WebObjects Web Applications Programming Guide

Tools > WebObjects

2007-07-11

Apple Inc.
1 Infinite Loop
Cupertino, CA 95014
408-996-1010

Simultaneously published in the United States and Canada.

Contents

Figures, Tables, and Listings

Introduction to WebObjects Web Applications Programming Guide

Important: The following tools are deprecated and no longer supported in WebObjects 5.4 and later: EOModeler, RuleEditor, WebObjects Builder, WOALauncher, and Java Client. WebObjects templates are not available for creating new projects in Xcode on Mac OS X v10.5 and later.

Note: This document was previously titled *Web Applications*.

Web applications are a type of WebObjects application that generates HTML-based dynamic webpages accessed via a client-side web browser. Web applications are object-oriented programs written in Java. Webpages are created from templates called web components. Web components are a combination of a WOComponent Java subclass and an HTML template. You create dynamic content in your webpages by adding dynamic elements to web components and binding them to variables and methods in your application. You can create web components graphically using WebObjects Builder or indirectly using Direct to Web. If you use Direct to Web, you can also freeze components, add them to your project, and edit them using WebObjects Builder.

Who Should Read This Document?

This document focuses on web application programming concepts and tasks. Read this document if you are developing a web application and need to learn more about programming web components, managing state in application and session objects, and using editing contexts. This document also explains how web applications work by tracing the request-response loop and explains how to create web application projects using Xcode. This document covers common tasks that web application developers need to know such as creating an EO model and deploying applications for testing.

Organization of This Document

This document contains the following articles:

- "How Web Applications Work" (page 9) describes the architecture of web applications and explains the messages invoked by the request-response loop.
- "Creating Projects" (page 19) explains the Xcode templates you can use to create a web application.
- "Creating Enterprise Objects" (page 29) explains how to create a simple Enterprise Objects (EO) model—the first step if you are using a back-end database to populate your webpages with dynamic content.

- "Creating Web Components" (page 37) explains how to create and reuse web components from a programmer's perspective. This article also covers more details about the methods invoked by the request-response loop.
- "Using the Application and Session Objects" (page 47) explains how to use the Application and Session objects in your web application to maintain state.

If you are new to WebObjects, read "How Web Applications Work" (page 9), "Creating Projects" (page 19), and "Creating Enterprise Objects" (page 29) first. Also, read *WebObjects Builder User Guide* for step-by-step instructions on how to create web components using WebObjects Builder. Read the rest of the articles in this document when you are ready to customize your web application and add advanced features.

See Also

For more information on related WebObjects subjects, see these documents"

- *WebObjects Overview* to learn about other WebObjects technologies.
- *WebObjects Builder User Guide* for how to create web components graphically.
- *WebObjects Direct to Web Guide* for how to use Direct to Web to create a web application.
- *WebObjects Enterprise Objects Programming Guide* for an in depth description of Enterprise Objects.
- *WebObjects 5.3 Reference* and for details about the WebObjects and Enterprise Objects APIs.
- *WebObjects Deployment Guide Using JavaMonitor* for details on how to deploy web applications.

How Web Applications Work

Web applications generate dynamic HTML-based webpages accessed through a web browser. Since WebObjects applications are object-oriented and written in Java, your application generates webpages by creating instances of objects called web components.

A **web component** is a combination of a Java subclass of WOComponent and an HTML template. Web components can contain any standard HTML elements and components including Flash animations, QuickTime movies, JavaScript programs, and Java applets. Web components also support Cascading Style Sheets (CSS).

You add dynamic content to your webpages by adding special WebObjects elements with HTML counterparts—called **dynamic elements**—to your web components. Some dynamic elements don't have HTML counterparts and are just used to control the generation of content—for example, content that is conditional or iterative. Dynamic elements are translated into static HTML when responding to client browser requests.

You can use either WebObjects Builder or Direct to Web to construct web components. **WebObjects Builder** is a graphical tool for creating web components and binding dynamic elements to variables and methods in your application.

Direct to Web is a rapid prototyping tool that creates a working web application from a given EO model. You use the Web Assistant to change the content of Direct to Web components. You can also freeze pages—create and add web components to your project—and modify them using WebObjects Builder.

You should have a basic understanding of the architecture of a web application before customizing your web application. This section describes the architecture of web applications and explains how dynamic elements work within the context of the application's request-response loop. It contains a brief description of the sequence of methods invoked when processing a request and generating a response page. This section also explains how backtracking works in WebObjects.

This document does not explain how to use the various WebObjects tools. Read *WebObjects Builder User Guide* for the steps involved in creating forms and binding dynamic elements. Read *WebObjects Direct to Web Guide* for how to use Direct to Web.

Application Architecture

Not only can your web application generate dynamic content but you can present forms to the user allowing them to author content. You obtain input from users using HTML-based forms, buttons, and other dynamic elements. Connecting form elements to variables and methods in your web component is similar to binding other dynamic elements that just display content.

You create forms by placing dynamic elements into a standard form element in your web component. The web component generates HTML that web browsers can interpret and display. This process includes translating user-entered data or selections back into variables in your application. If you are programming web components, it helps to understand how web applications process user input.

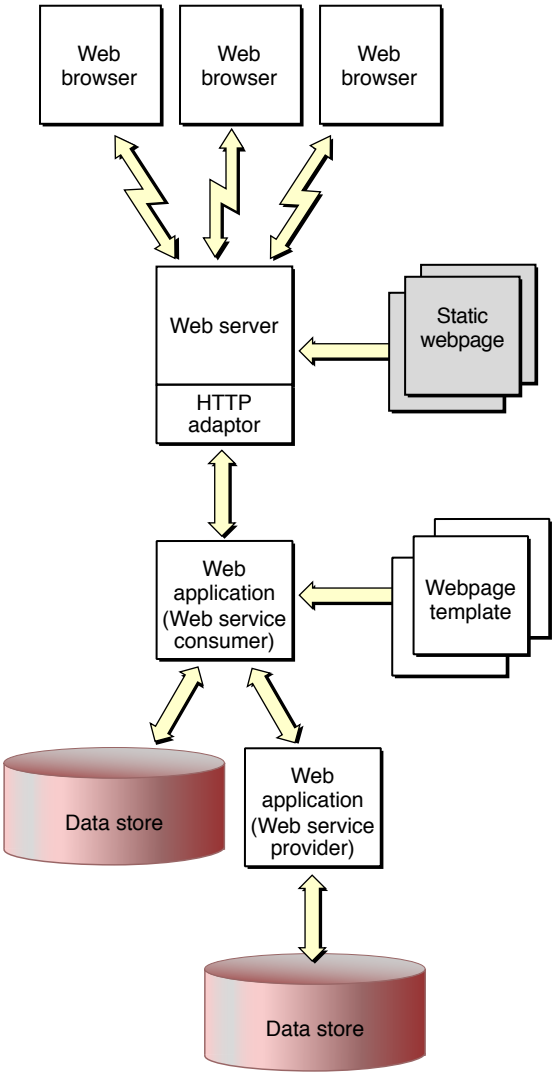
WebObjects applications are event driven, but instead of responding to mouse and keyboard events, they respond to HTTP (Hypertext Transfer Protocol) requests. The application receives an HTTP request for an action, responds to it, and then waits for the next request. The application continues to respond to requests until it terminates. The main loop that handles these requests is called the **request-response loop**.

Inside the request-response loop, WebObjects fills in the content of dynamic elements when the page needs to be generated in response to a request. The information your applications publish can reside in a database or other data-storage medium or it can be generated at the time a page is accessed. The pages are also highly interactive—you can fully specify the way the user navigates through them and what data they can view and modify.

Figure 1 shows a WebObjects-based website. Again, the request (in the form of a URL) originates from a web browser. The web server detects that the request should be handled by a WebObjects application and passes the request to an HTTP adaptor. The adaptor packages the incoming request in a form the WebObjects application can understand and forwards it to the application. Based upon web components you define and the relevant data from the data store, the application generates a webpage that it passes back through the adaptor to the web server. The web server sends the page to the web browser, which renders it.

This type of WebObjects application is referred to as a **web application**, since the result is a series of dynamically generated HTML webpages.

Figure 1 A dynamic publishing website

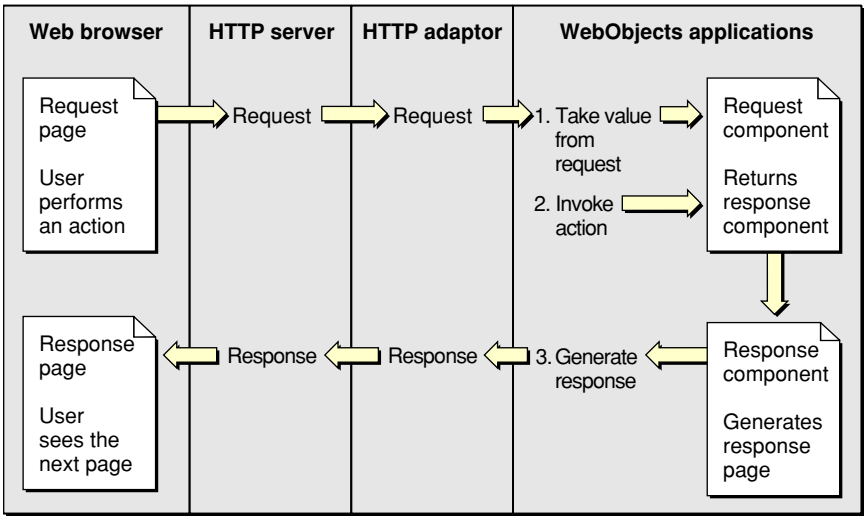


Request-Response Loop

Each action taken by a user is communicated to your application via the web server and the WebObjects adaptor. All the pertinent details of the user's action—the contents of text fields, the state of radio buttons and checkboxes, and the selections in pop-up menus—as well as information about the session and the button or link activated is encoded in the HTTP request.

The request is decoded by the action of the WebObjects adaptor and default application behavior. This decoding process, which culminates in the generation of a response page to be returned to the web browser, constitutes the request-response loop. Figure 2 shows the sequence of messages invoked when processing a request.

Figure 2 The request-response loop



WebObjects has two request-processing models: component actions and direct actions.

- The **component actions** model allows you to maintain state in applications; therefore, it requires and uses session objects. By default, web applications use this model.
- The **direct actions** model is used by applications that don't require state management—for example, search engines, product catalogs, document libraries, and dynamic publishing. Applications that use this model don't have session objects by default.

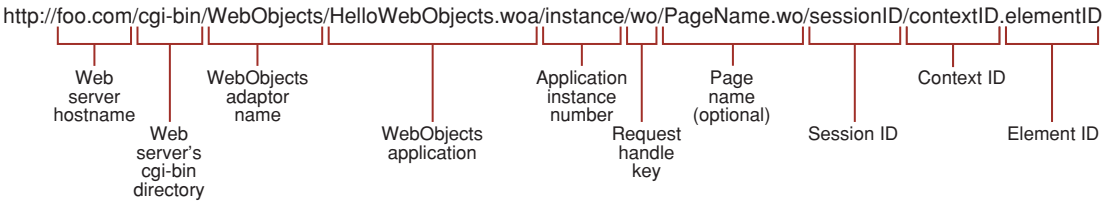
When developing an application, you are not restricted to one request-processing model. Applications can use the model most appropriate to implement specific features. Component actions are generally useful in web applications with interconnected components; however, they do not give the user a great deal of control over an application's flow. For example, a user cannot directly execute a method defined in the Java source file of a web component. Direct actions, on the other hand, are better suited at providing users such access. For example, using the appropriate URL, users can execute specific methods of an application.

Refer to the API documentation for the WODirectAction class in *WebObjects 5.3 Reference*. This article explains component actions in more detail.

Component Action URLs

When you deploy a web application and access it from a web browser, the URL displayed by the browser has a specific format that identifies the web application, page, session, context, and even the element. Figure 3 shows the parts of the URL. The URL contains all the information necessary for an application to reconstruct the state of the session and web components that were last generated for a given client. Listing 1 shows an example of a component action URL.

