Cincom®

Cincom Smalltalk™

Web Application Developer's Guide

P46-0137-05

SIMPLIFICATION THROUGH INNOVATION®

Cincom Systems, Inc.

55 Merchant Street

Cincinnati, Ohio 45246

Phone: (513) 612-2300

Fax: (513) 612-2000

World Wide Web: http://www.cincom.com

Contents

About This Book

This guide is designed to help VisualWorks programmers create Web applications effectively using the VisualWorks Application Server.

This book accompanies the *VisualWorks Application Developer's Guide*, providing additional information that will help you effectively use the features of the VisualWorks Application Server.

Audience

The discussion in this book presupposes that you have at least a moderate familiarity with object-oriented concepts and the VisualWorks environment. It also presupposes that you have a good understanding of the World Wide Web, web (HTTP) servers, browsers, and HTML.

For an overview of Smalltalk, the VisualWorks development environment and its application architecture, see the *VisualWorks Application Developer's Guide*.

The Web Toolkit support for Smalltalk Server Pages assumes a level of familiarity with server pages (ASP/JSP), and servlets.

In addition to this book, the documentation set for the VisualWorks Application Server includes the following:

- *Web GUI Developer's Guide:* Provides detailed information about building Web applications using the VisualWorks UI Painter and VisualWave.
- *Web Server Configuration Guide:* Provides more detailed information about installing and configuring server applications, the internal architecture of the VisualWorks Application Server, and its interface with commercial HTTP servers.

Conventions

We have followed a variety of conventions, which are standard in the VisualWorks documentation.

Typographic Conventions

The following fonts are used to indicate special terms:

Example	Description
template	Indicates new terms where they are defined, emphasized words, book titles, and words as words.
`cover.doc`	Indicates filenames, pathnames, commands, and other constructs to be entered outside VisualWorks (for example, at a command line).
`filename.xwd`	Indicates a variable element for which you must substitute a value.
windowSpec	Indicates Smalltalk constructs; it also indicates any other information that you enter through the VisualWorks graphical user interface.
Edit menu	Indicates VisualWorks user-interface labels for menu names, dialog-box fields, and buttons; it also indicates emphasis in Smalltalk code samples.

Special Symbols

This book uses the following symbols to designate certain items or relationships:

Examples	Description
File → New	Indicates the name of an item (New) on a menu (File).
<Return> key <Select> button <Operate> menu	Indicates the name of a keyboard key or mouse button; it also indicates the pop-up menu that is displayed by pressing the mouse button of the same name.
<Control>-<g>	Indicates two keys that must be pressed simultaneously.
<Escape> <c>	Indicates two keys that must be pressed sequentially.
Integer>>asCharacter	Indicates an instance method defined in a class.
Float class>>pi	Indicates a class method defined in a class.

Mouse Buttons and Menus

VisualWorks supports a one-, two-, or three-button mouse common on various platforms. Smalltalk traditionally expects a three-button mouse, where the buttons are denoted by the logical names <Select>, <Operate>, and <Window>:

<Select> button	*Select* (or choose) a window location or a menu item, position the text cursor, or highlight text.
<Operate> button	Bring up a menu of *operations* that are appropriate for the current view or selection. The menu that is displayed is referred to as the *<Operate> menu*.
<Window> button	Bring up the menu of actions that can be performed on any VisualWorks *window* (except dialogs), such as **move** and **close**. The menu that is displayed is referred to as the *<Window> menu*.

These buttons correspond to the following mouse buttons or combinations:

	3-Button	**2-Button**	**1-Button**
<Select>	Left button	Left button	Button
<Operate>	Right button	Right button	<Option>+<Select>
<Window>	Middle button	<Ctrl> + <Select>	<Command>+<Select>

Getting Help

There are many sources of technical help available to users of VisualWorks. Cincom technical support options are available to users who have purchased a commercial license. Public support options are available to both commercial and non-commercial license holders.

Commercial Licensees

If, after reading the documentation, you find that you need additional help, you can contact Cincom Technical Support. Cincom provides all customers with help on product installation. For other problems there are several service plans available. For more information, send email to supportweb@cincom.com.

Before Contacting Technical Support

When you need to contact a technical support representative, please be prepared to provide the following information:

- The *version id,* which indicates the version of the product you are using. Choose **Help → About VisualWorks** in the VisualWorks main window. The version number can be found in the resulting dialog under **Version Id:**.
- Any modifications (*patch files*) distributed by Cincom that you have imported into the standard image. Choose **Help → About VisualWorks** in the VisualWorks main window. All installed patches can be found in the resulting dialog under **Patches:**.
- The complete error message and stack trace, if an error notifier is the symptom of the problem. To do so, select **copy stack** in the error notifier window (or in the stack view of the spawned Debugger). Then paste the text into a file that you can send to technical support.

Contacting Technical Support

Cincom Technical Support provides assistance by:

Electronic Mail

To get technical assistance on VisualWorks products, send email to:

supportweb@cincom.com.

Web

In addition to product and company information, technical support information is available on the Cincom website:

http://supportweb.cincom.com

Telephone

Within North America, you can call Cincom Technical Support at (800) 727-3525. Operating hours are Monday through Friday from 8:30 a.m. to 5:00 p.m., Eastern time.

Outside North America, you must contact the local authorized reseller of Cincom products to find out the telephone numbers and hours for technical support.

Non-Commercial Licensees

VisualWorks Non-Commercial is provided “as is,” without any technical support from Cincom. There are, however, on-line sources of help available on VisualWorks and its add-on components. Be assured, you are *not* alone. Many of these resources are valuable to commercial licensees as well.

