How WebObjects Works

WebObjects is a framework that confers the benefits of object-orientation to the process of receiving a request for a page on the World Wide Web and returning a response. How does WebObjects accomplish this work? The following pages answer this question by delving layer by layer into the architecture and mechanics of WebObjects. The understanding acquired by reading this chapter will make you better prepared for WebObjects application development.

A Wide-Angle View of WebObjects

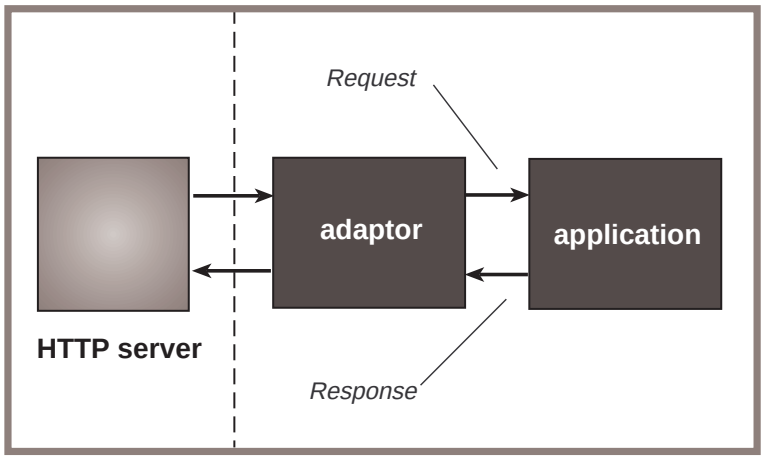
WebObjects applications are event-driven, but instead of responding to mouse and keyboard events, they respond to HTTP requests. A WebObjects application receives a request, responds to it, then waits for the next request. The application continues to respond to requests until it terminates. On each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

One way to get a sense of how WebObjects does this work is to survey incrementally the relationships and dynamics of the public classes. By learning about the basic role of each class, you can see how objects of that class fit into the mechanics of request handling.

Let's start the tour with the typical opening scenario: An incoming message (URL) from a client browser is handled by the HTTP server. From that point until an HTTP response is returned, WebObjects is working.

Server and Application Management

The HTTP sends a request to the application's adaptor, a WOAdaptor object. This object packages the incoming HTTP request in a WORequest object and forwards it to the application object (WOApplication) in a `handleRequest:` message. The application initiates and manages the process of request handling and returns the completed response (WOResponse) to the adaptor, which gives it to the HTTP server in a form the server can understand.



– WOAdaptor

  An abstract class that defines the interface for objects mediating the exchange of data between an HTTP server and a WebObjects application.

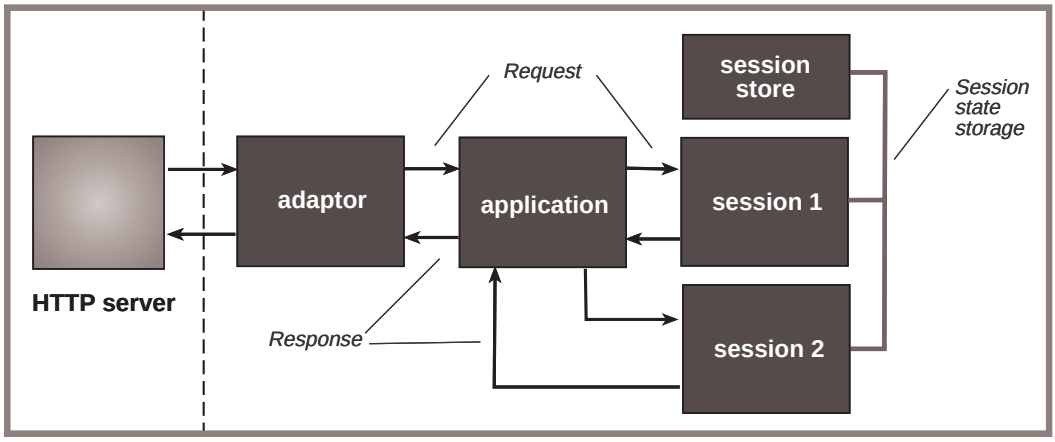
– WOApplication

Objects of this class (or subclass) receive requests from the adaptor and initiate and coordinate the request-handling process, at the end of which they return a response to the adaptor. They also create dynamic elements "on the fly" and manage adaptors, sessions, application resources, and pages.

Session Management

In WebObjects sessions are periods during which one user is accessing the application. An application can have multiple concurrent sessions. The objects dedicated to session management ensure that state with session-wide scope persists between cycles of the request-response loop.