Figure 3 Structure of a component action URL



Listing 1 Example of a component action URL

```
http://foo.com:49663/cgi-bin/WebObjects/TimeDisplay.woa/wo/NDdW3uF2xRVjvbXUgRCVM/0.5
```

Request-Response Loop Messages

Table 1 lists the phases of the request-response process. Table 2 shows the order in which the methods involved are invoked. The process is explained in detail in "Processing the Request" (page 14) and "Generating the Response" (page 16). The primary objects that receive messages from the request-response loop are the application, session, and web component objects.

The **application object** is an instance of Application where Application is a subclass of WOApplication. A **session object** is an instance of Session where Session is a subclass of WOSession. An instance of Application is created when your application launches, and an instance of Session is created for each initial user. Note that sessions may time out. You can configure the time out duration when deploying an application.

If you select one of the web applications templates in Xcode when creating a project, Application and Session classes are automatically added to your project. Read "Creating Projects" (page 19) for how to create a WebObjects Xcode project.

Table 1 Request-response processing phases

Phase	Method	Description
Awake	`public void awake()`	The application, session, and component objects are awakened. Custom initialization logic can be added in this phase.
Sync	`public void takeValuesFromRequest (WORequest, WOContext)`	Form data is read into the instance variables the WebObjects elements are bound to. Key-value coding set methods are invoked.
Action	`public WOActionResults invokeAction (WORequest, WOContext)`	The action the user triggered—with a link or a submit button—is performed. The action could create a new page.
Response	`public void appendToResponse (WOResponse, WOContext)`	The response page is generated. The form elements' contents are set to the values stored in the instance variables the WebObjects elements are bound to. Key-value coding accessor methods are invoked.

Phase	Method	Description
Sleep	`public void sleep()`	The application, session, and component objects are put to sleep. Custom deactivation logic can be added in this phase.

Table 2 Request-response processing timeline

Application	Session	Component
`awake`		
	`awake`	
		`awake`
`takeValuesFromRequest`		
	`takeValuesFromRequest`	
		`takeValuesFromRequest`
		Set methods invoked.
`invokeAction`		
	`invokeAction`	
		`invokeAction`
`appendToResponse`		
	`appendToResponse`	
		`appendToResponse`
		Accessor methods invoked. Response page generated.
		`sleep`
	`sleep`	
`sleep`		

Processing the Request

Request processing takes place in three stages: awake, sync, and action.

- **Awake**. This stage is carried out when WebObjects sends `awake` messages to several objects.

In a multi-user system, limited resources need to be used as efficiently as possible. To this end, applications are active only while they perform a task. A single server can be running several applications or many instances of the same application. Application instances are active only while processing requests. See "Generating the Response" (page 16) for more information.

The application object's `awake` method is invoked first, then the session object's `awake` method, and, for component action–based requests, the web component's `awake` method. You can customize the method in each of the corresponding classes to add logic that needs to be performed before processing the request. Even though the default implementations of these `awake` methods do nothing, you should invoke the superclass implementation before executing custom logic, as here:

```
public void awake() {
    super.awake();

    /* Custom logic goes here. */
}
```

- **Sync**. During this stage, the `takeValuesFromRequest` method is invoked, which causes the values entered in form elements by the user to be copied into the corresponding instance variables. This stage is skipped if the component contains no form elements or if the values of the form elements are not changed.

  WebObjects invokes the application object's `takeValuesFromRequest` method. The application then invokes the session object's corresponding method, which in turn invokes the web component's method (for component action–based requests). The component invokes each dynamic element's `takeValuesFromRequest` method, which causes form elements to copy the values from the request into the appropriate component bindings. WebObjects uses key-value coding—implemented by the NSKeyValueCoding interface in (`com.webobjects.foundation`—to determine how to set the value of the binding.

  To set the value of a key named `key`, key-value coding looks for an available set method or an instance variable in the following order:

  1. `public void setKey()`
  2. `private _setKey()`
  3. `_key`
  4. `key`

- **Action**. During this stage, the action the user chose is executed by invoking the `invokeAction` method.

  Like the `takeValuesFromRequest` method, WebObjects invokes the application's `invokeAction` method. The application then invokes the session's method, which in turn invokes the web component's method (for component action–based requests). The component then sends `invokeAction` to each of its dynamic elements.

  When the `invokeAction` method of the dynamic element that triggered the request is invoked—for example, a submit button—the dynamic element sends the message bound to its `action` attribute.

Generating the Response

After the form values are gathered and the action method is invoked, the application creates a response page. This is the web component returned by the action method. The response-generation process has two phases: response and sleep.

- **Response**. The response page is generated during this phase. Each dynamic element's `appendToResponse` method is invoked, so that it can add its content to the rendered webpage.

  WebObjects invokes the application's `appendToResponse` method. Then the application invokes the session's method, which in turn invokes the web component's method. The component goes through its HTML code creating the page's content. When it finds a `WEBOBJECT` element, it invokes its `appendToResponse` method, so that it can get the values of its bindings and add the resulting content to the page. The process continues recursively until the entire response page is generated.

  Again, WebObjects uses key-value coding when a variable needs to be accessed or set. When the value of a key named `key` is requested, key-value coding first looks for an accessor method. If one is not found, it accesses the instance variable itself. The order in which key-value coding tries to obtain the value for `key` is as follows:

  1. `public [...] getKey()`
  2. `public [...] key()`
  3. `private [...] _getKey()`
  4. `private [...] key()`
  5. `[...] _key`
  6. `[...] key`

- **Sleep**. When the response process is completed, the `sleep` methods of the web component, session, and application objects are invoked. (The order in which the objects' `sleep` method is called is the opposite of the order in which the `awake` methods are invoked in the awake phase.) When overriding the `sleep` method, you should incorporate the superclass implementation at the end of the method as shown in Listing 2. After all the objects involved in the request-response process are put to sleep, the new page is sent to the WebObjects adaptor.

Listing 2 Overriding the `sleep` method

```
public void sleep() {
    /* Custom logic goes here. */

    super.sleep();
}
```

Backtracking Cache

WebObjects supports the use of a web browser's Back button by keeping a cache of recently viewed pages on the server. This process is called **backtracking**. By default, a cache is configured to hold 30 pages per session, but you can customize it to meet your needs. To change the default size of the cache, add code to the Application class's constructor. For example, to change the page cache size to 45 pages, you add this code line:

```
setPageCacheSize(45);
```

When a response page is generated, it and its state information are added to the cache. That way, when the user clicks the browser's Back button, WebObjects can retrieve the correct web component and its state.

For backtracking to work properly with dynamic data, a web browser's own cache must be disabled, so that all page requests go to the web server and, therefore, your application. You can do this by adding this code to the Application class's constructor method:

```
setPageRefreshOnBacktrackEnabled(true);
```

When the cache becomes full, the oldest page in it is discarded to make room to store a new page. When the user backtracks past the oldest page in the cache, WebObjects alerts the user with a special webpage.

For more information on backtracking, read "Backtracking and Cache Management" (page 51).

Creating Projects

A WebObjects project contains all the files you need to build and run your application. You use Xcode to create a new WebObjects project. In Xcode, you select the appropriate WebObjects project template and an assistant guides you through the process of creating the project. The types of files added to your Xcode project and their organization depends on the Xcode template you choose. The organization of web applications—applications that generate dynamic HTML content—are very similar although the frameworks, targets, settings, and build configurations may differ slightly.

This article explains how to use Xcode to create web applications. This article describes the different templates, provides step-by-step instructions to create your project, explains the organization of files in the project, explains web application specific targets, and contains tips on building and installing your application. Read *Xcode 2.2 User Guide* for complete instructions on how to use Xcode.

Choosing a Template

When you create a project in Xcode, you need to select the appropriate WebObjects template in the assistant. The templates that create a web application are Direct To Web Application, Display Group Application, and WebObjects Application. You can also select WebObjects Framework.

- Choose Direct To Web Application if you have an EO model you created earlier with either EOModeler or Xcode and want to build a quick prototype. This is a good choice for developers new to WebObjects.
- Choose Display Group Application if you have an EO model or intend to create one—that is, you want to populate your webpages with content from a back-end database—and you want to build custom web components.
- Choose WebObjects Application if you don't want to use Enterprise Objects.
- Choose WebObjects Framework if you want to create a framework. Typically, you select this template to create a framework containing your business logic—your enterprise objects and EO model—which can be reused in other types of applications such as Web Services. You can also create a framework of reusable web components.

If you want to create a Direct to Web or display group application, read "Creating Enterprise Objects" (page 29) for how to create your EO model.

Creating a Web Application Project

When you create a project from a template, the Xcode Assistant will guide you through the process by displaying a number of panes. The first few panes are the same for all types of web applications. The later panes may differ depending on the template you select. The default settings in the assistant will work for

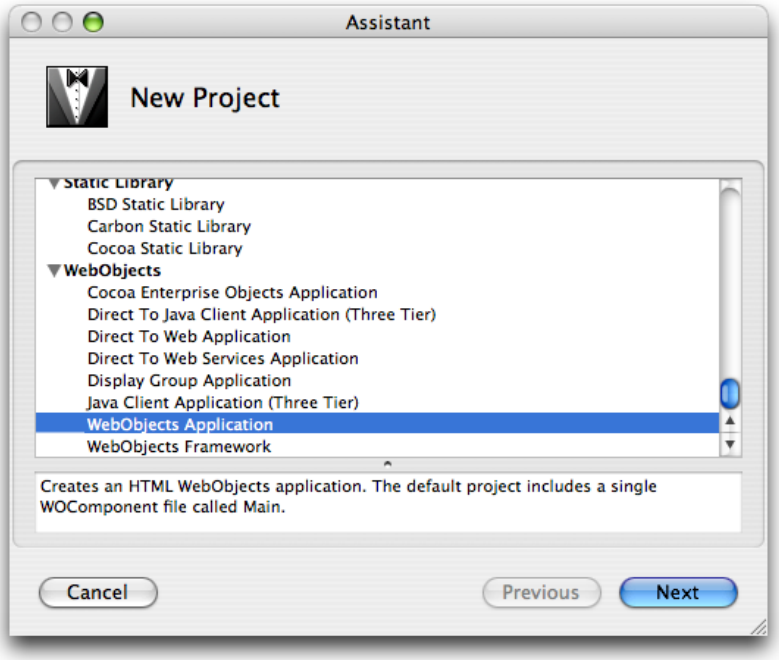
most applications. Typically, you just need to enter a project name and click the Next button and on the final pane, click Finish. Follow these general steps to create a web application. Read *WebObjects Direct to Web Guide* for details on using the Direct to Web Application template.

1. Launch Xcode located in `/Developer/Applications`.
2. Choose File > New Project.

   The Assistant panel appears displaying a list of templates.
3. Select one of the WebObjects templates and click Next as shown in Figure 1.

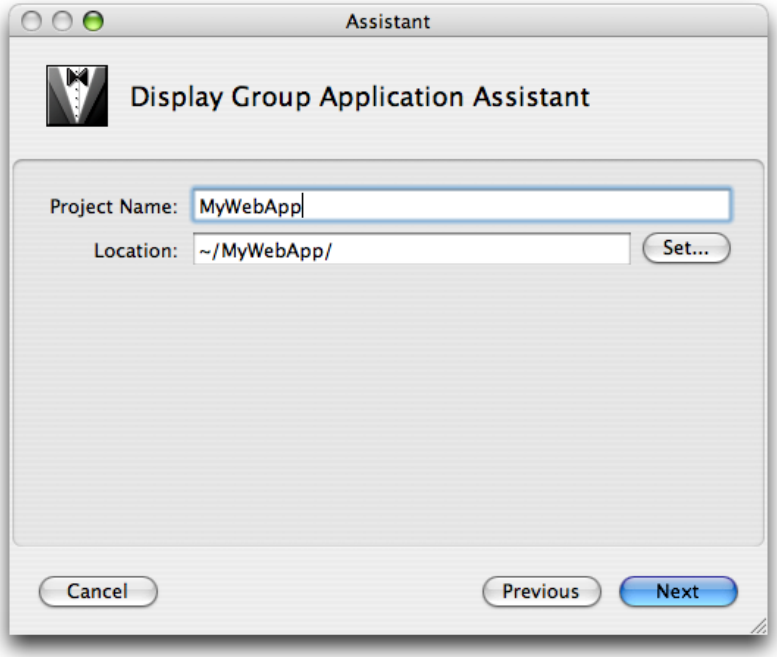
   Read "Choosing a Template" (page 19) if you are not sure what template to use.