The University of Illinois at Urbana-Champaign very kindly provides several resources on VisualWorks and Smalltalk:

- A mailing list for users of VisualWorks Non-Commercial, which serves a growing community of VisualWorks Non-Commercial users. To subscribe or unsubscribe, send a message to:

  vwnc-request@cs.uiuc.edu

  with the SUBJECT of "subscribe" or "unsubscribe".

- An excellent Smalltalk archive is maintained by faculty and students at UIUC, who are long-time Smalltalk users and leading lights in the Smalltalk community, at:

  http://st-www.cs.uiuc.edu/

- A Wiki (a user-editable web site) for discussing any and all things VisualWorks related at:

  http://wiki.cs.uiuc.edu/VisualWorks

- A variety of tutorials and other materials specifically on VisualWorks at:

  http://wiki.cs.uiuc.edu/VisualWorks/Tutorials+and+courses

The Usenet Smalltalk news group, comp.lang.smalltalk, carries on active discussions about Smalltalk and VisualWorks, and is a good source for advice.

Additional Sources of Information

This is but one manual in the VisualWorks library. The Cincom Smalltalk publications website:

http://www.cincom.com/smalltalk/documentation

is a resource for the most up to date versions of VisualWorks manuals and additional information pertaining to Cincom Smalltalk.

Online Help

VisualWorks includes an online help system.

To display the online documentation browser, open the **Help** pull-down menu from the VisualWorks main menu bar and select one of the help options.

News Groups

The Smalltalk community is actively present on the internet, and willing to offer helpful advice. A common meeting place is the comp.lang.smalltalk news group. Discussion of VisualWorks and solutions to programming issues are common.

VisualWorks Wiki

A wiki server for VisualWorks is running and can be accessed at:

http://brain.cs.uiuc.edu:8080/VisualWorks.1

This is becoming an active place for exchanges of information about VisualWorks. You can ask questions and, in most cases, get a reply in a couple of days.

Commercial Publications

Smalltalk in general, and VisualWorks in particular, is supported by a large library of documents published by major publishing houses. Check your favorite technical bookstore or online book seller.

Examples

There are a number of examples in file-in format in the **`examples`** subdirectory, under the VisualWorks install directory.

Web Toolkit examples are located in the **`\web\examples`** subdirectory.

1 Overview

The VisualWorks Application Server is a full-featured environment for creating and maintaining Web business applications using VisualWorks. A flexibile, scalable architecture provides support for industry standard Web technologies, and three distinct application frameworks: Smalltalk Server Pages, Servlets, and VisualWave.

Web applications can be built either using the VisualWorks IDE, or by using commercial Web design tools such as Macromedia Dreamweaver. The VisualWorks Application Server provides an open development model that accommodates either strategy, or a combination of both.

Many existing VisualWorks applications can run as VisualWave applications, though typically some modifications are required to optimize the application for the Web.

This chapter describes:

Application Server Architecture

Architecturelly, the VisualWorks Application Server is an add-on to the VisualWorks development environment. By building on VisualWorks' platform portability, the Application Server makes it easy to develop cross-platform, cross-browser Web applications.

VisualWorks Application Server supports the following:

- Static and dynamic HTML pages
- Smalltalk Server pages (ASP version 3.0 or JSP version 1.1)
- Servlets (Java specification version 2.2)
- Content management
- Security framework (HTTPS)
- CGI, ISAPI, FastCGI
- XML
- Database connects (Oracle, SQL, ODBC)
- Server administration tools

Web applications can be designed using either a formal approach that separates the development of presentation from domain logic, as well as a rapid-prototyping approach that begins directly with the business logic.

The VisualWorks Web Toolkit provides an application model well-suited for projects in which the visual layout of the application is created by page designers or marketing groups, while the domain logic is authored by programmers.

Content management features add support for associating domains with Web applications, performing URL aliasing, logical naming of application resources, and server-side includes. Server page and servlet protocol is extended for more flexibility when generating dynamic content.

Web Application Design

From the client's perspective, a Web application is a collection of pages. The content of these pages may be *static* or *dynamic*, the difference being that a dynamic page contains content generated by the Web server each time the client requests the page.

The mechanism for producing the dynamic content of any given page depends upon the server technology that is used. Dynamic pages may be generated from HTML templates (using server pages that contain script), or they may be generated programmatically by components of the application (using servlets or VisualWave).

VisualWorks Application Server provides three distinct server technologies: Smalltalk Server Pages (SSP), servlets, and VisualWave. For maxiumum flexibility, the Application Server can simultaneously host a number of different types of applications, each being constructed with any combination of server pages, servlets, or VisualWave.

When starting to design your Web application, it may be helpful to begin by identifying the type of server technology that best meets your requirements. Some criteria include:

- Whether the bulk of your site design is done commercial page applications (e.g., Macromedia Dreamweaver)
- Whether your application requires extensive manipulation of form data
- Whether you are converting an existing VisualWorks application into a VisualWave application

As a general guideline, applications built with commercial page design software are best implemented using server pages. This approach is most familiar to web designers. If the application makes extensive use of dynamic form data (results from database queries, for example), then an implementation using server pages and servlets may be better suited.

When converting an existing VisualWorks application, you may rapidly prototype a Web version using VisualWave. For applications that do not require dynamic page data or precise control over graphical presentation, conversion using VisualWave can save time.

Once you have selected an appropriate server technology, you use the corresponding *server behavior* to help structure your application logic. Each server technology supported by the Application Server comes with an ensemble of classes for creating an application model.