When a user makes an initial request to a WebObjects application, the application creates a session object (WOSession). At the end of the request-response cycle, the application stores the state-bearing session object using the facilities of WOSessionStore. With each subsequent cycle of the request-response loop for that user, the application restores the state of the session at the beginning of the cycle and stores it again at the end of the cycle.



– WOSession

Encapsulates the state of a session. WOSession objects persist between the cycles of the request-response loop (or *transactions*) that occur while a user is accessing a WebObjects application. WOSession objects store (and restore) the pages of a session, the values of session variables, and any other state that components want to persist throughout a session. The number of pages stored by the session object is dependent on the page-cache size set in WOApplication. Each session object is identified by a unique session ID, which is reflected in the URL.

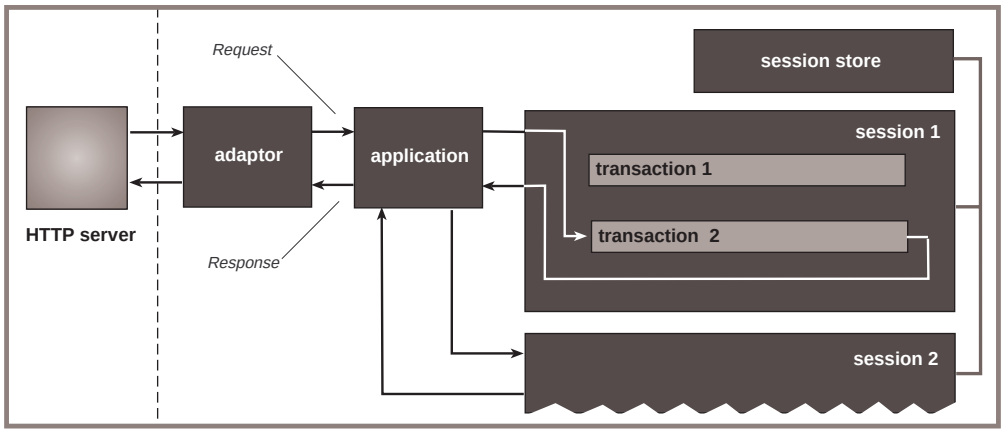
– WOSessionStore

Provides the strategy or mechanism through which WOSession objects are made persistent. An WOSession-Store object stores WOSessions in the server or in the page (which can include Netscape cookies), and restores them upon request by the application.

Request-Response Loop Information

A request-response cycle (sometimes called a *transaction*) has three phases, the first for transferring user-entered data to the objects associated with the request page, the second for invoking an action method, and the third for generating and returning the response. The application initiates each phase by sending a messages:

takeValuesFromRequest:inContext:, invokeActionForRequest:inContext:, and appendToResponse:inContext: It passes in WORequest, WOResponse, and WOContext objects as arguments to one or more of these methods. From these objects, the components, dynamic elements, and other objects involved in the transaction get essential information. (See "The Phases of the Request-Response Loop" on page 7 for more on the mechanics of request handling).



– WORequest

  Stores essential data about an HTTP request, such as header information, form values, HTTP version, host and page name, and session, context, and sender IDs.

– WOResponse

  Stores and allows the modification of HTTP response data, such as header information, status, and HTTP version. It also provides convenience methods for appending HTML and simple textual data to the content of the response (that is, the response page).

– WOContext

  Provides access to the objects involved in the current transaction, including the current request, response, session, and application objects. It also stores the component (which is either the current page or one of its subcomponents) to which the elements of the page make reference when they "push and pull" values via association (see "Component and Element" for an explanation of this). The WOContext object acts as a "cursor" for traversing the object graph during a phase of the request-response loop. The WOContext for a transaction is identified by a unique context ID, which appears in the URL.

Page Composition

Objects of many classes (most of them private) work together to compose the HTML content of response pages. Many of the same objects also set their variable values from data entered into request pages and respond to user actions. Two major branches of these objects descend from WOElement: WOComponent objects, which usually represent pages, and WODynamicElement objects, which represent dynamic HTML elements on the page (that is, elements with changeable state or the ability to trigger actions). For details on how this happens and for more on these classes, see "Component and Element" on page 11.