Figure 1 Selecting a WebObjects template



4. Enter a project name and location and click Next as shown in Figure 2 (page 21). If you selected the WebObjects Framework template, click Finish and skip the remaining steps.

Figure 2 Entering a project name



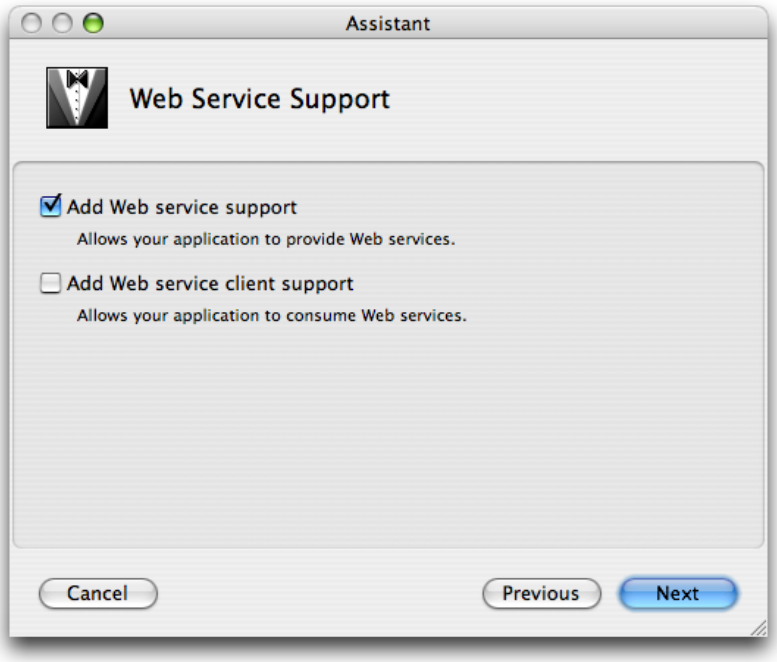
5. If you want to deploy your web application in a J2EE servlet container, select the "Deploy in a servlet container" option on the J2EE Integration pane and then click Next.

   J2EE integration is optional. Just click Next if you don't want to use this feature.

6. If your application is a web service, select "Add Web service support" on the Web Service Support pane. If your application uses a web service, click "Add Web services client support." Then click Next as shown in Figure 3.

Using web services is optional. Just click Next if you don't want to use this feature.

Figure 3 Adding web service support



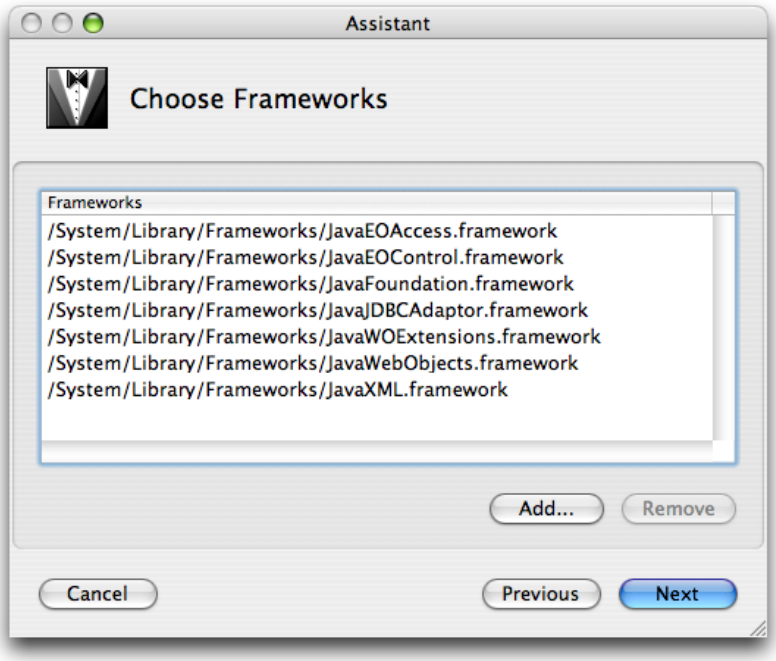
7. If you use the JDBC adaptor, select `JavaJDBCAdaptor.framework` on the Choose EOAdaptors pane. If you use the JNDI adaptor, select `JavaJNDIAdaptor.framework`.

   The default database adaptor is JDBC since most modern databases support JDBC. Just click Next if you are unsure about which database you will use.

8. Click Add on the Choose Frameworks pane if you need to add additional frameworks to your project—for example, third-party database frameworks—as shown in Figure 4. Otherwise, click Next to continue.

The assistant adds the appropriate Java and WebObjects frameworks to your project depending on the template you selected. Just click Next if you don't need to add any more frameworks.

Figure 4 Choosing frameworks



9. Next you add an EO model by clicking Add on the Choose EOModels pane.

   If you have an existing EO model that you created using either EOModeler or Xcode, you can add it to your project now. If you selected the Direct to Web Application or Display Group Application template, then selecting an EO model is mandatory.

10. If you selected the Web Application template, click Finish and skip the remaining steps.

11. If you selected the Direct to Web Application template, then a few panes specific to Direct to Web appear. Read *WebObjects Direct to Web Guide* for how to create a Direct to Web application.

12. If you selected the Display Group Application template, then a few panes specific to configuring a display group appear.

13. Choose the main entity on the Choose the Main EOEntity pane. Select the entity that represents the root objects and click Next.

14. Choose a layout for the page on the Choose a Layout pane and click Next.

15. Choose the properties to display on the page similar to configuring a Direct to Web application on the Choose Attributes to Display pane. Click Finish when done.

When you click the Finish button, the Assistant panel closes and a project window opens containing all your application files. Read "Project Groups and Files" (page 24) for a description of your project files.

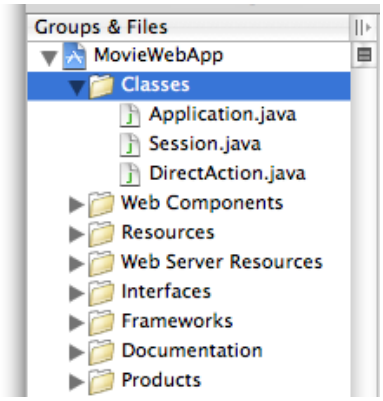
Project Groups and Files

The groups and files displayed in Xcode are different depending on which WebObjects template you choose when creating a Xcode project. This section describes some of the groups that appear when you create a web application.

Classes

The Classes group contains Java classes that do not correspond to web components as shown in Figure 5 (page 24). Typically, this group contains `Application.java`, `Session.java`, and `DirectAction.java`. Classes that corresponding to web components are located in the Web Component group. For example, `Main.java` is located in `Web Components/Main`.

Figure 5 Classes group



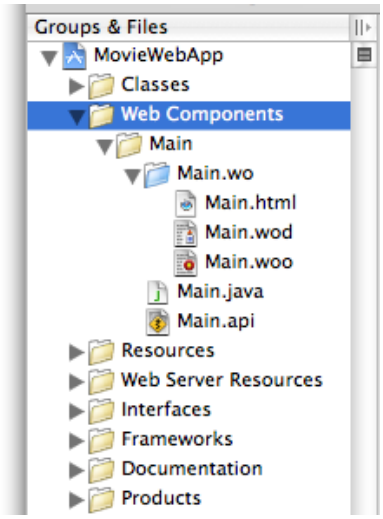
The default classes in a WebObjects project are:

- `Application.java` is a subclass of WOApplication. The application object is automatically created when your application launches and corresponds to the application instance.
- `Session.java` is a subclass of WOSession. A session object is automatically created when a user makes a connection to your web application.
- `DirectAction.java` is a subclass of WODirectAction.

Web Components

The Web Components group contains all the files pertaining to web components as shown in Figure 6. A web component represents a page, or part of a page, in your application. An application can have one or more web components. For example, every WebObjects application has at least one component called `Main` which appears in the Web Components group. The `Main` component implements the first page displayed by your web application.

Figure 6 Web components group



There's a folder for each web component in the Web Components group. A web component folder contains several files that specify the component's look and behavior. Each file has the same prefix but with different extensions. These are the files contained in a web component folder:

- A web component with a `.wo` extension that stores the layout of HTML elements and bindings to dynamic elements.
- A class file with a `.java` extension that implements the component's behavior. Each component is a subclass of WOComponent. Your class typically implements variables and methods that are bound to dynamic elements.
- An API file with a `.api` extension that contains the keys defined by a component that other components can bind to and the rules for binding the keys. WebObjects Builder uses these files to check if a reusable component is used correctly.

Typically, you edit a component using WebObjects Builder—just double-click a folder with a `.wo` extension to edit it in WebObjects Builder. However, you can sometimes edit these files directly if you understand the format. For example, these are the files in the `Main.wo` folder:

- `Main.html` is the HTML template for the component. This file contains HTML tags, just like any webpage; in addition, it can contain tags for dynamic elements.
- `Main.wod` is the declarations file that specifies bindings between the dynamic elements and variables or methods in your Java file.
- `Main.woo` is used to store information about display groups—for example, if your project accesses a database—and encodings for HTML templates. You rarely edit this file directly.

Resources

The Resources group contains files that are needed by your application at runtime, but which do not need to be in the web server's document root and hence will not be accessible to users. Resource files may include miscellaneous configuration files, EO model files, and icons.

Web Server Resources

The Web Server Resources group contains files, such as images and sounds, that must be under the web server's document root at runtime. When developing your application, you place these files in your project directory and add them to the project in the Web Server Resources group. When you build your project, Xcode copies the files in this group into the `WebServerResources` folder of your application wrapper.

Frameworks

The Frameworks group contains the frameworks you selected in the assistant when creating your WebObjects project, as well as the frameworks that Xcode adds to your project automatically.

A framework is a collection of classes and resources that multiple applications can use. By storing items such as components and images in frameworks, you can reuse them in multiple projects without having to create multiple copies.

Every WebObjects project includes several frameworks by default depending on the template you select in Xcode. A WebObjects application that uses Enterprise Objects contains these frameworks:

- `JavaEOControl.framework` corresponds to the **control layer** in Enterprise Objects, which provides infrastructure for creating and managing enterprise objects.
- `JavaEOAccess.framework` corresponds to the **access layer** in Enterprise Objects which, provides the data access mechanisms for the Enterprise Objects technology.

Read *WebObjects Enterprise Objects Programming Guide* for more information about Enterprise Objects.

If you selected the JDBC adaptor to access your database, your project contains this framework:

- `JavaJDBCAdaptor.framework` provides an implementation of an Enterprise Objects adaptor for JDBC data sources.

A web application contains these frameworks:

- `JavaFoundation.framework` provides a set of robust and mature core classes, including utility, collection, key-value coding, time and date, notification, and debug logging classes.
- `JavaWebObjects.framework` contains the core web application server, session management, web component, and request-response loop classes.
- `JavaWOExtensions.framework` contains additional reusable web components.
- `JavaXML.framework` contains support for XML content—for example, contains Apache XML parsers.

A Direct to Web application contains these frameworks:

- `JavaDTWGeneration.framework` provides support for generating Direct to Web pages.
- `JavaDirectToWeb.framework` provides classes for rapid development of HTML-based web applications.
- `JavaEOProject.framework` provides services for WebObjects Builder—for example, returns the keys and actions from a web component Java file.

Products

After you build your application, the Products group contains the application wrapper, which is a folder whose name is the project name with a `.woa` extension—for example, `MyWebApp.woa` if the project name is MyWebApp. The application wrapper has a structure similar to that of a framework. It contains the following:

- The executable application—for example, `MyWebApp`.
- The application's resources in the Contents group.

