Inside the Request-Response Loop

WebObjects applications are event driven, but instead of responding to mouse and keyboard events, they respond to HTTP requests. A WebObjects application receives a request, responds to it, then waits for the next request. The application continues to respond to requests until it terminates. On each cycle of this *request-response loop*, the application stores user input, invokes an method if one is associated with the user's action, and generates a response—usually an HTML page.

Most of your program's activity is a reaction to messages the application sends out during a cycle of the request-response loop. These messages are *hooks* into the loop from which you can invoke your application's custom behavior. This chapter discusses the flow of control inside the request-response loop and the request-response loop hooks.

Starting the Request-Response Loop

This section looks at handling requests in a typical application, beginning where the application itself begins, with the code that sets up the request-response loop. A WebObjects application begins just as a C program does, by calling its **main()** function. In a WebObjects application, **main()** is usually very short. Its job is to set up a WOApplicationAdaptor object and turn over control of the program to it. Generally, **main()** is just a few lines of code:

```
void main(int argc, char *argv[]) {
    NSAutoreleasePool *pool;
    WOApplicationAdaptor *adaptor;
    WOWebScriptApplication *application;

    pool = [[NSAutoreleasePool alloc] init];
    adaptor = [[[WOApplicationAdaptor alloc] init] autorelease];
    application = [[[WOWebScriptApplication alloc]
            initWithArgC:argc argV:argv]
            autorelease];
    [adaptor runWithApplication:application];
    [pool release];
    exit(0);
}
```

The **main()** functions you write should look identical or much the same.

This version of the **main()** function creates an autorelease pool that's used for the automatic deallocation of objects that receive an **autorelease** message. Next, it creates a WOApplicationAdaptor object that handles communication between an HTTP server and the WOWebScriptApplication object, which is created in the next statement.

The **runWithApplication:** method initiates the request-response loop. As shown in Figure 1, in each cycle of the loop the WOApplicationAdaptor receives an incoming request, forwards it to the WOWebScriptApplication object, and returns the outgoing response from the WOWebScriptApplication.

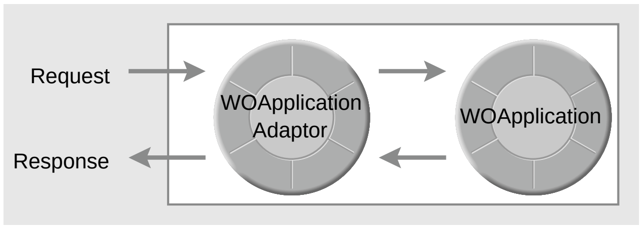


Figure 1. The Request-Response Loop

The last statement in this version of **main()** releases the autorelease pool, which sends release messages to any object that has been added to the pool since the application began.

Most of an application's time is spent in the request-response loop started by **runWithApplication:**, getting and responding to requests. You can participate in request and response handling by taking advantage of *hooks* into the request-response loop. For example, you can determine what page to return based on user input and modify the header lines of a generated HTTP response.

There are three types of hooks:

- *awake methods* that are invoked at the point in the request-response loop just before the receiver begins to participate in the request handling
- *Action methods* that are associated with a particular user action such as clicking a button or hyperlink
- *Request and response handling methods* that are invoked at a particular point in the request-response loop if you implement them:

  willPrepareForRequest:inContext:
  didPrepareForRequest:inContext:
  willGenerateResponse:inContext:
  didGenerateResponse:inContext:

To understand when these methods are invoked and what you can use them for, you need to understand the sequence of events in a cycle of the request-response loop. The following section describes the sequence. Information on using the request-response loop hooks is presented later in this chapter.

Flow of Control in the Request-Response Loop

Each cycle of the request-response loop has three phases:

- Preparing for the request
- Invoking an action
- Generating a response

The following sections describe the sequence of events that occur in each phase.

Preparing for the Request

All requests received by a WebObjects application are associated with one of the application's pages—the *request page*. If the request doesn't explicitly specify a page, the WOWebScriptApplication object associates the request with a page named "Main". During the first phase of the request-response loop, the application finds or creates a component to represent the request page. This component—called the *request component*—is represented with an instance of a custom WOComponentController subclass or a WOWebScriptComponentController.

If the WOWebScriptApplication object has already created a component to represent the request page, it uses that same component. If it hasn't, the WOWebScriptApplication object performs the following steps to create it:

1. It looks in the Objective-C runtime system for a class with the same name as the request page. If it finds a class with the same name, it creates an instance of that class. For example, if the request specifies a request page named "LoginPanel" and a class with the same name is present in the Objective-C runtime system, the WOWebScriptComponentController instantiates a LoginPanel object as the request component.