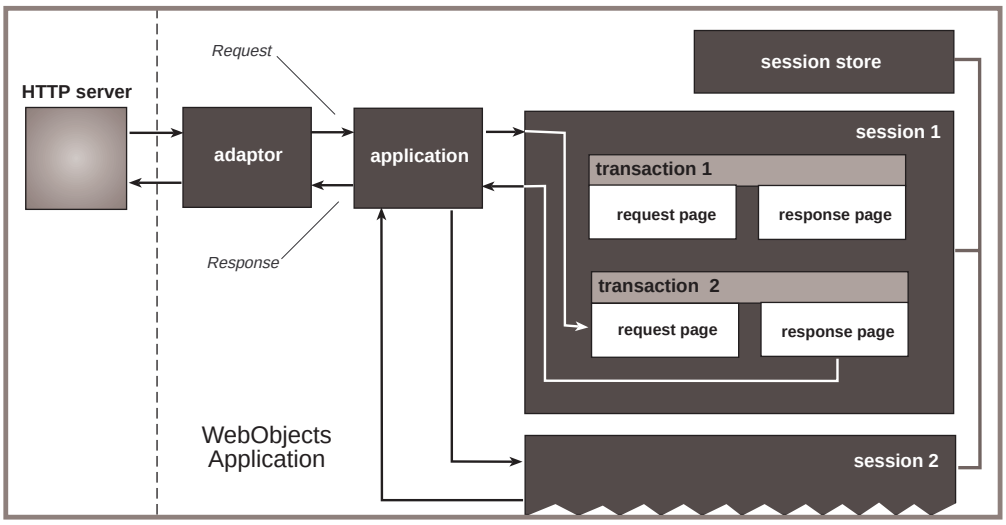


Figure 1: WebObjects Classes in the Context of Request Handling

– WOComponent

  An object that represents a integral, reusable page (or portion of a page) for display in a web browser.

– WOElement

  An abstract class that declares the three request-handling methods: takeValuesFromRequest:inContext:, invokeActionForRequest:inContext:, and appendToResponse:inContext:. Each node in an object graph, which represents the HTML elements of a component and their relationships, is an object that inherits from WOElement.

– WODynamicElement

  An abstract class for subclasses that generate particular dynamic elements.

– WOAssociation

  An object that knows how to find and set a value by reference to a key. Instance variables and action methods of dynamic elements are WOAssociations.

Database Integration

WebObjects has been extended and enhanced in many ways to interact smoothly with the Enterprise Objects Framework, thereby enabling WebObjects applications to fetch, write, and query database records. A major factor behind this integration is WODisplayGroup. Instances of this class (commonly referred to as display groups) provide a simple interface allowing objects in a WebObjects application to interact with relational databases. Enterprise Objects Framework and WODisplayGroup convert operations on objects to database operations on records. Display groups use classes defined in Enterprise Objects to:

- Fetch data from the data.
- Insert, update, and delete database records.
- Build qualifiers from user input and order search results.

- Manage batches of search results.

The WOSession class also provides access to an EOEditingContext object that is used, for example, when changed data is saved to the database. Each session uses an EOEditingContext to manage graphs of objects fetched from a database and to ensure that all parts of an application remain synchronized. For read-only applications you can customize WOSession to return a per-application EOEditingContext.

Starting the Request-Response Loop

A WebObjects application can start up in one of two ways: automatically, when it receives a request (autostarting), or manually, when it's run from the command line. Either way, its entry point is the same as any C program: the main() function. In a WebObjects application, main() is usually very short. Its job is to create and run the application:

```
int main(int argc, const char *argv[])
{
   NSAutoreleasePool *pool = [[NSAutoreleasePool alloc] init];
   WOApplication *application = [[[WOApplication alloc] init] autorelease];

   [application run];

   [pool release];

   exit(0);       // ensure the process exit status is 0
   return 0;      // ...and make main fit the ANSI spec.
}
```

The WODefaultApp application uses a main() function very much like this one. When Project Builder creates a WebObjects application project (for compiled applications) it also generates a similar main() function. If you write a main() function, it should look identical or much the same.

This version of the main() function is deceptively simple. First, it creates an autorelease pool that's used for the automatic deallocation of objects that receive an autorelease message. Then it creates a WOApplication object. This seems fairly straightforward, but in the init method the application creates and stores, in an instance variable, one or more adaptors. These adaptors, all instances of a WOAdaptor subclass, handle communication between an HTTP server and the WOApplication object. The application first parses the command line for specified adaptors (with necessary arguments); if none are specified, as happens when the application is autostarted, it creates a suitable default adaptor.

The run method initiates the request-response loop. When run is invoked, the application sends registerForEvents to each of its adaptors to tell the adaptor to begin receiving events. Then the application begins running in its run loop.

