as much information about its enterprise objects as is needed to reconstitute the object graph at a later time. For example, unmodified objects are stored as simple references that will allow the EOEditingContext to recreate the object from the database at a later time. Thus, your application can store state very efficiently by letting an EOEditingContext archive your enterprise objects.
- During unarchiving, an EOEditingContext can recreate individual objects in the graph only as they are needed by the application. This approach can significantly improve an application's perceived performance.

An enterprise object (like any other object that uses the OpenStep archiving scheme) makes itself available for archiving by declaring that it conforms to the NSCoding protocol and by implementing the protocol's two methods, **encodeWithCoder:** and **initWithCoder:**. It implements these methods like this:

```
- (void)encodeWithCoder:(NSCoder *)aCoder {
    [EOEditingContext encodeObject:self withCoder:aCoder];
}

- (id)initWithCoder:(NSCoder *)aDecoder {
    [EOEditingContext initObject:self withCoder:aDecoder];
    return self;
}
```

The enterprise object simply passes on responsibility for archiving and unarchiving itself to the EOEditingContext class, by invoking the **encodeObject:withCoder:** and **initObject:withCoder:** class methods and passing a reference to itself (**self**) as one of the arguments. The EOEditingContext takes care of the rest. (See the EOEditingContext class reference for more information.)

Using the NSCoding Protocol to Archive Custom Objects

Custom classes that can't take advantage of an EOEditingContext for archiving must take a different approach. These 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 message 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 {
    [coder encodeObject:carID];
    [coder encodeObject:colorID];
    [coder encodeObject:colorPicture];
    [coder encodeObject:packagesIDs];
    [coder encodeObject:downPayment];
    [coder encodeObject:leaseTerm];
}
- initWithCoder:(NSCoder *)coder {
    self = [super init];
    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;
}
```

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

Controlling Session State

Maintaining state in memory on the server has the potential of consuming considerable resources, so WebObjects provides a number of mechanisms to control session and page caching. This section takes a closer look at how you store, access, and manage session-wide and component state.

Creating and Accessing Session State

You typically store session state as instance variables in your application's session object. Using WebScript, you add these variables to a session script file named **Session.wos**, which is located in the application directory (for example, **MyApp.woa/Session.wos**). A less common but equally effective alternative is to add these instance variables to a compiled session object. It's also possible to store session state within a special dictionary provided by the session object, as we'll see shortly.

Session state is directly accessible to any component within the application. One way to access a session variable is to bind it to an attribute of a dynamic element. For instance, if you open the Visitors example application in WebObjects Builder, you'll see that the value of a WOString dynamic element is bound to the session variable that stores the elapsed time since the session began:

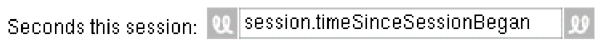


Figure 8. Binding a WOString to a Session Variable

Another way to access session variables is from a component's scripted or compiled code:

```
elapsedTime = [[self session] timeSinceSessionBegan];
```

(A component inherits from the WOComponent class, which defines convenience methods such as **session** and **application**, making it easy for a component to access these other objects simply by sending a message to **self**.)

The WOSession class also provides a dictionary where state can be stored by associating it with a key. WOSession's **setObject:forKey:** and **objectForKey** methods give access to this dictionary. For an example of when this session dictionary might be useful, consider a web site that collects users' preferences about movies. At this web site, users work their way through page after page of movie listings, selecting their favorite movie on each page. A "Choices" component at the bottom of each page displays the favorites that have been picked so far in the user's session. The Choices component is a general purpose reusable component that might be found in various applications.

The designer of the Choices component decided to store the session-wide list in the session dictionary:

```
[[self session] setObject:usersChoiceArray forKey:@"Choices"];
```

By storing the information in the session dictionary rather than in a discrete session instance variable, this component can be added to any application without requiring code changes such as adding variables to the session object.

This approach works well until you have multiple instances of a reusable component in the same page. For example, what if users were asked to pick their most *and least* favorite movies from each list, with the results being displayed in two different Choices components in each page. In this case, each component would have to store its data under a separate key, such as "BestChoices" and "WorstChoices".

A more general solution to the problem of storing state when there are multiple instances of a reusable component is to store the state under unique keys in the session dictionary. One way to create such keys is to concatenate the component's name, context ID, and element IDs:

```
id componentName;
id context;
id contextID;
id elementID;
id uniqueKey;

context = [self context];
componentName = [[context component] name];
contextID = [context contextID];
elementID = [context elementID];
uniqueKey = [NSString stringWithFormat:@"%@-%@-%@", componentName, contextID,
    elementID];
[[self session] setObject:someState forKey:uniqueKey];
```

Since, for a given context, each element in a page has its own element ID, combining the context and element IDs yields a unique key. We added the component name to the key for readability during debugging.

Managing Session Resources

If you choose to store state in memory on the application server, memory usage can become an issue. (See "State Storage Strategies" for alternative ways of storing this information.) Take care that your application only stores state for active sessions and stores the smallest amount of state possible. WOSession lets you control these factors by providing a timeout mechanism for inactive sessions and by providing a way to specify exactly what state to store between transactions.

Setting Session Time-out

By assigning a timeout value to a session, you can ensure that the session will be deallocated after a specific period of inactivity. WOSession's **setTimeOut:** method lets you set this period and **timeOut** returns it.

Here's how the session time-out works: After a transaction, WebObjects associates a timer with the session object that was involved in the transaction and then puts the session object into the session store. The timer is set to the value returned by the session object's **timeOut** method. If the timer goes off before the session is asked to handle another transaction, the session and its resources are deallocated. A user submitting a request to a session that has timed out receives an error message:

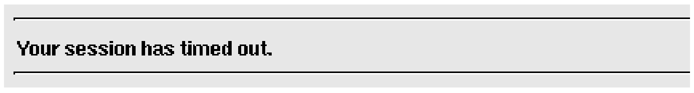

Your session has timed out.

Figure 9. Session Time-out Error Message

By default, a session object's timeout value is so large that sessions effectively never time out. You should set the session timeout for your application to the shortest period that seems reasonable. For example, to set the timeout to ten minutes, you could send this **setTimeOut:** message in your application's **Session.wos** script:

```
- init {
    [super init];
    [self setTimeOut:600];
    return self;
}
```

Note that the argument to **setTimeOut:** is interpreted as a number of seconds.

At times, a user's choice signals the end of a session (such as when the Yes button is clicked in response to the query, "Do you really want to leave the Intergalactic Web Mall?"). If you are sure a session has ended, you can send a **terminate** message to the session object, marking it (and the resources it holds) for release.

A session marked for release won't actually be released until the end of the current request-response loop. Other objects may need to know whether a particular request-response loop is their last, so they can close files or do other clean up. They can learn their fate by sending the session object an **isTerminating** message.

Using awake and sleep

Another strategy for managing session state is to create it at the beginning of the request-response loop and then release at the end. WOSession's **awake** and **sleep** methods provide the hooks you need to implement this strategy. A session object receives an **awake** message at the beginning of the request-response loop (where you could reinitialize the session state) and a **sleep** message at the end (where you could release it).

Controlling Component State

As mentioned previously, a WOComponent can represent a portion of a page (as with reusable components) or a complete page. State associated with the component is generally stored in the component object's instance variables and so persists for the life of the object. Component objects exist within a particular session and are stored along with the session object between each cycle of the request-response loop. Since a user can visit many pages during a session, managing component state can be crucial to reducing your application's storage requirements.

Creating and Accessing Component State

Common uses for component state include storing:

- A list of items that a user can choose from within a particular page
- The user's selection from that list
- Information that the user enters in a form
- Default values for a component's attributes

A simple example of component state can be seen in the first page of the DodgeLite example application, which list models, prices, and types of vehicles for the user to choose from:

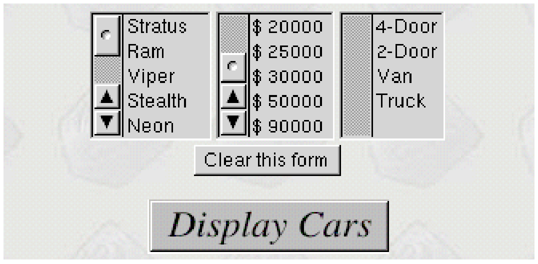


Figure 10. First Page of the DodgeLite Example

The script for this component (**Main.wos**) declares instance variables for the values displayed in the browser and for the user's selection from the browsers. Before the page can be sent to the user, the instance variables that hold the values to be displayed (**model**, **price**, **type**) are initialized:

```
id models, model, selectedModels;
id prices, price, selectedPrices;
id types, type, selectedTypes;

- init
```

```
{
    [super init];
    models = [[WOApp modelsDict] allValues];
    types = [[WOApp typesDict] allValues];
    prices = [WOApp prices];
    return self;
}
```

(The **selectedModels**, **selectedPrices**, and **selectedTypes** instance variables are bound to the **selections** attributes of the three WOBrowsers and so will contain the user's selections when the Display Cars button is clicked.)

When a user starts a session of the DodgeLite application, the Main component's **init** method is invoked, initializing the component's instance variables from data accessed through the application object. From this point on (subject to conditions discussed below), the Main component and its instance variables become part of the state stored for that user's session of the DodgeLite application. When the session is released, the component is also released. However, there are other techniques that allow you to control resource allocation on a component basis, as you'll see in the next section.

As with the session state, a component's state is accessible to other objects within the same session. As the result of a user's action, for example, it's quite common for one component to create the component for the next page and set its state. Looking again at the DodgeLite application, consider what happens when the user makes a selection in the first page and clicks Display Cars. The **displayCars** method in the Main component is invoked:

```
- displayCars
{
    id selectedCarsPage = [[self application] pageWithName:@"SelectedCars"];

    ...