2. If the WOWebScriptApplication object fails to find a class in the runtime, it creates an instance of WOWebScriptComponentController.

Note: A WOWebScriptApplication object follows this same procedure whenever it's called upon to provide a component.

During the first phase of the request-response loop, the request component assigns user input to variables. This is the basic sequence of events in preparing for a request:

1. The WOWebScriptApplication object restores the values of all session and persistent variables. For more information on how a WebObjects application manages state, see the chapter "Managing State."

2. The WOWebScriptApplication object receives a **willPrepareForRequest:inContext:** message.

3. The request component receives a **prepareForRequest:inContext:** message. **prepareForRequest:inContext:** invokes the request component's **willPrepareForRequest:inContext:** method, stores user input in variables according to the component's declarations file, and then invokes the request component's **didPrepareForRequest:inContext:** method.

4. The WOWebScriptApplication object receives a **didPrepareForRequest:inContext:** message.

Invoking an Action

In the second phase the request-response loop, the request component receives an **invokeActionForRequest: inContext:** message. **invokeActionForRequest: inContext:** determines whether or not the user has triggered an action. If an action has been triggered—for example, if the user clicked a button or a hyperlink—the application invokes the action method that corresponds to what the user did. An action method returns the *response component*—the component responsible for generating an HTTP response. If the user has not triggered an action, the request component is used as the response component.

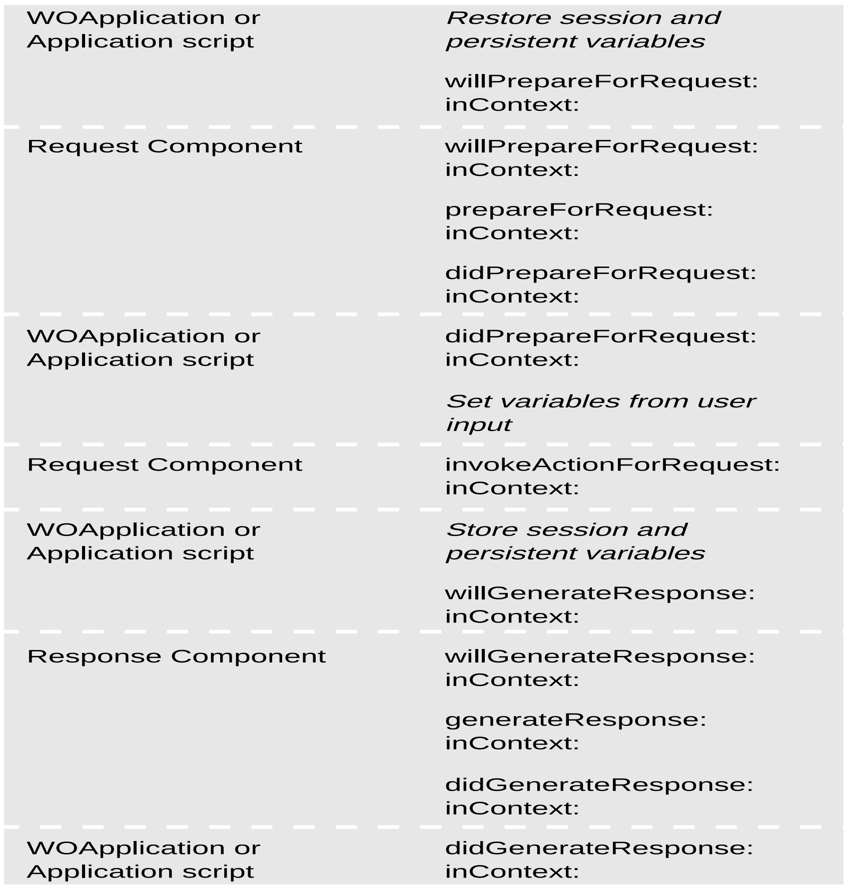
Generating a Response

In the final phase of requst-response loop, the response page generates an HTTP response. Generally, the response contains a dynamically generated HTML page.

This is the basic sequence of events in generating a response:

1. The WOWebScriptApplication object stores the values of all session and persistent variables. Subsequent changes to session and persistent variables will not be preserved in the next cycle of the request-response loop. For more information on how a WebObjects application manages state, see the chapter "Managing State."