As shown in Figure 2, in each cycle of the loop a WOAdaptor object for the application receives an incoming HTTP request, packages the request in a WORequest object, forwards this object to the WOApplication object in a handleRequest message, and returns the response from the WOApplication to the HTTP server in a form it can understand.

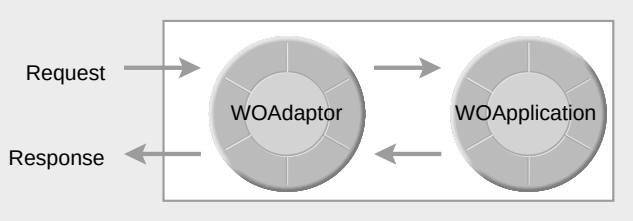


Figure 2: Adaptor and Application in the Request-Response Loop

The last message in this version of main() releases the autorelease pool, which in turn releases any object that has been added to the pool since the application began. See the introduction to the Foundation Framework for a discussion of autorelease pools and object disposal.

A Note on Scripted Classes

The following sections discuss in detail the roles performed by the application (WOApplication), session (WOSession) and component (WOComponent) objects in request handling. In WebObjects, the behavior of these objects can be scripted or compiled. In the latter case, you would might create a subclass of, say, WOSession, and then implement the request-handling methods. Scripted behavior, however, is more common in WebObjects than compiled behavior. In a scripted application, you write WebScript code (or another supported scripting language) in an application's Application.wos and Session.wos files, and in each component's ".wos" files. At run time, the application uses a special subclass of WOApplication and WOSession—and a unique WOComponent subclass for each component—to generate instances. It makes the code it finds in the associated ".wos" files the implementation code of these subclasses.

For more on scripting, including the script files allowed in a WebObjects application, see the chapter "Using WebScript."

The Phases of the Request-Response Loop

When the adaptor receives a request for a page from the HTTP server, it sends the message handleRequest: to the WebObjects application that "owns" the page. This message sets in motion a cycle of the request-response loop. This cycle consists of three phases:

- Taking values from the request
- Invoking an action
- Generating a response

Each phase is associated with a message originated by the application object. Each message passes from application to session, from session to component, and from component to (potentially) each HTML element the component contains.

A note on terminology: In WebObjects, a *page* in a browser is represented by a WOComponent object, or simply, a *component*. Conceptually, this relationship is so strong, that "page" and "component" are synonymous in many of the discussions that follow. In other words, "request page" usually refers to the same thing as "request component."

However, components are not always identified with an entire page; you can have nested components, called *subcomponents* or *reusable components*, that occupy only a portion of a page. See "Subcomponents and Component References" on page 14 for more on subcomponents.

Taking Values From the Request

The purpose of this first phase is to synchronize the state of the request component with the HTML page as submitted by the user. In this phase, the appropriate dynamic elements extract the values that users enter and the choices they make in the request page and assign them to declared variables. For example, if the user clicked a check box, the dynamic element that represents that check box must be set to the "checked" state. In other words, the checked attribute of the appropriate WOCheckbox dynamic element must be set to YES.

Note: The discussion of each phase includes a diagram that—using a matrix of object, relationship, and sequence—lays out the messages invoked and the tasks accomplished in the phase. The diagram is thus mean to accompany the discussion of these specific messages and tasks. The three diagrams, taken together, constitute a diagram of the entire request-response loop. The following diagram shows what goes on in the first phase:

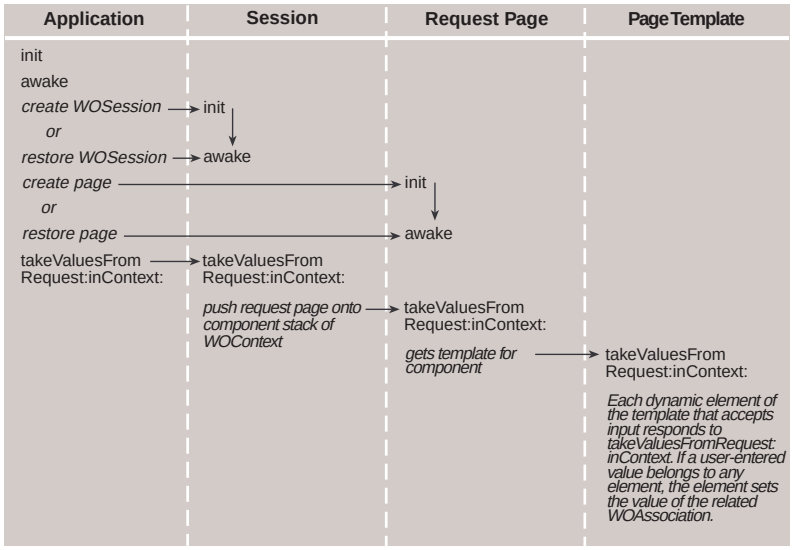


Figure 3: Taking Values from the Request

Upon receiving handleRequest: from a WOAdaptor, the application first creates the WOResponse and WOContext objects that will be needed (the WORequest object is passed in). Then it does the following:

- It invokes its own awake method.
- It restores the WOSession object for the transaction (thereby restoring all session variables) or, if this is the first request from a particular user, it creates a WOSession object. In the latter case, the WOSession's init method is invoked.
- It invokes the session's awake method.

Creating or Restoring the Request Page

Each request received by a WebObjects application is associated with one of the application's pages—the *request page*. If the request doesn't explicitly specify a page, the WOApplication object associates the request with a page named "Main." By default, an application caches page instances once they're created, primarily to facilitate backtracking: when users backtrack, they're revisiting pages restored by the application. During the first phase of the request-response loop, the application restores the component instance that generated the response page earlier in the current session; if this instance doesn't exist, the application creates a component to represent the request page.

The component instance for the request page cannot be restored and must be created if:

- The page-caching feature is turned off.
- The request is the first for that page during the session.

This component—called the *request component*—is an instance of a WOComponent subclass (created automatically for scripted applications). The WOApplication object performs the following steps to create a WOComponent object:

1. It looks in the Objective-C run-time system for a WOComponent subclass with the same name as the request page. If it finds such 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 WOComponent instantiates a LoginPanel object as the request component.
2. If the WOApplication object fails to find a class in the run-time system, it looks for a scripted component with the name of the request page. When it finds the ".wo" directory, it creates a component object using a unique WOComponent subclass for the scripted component and makes the scripted code the class implementation.

When a component is created its init method is invoked. The awake method of both restored and created components is also invoked. In other words, the awake method of request components, as well as for the application and session objects, is invoked once per transaction. (There is one exception to this: When the response component is the same as the request component, awake is not invoked.)

Assigning Input Values

During the first phase of the request-response loop, the request component extracts user-entered values and assigns them to transaction variables. This is the basic sequence of events in preparing for a request:

1. The WOApplication object sends takeValuesFromRequest:inContext: to itself; its implementation simply invokes the WOSession object's takeValuesFromRequest:inContext:.
2. The session, in its implementation of takeValuesFromRequest:inContext:, gets the *template* of the component and forwards the message to it. A template consists of the static HTML elements, dynamic HTML elements, and subcomponents that together compose the page associated with a component instance. (See "Templates" on page 12 for more on this subject.)
3. All dynamic elements in the page template, and in the templates of subcomponents, receive the takeValuesFromRequest:inContext: message. If one of these elements "owns" a user-entered value, it responds to the message by storing the value in the appropriate variable defined in the request component's declarations file.

For more on how components are associated with templates, and on how HTML elements participate in request-handling, see "Component and Element" on page 11.

Invoking an Action

In the second phase of the request-response loop, the application sends the invokeActionForRequest:inContext: message to itself, and eventually to all other objects involved in handling this request. The purpose of this message is to locate the dynamic element which the user has clicked (or otherwise activated) and have that element trigger the appropriate action method in the request component. This method returns the *response page*—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.

This is the basic sequence of events in invoking an action:

1. The WOApplication object sends invokeActionForRequest:inContext: to itself; its implementation simply invokes the WOSession object's invokeActionForRequest:inContext:.

2. The session, in its implementation of invokeActionForRequest:inContext:, gets the template of the component and forwards the message to it.

3. Suitable dynamic elements in the request-page template, and in subcomponent templates, handle the invokeActionForRequest:inContext: message. To be suitable, an element must be able to respond to user actions (a WOSubmitButton or a WOActiveImage, for example). Each of these elements evaluates the invoked action to determine if it "owns" it. If so, it invokes the appropriate action method in the request page, which returns the response page.

For more on how components are associated with templates, and on how HTML elements participate in request-handling, see the section "Component and Element" on page 11.