  The Resources group includes the application's web components as well as other files that are needed by your application at runtime.
- The application's web server resources in the Contents group. If your application has no web server resources, then the Web Server Resources group does not appear in the Contents group.

When you build and install your application, Xcode copies all the files from your Web Server Resources group to a folder called `WebServerResources` inside the application wrapper. If you have client-side Java components in your project, these are also copied to the `WebServerResources` folder.

Targets

The targets of a web application are:

- The application—for example, `MyWebApp`.
- `Application Server` builds the part of your application that creates web components and enterprise objects.
- `Web Server` sets up the resources that can be used by the HTTP server, such as images and QuickTime movies not stored in the database.

Building Your Application

Building and running your web application is simple. Just select the application target and click the Build and Go button in Xcode. You use a web browser to run and test your application. For example, if you selected the Direct to Web template, your direct to web application is built and launched. Safari will also launch and connect to your application via the WebObjects application URL.

Installing Your Application

You may wish to install your application on your development machine for testing. Before installing or deploying your application, you should understand how a web server works and where files need to be installed.

Some files in a web application—for example, images and sound files—must be stored under the web server's document root in order for the server to access them. This is because the files are part of the dynamic HTML that the web server sends to web clients. The remaining files—for example, your components and source code—must be accessible by your application but not necessarily by the web server itself. Therefore, when you install or deploy a web application, your product files are split—those files needed by the web server are placed in the document root, and all other files are stored elsewhere. This type of installation is referred to as a **split install**.

Creating Enterprise Objects

If you want to populate your dynamic webpages with content from a back-end data store, the first task is to design your enterprise objects and create your object model. The enterprise objects encapsulate your business data and logic. They are the objects that persist beyond the lifetime of your application. They are also the objects that can be reused in other types of web and non-web applications.

However, before doing so, you should understand a few design patterns that are fundamental to how WebObjects works. Specifically, WebObjects relies on the Model-View-Controller design pattern, object modeling, and key-value coding. These concepts are fundamental to how enterprise objects, dynamic elements, and web components work.

After you understand these concepts, you can create your model and add your business logic. Optionally, create a framework containing your enterprise objects so that you can reuse them in multiple applications. If you are creating a Direct to Web or display group application, then you need to create your EO model first. Then read "Creating Projects" (page 19) for how to create a web application in Xcode.

Model-View-Controller Design Pattern

Many Apple frameworks use a Model-View-Controller (MVC) design pattern including the Application Kit, Core Data, Sync Services, and WebObjects. MVC has been around since the early days of object-oriented programming and is a proven design pattern used to build robust, extensible, and maintainable applications. The design pattern has three components: a model, a view, and a controller.

Models

Models represent special knowledge and expertise. They hold an application's data and define the logic that manipulates that data. A well-designed MVC application has all its important data encapsulated in model objects. Any data that is part of the persistent state of the application should reside in the model objects once the data is loaded into the application. In WebObjects, models are enterprise objects.

Views

Views know how to display and possibly edit data from the application's model. A view should not be responsible for storing the data it displays. A view object can be in charge of displaying just one part of a model object, or a whole model object, or even many different model objects. Views come in many different varieties. In web applications, views are the HTML-based elements and components you use to construct your web component.

Controllers

Controllers act as the intermediary between the application's view objects and its model objects. Typically controller objects have logic in them that is specific to an application. Controllers are often in charge of making sure the views have access to the model objects they need to display and often act as the conduit through which views learn about changes to models. In web applications, high-level controllers are the Session and Application objects. Other examples of controllers are the web component objects and display groups.

Object Modeling

Object modeling is a way of representing objects typically used to describe a data source's data structures in a way that allows those data structures to be mapped to objects in an object-oriented system. It is a representation that facilitates storage and retrieval of objects in a data source. A data source can be a database, a file, a web service, or any other persistent store. Because it is not dependent on any type of data source, it can also be used to represent any kind of object and its relationship to other objects. Object modeling is similar to **entity-relationship modeling**, a popular discipline with a set of rules and terms that are documented in database literature. This section defines **object modeling** terms used throughout WebObjects APIs and tools.

Entities

In the MVC design pattern, models are the objects in your application that encapsulate specified data and provide methods that operate on that data. Models are usually persistent but more importantly, models are not dependent on how the data is displayed to the user.

In the object model, models are called **entities**, the components of an entity are called **attributes**, and the references to other models are called **relationships**. Together, attributes and relationships are known as **properties**. With these three simple building blocks (entities, attributes, and relationships), arbitrarily complex systems can be modeled.

Attributes

Attributes represent structures that contain data. An attribute of an object may be a simple value, such as a scalar—for example, `integer`, `float`, or `double`—but can also be a C structure or an instance of a primitive class. An attribute may correspond to a model's instance variable or accessor method. For example, Employee has `firstName`, `lastName`, and `salary` instance variables.

Relationships

Not all properties of a model are attributes—some properties are **relationships** to other objects. Your application is typically modeled by multiple classes. At runtime, your object model is a collection of related objects that make up an **object graph**. These are typically the persistent objects that your users create and save to some data store. The relationships between these model objects can be traversed at runtime to access the properties of the related objects.

Cardinality

Every relationship has a **cardinality**; the cardinality tells you how many destination objects can (potentially) resolve the relationship. If the destination object is a single object, then the relationship is called a **to-one relationship**. If there may be more than one object in the destination, then the relationship is called a **to-many relationship**.

Mandatory

Relationships can be mandatory or optional. A **mandatory relationship** is one where the destination is required—for example, every employee must be associated with a department. An **optional relationship** is, as the name suggests, optional—for example, not every employee has direct reports.

Ownership

A delete rule is used to specify ownership. You can specify that the destination of a relationship be deleted when the source object is deleted. For example, should an employee be deleted if its department is deleted? What delete rule you use is application-specific.

Key-Value Coding

In order for models, views, and controllers to be independent of each other, you need to be able to access properties in a way that is independent of a model's implementation. This is accomplished by using key-value coding.

Keys

You specify properties of a model using a simple **key**, often a string. The corresponding view or controller uses the key to look up the corresponding attribute **value**. The "value for an attribute" construction enforces the notion that the attribute itself doesn't necessarily contain the data—the value can be indirectly obtained or derived.

Key-value coding is used to perform this lookup—it is a mechanism for accessing an object's properties indirectly and, in certain contexts, automatically. Key-value coding works by using the names of the object's properties—typically its instance variables or accessor methods—as keys to access the values of those properties.

For example, you might obtain the name of a Department object using a `name` key. If the Department object either has an instance variable or method called `name`, then a value for the key can be returned. Similarly, you might obtain Employee attributes using the `firstName`, `lastName`, and `salary` keys.

Values

All values for a particular attribute of a given entity are of the same data type. The data type of an attribute is specified in the declaration of its corresponding instance variable, the return value of its accessor method, or simply in the object model. For example, the data type of the Department object `name` attribute may be an String object in Java. Note that key-value coding returns only object values.

The value of a to-one relationship is simply the destination object of that relationship. For example, the value of the `department` property of an Employee object is a Department object.

The value of a to-many relationship is a collection object (an array) that contains the destination objects of that relationship. For example, the value of the `employees` property of Department object is a collection containing Employee objects.

Key Paths

A **key path** is a string of dot-separated keys that specify a sequence of object properties to traverse. The property of the first key is determined by, and each subsequent key is evaluated relative to, the previous property. Key paths allow you to specify the properties of related objects in a way that is independent of the model implementation. Using key paths you can specify the path through an object graph, of arbitrary depth, to a specific attribute of a related object.

The key-value coding mechanism implements the lookup of a value given a key path similar to key-value pairs. For example, you might access the name of a department via an Employee object using the `department.name` key path where `department` is a relationship of Employee and `name` is an attribute of Department.

Not every relationship in a key path necessarily has a value. For example, the `manager` relationship can be `null` if the employee is the CEO. In this case, the key-value coding mechanism does not break—it simply stops traversing the path and returns an appropriate value, such as `null`.

Enterprise Object Models

Enterprise Objects is a suite of tools and frameworks that allow you to create applications that store your models, called **enterprise objects**, in a database. It is divided into several layers concerned with connecting to the database, converting result sets to enterprise-object instances, and ensuring that the state of the enterprise objects and the database are always synchronized. WebObjects adds many more classes used to manipulate enterprise objects and display their data.

The **enterprise object (EO) model** is a folder added to your Xcode project that defines the mapping between your entities and the tables in the database. It also defines relationships between entities, which are reflected in the database tables using primary and foreign keys.

The EO model maps attributes to table columns for each entity. It also maps Java to database data types. For example, an EO model specifies whether an attribute value of type Number (`java.lang`) is mapped to `int` when an enterprise object is stored in a database.

The EO model contains many more specifications for how enterprise objects are added, stored, fetched, and deleted. The EO model can also specify the information needed to connect to the database, including network and password information.

Creating an EO Model

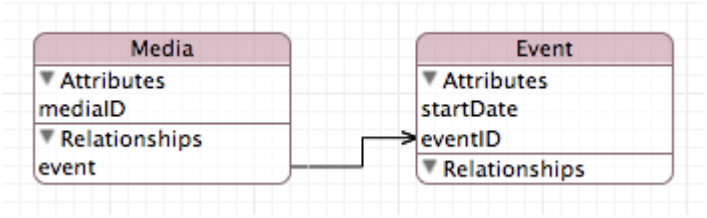
If you do not have an existing EO model or database, you need to design your enterprise objects per your application requirements. You should do some object-oriented analysis and design before creating your first model. Fortunately, WebObjects supports an iterative development cycle so your model can evolve with your application.

You can create your object model using either EOModeler or the EO Model design tool in Xcode. Whatever tool you choose, the steps to create your EO model are similar:

1. Create your model file using the tool.
2. Add entities that represent your objects.
3. Add attributes to each entity.
4. Add primary keys to each entity.
5. Add relationships to each entity.
6. Optionally, add fetch specifications and sort orderings to your entities.
7. Verify your model.
8. Generate your schema.

Figure 1 shows an example of an EO model in the graphical view of the EO Model Xcode design tool. In the example, the Media entity has a to-one relationship to the Event entity and the Event entity has a `startDate` attribute.

Figure 1 Example EO model



If you have an existing database that has a JDBC or other SQL-based interface, you can use EOModeler to create your model directly from the database schema—essentially reverse-engineer your EO model from a legacy database. Read *EOModeler User Guide* for specific steps on how to use EOModeler, and read *Xcode 2.2 User Guide* for how to create a model using Xcode. Read *WebObjects Enterprise Objects Programming Guide* for a deeper understanding of how Enterprise Objects works. Refer to *WebObjects 5.3 Reference* for API details.

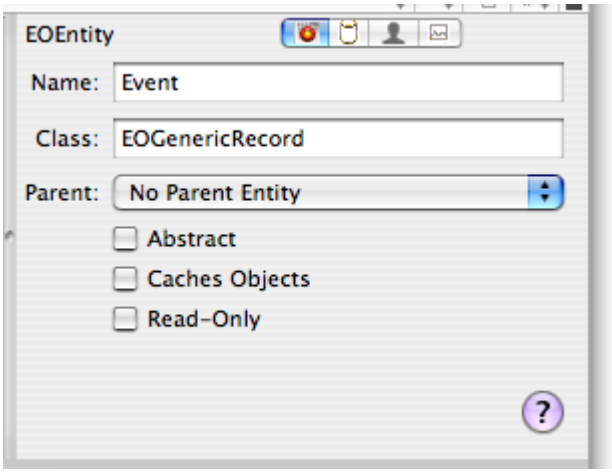
Adding Business Logic

Once you create your EO model, you can add business logic to your enterprise objects. All enterprise objects inherit from a root class that provides common behavior such as support for key-value coding and introspection. You create a subclass of this class to add business logic but need to follow some guidelines so you don't lose enterprise object behavior.