2. The WOWebScriptApplication object receives a **willGenerateResponse:inContext:** message.
3. The response component receives a **generateResponse:inContext:** message. **generateResponse:inContext:** invokes the request component's **willGenerateResponse:inContext:** method, generates an HTTP response using the response component's HTML template, and then invokes the response component's **didGenerateResponse:inContext:** method.
4. The WOWebScriptApplication object receives a **didGenerateResponse:inContext:** message.



WOApplication or Application script	*Restore session and persistent variables* willPrepareForRequest: inContext:
Request Component	willPrepareForRequest: inContext: prepareForRequest: inContext: didPrepareForRequest: inContext:
WOApplication or Application script	didPrepareForRequest: inContext: *Set variables from user input*
Request Component	invokeActionForRequest: inContext:
WOApplication or Application script	*Store session and persistent variables* willGenerateResponse: inContext:
Response Component	willGenerateResponse: inContext: generateResponse: inContext: didGenerateResponse: inContext:
WOApplication or Application script	didGenerateResponse: inContext:

Figure 2. The Sequence of Events in One Cycle of the Request-Response Loop

awake Methods

There are two types of **awake** methods: an application **awake** method and a component **awake** method. Both **awake** methods provide an opportunity for the receiver to perform initialization before it participates in request handling. The biggest difference between the two is that an application **awake** method is only invoked once, whereas component **awake** methods can be invoked many times.

Application awake

The application **awake** method is invoked when the application receives its first request and never again. It's common to initialize global variables in an application **awake** method. For example, the **Application.wos** script in the DodgeLite example initializes the global variable **dodgeData** from a file:

```
- awake {
    id filePath = [WOApp pathForResource:@"DodgeData" ofType:@"dict"];
    dodgeData = [NSDictionary dictionaryWithContentsOfFile:filePath];
}
```

See **DocumentRoot/WebObjects/Examples/DodgeLite** for the complete DodgeLite source.

The WOWebScriptApplication class defines the **awake** method, which is invoked when an instance receives its first request from a WOApplicationAdaptor object. WOWebScriptApplication's **awake** implementation invokes the **awake** method defined in the corresponding **Application.wos** if one exists. You can subclass WOWebScriptApplication and override **awake** to perform any necessary initialization. It is more common, however, to implement the **awake** method in an application script.

Because the application **awake** method is invoked only once in an application's lifetime, changing a scripted application **awake** method has no effect on a running WebObjects application that has received its first request. You must restart an application for changes to a scripted application **awake** method to have an effect.

In addition to using the application **awake** method to initialize global variables, you can also use it to configure the application's behavior. For example, you can set a session timeout and enable component caching:

```
// Timeout sessions that have been inactive for more than 2 minutes.
[WOApp setSessionTimeOut:120];
[WOApp setCachingEnabled:YES];
```

Component awake

A component **awake** method is invoked at the point in a cycle of the request-response loop just before the receiver begins to participate in request handling. It's common to implement a component **awake** method that initializes transaction and persistent variables. For example, the **Main.wos** script in the CyberWind application uses **awake** to initialize the **options** transaction variable:

```
- awake {
    options = @("See surfshop information", "Buy a new sailboard");
    return self;
}
```

See **DocumentRoot/WebObjects/Examples/CyberWind** for the complete CyberWind source.

The WOComponentController class—an abstract class that implements basic component behavior—defines the **awake** method. In WOComponentController, **awake**'s implementation does nothing, but you can subclass WOComponentController and override **awake** to perform any necessary initialization. It is equally common, however, to implement the **awake** method in a component script. The WOComponentController subclass WOWebScriptComponentController overrides **awake** to invoke the **awake** method defined in the corresponding script file if one exists.

For a given component, the **awake** method is invoked only once in a cycle of the request-response loop. Furthermore, a component's **awake** method is invoked only in cycles in which the component is participating. Generally, a component participates in a cycle of the request-response loop if:

- It represents the request page—the page associated with the request.
- It represents the response page—the page returned to the server.
- It's nested in either the request or response page.
- It's messaged in any other way during the current cycle.

As an example of the latter, the following method messages the component associated with the "LoginPanel" page:

```
- messageCountString
{
    id loginPage = [WOApp pageWithName:"LoginPanel"];
    id userName = [loginPage userName];
    id messagesCount = [messages count];
    id countString;

    if (messagesCount == 0)
        countString = @"No messages";
    else if (messagesCount == 1)
        countString = @"1 message";
    else
        countString = [NSString stringWithFormat:@"%@ messages", messageCount];

    return [NSString stringWithFormat:@"%@ for %@", countString, userName];
}
```

If the “LoginPanel” component is not already participating in the request-response loop when **messageCountString** is invoked, the **pageWithName** message to WOApp creates the component and sends it an **awake** message.