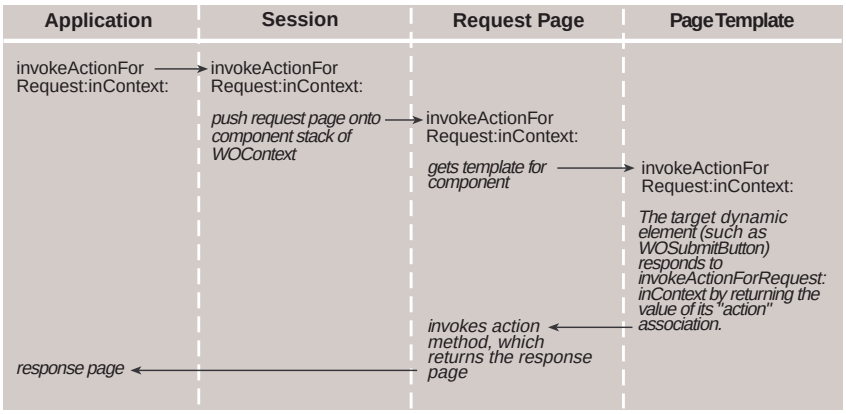


Figure 4: Invoking an Action

Generating the Response

In the final phase of request-response loop, the response page generates an HTTP response. Generally, the response contains a dynamically generated HTML page. Each element (static and dynamic) that makes up the response page appends its HTML to the total stream of HTML code that will be interpreted by the client browser.

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

1. The WOApplication object stores the object returned from an invoked action method as the current page for the transaction.

2. Then it sends appendToResponse:inContext: to itself; its implementation simply invokes the WOSession object's appendToResponse:inContext:.

3. The session, in its implementation of appendToResponse:inContext:, pushes the response component onto the WOContext stack, gets the template for the component, and sends appendToResponse:inContext: to the template.

4. All static and dynamic HTML elements in the response-page template, and in subcomponent templates, receive the appendToResponse:inContext: message. In it, they append to the content of the response the HTML code that represents them. For dynamic elements, this code includes the values assigned to variables.

After the response has been generated, but before returning the response to the adaptor, the application object concludes request handling by doing the following:

1. It causes the sleep method—the counterpart of awake—to be invoked in all components involved the transaction (request, response, and subcomponents). In sleep objects can release resources that don't have to be saved between transactions.

2. Then the application requests the session object to save (cache) the response page.

3. It invokes the session object's sleep method and saves the session object.

4. It invokes its own sleep method.

When an object is about to be destroyed, its dealloc method is invoked. This happens some at some indefinite point after a transaction (indicated by vertical ellipses in the diagram below). In the dealloc method, the object should release any retained instance variable. (In scripted applications this is unnecessary since "garbage collection" occurs automatically in this case.)

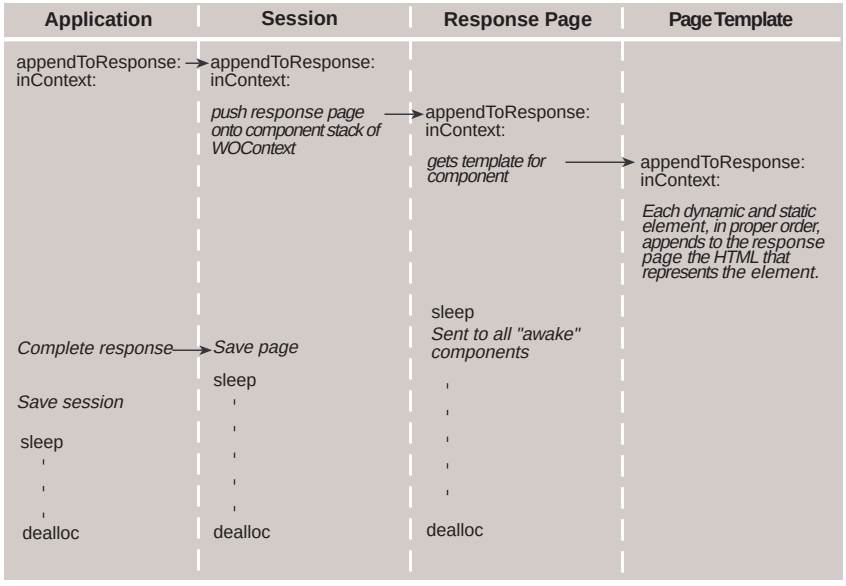


Figure 5: Generating the Response

Component and Element

So how exactly are request-handling messages propagated from a component to its HTML elements? This question begs another: What is the relationship between component and HTML element?

Both WOComponent objects and HTML elements (static and dynamic) share a common ancestor, WOElement. WOElement declares, but does not implement, the three request-handling messages: takeValuesFromRequest:inContext:, invokeActionForRequest:inContext:, and appendToResponse:inContext:. This common inheritance, of course, makes it possible for both components and HTML elements to participate in request handling. But there the inherited similarities end. Although components can generate HTML content, this capability is not a essential characteristic, as it is with objects on the other branch of the inheritance tree.