    [selectedCarsPage setModels:selectedModels];
    [selectedCarsPage setTypes:selectedTypes];
    [selectedCarsPage setPrices:selectedPrices];
    ...
    [selectedCarsPage fetchSelectedCars];

    return selectedCarsPage;
}
```

The new component is created by sending a **pageWithName:** message to the WOApplication object, and then a series of messages is sent to this new object to set its state before the object is returned as the response page.

Managing Component Resources

Typically, page caching occurs on both the client machine and on the WebObjects application server. WOApplication provides methods to control caching on either end of a web connection. This section discusses server-side caching and the section "Client-Side Page Caching" looks at the consequences of page caching on the client.

Techniques for controlling component resources include:

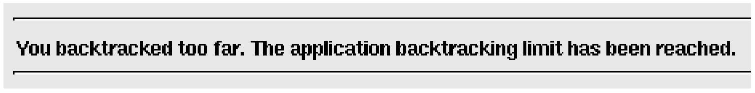
- Adjusting the page cache size
- Using **awake** and **sleep** to initial and release resources
- Controlling page instantiation by implementing **pageWithName:**

Adjusting the Page Cache Size

As noted earlier, except for the first request, a request to a WebObjects application contains a session ID, page name, and context ID. The application uses this information to ask the appropriate session object for the page identified by the name and context ID. As long as the page is still in the cache, it can be retrieved and enlisted in handling the request.

By default, a WebObjects application caches the last 30 pages that a user has visited within a session.You can change the size of the cache using WOApplication's **setPageCacheSize:** method and retrieve the cache size with the **pageCacheSize** method. Within each session, new pages are added to the cache until the cache size limit is reached. Thereafter, for each new page added to the cache, the cached page object representing the least recently visited page is released.

To reduce the resource requirements for an application, you could set the page cache to a smaller number. However, doing so increases the possibility that a request could address a page that is no longer in the cache. For example, if you set the page cache size to four, a user could backtrack five pages to a order form, make some changes, and resubmit the form. The result would be an error page like this:



You backtracked too far. The application backtracking limit has been reached.

Figure 11. Backtracking Error Message

To keep users from encountering this error, your application should maintain a moderate sized cache of pages. (Another strategy is to limit the number of identical page instances that your application creates; see “pageWithName: and Page Caching” for one way to do this.) The default cache size of 30 pages is a reasonable value that protects users from reaching the backtracking limit under normal conditions; however, you can adjust the limit to any positive value you like or even zero.

Setting the page cache size to zero has two effects. As expected, it disables page caching. But furthermore, it signals to WebObjects that you intend to provide for component state persistence rather than rely on WebObjects’ inherent support. Thus, if you set the cache size to zero, no error page is generated if a request addresses a page that can’t be found in the cache. Instead, WebObjects creates a new page by sending the application object a **pageWithName:** message. Since with this model pages do not persist from one transaction to the next, you assume responsibility for maintaining any needed component state. For this reason, it’s rarely advisable to turn off page caching.

Using awake and sleep

Another way to control the amount of component state that’s maintained between transactions is to make use of WOComponent’s **awake** and **sleep** methods. Unlike the component’s **init** method that’s invoked just once in the life of the component, a component’s **awake** and **sleep** methods are invoked at the beginning and end of any request-response loop that involves the component.

By moving a component’s variable initialization routines from its **init** method to its **awake** method and implementing a **sleep** method to release those variables, you can reduce the space requirements for storing a component. For example, the code for DodgeLite’s Main component that we looked at earlier could be changed to:

```
id models, model, selectedModels;
id prices, price, selectedPrices;
id types, type, selectedTypes;

- awake {
    models = [[WOApp modelsDict] allValues];
    types = [[WOApp typesDict] allValues];
    prices = [WOApp prices];
}



- sleep {
    models = nil;
    types = nil;
    prices = nil;
}
```

Note that in WebScript you set a variable to **nil** to mark it for release; whereas, in Objective-C you send the object a **release** message:

```
- sleep {
    [models release];
    [types release];
    [prices release];
}
```

Of course, what you save in storage by moving variable initialization to the **awake** method is lost in performance since these variables will be reinitialized on each cycle of the request-response loop.

pageWithName: and Page Caching

When the application object receives a **pageWithName:** message, it creates a new component. For example, in the HelloWorld example a user enters a name in the first page (the Main component), clicks Submit, and is presented with a personal greeting on the second page (the Hello component). Clicking the Submit button in the first page invokes the **sayHello** method in the Main component. As part of its implementation **sayHello** sends a **pageWithName:** message to the application object:

```
id visitorName;

- sayHello {
    id nextPage;

    // Create the next page.
    nextPage = [[self application] pageWithName:@"Hello"];

    // Set state in the Hello page
    [nextPage setVisitorName:visitorName];

    // Return the 'Hello' page.
    return nextPage;
}
```

Each time the **sayHello** method is invoked, a new Hello component is created. For example, if the user backtracks to the main page and clicks the Submit button again, another Hello page is created. It's unlikely this duplication of components will be a problem for the HelloWorld application, since users quickly tire of its charms. But, depending on design, some applications may benefit by modifying the operation of **pageWithName:** so that an existing component can be reused.

If you want to extend WebObjects' page caching mechanism to include pages returned by **pageWithName:**, you must implement your own solution. Fortunately, it's easy. One approach is to have the session maintain a dictionary that maps page names to page objects. Here's the code you would add to an application's **Session.wos** file:

```
id pageDictionary;

- init {
    [super init];
    pageDictionary = [NSMutableDictionary dictionary];
    return self;
}

- pageWithName:aName {
    id aPage = [pageDictionary objectForKey:aName];

    if (!aPage) {
        aPage = [[self application] pageWithName:aName];
        [pageDictionary setObject:aPage forKey:aName];
    }
    return aPage;
}
```

Note that we implement **pageWithName:** in the session object since we want to cache these pages on a per-session basis. (Overriding the method in the application object would cache pages on a per-application basis.) Since the **pageWithName:** method that we want to use now resides in the session object, one line in the **sayHello** method has to change (change in bold):

```
- sayHello {
    id nextPage;

    nextPage = [[self session] pageWithName:@"Hello"];
    [nextPage setVisitorName:visitorName];
    return nextPage;
}
```

Client-Side Page Caching

When accessing a web page, the user's browser associates the URL with the HTML page it downloads from the server and stores this information on the user's machine. If the browser is asked to display the URL again at a later date, it fetches the cached page rather than emitting another request. In many cases, this short-circuit is desirable since it reduces network traffic and increases a web site's perceived responsiveness.

Sometimes, however, you need to make sure the user is seeing the most up-to-date information, so you must disable client-side caching. WOApplication provides the **setPageRefreshOnBacktrackEnabled:** method for this purpose. In general, you send this message in the **init** method of your application script (**Application.wos**):

```
- init {
    [super init];
    [self setPageRefreshOnBacktrackEnabled:YES];
    return self;
}
```

The **setPageRefreshOnBacktrackEnabled:** method adds a header to the HTTP response that sets the expiration date for an HTML page to the date and time of the creation of the page. Later, when the browser checks its cache for this page, it finds that the page is no longer valid and so refetches it by resubmitting the request URL to the WebObjects application.

A WebObjects application handles a page-refresh request differently than it would a standard request. When the application determines that the request URL is identical to one it has previously received (that is, the session and context IDs in the request URL are identical to those in a request it has previously received), it simply returns the response page that was associated with this earlier request. The first two steps of a normal request handling loop (value extraction from the request and action invocation) don't occur.

Page Refresh and WODisplayGroup

If you're using a WODisplayGroup object in your application, you must enable page refresh so that the application and the client browser stay in agreement about which objects are being displayed.

A WODisplayGroup holds a set of objects (generally enterprise objects fetched from a database) and provides “batched” access to these objects. For example, if a user submits a query (such as, “Show me the movies released in 1996.”) to a Movies application, a WODisplayGroup might return ten records at a time to the user’s browser. The application would offer controls to let the user display the next and previous batches of ten movie titles. When the user decides to order one of the movies, the WODisplayGroup needs to know which batch the item comes from.

As the user presses the Next Ten Movies or Previous Ten Movies buttons, the WODisplayGroup updates its record of the which ten movies are being displayed. When the user decides to order the second movie in the list, the WODisplayGroup can determine the actual record since it knows which batch is being displayed and which record is number two in that batch. But if the user backtracks to a previous page (with page refresh disabled) and chooses the second record, the WODisplayGroup will erroneously pick the second record from its current batch. By enabling page refresh, the WODisplayGroup is alerted each time the user backtracks and can update its notion of the current batch, eliminating this problem.