Building Web Applications with Smalltalk Server Pages

In an application that uses Smalltalk Server Pages, both static and dynamic pages are represented as HTML files, and both can contain client-side scripts (e.g., JavaScript) that are evaluated by the client's browser. The dynamic pages contain scripts or simple classes written in Smalltalk that are evaluated by the VisualWorks Application Server.

VisualWorks Application Server evaluates server pages using the familiar model: when the client requests a server page, the application server loads the HTML page, treating it as a template for an embedded script written in Smalltalk. The Smalltalk expressions embedded in the page are evaluated by the server, and the results are merged into the stream of HTML sent to the client. The template provides a simple way to combine dynamic content with static HTML.

The process of developing a Web application can be greatly simplified by designing with server pages, which provide a general framework for presenting a complex application model to the client while separating the application logic from the page design. The application logic that appears in the page is minimal.

Architecturally, Smalltalk server pages are composed of two elements: a scripting language and an application model.

The scripting language is Smalltalk, which gives your application full access to business logic that uses the Smalltalk class library, language features such as name spaces, and database and DLL access via add-ons. Syntactically, server page scripts may be viewed as a simple extension of HTML. For a complete discussion of how server pages are embedded in HTML, see “Server Page Syntax” on page 8-1.

The application model may follow either that of ASP or JSP. For a complete description, see “Server Page Applications” on page 7-1.

Building Web Applications with Smalltalk Servlets

Servlets are a component framework for processing HTML directly within the Web application. They use a request/response application model similar to that of a CGI, though they offer distinct advantages over CGIs in terms of performance, security, and reliability.

In most respects, servlets offer nearly identical functionality to server pages. Both provide session and application (context) variables, transparent session tracking, security, and direct use of the Smalltalk class library.

Servlets are well-suited for use in conjunction with server pages, as they provide a better model for concurrency and transaction handling.
A common way to organize an application is to divide the presentation components from the components that actually connect to the application model, using redirection to pass requests from the first to the second. The presentation components are then implemented with server pages, while servlets are used to connect to the application model.

For a complete discussion, see "Servlets" on page 6-1.

Building Web Applications with VisualWave

You build a Web application with VisualWave the same way that you build an application with VisualWorks:

- Design the application using an object-oriented architecture.
- Create or adapt classes that define the data and behavior of the application objects.
- Create the user interface using the UI Painter and related editor tools.
- Build the application domain model and the application model (or controller), you create classes and method definitions in Smalltalk code.

For a complete discussion of building applications for VisualWave, see the *VisualWorks Web GUI Developer's Guide*.

2

Web Concepts

Web applications primarily use HTTP to communicate between client and server. Regardless of whether a particular application makes use of static or dynamic pages, client- or server-side scripting, it uses features of HTTP to structure all interactions between client and server.

Since the requirements for client- and server-side application development differ, the following chapter reviews basic Web concepts, but with an emphasis placed upon the features of HTTP most often used when developing server-based applications.

If you are familiar with the basic concepts of server-based applications (HTTP transactions, GET vs. POST, Cookies, etc.), you may wish to skip ahead to the following chapters.

Additional information about HTTP can be found here:

- Hypertext Transfer Protocol -- HTTP 1.0
  http://www.ietf.org/rfc/rfc1945.txt
- Hypertext Transfer Protocol -- HTTP 1.1
  http://www.ietf.org/rfc/rfc2616.txt

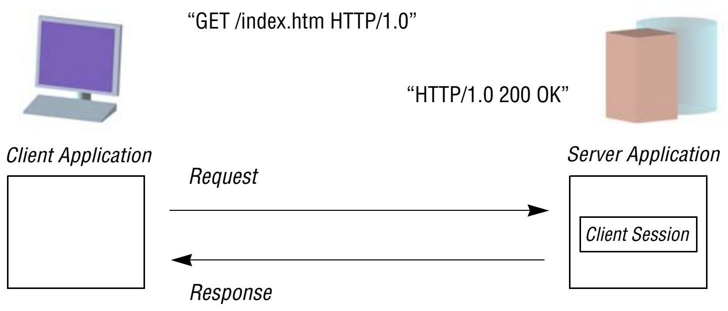
Web Transactions

The basic structure of a Web application model can be understood by briefly looking at the request-response form of an HTTP transaction.

An HTTP transaction begins when a client opens a stream connection to a server and then sends a *request message*. The server replies with a *response message*, and then generally closes the connection. Since the connection is typically opened and then closed for each transaction, HTTP is often described as a *stateless protocol*.

Although HTTP/1.1 allows clients to hold an open connection, the Web transaction model does not generally use it to maintain state.

There are a number of different HTTP request types, although typically a request is either for a *resource* (a resource being a file, a graphic image, or some kind of document), or else a request that the server accept data from the client. In both cases, the client's request includes a uniform resource identifier (URI) to specify the resource to retrieve or accept.



On the server side, the HTTP stream is first converted into request message objects. The server may host a number of different Web applications, but it resolves the request to just one of them. If the request is simply for a resource contained in a file, the Web application may not be involved at all. If the request is for dynamically-generated data, then the Web application handles the request.

If the Web application completes the requested activity, it prepares a *response object* to be returned to the client. The transaction is complete when the application satisfies the request and the server sends the response to the client.

HTTP Request

The VisualWorks Application Server resolves incoming requests to files, server pages, servlets, or directly to Smalltalk methods. If the request is for a resource located in a file, the server simply returns the named resource to the client. If, on the other hand, the request requires dynamic activity on the part of the server, it converts the HTTP message into a new *request object* which is then passed to the Web application.