Default EOGenericRecord Class

By default, the instances of entities you create are instances of EOGenericRecord. When you create an entity, the entity's class is set to EOGenericRecord as shown in Figure 2. Some entities never have a specific class. For example, PlotSummary, Review, Movie, and Director in `Movies.eomodeld` are all EOGenericRecord classes.

Figure 2 Creating a new entity



Instances of EOGenericRecord are generic containers—they have attributes and relationships defined by their entity in the EO model but have no special methods that process those properties. Instances of EOGenericRecord and its subclasses simply represent database rows or records. EOGenericRecord is suitable for many entities and saves you time in implementing Java classes with key-value coding compliant methods.

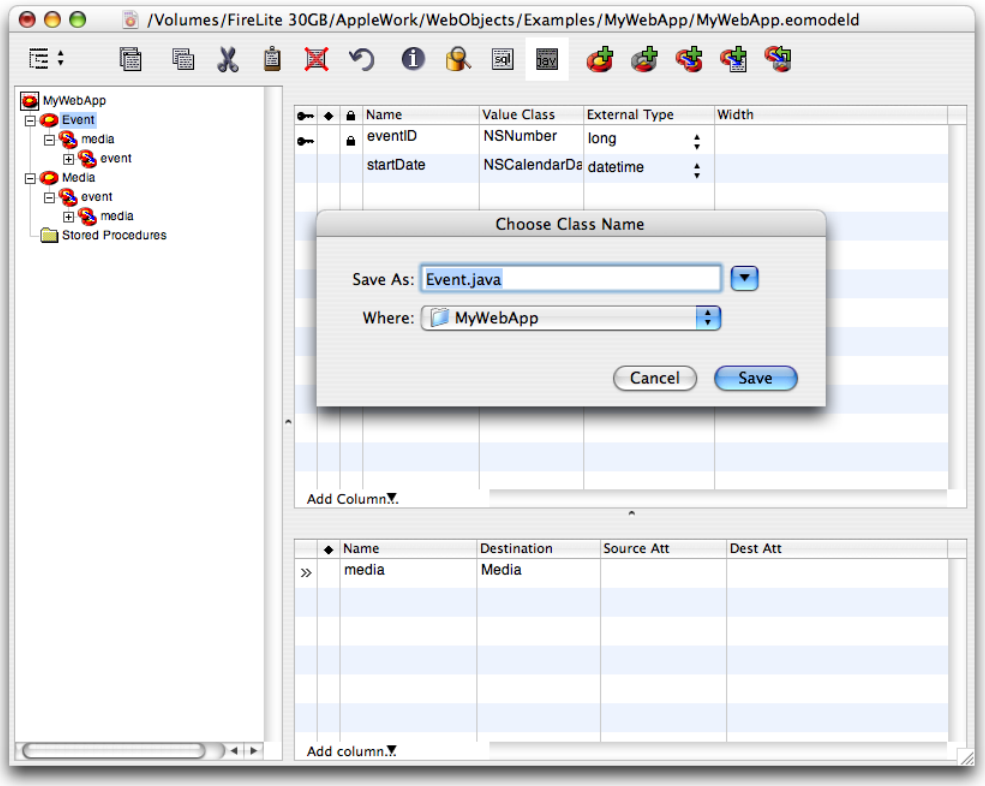
However, if you want to add some business logic or implement derived properties, you need to create a corresponding Java class for your entity. **Derived properties** are properties that are computed at runtime from the values of other properties or the state of your application. For example, you can create a derived property called `fullName` that is a concatenation of `firstName` and `lastName`. To do this, you create a custom enterprise-object class as a subclass of EOGenericRecord, so it inherits the default enterprise-object behavior. Then you add a `fullName` accessor method to the class.

EOGenericRecord uses the key-value coding mechanism to store entity properties. Each key is named for the database column it represents. When an enterprise object is instantiated from a row in the database, the values of its keys are obtained from their corresponding columns in the row. WebObjects dynamic elements use key-value coding to get and set the values of enterprise-object attributes.

Subclassing EOGenericRecord

EOModeler provides an easy way to create a custom enterprise object, a Java class for your entity. Just select the entity in EOModeler, enter a class name in the inspector window, and click the jav button. The Java source files is added to your Xcode project as shown in Figure 3.

Figure 3 Generating Java source code



The class created by EOModeler is a subclass of EOGenericRecord. The subclass has no instance variables although properties may be defined in EOModeler. Instead, property values are accessed using key-value coding methods: `valueForKey()` and `takeValueForKey()`. By using these key-value coding methods to modify properties of a custom enterprise object, you ensure that your changes are stored in the database and all controller objects are notified of changes.

See *WebObjects 5.3 Reference* for the order in which NSKeyValueCoding searches for an accessor method. Read *EOModeler User Guide* for complete instructions on how to use EOModeler to generate Java code.

Creating Frameworks

You might create a framework using Xcode to contain your EO model and any custom enterprise object classes. You especially need to do this if you plan to reuse your enterprise objects in multiple applications. You can use the same EO model and business logic to implement any other type of WebObjects applications—for example, Direct to Web, Direct to Java Client, and Direct to Web Services. You can use these

prototypes to evaluate different approaches or simply to verify and test your enterprise objects. It is not uncommon to maintain two WebObjects applications in parallel. Read "Creating Projects" (page 19) for how to create a WebObjects framework.

Creating Web Components

Web components are fundamental to how dynamic content works in WebObjects. Typically, you choose WebObjects if the information on your website changes frequently or varies based on some conditions. Examples of dynamic websites include online news, stores, polls and statistics. WebObjects is also ideal for any website that tracks user sessions and offers personal services such as authoring content and custom pages. Dynamic content can be tuned to user preferences and search criteria.

You use web components to represent webpages or partial webpages generated by your website. Web components are actually templates for generating HTML pages. Web components are constructed from static and dynamic elements. You use dynamic elements to bind HTML counterparts to variables and methods in your web component class. Some elements are abstract and are used just to control the generation of HTML—for example, conditionals and repetitions.

Although you can create a web application without using Enterprise Objects, typically, your website renders HTML pages that are populated with data obtained from your enterprise objects stored in a back-end database. Hence, web components behave similarly to controllers in the MVC design pattern by being the intermediary between the views (dynamic elements) and your models (enterprise objects). You can also use abstract elements and display groups—controllers that manipulate many enterprise objects—in interesting ways to create smart webpages.

Web components benefit from all the advantages of object-oriented systems. Web components are reusable, extensible, and maintainable. Web components can contain other web components–those that represent partial pages—as well as dynamic elements, static elements, and plain text. Any HTML tag can be added to a web component's HTML template.

Web components are folders you add to your Xcode project. Each folder contains an HTML, WOD, API and Java file. The HTML file represents the template, the WOD file contains the dynamic element bindings, the API file contains any bindings that your component exports, and the Java file is the class that implements controller logic. You add your controller logic—variables and methods—to the Java class using Xcode. You can edit the other files directly but typically, you use WebObjects Builder, a graphical editor, to design your web components. WebObjects Builder creates the files located in the web component folder.

This section describes how to reuse and extend web components from a Java programmer's perspective. For a complete guide on how to create web components graphically, read *WebObjects Builder User Guide*.

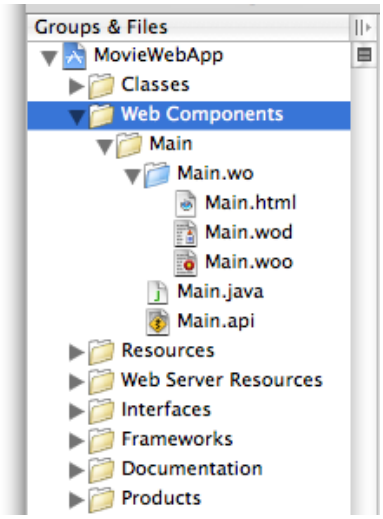
If you use a back-end database to store your enterprise objects, you should create your EO Model first and then your Xcode project before you create web components. Read "Creating Enterprise Objects" (page 29) to create an EO model and custom enterprise objects. Read "Creating Projects" (page 19) for how to create an Xcode project.

Main Component

By default, every WebObjects application includes a Main component. This component, initially empty, is the first page displayed to users unless you specify otherwise. It is the tunnel or login page for the rest of your application.

For example, if you select the Web Applications template when creating your Xcode project, `Main.wo` appears in the Web Components group as shown in Figure 1. Double-click `Main.wo` to open the Main component in WebObjects Builder.

Figure 1 Web component files



Java Files

Every web component contains a Java file that you use to add controller and business logic.

For example, if your Main component is a login page, it might contain a user and password field to allow the user to log in to your website, then you might have `username` and `password` variables to store the text that the user enters. You also need an action method when the user enters Return or clicks the Login button. You add these variables and methods to `Main.java` which is in the Web Components group, as shown in Figure 1 (page 38). Then you bind the dynamic elements, the WOTextField, WOPasswordField, and WOSubmitButton elements, to your variables and methods using WebObjects Builder.

Alternatively, you can add variables and methods to your component using WebObjects Builder which will edit the Java file for you.

HTML and WOD Files

Occasionally, you might need to edit one of the web component files directly. However, since these files are created by WebObjects Builder, you should be aware of their file formats before doing so. If you change the format of these files, the component may not validate or open in WebObjects Builder.

For example, suppose you create a main component using WebObjects Builder that contains a text string, "The current time is", followed by a WOString element that displays the current time. In addition, you bind the WOString element to the `currentTime` method in your main component that returns the current time. This information is stored in the web component's HTML and WOD files that you can view in Xcode.

The HTML file shown in Listing 1, contains a `WEBOBJECT` element that represents the location where the WOString inserts the value returned by the `currentTime` method. Notice that the element is defined as `<WEBOBJECT NAME=String1></WEBOBJECT>`. There's a corresponding entry in the WOD file that uses the same name, `String1`.

Listing 1 Sample HTML file

```
<BODY>
    The current time is <WEBOBJECT NAME=String1></WEBOBJECT>
</BODY>
```

The connection between the WOString element and the `currentTime` method is declared in the WOD file shown in Listing 2. The entry has only one binding listed, the connection between the `value` attribute and the `currentTime` method. This method is called whenever the WOString element needs to display its value. The WOD file can contain other element settings that do not appear in WebObjects Builder.

Listing 2 Sample WOD file

```
String1: WOString {
    value = currentTime;
}
```

How Dynamic Elements Work

When programming with web components, it helps to understand how dynamic elements work in the context of the request-response loop.

When you run a simple application that displays the current time on the main page as described in "HTML and WOD Files" (page 38), the page displayed by the web browser replaces the WOString element you added to the Main component with the current time. If you reload the page, the time gets updated. WebObjects assembles the page dynamically during the request-response loop.

When you access the URL corresponding to your application in a web browser, the web server hands control to the **WebObjects adaptor**—a process that connects WebObjects application instances to web servers. This program goes through a couple of steps in generating the response:

1. Read the HTML file

   Much like a regular web server, WebObjects first reads an HTML file.

2. Render WebObjects elements

   Unlike a regular web server, WebObjects parses `WEBOBJECT` elements before handing the file to the web server.

   When WebObjects encounters a `WEBOBJECT` element, it consults the WOD file for the corresponding web component. All the `WEBOBJECT` elements in the HTML file of a web component template are named, and each one is listed by its name in the WOD file as shown in Listing 2 (page 39).

   Each type of WebObjects element has special logic for constructing the HTML code to return to the web server. Customization of this process is done with attributes defined by the web component's developer. Each binding in a WOD file can be either static or dynamic. If a binding is static, the value supplied is used directly.

If a binding is dynamic—that is, an attribute is bound to a method or instance variable—WebObjects invokes the method or accesses the instance variable to obtain the value at runtime. For example, when the WOString element is evaluated, it invokes the method named in its `value` binding, `currentTime`, to get the value to display. The implementation of WOString turns the NSTimestamp object into text that's incorporated into the webpage returned to the web browser. Before returning the webpage to the browser, WebObjects converts the component's HTML code shown in Listing 3 to a markup similar to the one shown in Listing 4.