A component **awake** method is invoked at different points in the request-response loop depending on when the component begins participating. Before an application dispatches a message to a component, the application invokes the component’s **awake** method if the component has not received an **awake** message in the current cycle of the request-response loop. Thus, the **awake** method is invoked before any other method, and won’t be invoked again until the next cycle in which the component participates.

The **awake** method is the best place to initialize transaction and session variables. The advantage of using **awake** to perform this type of initialization is that the variables are guaranteed to be initialized before any other methods are invoked.

Action Methods

An action method is a method that’s associated with a user action. You associate methods with a user action using a dynamic element. For example, WOSubmitButton has an attribute named **action** to which you can assign a method. When the submit button in the corresponding HTML page is clicked, the action method is invoked in the subsequent cycle of the request-response loop. This declaration in the HelloWorld application associates the action method **sayHello** with a submit button:

```
SUBMIT_BUTTON: WOSubmitButton {action = sayHello};
```

Clicking the submit button sends a request to the HelloWorld application, initiating a cycle of the request-response loop in which **sayHello** is invoked.

Note: The WOActiveImage, WOHyperlink, and WOForm dynamic elements can also be used to associate action methods to a user action.

Action methods take no arguments and return a component responsible for generating an HTTP response. For example, the **sayHello** action method is defined as follows:

```
- sayHello
{
    id nextPage = [WOApp pageWithName:@"Hello"];
    [nextPage setNameString:nameString];
    return nextPage;
}
```

As in **sayHello**, most action methods perform page navigation. It is common for action methods to determine the response page based on user input. For example, the following action method returns an error page if the user has entered an invalid part number (stored in the transaction variable **partnumber**) or an inventory summary otherwise.

```
- showPart {
    id errorPage;
    id inventoryPage;

    if ([self isValidPartNumber:partnumber]) {
        errorPage = [WOApp pageWithName:@"Error"];
        [errorPage setErrorMessage:@"Invalid part number."];
        return errorPage;
    }
    inventoryPage = [WOApp pageWithName:@"Inventory"];
    [inventoryPage setPartNumber:partnumber];
    return inventoryPage;
}
```

Action methods don’t have to return a new page. They can instead direct the application to regenerate the request page. When an action method returns **nil**, the application uses the request component as the response component.

Note: Returning **self** in an action method generally has the same effect as returning **nil**. However, there’s a difference when the action method is implemented in a nested component. When a nested component—a component representing only a portion of the request page—returns **self** in an action, the application attempts to use the nested component to generate the response page. Since the component only represents a portion of a page, returning **self** is probably an error. Returning **nil** always has the effect of using the request component—the component representing the whole request page—as the response component. As a result, returning **nil** is considered to be a better practice than returning **self**.

In the Visitors example, the request page is also used as the response page. The WebScript **recordMe** action method records the name of the last visitor and clears the text field:

```
- recordMe
{
    if ([aName length]) {
        [WOApp setLastVisitor:aName];
        [self setAName:@""]; // clear the text field
    }
}
```

Request and Response Handling Methods

WebObjects defines four request and response handling methods that are invoked at particular points in the request-response loop if you implement them:

- willPrepareForRequest:inContext:
- didPrepareForRequest:inContext:
- willGenerateResponse:inContext:
- didGenerateRequest:inContext:

As with **awake** methods, the request and response handling methods can be implemented for an application and its components. Both versions of a request-handling method work identically, and can be used for identical purposes. You choose to implement a method for the application or for a component based on which is more appropriate for the behavior you need to provide. Generally, you implement request and response handling methods in the application when the request handling behavior should be invoked for every request. You implement request and response handling methods in a component when the request handling behavior should be invoked only for a particular page.

The WOWebScriptApplication class defines each of the request and response handling methods. Its implementations invoke the corresponding method defined in the corresponding **Application.wos** if one exists. For example, WOWebScriptApplication's **willPrepareForRequest:inContext:** invokes the **willPrepareForRequest:inContext:** method in the corresponding **Application.wos** if it exists. You can subclass WOWebScriptApplication and override the request and response handling methods, but it is more common to implement them in an application script.