Request objects may contain parameter and query data, cookies, x.509 certificate fields, or CGI environment variables (a.k.a. *server variables*).

GET versus POST

HTTP/1.1 defines a number of different *request methods*, though in practice, only two of them are commonly used: **`GET`** and **`POST`**:

In general, the client sends **`GET`** as a simple resource request (the resource may be a file, an image, or the output of a script) while **`POST`** is used to pass parameter data, usually from an HTML form object. It should be noted, though, that a **`GET`** message may also be used to pass a limited amount of parameter data.

If the resource is available, the server will respond with an **`OK`**, returning the contents of the resource in the message body. If the resource doesn't exist, the server will respond with a **`Not Found`** error. For details on response types, see "Response Status" on page 2-6.

All HTTP messages have the same general form: all contain a *header* and a *body* section (HTTP message headers are specified by RFC-822). Since the parameter data passed with HTTP messages may be located in either the header or the body, the VisualWorks Web request classes provide methods to access both parts of an HTTP message.

When the client passes parameters in a request message, the actual parameter data will be located either in the message body (in the case of a **`POST`** message) or else in the message header (in the case of a **`GET`** message). The principal difference between these two message types is that **`POST`** may pass parameters in the message body while **`GET`** may not.

As a rule, there should be no active code in a page that is invoked by **`GET`**, e.g., a **`GET`** should not cause some change to a database. When committing any sort of transaction between the client and the Web application, by convention a **`POST`** message is used.

When a client passes parameters using **`GET`**, the request is often referred to as a *query*, and when the parameters are passed using **`POST`**, they are often referred to as *form data*.

Server Variables

Each HTTP request is also associated with a collection of server environment variables containing details available to the Web application.

Server variables represent a combination of information sent by the client and information supplied by the Web server. Not all of these variables are available, depending on the server and gateway in use.

The following table shows the available server variables:

Code	Description
AUTH_TYPE	Protocol used to authenticate the user, e.g., NTLM or BASIC.
CONTENT_LENGTH	Length of the content portion of a client request.
CONTENT_TYPE	Data type of the client request, e.g. application/x-www-form.
DOCUMENT_ROOT	Path to the root data directory relative to the system root.
GATEWAY_INTERFACE	Revision of the CGI specification to which this server complies; format: CGI/revision.
HTTP_ACCEPT	MIME types that the client accepts, for example: image/jpeg, */*; format: type/subtype
HTTP_HOST	Contents of "Host:" header supplied by the client.
HTTP_REFERER	URL of the page containing the link that was referenced to arrive at the current page.
HTTP_USER_AGENT	String describing the client’s browser; format: software/version library/version.
PATH_INFO	Virtual path of the script being executed, e.g., /MyApp/Search.ssp.
PATH_TRANSLATED	Physical path of the script being executed, e.g., C:\MyAppServer\www\Search.ssp.
QUERY_STRING	String used for queries — all that follows the ‘?’ character in the URL.
REMOTE_ADDR	IP address of the client’s machine.
REMOTE_HOST	DNS name of the client’s machine (if available).
REQUEST_METHOD	HTTP method used for the request, e.g., GET, POST, etc.
SCRIPT_NAME	Virtual path of the script being executed.
SERVER_NAME	DNS name of alias of the server machine.

Code	Description
SERVER_PORT	Post number receiving the request.
SERVER_PROTOCOL	Name and revision number of the request protocol, e.g., HTTP/1.1.
SERVER_SOFTWARE	Name and version of the server software that responds to the request; format: name/version.

Cookies

Cookies allow Web applications to store small pieces of information on a client's machine, and to access this persistent state across web sessions. This mechanism enables application developers to recover state information each time a client visits a particular page, greatly simplifying the task of working with a stateless protocol like HTTP.

When the client requests a particular Web page, the browser uses the URL or domain of the request to look up any associated cookies saved on the client's machine. If cookies are found that belong to the page, the browser attaches them to the message header of the HTTP request.

Web applications may likewise attach cookies to HTTP responses, and these are in turn saved by the client's machine when the browser receives the response. Future requests to the same page will include these cookies.

HTTP Response

Each time the client initiates a transaction, the Web server invokes the application logic with a single request object and a single response object.

A *response object* represents an HTTP response message sent from the Web application to the client.

Like HTTP requests, a response message consists of several parts: a status code, header information specifying the content's data type, page content (the response body), and cookies. Web applications can access each part of the response object.

The Web application framework also provides protocol for controlling how a response is buffered, whether or not it has an expiration time, and whether it may be cached by proxy servers.

Response Status

Each HTTP response starts with a three-digit general *status string* passed to the client's browser. This status string may be specified by the server and/or the Web application, e.g.:

```
200 OK
...
404 The requested page could not be found.
```

The first digit of the status string indicates the category of the response. The codes used in HTTP 1.1 are summarized in the following table (see RFC 2616 for a complete description of the codes in each category):

Code	Category	Description
1xx	Informational	Information only — request processing in progress.
2xx	Success	The request was received, understood, and accepted.
3xx	Redirection	The request is incomplete because a redirection is in progress; this range is used to indicate pages that have been moved to another URL.
4xx	Client Error	The request contains bad syntax or otherwise cannot be fulfilled.
5xx	Server Error	The server encountered a runtime error (anomaly or insufficient resources to fulfill the request).

Response Buffering

As the Web application assembles the content of a response, the Application server can either store it in a buffer, or else send each element to the client as it is passed to the HTTP output stream.

When responses are unbuffered, the HTTP connection remains open and client’s browser receives the unbuffered stream as it is being assembled by your application.