Listing 3 HTML code interpreted by WebObjects

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 3.2//EN">
<HTML>
    <HEAD>
        <META NAME="generator" CONTENT="WebObjects 5">
        <TITLE>Untitled</TITLE>
    </HEAD>
    <BODY>
        The current time is <WEBOBJECT NAME=String1></WEBOBJECT>
    </BODY>
</HTML>
```

Listing 4 HTML code WebObjects sends to web browser

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 3.2//EN">
<HTML>
    <HEAD>
        <META NAME="generator" CONTENT="WebObjects 5">
        <TITLE>Untitled</TITLE>
    </HEAD>
    <BODY>
        The current time is 01:19:47 PM
    </BODY>
</HTML>
```

This process takes place each time a web browser requests the Main page. If you reload the page, the method is invoked again and a new time value is displayed.

For more information on the request-response loop, read "Request-Response Loop" (page 11).

Maintaining State

Understanding the connection between a web component's HTML, WOD, and Java files is an important part of WebObjects development. Not only do you add variables and methods to your component to bind dynamic elements, but you can also add variables and methods to maintain state of a component.

When you add methods to a component in WebObjects Builder, you are actually editing the component's Java source file. When you modify how the component looks by adding elements, you modify its HTML file. When you bind dynamic elements to your component variables, you modify its WOD file.

Typically, after using WebObjects Builder to define the major parts of a web component, you can add details by editing the HTML and Java files directly (you rarely need to edit a WOD file). To edit the HTML file of a web component in WebObjects Builder, simply use the Source mode—click the Source button on the toolbar to use the HTML editor. You can edit the Java file using Xcode.

When you deploy your application and users connect to your website via a web browser, webpages are created from web components as needed. In programming terms, web components are all subclasses of WOComponent and instantiated as needed. For example, when the user requests the main component described in "HTML and WOD Files" (page 38), a Main instance is created. When it's time for WebObjects to add the content for the WOString, it looks up the element's `value` binding in the WOD file. The `value` binding is set to the `currentTime` method. Therefore, WebObjects sends `currentTime` to the web component instance, which returns the current time.

An instance of a web component lives for at least two cycles of the request-response loop: In the first cycle the webpage is rendered, and in the second cycle the component determines which page to display next. If the page is not the same as the previous page, WebObjects creates an instance of the new component. The old component is then discarded or stored in a server cache to allow users to backtrack to previous pages. However, if the component to display is the same, the instance lives on. In this case, the previous version of the component is stored in the backtracking cache. Read "Backtracking and Cache Management" (page 51) for more information on backtracking.

Another way to use variables and methods in your Java source file is to maintain other state—for example, tracking user actions as they interact with your application.

Example: Displaying the Page Count

The following sections show how to maintain state in your web component by implementing a simple web application that counts the number of times the main component is displayed by a single user. This example adds a counter and page refresh hyperlink to the main component. It uses the web component to maintain the state of the counter for the duration of the session.

Create Your Project

Create a simple web application as follows:

1. Launch Xcode and choose File > New Project.
2. Select the WebObjects Application template and click Next.
3. Click Next on all the remaining assistant panes to select all the default values.
4. Click Finish on the final pane to create the project.

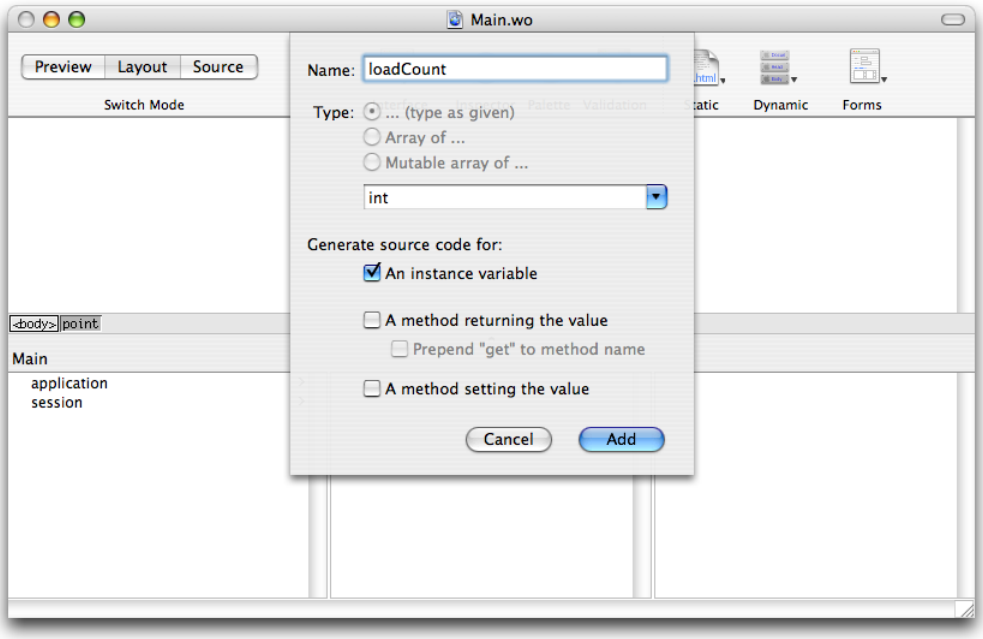
Add Variables to Your Component

First you add a variable and code to the Main component to increment a counter each time the page is displayed.

1. Open `Main.wo` in WebObjects Builder by double-clicking it in Xcode.
2. Choose Add Key from the Interface menu on the toolbar of the `Main.wo` window.

3. Add a key of type `int` named `loadCount` and click the Add button as shown in Figure 2.

Figure 2 Adding a key



4. Examine the `Main.java` file in Xcode to confirm that the variable was added. Modify the code to initialize `loadCount` to `1` as shown in Listing 5:

Listing 5 Adding a variable

```
public class Main extends WOComponent {
        public int loadCount = 1;

    public Main(WOContext context) {
        super(context);
    }
}
```

Add Dynamic Elements to Your Component

Next you use the variable to display the number of times the page is loaded. To display the load count in the webpage, you need to add a WOString element to the Main component using WebObjects Builder.

1. Add a label and a WOString element to `Main.wo`.

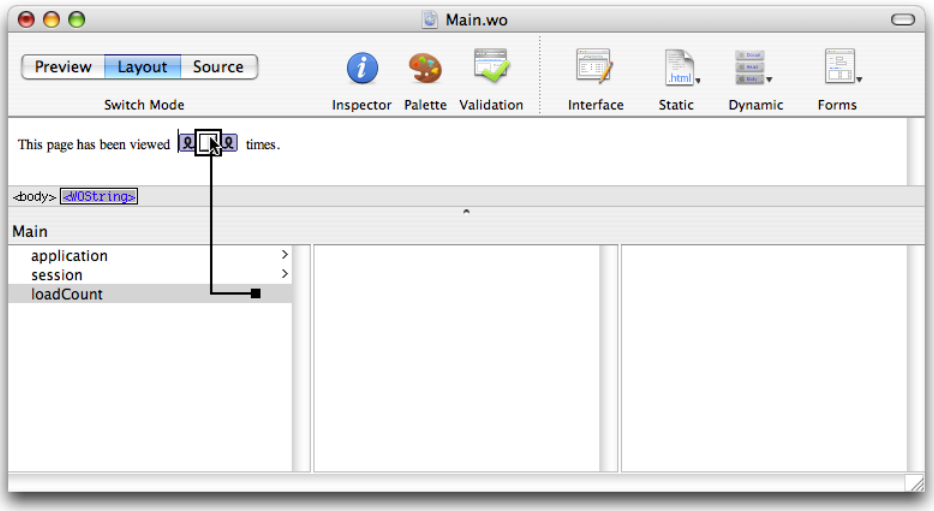
    a. Enter `This page has been viewed`.

    b. Add a space and a WOString element to the right of the label.

    c. Add a space and `times.` to the right of the WOString element.

2. Drag from `loadCount` in the object browser to the WOString element to bind it to the `value` attribute of the WOString element as shown in Figure 3.

Figure 3 Binding a WOString



Add Methods to Your Component

Now, you need to add a way to reload the page using a hyperlink.

In WebObjects, regular hyperlinks, WOHyperlink elements, can call web component methods—methods defined in the component's Java source file. These methods are called **action methods**. All action methods return a web component representing the next page. If an action method returns `null`, then the same page is redisplayed. Action methods are covered in greater detail in "Processing the Request" (page 14).

Follow these steps to add a `refreshTime` action method:

1. Add the action method.

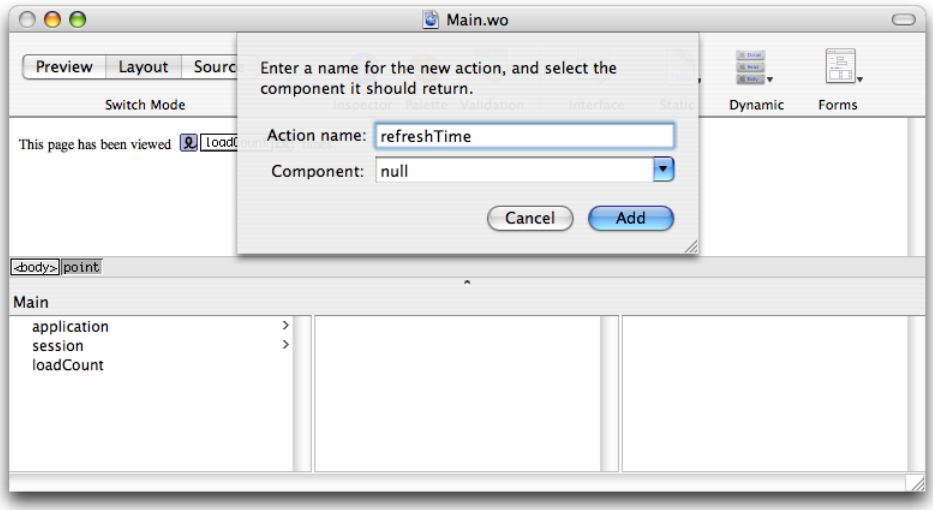
   Open the Main component template in WebObjects Builder and choose Add Action from the Interface menu on the toolbar.

   a. Name the action `refreshTime`.

   b. Select `null` from the Component pop-up menu as shown in Figure 4.

The value returned by an action method represents the next page (web component) to be displayed. When you return `null`, the current page is redrawn.

Figure 4 Adding an action



c. Click Add.

2. Add a hyperlink.

Position the cursor below the line where the load count is displayed.

Choose Dynamic > WOHyperlink.

By default, the text for a new link is `Hyperlink`. You can replace this by selecting the text and typing something more appropriate over it, such as `Refresh Time`.

3. Connect the `refreshTime` method to the WOHyperlink element.

Much like a WOString element, a WOHyperlink element has several attributes. Bind the `refreshTime` method to the `action` attribute of WOHyperlink.

Drag from the `refreshTime` method in the Main list to the WOHyperlink element. When you release the mouse button, a pop-up list of attributes appears. Choose the `action` attribute to indicate that you want the `refreshTime` method called when the link is clicked.

4. Save `Main.wo`.

Add Logic to Your Methods

Finally, modify the `refreshTime` method using Xcode so that it increments the `loadCount` variable each time it is invoked as shown in Listing 6.

Listing 6 Implementing an action method

```
public WOComponent refreshTime() {
    loadCount++;
    return null;
}
```

How Maintaining State Works

If you build and run the application created in "Example: Displaying the Page Count" (page 41), and click Refresh Time, the time and the load count are updated. The load count increments every time a specific user displays the page. This is accomplished using the Session object, an object that represents a connection between an application and a particular client.

Each time the user clicks the hyperlink WebObjects creates a Main object and associates it with your web browser window through a session object. Each time you interact with the application, by clicking Refresh Time, the same Main object is used. If you open another browser window and connect to the application again using the URL shown in Xcode's Run pane, a separate instance of Main is created and associated with that window. From then on, you can work with both windows individually. As a matter of fact, not only is a new instance of Main created, a new Session object is created as well.