Components are reusable pages, or portions of pages, displayed in a World Wide Web browser. As with all objects, components contain a unique set of data, although the structure of that data is the same for each component instance. Through a script or compiled code, components also include logic that manipulates the data and otherwise affects behavior, especially the behavior of returning another page based on a user's request. Finally, each component is associated with a template.

Templates

A template is a fixed and hierarchical network of elements and subcomponents shared by instances of a component. An application composes the template for a component at run time when the component is first requested. Template creation involves the parsing and integration of the component's “.html” and “.wod” files. This activity results in a *component definition*, which contains the template. A component definition captures essential information about a component and facilitates access to component resources. It makes it possible for component instances to share resources; instances need carry only the instance-variable values that are distinctive to them. If caching of component definitions is enabled, the application subsequently stores templates; in this case, parsing occurs only once during an application's lifetime.

When a component requests its template in a request-handling method, the WOElement object that is returned is the root object of an object graph. The root object can reference, directly or indirectly, every other element of the template. The network of reference corresponds to location on the page and to parent-child relationships; for instance, a WOForm would probably have WOTextField and WOSubmitButton children.

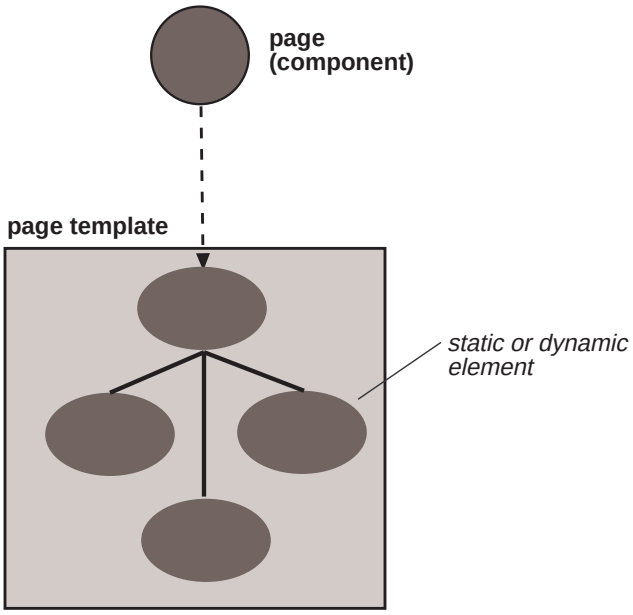


Figure 6: An Object Graph for a Page's Template

For each request-handling message, the default behavior of WOComponent is to forward the message to its template, which is represented by the root element. The root element, in turn, forwards the message to each of its child elements; if they have any children, these elements send it to them. Thus each element has an opportunity, if appropriate, to extract user data from the request, to invoke an action in the component, and to append its HTML representation to the response.

Each HTML element on a template has an element ID to identify it within the object graph. An element ID is implemented as an extension of the sender ID in the URL. You can request the current element ID from the WOContext object.

Associations and the Current Component

A dynamic HTML element such as a text field or a pop-up button differs from a static HTML element (for example, a heading) in that its attributes can change over a cycle of the request-response loop. These attributes can include values that determine behavior or appearance (a "disabled" attribute, for instance), values that users enter into a field, values that are returned from a method, and actions to invoke when users click or otherwise activate the element. Each dynamic element stores its attributes as instance variables of type WOAssociation.

WOAssociation objects know how to obtain and set the value they represent. Usually WOAssociations work by key-value coding. The key to a value can be represented as a sequence of keys separated by periods. The resolution of a key by yielding its value makes possible the resolution of the next key. For instance:

```
self.aRepetition.list.item
```

means that `self` (identifying the current component) has a WORepetiton named `aRepetition`. The `list` key denotes the list of elements displayed by the WORepetiton, and the `item` is the key to the current item in that list. Keys (including actions) are WOAssociations defined for each dynamic element. The values for these keys are constants assigned in the ".wod" file, or they derive from bindings to variables or methods, also made in a component's ".wod" file.

WOAssociations refer to the *current component* for the initial value of this sequence. It gets this object from the transaction's WOContext object. Often the current component is the request or response page of the transaction, but it can be a reusable component embedded in a page, or even a component incorporated by one of those subcomponents (see "Subcomponents and Component References" on page 14, below, for more on this). The WOContext stores the current component on a stack, "pushing" and "popping" components onto and off of the stack as necessary.