Depending upon the requirements of the Web application, it may be preferable to disable output buffering.

Cookies

Just as the request object provides protocol for accessing the value of cookies stored on the client machine, the response object provides protocol for assigning values to cookies held by the client. When the client’s browser receives a response message containing cookies, it will create new cookies or update existing ones to reflect the values of any cookies passed in the message.

Redirection

Web applications may perform two types of redirection on requests: client- or server-side.

In a client-side redirection, the server returns an HTTP response message with a special status code that directs the client’s browser to another URL.

The redirection may be to a full URL (e.g., `http://mycorp.com/index.html`), a relative URL (alias), or to a file stored in the same directory on the server. If any page content has been written to the response object, the client’s browser may display it while the redirection is in progress.

The second type of redirection happens within the server. In this case, the Web application redirects the flow of control from one servlet or server page to another. A server-side redirect is completely invisible to the client. This is typically referred to as a “forward”, “include”, or “transfer” operation.

Secure Sockets

To support HTTP transactions via the Secure Sockets Layer (SSL), client certificates are used to communicate the X.509 certificate information for each request. This is generally handled by the front-end Web server.

For details on working with SSL, refer to the *VisualWorks Server Configuration Guide*.

3

Application Development

To begin developing a Web application, you must first create a *development image* by loading the Application Server parcels into the VisualWorks environment.

Complete details on working with image files and loading parcels can be found in the *VisualWorks Application Developer's Guide*.

Application Server Parcels

The VisualWorks Application Server is contained in a set of parcels. To develop Web applications, you must load one or more of these parcels.

The main parcels, which are contained in the **`/web`**, **`/wavedev`**, and **`/waveserver`** directories of the VisualWorks distribution, are:

VisualWave	Support for VisualWave applications
WaveLoad-Server	Load-balancer component for the Server
WaveLoad-Client	Load-balancer component for secondary servers
WebToolkit	Support for Smalltalk server pages and servlets

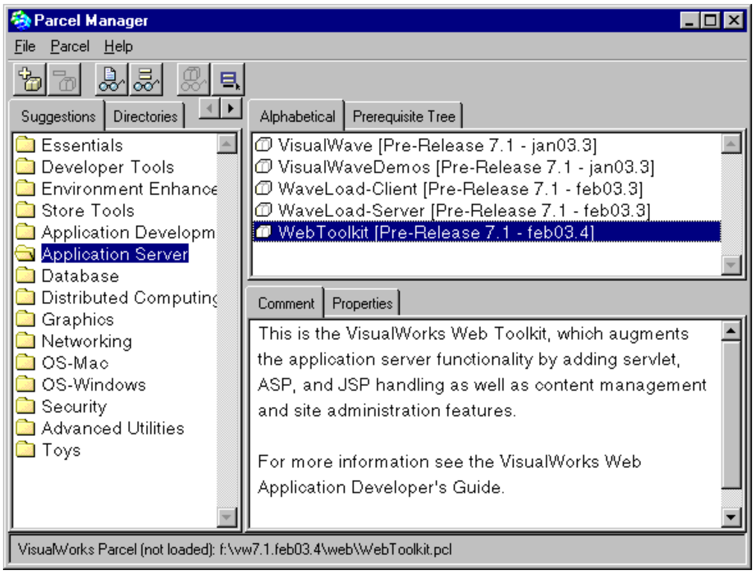
When beginning development of a Web application, generally you will start with either the **VisualWave** or the **WebToolkit** parcel.

Creating a Development Image

To load the Application Server into the VisualWorks environment:

1. Open the Parcel Manager by selecting **System → Parcel Manager** in the Launcher window.
2. Select the category **Application Server** on the **Suggestions** tab.

   A list of the Application Server parcels appears on the left:



This is the VisualWorks Web Toolkit, which augments the application server functionality by adding servlet, ASP, and JSP handling as well as content management and site administration features.

For more information see the VisualWorks Web Application Developer's Guide.

VisualWorks Parcel (not loaded): f:\vw7.1.feb03.4\web\WebToolkit.pcl

3. Select the **VisualWave** parcel and select **Load** from the <Operate> menu. If you wish to use the features of the Web Toolkit (i.e., server pages or servlets), select and load the **Web Toolkit** parcel instead.
4. VisualWorks loads the chosen parcel and any prerequisites, showing its progress in the Transcript window.

The Application Server is now loaded and ready for use.

Saving the Development Image

To save an image containing the Application Server:

1. In the VisualWorks Launcher window, choose **File → Save Image As...**

   A dialog box prompts you with the name of the current image.

2. Enter a name for the image, and click on **OK**.

   You can use the current image name, and overwrite the current image file, or enter another name. Do not include the **`.im`** file extension.VisualWorks does not create a directory for you.

   Caution: If you save an image using the same name as that of the standard image, make sure that you can restore the original image shipped with VisualWorks, if needed.

A new image file has been created.

Saving the image also saves any servers you create, and any applications you load.

Exiting the Development Image

To exit the development environment:

- In the Launcher Window, choose **File → Exit VisualWorks...**

When exiting, you are given an option to save before exiting. This allows you to save the server environment image, as described above. Any unsaved changes to the server environment or servers are lost.

Exiting the server environment renders all of the servers in that environment unavailable.

Deploying Web Applications

You deploy a VisualWorks Web application the same way you deploy a VisualWorks desktop application: by delivering an image or any number of parcels. For applications built with server pages or static HTML, all content files are typically located in one or two directories.