WOComponentController defines each of the request and response handling methods as well, but WOComponentController's implementations do nothing. You can subclass WOComponentController and provide your own implementations, but it is equally common to implement a component's request and response handling methods in a component script. The WOComponentController subclass WOWebScriptComponentController overrides the request and response handling methods to invoke them in the corresponding component script if they exist.

The remainder of this section discusses some of the common uses of the request and response handling methods.

willPrepareForRequest:inContext:

This method is invoked before the application stores user input. It is common to use this method to access request and context information. For example, the following implementation of **willPrepareForRequest:inContext:** records the kinds of browsers—user agents—from which requests are made:

```
- willPrepareForRequest:request inContext:context {
    id requestHeaders = [request headers];
    id userAgent = [requestHeaders objectForKey:@"user-agent"];
    [WOApp recordUserAgent:userAgent];
    return nil;
}
```

The first argument to **willPrepareForRequest:inContext:** is a WORequest object. A WORequest object encapsulates information from an HTTP request such as the method line, request headers, URL, and form values. The second argument is a WOContext object. A WOContext object contains application specific information such as the path to the request component's directory, the version of WebObjects that's running, the name of the application, and the name of the request page.

You can also use **willPrepareForRequest:inContext:** to substitute a different object for the request page. If this method returns a non-**nil** value, **prepareForRequest: inContext:** is sent to the substituted object. If you implement this method in the application and return a non-**nil** value, the substitute object will receive a **willPrepareForRequest:inContext:** message as well.

didPrepareForRequest:inContext:

This method is invoked after the request page has processed user input. You can use this method to substitute a different object for the request page before an action method is invoked. If this method returns a-non **nil** value,

invokeActionForRequest:inContext: is sent to the substituted object. The use of this method to substitute an object at this point in the cycle is very rare.

willGenerateResponse:inContext:

This method is invoked before the application generates HTML for the response. You can use this method to substitute a different object for the response page before a response is composed. If this method returns a non-**nil** value, **generateResponse: inContext:** is sent to the substituted object. If you implement this method in the application and return a non-**nil** value, the substitute object will also receive a **willGenerateResponse:inContext:** message.

The most common use of this method is to implement security features. You can use **willGenerateResponse:inContext:** to return a login page if the user has not yet provided valid login information. The advantage of using **willGenerateResponse: inContext:** over any other hook is that this method is always invoked before generating a response.

You can't rely upon an action to implement the same functionality. By specifying a page in a URL, a user can attempt to access any page in an application without invoking an action. For example, you can access the second page of HelloWorld without invoking the **sayHello** action by opening the URL:

```
http://serverhost/cgi-bin/WebObjects/Examples/HelloWorld/Hello.wo/
```

When a WebObjects application receives such a request, it cycles through the request-response loop as usual, but there is no user input to store and no action to invoke. As a result, the object representing the requested page—Hello in this case—generates the response.

By implementing a login mechanism in **willGenerateResponse:inContext:**, you can prevent users from accessing pages without authorization.

didGenerateRequest:inContext:

This method is invoked after the response page has generated a response. You can use this method to substitute a different object for the response page after a response has already been composed. If this method returns a non-**nil** value, **willGenerateResponse:inContext:** and **generateResponse: inContext:** messages are sent to the substituted object.

A more common use of **didGenerateRequest:inContext:** is to perform session clean up.

Summary

How is the request-response loop started?

The request response loop is started in the **main()** function with the WOApplicationAdaptor method **runWithApplication:**.

What request-response loop hooks can you implement?

There are three types of hooks into the request-response loop:

- *Action methods* that are associated with a particular user action such as clicking a button or hyperlink
- *awake methods* that are invoked at the point in the request-response loop just before the receiver begins to participate in the request handling
- *Request and response handling methods* that are invoked at a particular point in the request-response loop if you implement them

You can participate in the request-response loop by implementing any of the hooks provided.

What you can use the hooks for?

The following list summarizes common uses of the request-response loop hooks:

- Action methods perform page navigation.
- Application **awake** methods initialize global variables and configure application behavior.
- Component **awake** methods initialize transaction and persistent variables.
- **willPrepareForRequest:inContext:** methods substitute a different object for the request page before user input is processed.
- **didPrepareForRequest:inContext:** methods substitute a different object for the request page before an action method is invoked.
- **willGenerateResponse:inContext:** methods substitute a different object for the response page before a response is composed. For instance, this method can be used authorize access with a login page.
- **didGenerateResponse:inContext:** methods substitute a different object for the response page after a response has already been composed and perform session clean up.