WebObjects determines that a new session needs to be created when the incoming URL does not contain a session ID. The first time you connect to the application using a URL like the one in Listing 7, WebObjects creates a session and assigns it a session ID and other information. That information is added to the URL returned to your browser together with the webpage to be displayed (see Listing 8). When you send another request from your browser—for example, by clicking Refresh Time—WebObjects uses the session ID encoded in the URL to locate the session that will process the request. This is the default mechanism WebObjects uses to keep track of the state of each user.

For more on managing state using application and session objects, read "Using the Application and Session Objects" (page 47).

Listing 7 URL that causes the instantiation of a Session object

```
http://foo.com:49361/cgi-bin/WebObjects/WebApp
```

Listing 8 URL with session ID

```
http://foo.com:49361/cgi-bin/WebObjects/WebApp.woa/wo/whcV5sauLNtG8Tfh6xCuvM/0.1
```

Using the Application and Session Objects

The web—by its nature—is a stateless medium. A web server receives a request, produces a response, and returns it to the requesting web browser—without any knowledge of previous requests from the same user.

A web application, however, can maintain state between requests from the same user to provide an acceptable user experience. For example, many websites allow you to purchase items using a shopping cart. Such applications have to remember the contents of your shopping cart as you navigate the website. WebObjects encodes a unique identifier with each incoming request. This identifier is used to maintain state over an otherwise stateless medium. Read "How Web Applications Work" (page 9) for more information on responding to requests.

While you can pass information back and forth between web components, you frequently need to maintain state that is shared between web components. Rather than pass this information from web component to component, you can store it at a higher level per application or per session.

The Application

When you create your application using one of the web application Xcode templates, `Application.java` is added to your project. Application is a subclass of WOApplication. WebObjects instantiates an Application object at startup. Every web component in your application has a relationship to this Application object—send `application()` to a web component to get the Application object programmatically. The application and session objects also appear in the object browser in WebObjects Builder so that you can bind dynamic elements directly to their properties.

You can override methods inherited from WOApplication to customize the behavior of your web application. For example, you can invoke WOApplication methods from the initializer to change the default behavior of backtracking (read "Backtracking and Cache Management" (page 51) for details).

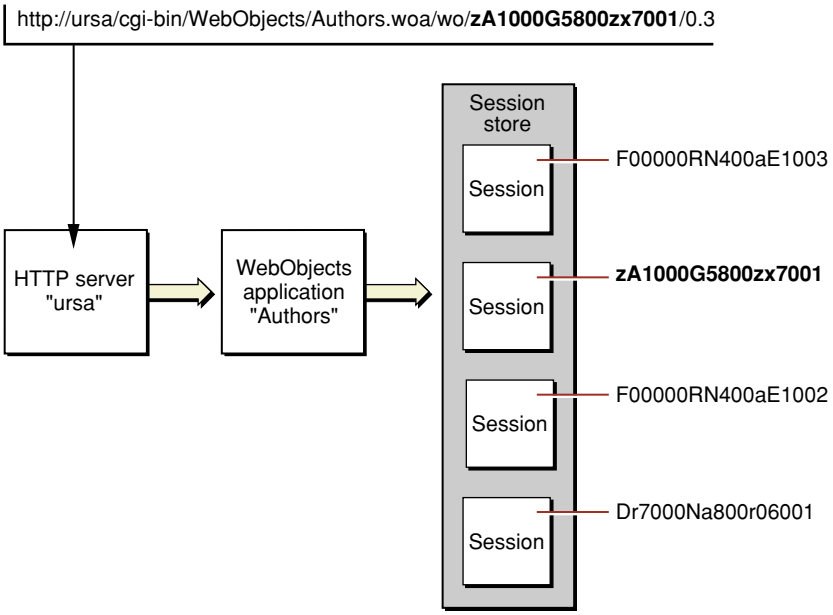
You can also add variables and methods to the Application class to store objects and add business logic that you want to share between sessions and web components. For example, any objects that are read-only and shared by sessions, can be stored in the Application object to improve performance.

However, use the EOSharedEditingContext, not the Application class, if you want to share enterprise objects. See *WebObjects Enterprise Objects Programming Guide* and *EOModeler User Guide* for how to configure a fetch specification in your model that places enterprise objects in the shared editing context.

The Session

A **session** is a period of time in which one user interacts with your application. Since each application can have multiple users simultaneously, it may have multiple open sessions. Each session has its own data and its own cached copies of the components that the user requests, as shown in Figure 1.

Figure 1 Relationship between application and session



The session is represented as an instance of the Session class (`Session.java` in a WebObjects application project). Session is a subclass of WOSession. Initially, Session has only inherited behavior, but you can add custom methods and variables. For example, if you are building an online shopping application, the session would be an appropriate place to store a user's shopping cart, because the session is tied to one particular user and persists as long as the user utilizes the application.

When an incoming request is processed, WebObjects automatically activates the Session object associated with the user who originated the request, as described in "Request-Response Loop" (page 11).

The WOComponent class includes a method for accessing the currently active session. The Java classes of web components are subclasses of WOComponent and WebObjects automatically activates the correct session when a request is processed. Sending the `session()` message to a WOComponent object returns the Session object for the current user.

Shopping Cart Example

If you are implementing an online store then your users need a shopping cart to store their purchases before they checkout. Since a shopping cart belongs to a single user and is only valid during the lifetime of a session, it is reasonable to store the shopping cart in the Session object. For example, the Pet Store example located in `/Developer/Examples/JavaWebObjects/PetStoreWOJava` adds `currentAccount` and `cart` attributes to the Session class as shown in Listing 1 (page 48).

Listing 1 Pet Store Session Class

```
public class Session extends WOSession {

    Account account;
    Cart cart;
```

```
    public Session() {
        super();
        cart = new Cart();
    }

   public Account currentAccount() {
       return account;
   }

   public void setCurrentAccount(Account newAccount) {
       account = newAccount;
   }

    public Cart cart() {
        return cart;
    }
}
```

Backtracking and Cache Management

Backtracking, client-side page caching, and web component caching are three closely related issues that cause many headaches for web application developers. Fortunately, WebObjects offers a number of mechanisms that help you deal with the collective problem of managing page state.

Dynamic web applications are possible because of, among other things, server-side state persistence and state management. HTTP, the protocol of the web, is inherently stateless. However, storing state in an application server makes persistence management in web applications possible. In WebObjects, the Session object holds state but is not solely responsible for state management. The Session object tracks sessions, flags WOComponent and WOElement objects with special identifiers, and uses other mechanisms to hold and manage state. WOComponent objects manage the state of their internal instance variables and dynamic elements.

Along with these mechanisms, caching plays an important role in managing the state of visual components. Caching allows a user to view a previously viewed webpage (even a dynamically generated one) without the application needing to regenerate the page. Caching also plays a crucial role in providing a good user experience in web applications. Caching lets users backtrack using their web browser's Back button, which often allows for instantaneous loading of pages from the client-side cache rather than requesting a previously viewed page from the application server. However, because there are diverse implementations of the HTTP protocol in web browsers, backtracking behavior is inconsistent and requires considerable attention when developing web applications.

In addition to client-side page caching, WebObjects also caches components in a server-side cache. If used correctly, this is a valuable feature that can improve performance and user experience. But you must be conscious of the relationship between server-side component caching and client-side page caching, and how inconsistencies in backtracking behavior affect the result when either or both caching features are active.

Client-Side Page Cache

A web component is the aggregate of WebObjects elements and subcomponents. When a web browser caches a webpage from a WebObjects application, it caches the static HTML code of a generated page (which does not include a web component's programmatic entities, such as instance variables). In contrast, server-side component caching caches a web component's definition and state.

Client-side page caching is a feature implemented by web browsers to improve performance and user experience. Although WebObjects applications primarily publish dynamic webpages, many websites serve static pages: They do not change as rapidly as content-driven dynamic sites.

For instance, consider a website that publishes news stories and other articles. Although the front page of the site probably changes a few times each day, it likely would not change in the few minutes an average user spends browsing headlines and reading a few articles.

With client-side page caching active, the front page of the news website is cached on the client's computer upon the first visit. The first page could be large, containing images, banner ads, and text. The user could select an article, read part of it, and access other articles through URLs in the first article. Then, having visited

five or six pages within the website, the user could backtrack to the main page. Since the content of that page is not likely to change in the time the user took to peruse the five or six pages, the page should be reloaded from the local cache. So the web browser—instead of requesting and downloading the main page from the web server again—would retrieve it from the local cache, avoiding a round trip over the network to the web server. In this case, page caching serves a sensible and user-friendly function.

Now, consider the case of an online store: A user chooses items to buy and adds them to a shopping cart. It's generally not a good idea for the user to view a cached webpage representing the shopping cart as it likely does not contain the most up-to-date information. If client-side page caching is active, however, this is a real possibility.

WebObjects offers a number of mechanisms to deal with the problems of backtracking and client-side caching. The first one you should use is a flag on the Application object that you set using the `setPageRefreshOnBacktrackEnabled` method of the WOApplication class (`com.webobjects.appserver`). When `pageRefreshOnBacktrackEnabled` is `true`, a number of HTTP headers are added to each response generated by the WebObjects application to disable client-side page caching. Table 1 shows these headers and their values.

Table 1 HTTP response headers that deactivate client-side page caching

Header	Value
`date`	*The time the response page was generated.*
`expires`	*The time the response is to expire.* (Same as `date`.)
`pragma`	`no-cache`
`cache-control`	`private`, `no-cache`, `no-store`, `must-revalidate`, `max-age = 0`

See section 14.9 of the HTTP 1.1 specification (RFC 2616) for more details on each of these headers.

The `pageRefreshOnBacktrackEnabled` property affects all responses generated by an application. If you want to restrict the behavior to a specific response, invoke the `disableClientCaching` method of the WOResponse object (`com.webobjects.appserver`). WOResponse also includes the methods `setHeader` and `setHeaders`, which allow you to explicitly set the HTTP headers for a particular response.

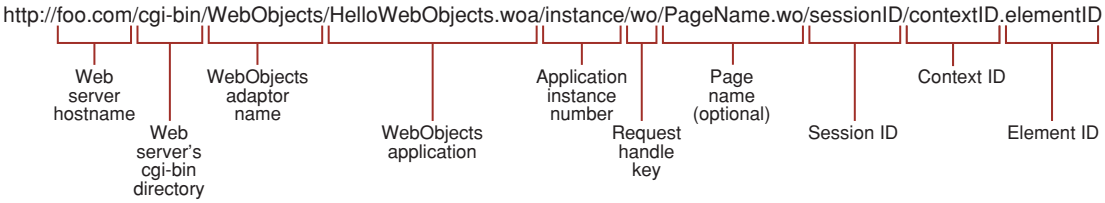
When a web browser receives a response page with the headers shown in Table 1, it should not add the page to its local cache and it should invalidate the page as soon as it is displayed. In other words, when users backtrack to retrieve previously viewed pages, the web browser should request the response page from the application server. However, not all web browsers follow this protocol, as demonstrated in "Web Browser Backtracking Behavior" (page 54). The first few times the user backtracks to previously viewed pages, most web browsers ignore the HTTP headers and render the page stored in the cache.

When a web browser needs to refresh an expired page, it sends a request to the application server, which accesses the server-side cache to reconstruct the page (see "Server-Side Page Cache" (page 53) for more information on server-side caching). "Request-Response Loop Messages" (page 13) explains the phases of the request-response loop in detail. The main phases are sync, action, and response. When processing a refresh request, an application does not go through the sync and action phases; it performs only the response phase.

So how does an application know to perform only the response phase (just returning the response page stored in the server cache, rather than regenerating it)? WebObjects assigns each response a context ID. The context ID is increased by 1 each time a web browser requests a specific page from the application server

during a session. It identifies a specific instance of the corresponding WOComponent. (Figure 1 shows the elements of a WebObjects URL.) Specifically, an application assigns the outermost component of a WOComponent a context ID each time that component is part of a response. So, if the same component is dynamically generated multiple times, each instance of the page (each response) is assigned a unique context ID.