For a Web application, a deployment image may be created by stripping the image using the Runtime Packager. A headless configuration is also available. Note that while previous versions of the Application Server used the ImageMaker to create deployment images, in VisualWorks 5i this was superseded by the Runtime Packager.

Deploying Web applications is described in more detail in the *VisualWorks Web Server Configuration Guide*. Parcels, the Runtime Packager, and strategies for deploying screen-oriented applications are described in detail in the *VisualWorks Application Developer's Guide*.

Server System Requirements

In addition to an image configured for the VisualWorks Server, applications require:

- An HTTP (Web) server.
    - For development, you can use the Smalltalk HTTP Server provided with the Application Server.
    - For production use, you may also use a commercial HTTP server in combination with VisualWorks Server. For more information, see the *VisualWorks Web Server Configuration Guide*.
- The VisualWorks object engine and associated files.

For hardware requirements, refer to the *VisualWorks Release Notes*.

Client System Requirements

Clients need a Web browser and the URL to reach your application. The URLs used during development typically have this form:

```
http://host.domain:port/pathPrefix/path
```

For example, the following URL starts the `Examples.CheckbookInterface` application using the Smalltalk server that is at `localhost:8008`:

```
http://localhost:8008/launch/Examples.CheckbookInterface
```

If you are using an External Web server, the URL will be different. For details, see the *VisualWorks Web Server Configuration Guide*.

4

Server Console

The Server Console is the tool you use to manage the HTTP servers and server applications running within the VisualWorks Application Server image. It enables you to manage and monitor all communication between the Application Server and the Internet.

VisualWorks Application Server includes a personal HTTP server, the Smalltalk HTTP Server, which you can use for developing and deploying server applications. As a web server, it listens for requests from a web browser, manages a session with the requested server application, and returns the response to the web browser.

For deployed applications, VisualWorks Application Server includes special features for working with high-performance, commercial web servers.

Smalltalk HTTP Server

The Smalltalk HTTP Server runs within your image and eliminates the need for a commercial web server and CGI script. When you use the personal HTTP server, the client's web browser is directed to the application in your image via a specially-coded URL.

The Smalltalk HTTP Server has two limitations:

- It does not automatically serve documents from the file system to the requesting web browser. To do this, you can set up a special `FileResponder` resolver to serve documents from the file system.
- It has no security features.

Using a Web Server

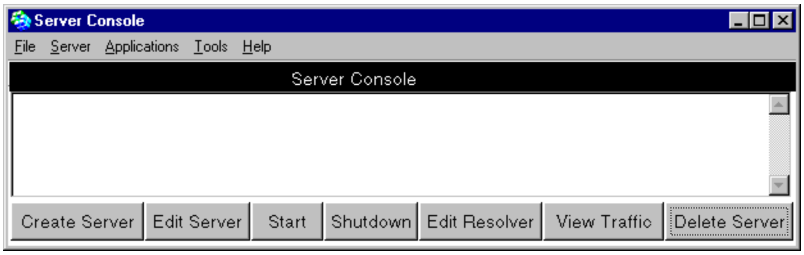
To set up a web server:

1. Open the Server Console.
2. Create a new server.
3. Start the server.

Opening the Server Console

To open the Server Console, choose **Web → Server Console** from the VisualWorks Launcher window, or click the Web button.

VisualWorks displays the Server Console, which lists any HTTP servers in the image, providing status information about those servers, and allowing you to configure them. Initially, no servers are listed.



From the Server Console you can:

- Create servers
- Edit the configuration of the selected server
- Start or shutdown the selected server
- Edit the server's resolver, which determines how the server interprets the request provided to it
- View server activity (web requests and responses)
- Delete servers

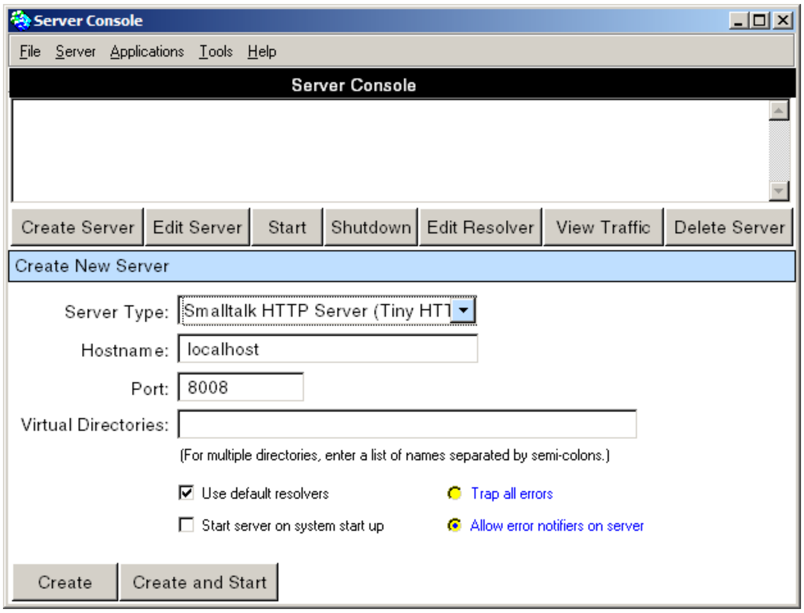
After you create servers and save the environment as an image, the Application server preserves any servers you create. All servers start in an inactive state, unless the server is set to start at system startup. Any applications that you had loaded and registries you created are also still present.

Creating a New Server

The first thing you need to do is create a server. To create a web server:

1 Click the **Create Server** button.

The Server Console extends to provide options for creating a new Web server.