Depending on the phase of the request-response loop, a dynamic element uses its WOAssociations to "pull" values from the request (that is, set its values to what the user specifies) or to "push" its values onto the response page. When a dynamic element that can respond to user actions (such as WOSubmitButton) requests the value of its "action" WOAssociation, the appropriate action method in the current component is invoked and the response page is returned.

The exchange of data through an association that binds an attribute of a parent component to an attribute of a child component is two-way. This two-way binding allows the synchronization of state between the two components. For example, consider this example declaration in Main.wod of the TimeOff example:

```
START:Calendar {

    selectedDate = startDate;

    callBack = "mainPage";

};
```

In this example, Main is the parent component and Calendar is the child component. The "startDate" variable belongs to the parent component while "selectedDate" is a variable of the child component. A change in the parent component instance variable is automatically communicated through the association to the child variable. Conversely, a change in value in the child component variable is communicated to the parent variable. Component synchronization occurs at the beginning and end of each of the three request-handling phases of a component (takeValuesFromRequest:inContext:, invokeAction:inContext:, and appendToResponse:inContext:) Synchronization is performed through the accessor methods of both components.

This aspect of synchronization has a couple of implications for developers. Since WebObjects invokes explicitly implemented accessor methods many times during the same request-response loop, accessor methods should not have any side effects. Accessor methods should simply set a variable's value or return a value. And if you return a value, there should be some way for WebObjects to set the value. This rule applies also when the binding involves a parent or a child component's method instead of an instance variable. To illustrate this, assume that "startDate" is a method of the Main component instead of an instance variable. Even in this case, WebObjects attempts to synchronize "startDate" with the "selectedDate" value. In other words, WebObjects attempts to invoke a "setStartDate:" method, and raises if such a method does not exist.

See the chapter "Reusable Components" for more on state synchronization between child and parent component.

Subcomponents and Component References

A "node" in a template's object graph can represent a subcomponent (also called "reusable component") as well as a dynamic or static HTML element. A dynamic element called a *component reference* represents all occurrences of the subcomponent in the parent component. At run time, the component reference binds itself to the separate instances.

A subcomponent can fire actions against its parent component (using performParentAction:) and, if the parent's state changes, it's state is synchronized accordingly. In other words, its state is updated to reflected changes based upon its bindings with the parent.

An element ID is assigned to each instance of a subcomponent. When the chain of request-handling messages goes down an object graph and reaches the component reference, it resolves references to its instances based on the element ID of each instance. Components keep track of all their subcomponents by storing them in an internal dictionary using element IDs as keys.

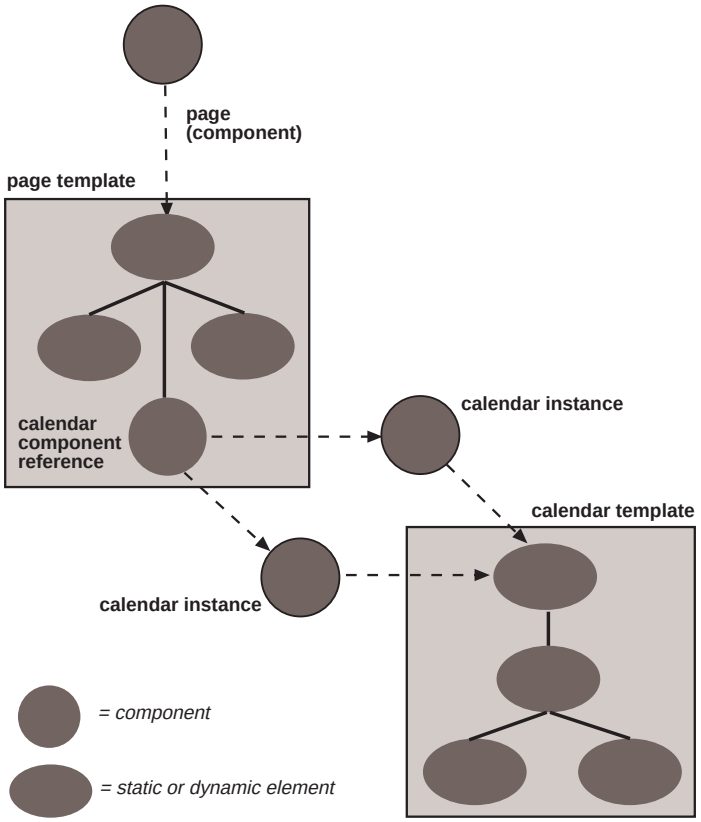


Figure 7: An Object Graph for a Page With a Subcomponent