Figure 1 Structure of a component action URL



Server-Side Component Definition Cache

When a web component is accessed for the first time, its definition is placed on the server-side cache. Subsequent requests for the same component use the definition stored in the cache. Using the web component cache improves performance because the application looks up a component's definition only one time during the lifetime of the application. You can control web component caching at the application level and the component level. You can set a caching policy for the application (either active or inactive) for all components, but you can also override such policy on specific components. To set the caching policy for an application or a web component you use the `setCachingEnabled` method of WOApplication or WOComponent, respectively. Sending `true` as the argument activates web component–definition caching, while sending `false` deactivates it.

Server-Side Page Cache

In addition to component-definition caching, WebObjects applications can also cache responses sent to a client. When an already-generated page is requested from the application server, WebObjects checks the context ID of the requested page with the context ID of pages in its cache. If it finds a match, it performs the response phase of the request-response loop. This returns a response that has a new context ID and updated content from the invocation of the response phase of the request-response loop (dynamic bindings are again resolved in the response phase).

By default, the WebObjects application server maintains a page cache for each session. Each page a user accesses is added to the session's page cache. When a user backtracks, accesses a URL, or selects a bookmark of a page that is cached but expired in the local cache, the web browser requests a refreshed version of that page from the application server. The server-side page cache preserves resources as it hands out the result of previously generated pages. When the page the user backtracks to is no longer in the cache, WebObjects returns an error page.

If you deactivate the server-side page cache (by passing `0` to the `setPageCacheSize` method of WOApplication), the application assumes that you intend to provide custom component state persistence rather than rely on WebObjects inherent support. Deactivating the component cache means that new WOComponent objects are instantiated (that is, each request for a component creates a new instance of that component) with each cycle of the request-response loop, even for component action requests that return

the invoking page. This means that any nondefault instance variable values are discarded with each subsequent cycle of the request-response loop. In large applications, this redundancy and overhead could hinder performance.

WebObjects also provides a permanent page cache that is useful for storing subcomponents such as navigation bars or page headers, or when using frame sets. You have to explicitly add components to it using the `savePageInPermanentCache` method of WOSession (`com.webobjects.appserver`). Read *WebObjects 5.3 Reference* for details.

Web Browser Backtracking Behavior

To better understand the concepts of backtracking, client-side page caching, and component-definition caching, perform the tasks described in the following sections.

Viewing the HTML Headers

Open the web application you developed in "Creating Web Components" (page 37) or any other simple web application.

In `Main.java`, add a method called `outgoingHeaders`:

```
public String outgoingHeaders() {
    return context().response().headers().toString();
}
```

This gets the headers that are attached to each outgoing WOResponse object. To view these headers, override the `sleep` method in the Main class so that it prints the headers to the console:

```
public void sleep() {
    System.out.println("<Main.sleep> headers=" + outgoingHeaders());
}
```

Build and run the application. You should see output similar to this in the console:

```
Welcome to WebApp!
[2003-01-08 17:53:56 PST] <main> Opening application's URL in browser:
http://17.203.33.19:8888/cgi-bin/WebObjects/WebApp.woa
[2003-01-08 17:53:56 PST] <main> Waiting for requests...
<Main.sleep> headers={cache-control = ("private", "no-cache", "no-store",
"must-revalidate", "max-age=0");
 expires = ("Thu, 09-Jan-2003 01:53:54 GMT"); date = ("Thu, 09-Jan-2003 01:53:54
 GMT"); pragma = ("no-
cache"); content-type = ("text/html"); }
```

The `expires` header is set to the time the component is generated, so that when the web browser receives the webpage, it is already expired in the web browser's cache. These headers (except `content-type`) are appended to the response when the `isPageRefreshOnBacktrackEnabled` method of WOApplication returns `true`, which it does by default.

In `Application.java`, set the `pageRefreshOnBacktrackEnabled` property to `false` in the constructor:

```
public Application() {
```

```
    super();
    System.out.println("Welcome to " + this.name() + "!");
    setPageRefreshOnBacktrackEnabled(false);
}
```

Build and run the application. You should see output similar to the following in the console:

```
Welcome to WebApp!
[2003-01-08 17:57:15 PST] <main> Opening application's URL in browser:
http://17.203.33.19:8888/cgi-bin/WebObjects/WebApp.woa
[2003-01-08 17:57:15 PST] <main> Waiting for requests...
<Main.sleep> headers={content-type = ("text/html"); }
```

Notice that the headers disabling client-side caching are not generated in the response.

Standard Webpage Backtracking

So, how does the `pageRefreshOnBacktrackEnabled` property of WOApplication affect user backtracking? You need to add some more code to trace what WebObjects does behind the scenes. Modify the constructor in the Main class to look like this:

```
public Main(WOContext context) {
    super(context);
    System.out.println("<Main> context ID="+ context().contextID());
}
```

Each time an instance of Main is created, this code outputs the context ID of the WOResponse object associated with the new instance. This allows you to see when user actions like clicking the Refresh hyperlink on the webpage or the web browser's Back button produce a new instance of the Main component. While this is useful information, you may also want to know when a user action causes the application to send a new response page to the client web browser. You can trace this by adding similar code to the `refreshTime` method:

```
public WOComponent refreshTime() {
    System.out.println("<Main.refresh> context ID=" + context().contextID());
    loadCount++;
    return null;
}
```

Now, remove the `sleep` and `outgoingHeaders` methods and build and run the application.

Click Refresh Time three times. This prints the incremental context ID of the instance of Main through which you navigate. When you click Refresh Time, the application invokes the `refreshTime` method, which outputs the context ID of the outgoing response to the console:

```
Welcome to WebApp!
[2003-01-08 18:56:18 PST] <main> Opening application's URL in browser:
http://17.203.33.19:8888/cgi-bin/WebObjects/WebApp.woa
[2003-01-08 18:56:18 PST] <main> Waiting for requests...
<Main> context ID=0
<Main.refreshTime> context ID: 1
<Main.refreshTime> context ID: 2
<Main.refreshTime> context ID: 3
```

Now, click your browser's Back button three times. Notice that nothing is printed to the console. This is because, when `pageRefreshOnBacktrackEnabled` is set to `false`, backtracking does not result in a request to the application; the page is simply rendered using the copy in the browser's cache. Similarly, choosing the bookmark of a page cached in the web browser does not result in a request to the application.

Refreshing Pages When Backtracking

When `pageRefreshOnBacktrackEnabled` is set to `true`, backtracking should result in a request to the application (you should see a context ID line with a new context ID) when a user backtracks, although the actual behavior differs among various web browsers.

In Mac OS X, web browsers that use the Gecko HTML rendering engine (such as Chimera and Mozilla), comply most closely to the HTTP specification. Clicking the Back button causes the browser to ask for an updated version of an expired webpage. Other browsers, such as Internet Explorer and OmniWeb, behave differently: The first few clicks (two to three, depending on the browser) of the Back button reload the page from the cache. Subsequent clicks cause the browser to send a request to the application.

Notice that when the browser requests the updated version of the webpage from the application, the page-load counter doesn't decrease, but the time is updated.

You must test your application on many configurations to ensure that it provides a good user experience.

Disallowing Server-Side Caching

A WebObjects application can hand back only the response of a previously generated page when server-side page caching is active, which is the default. When this feature is inactive, the `println` statement in the constructor of the Main class (of the web application described earlier in this article) is invoked each time you click the Refresh Time link. This indicates that the application instantiates a Main object each time the `refreshTime` method of Main is invoked, instead of returning the current Main object.

Modify the constructor in the Main class by adding a call to `setPageCacheSize`:

```
public Application() {
    super();
    System.out.println("Welcome to " + this.name() + "!");
    setPageRefreshOnBacktrackEnabled(true);
    setPageCacheSize(0);
}
```

Build and run the application. After clicking Refresh Time three times, you should see the following console output:

```
Welcome to WebApp!
[2003-01-08 20:31:58 PST] <main> Opening application's URL in browser:
http://17.203.33.19:8888/cgi-bin/WebObjects/WebApp.woa
[2003-01-08 20:31:57 PST] <main> Waiting for requests...
<Main> context ID=0
<Main> context ID=1
<Main.refreshTime> context ID: 1
<Main> context ID=2
<Main.refreshTime> context ID: 2
<Main> context ID=3
<Main.refreshTime> context ID: 3
```

Notice that the constructor in the Main class is invoked each time you click Refresh Time, before the `refreshTime` method is executed. An instance of Main is created during each cycle of the request-response loop. Also notice that the page-view counter does not increase. The primary consequence of deactivating server-side page caching is that the values of variables in components are lost after each response is generated.

Setting the Size of the Server-Side Cache

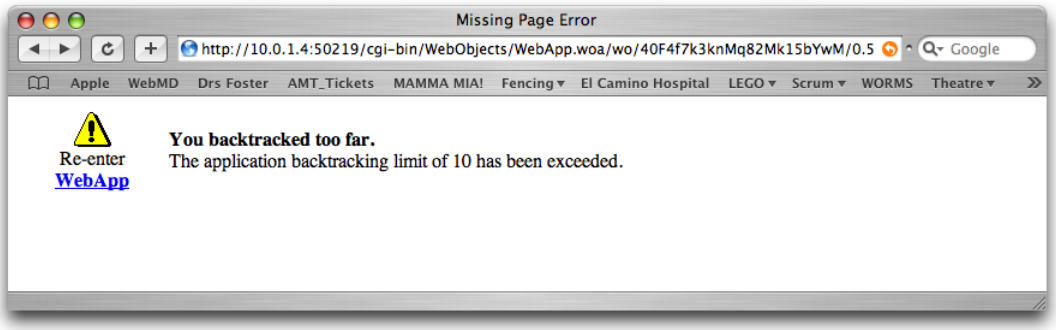
Instead of completely disallowing server-side caching, you can use the `setPageCacheSize` method of WOApplication to define the number of instances of a component an application is to keep in its cache. For example, if you want to maintain state between cycles of the request-response loop (that is, to ensure that state is transferred between user actions), set the `pageCacheSize` to `1`.

Modify the constructor in the Application class by adding a call to `setPageCacheSize`, setting the `pageCacheSize` property to `10`.

```
public Application() {
    super();
    System.out.println("Welcome to " + this.name() + "!");
    setPageRefreshOnBacktrackEnabled(true);
    setPageCacheSize(10);
}
```

Figure 2 shows the page an application sends to a web browser when a user backtracks too far (the page is no longer in the cache).

Figure 2 Backtracking error page



You can customize the error page users receive by implementing the `handlePageRestorationErrorInContext` method in the Application class:

```
public WOResponse handlePageRestorationErrorInContext(WOContext aContext) {
    WOComponent nextPage;
    nextPage = (Error)pageWithName("Error", aContext);
    return nextPage.generateResponse();

}
```

In this code listing, a page is instantiated from a web component named `Error`, which you must build. The contents of the component are completely up to you, but should include the name of the application, your company's name, and a friendly message that tells the user that something went wrong and suggests ways they can return to normal operation.

Document Revision History

This table describes the changes to *WebObjects Web Applications Programming Guide*.

Date	Notes
2007-07-11	Updated for WebObjects 5.4.
2006-01-10	Completely rewritten to describe how to create HTML-based WebObjects applications. This document was previously titled "Web Applications."
2003-04-01	Corrected `addUser` method's definition in "Conditional Display With WOConditional Elements" to suppress NullPointerException generated when either `personName` or `favoriteFood` are `null`. Made the same correction in the `projects/Managing_User_Input/1-Using_Instance_Variables/UserEntry/Main.java`.
	Changed incorrect reference to *WORepetition* to *WOForm* in "Create the ConfirmAuthorDelete Component."
2003-02-01	Added Chapter 6, "Backtracking and Cache Management" (page 51).
	Made editorial changes, including changing *Web* to *web*, *Web page* to *webpage*, and *keypath* to *key path*.
2002-09-01	Document name changed to *Inside WebObjects: Web Applications*.
	Project examples now in `/Developer/Documentation/WebObjects/Web_Applications/projects`.
	Revised for WebObjects 5.2.
2001-05-01	Document published as *Inside WebObjects: Discovering WebObjects for HTML*.