Server Type: Smalltalk HTTP Server (Tiny HTT

Hostname: localhost

Port: 8008

Virtual Directories:

(For multiple directories, enter a list of names separated by semi-colons.)

[✓] Use default resolvers

[ ] Start server on system start up

( ) Trap all errors

(●) Allow error notifiers on server

For a server, you can specify:

Type: During development, it is easiest to use the Smalltalk HTTP Server. The External Web Server is for use with CGIs and commercial web servers.

Hostname: The name of the machine on which the VisualWorks Server is running.

Port: The port on which the IP Server listens.

Virtual Directories: The names of any virtual directories used on the server (for use with External Web Server only).

Resolvers: The set of resolvers used to map information provided in the URL to actions to take and/or applications to invoke. A set of default resolvers is provided with VisualWorks Application Server.

Errors: The destination for error messages. Servers can display errors on the host machine's screen or they can redirect errors back to the web browser.

2 Choose the options for your configuration. For a VisualWorks image running on the same machine as the Web browser (during testing), the most common options are:

Setting	Value
Server Type	Smalltalk HTTP Server
Hostname	localhost
Port	8008
Virtual Directories	
Use default resolvers	Yes
Allow error notifiers on server	Yes

3 Click the **Create** button at the bottom of the Server Console.

VisualWorks creates the specified server and lists it at the top of the console.

You now have a personal web server to use for interactions between a VisualWorks image and a web browser that are on the same host machine. That server, however, is not yet running.

Starting a Server

To start the HTTP server that you have just created:

1 In the Server Console, select the server's entry from the list view.

2 Click the **Start** button.

The HTTP Server changes from inactive to active.

The entry for a server shows (from left to right):

- The type of server (Smalltalk or External)
- Whether the server is active (started) or inactive (shutdown)
- The hostname and port number
- Installed paths

VisualWorks is now ready to accept requests from a web browser.

Testing a Server

To test VisualWorks and your web server:

1 Start your web browser.

2 In your web browser, request the following URL:

```
http://localhost:8008/echo
```

3 The web browser displays a list of environment variables received from the Application server. If you did not receive a list of variables, make sure that the Server is active in the Server Console.

In the requested URL:

- The first section specifies the *protocol* to transmit the request and response. HTTP (HyperText Transfer Protocol) is the protocol most commonly used by web browsers and servers.
- The next section specifies the *recipient* of the request by hostname and port number (e.g., `localhost:8008`). It matches the hostname and port of the web server that you created.
- The next section of the URL (the *path*) contains information for the recipient. VisualWorks Web servers use resolvers to break apart the path and determine what action to take. Resolvers are set up as part of the server. When you created the Smalltalk HTTP Server, you chose to use the default resolvers. When a server using the default resolvers receives the path echo, it simply returns the complete web request to the browser.

Shutting Down the Server

To shut down a server:

1 Display the Server Console.

2 Select the server to shut down.

3 Click the **Shutdown** button.

You do not need to shut down the server when your application isn’t running. In fact, when you’re using VisualWorks to serve your clients via the web, you probably want to leave a server running at all times.

You do not need to delete servers that are not currently being used. When you delete a server, you also lose the configuration set up for that server. Shutting down a server makes that server unable to respond to any web requests, but it is simple to restart.

5

Web Sites

A Web application may be defined as a collection of static and/or dynamic pages, their supporting files, and any server-side code needed to make the dynamic pages run. A Web application is generally configured for a particular server or set of servers at deployment time, and what we call a *Web site* refers specifically to this deployment information.

The *site configuration* defines all parameters that concern how a Web application is to be hosted on a particular server. VisualWorks Application Server provides several mechanisms for creating and managing Web site configurations.

This chapter shows how to configure server support for Web sites. Additional information on configuring servers can be found in the *VisualWorks Web Server Configuration Guide*.

Working with Web Sites

A single VisualWorks Application Server may host a number of distinct *Web site configurations*, each being held in the server's virtual image as an instance of class WebSite. Distinct sites must be unique by name.

When the Application Server is running, a special administrator's Web interface is available to interactively set configuration parameters for any WebSite contained in the server image, as well as those which apply to the Application Server itself. The administrator's interface may be accessed using a standard Web browser.

Since any changes to these configurations only affect the objects in the virtual image, at some point, you may want to create configuration files that define all configuration parameters for the server and its sites.

For more advanced site configurations, including use of content management features, or for deploying an application in a production environment, you must use a configuration file. For details, see the discussion of "Deployment" on page 11-1.

Managing Web Site Configurations

All server and site properties may be viewed and modified using a standard Web browser. With the Application Server running, you can view all site configurations defined by the server using a Web browser.

The Application server provides a special Welcome page that is configured by default to appear at the server's base URL.

For example, to view the Welcome page for a server running on the same machine as the Web browser:

1 Start the server using either the prebuilt image included on the release media (**`\web\runtime.im`**), or a standard VisualWorks image with the Application Server and Web Toolkit loaded.

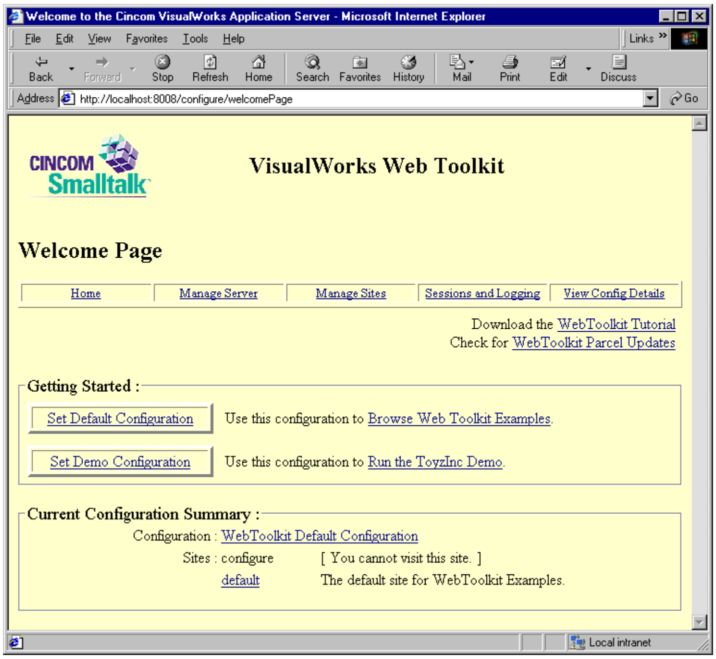
   a If you are starting from the prebuilt image, skip to step 3.

2 Open a Server Console, then create and start a Smalltalk HTTP Server (for instructions, see "Smalltalk HTTP Server" on page 4-1).

3 Start a Web browser and open the following URL:

```
http://localhost:8008/
```

The browser will redirect you to the Web Toolkit's Welcome page:


Welcome to the Cincom VisualWorks Application Server - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Forward Stop Refresh Home Search Favorites History Mail Print Edit Discuss
Address http://localhost:8008/configure/welcomePage
CINCOM Smalltalk
Home | Manage Server | Manage Sites | Sessions and Logging | View Config Details
Set Default Configuration
Set Demo Configuration
Local intranet

VisualWorks Web Toolkit

Welcome Page

Download the WebToolkit Tutorial
Check for WebToolkit Parcel Updates

Getting Started :

Set Default Configuration — Use this configuration to Browse Web Toolkit Examples.

Set Demo Configuration — Use this configuration to Run the ToyzInc Demo.

Current Configuration Summary :

Configuration :	WebToolkit Default Configuration	
Sites :	configure	[You cannot visit this site.]
	default	The default site for WebToolkit Examples.

From the Welcome page, you may manage the attributes of an existing site, create new sites, remove sites, or manage server attributes that govern the behavior of all sites.

If you have previously defined a home page for the **default** site, or if you do not have a **default** site in your server's configuration, you must access the Welcome page using its explicit URL:

```
http://localhost:8008/configure/
```

The Welcome page also allows you to configure the server with one of two pre-set configurations distributed with the Web Toolkit: **default** or **demo**. You may browse examples or run demos directly from this page.

Note that selecting one of the pre-set configurations restores the server to a specific state, removing any other sites that you have created.

Note: Use of Netscape or Internet Explorer version 6 or later is strongly recommended. Some of the Welcome page features may not work correctly with Netscape 4.x.

Selecting the Default Configuration

The default configuration is provided to help orient you during the initial phase of working with site definitions. The prebuilt image (**`runtime.im`**) is configured to use this configuration.

You may restore this configuration from the server's Welcome page, by clicking on **Set Default Configuration**.

The default configuration includes two sites:

Site Name	Description
configure	Manages the Administrator's Web interface.
default	A placeholder containing example pages and servlets.

For convenience, each example page in the **default** site is located in a separate subdirectory of **`\web\examples`**, and each example is accompanied by a **`Readme`** file. To view the general **`Readme`** file for these examples, click on **Browse Web Toolkit Examples**. From the examples page, you can navigate to specific **`Readme`** files and access example pages.

Alternately, you may enter a URL directly into a Web browser., e.g., for the frames example:

http://localhost:8008/examples/frame0.ssp

Selecting the Demo Configuration

The Web Toolkit also contains a simple demo application. Before running this application, though, you must set its specific site configuration. The configuration files and the pages for this application are located within or below the **`\web\demo`** directory.

To select this pre-set configuration, open the server's Welcome page, and click on **Set Demo Configuration** (note that this will remove all other sites you have created). The demo configuration contains the four following sites:

Site Name	Description
configure	Manages the Administrator's Web interface.
default	A placeholder containing example pages and servlets.
images	A site alias that allows the demo to serve general image content separately from the server pages.
toyzinc	Contains the Toyzinc demo application.

For convenience, each example site is located in a separate subdirectory of **`\web`**, though this placement is not a requirement imposed by the Application Server. Each example is accompanied by a **`Readme`** file.

With the example configuration selected, a number of demonstration files may be accessed in the **default** site. Alternately, you may enter a URL directly into a Web browser.

For example, to start the Toyzinc demo: open the Welcome page, click on **Run the ToyzInc Demo** or else launch:

http://localhost:8008/toyzinc/main.ssp

When running the ToyzInc demo, if you have not already configured the demo configuration, you are prompted to do so.

Viewing Site Attributes

You may use the administrator's Web interface to view the attributes of any site in the currently selected configuration.

To view a summary of all site attributes in the current configuration, jump to the **View Config Details** link from the server's Welcome page.

The Configuration Details page shows the server's global configuration, followed by the details for each site in the configuration. Any errors are also noted here (for details, see "Configuration Errors" on page 11-8).

The Configuration Details page may also be accessed using this URL:

http://localhost:8008/configure

To view the attributes of a particular site, click on its linked name as it appears in the site list on the server's Welcome page.

This opens a page showing the complete site configuration, and any server pages or static HTML pages located in its home directory.

Creating and Configuring a Site

You may change the default site configuration that is included with the Application Server release image, or you may create new site instances, but the procedure for configuring a site is the same in both cases.

To change the configuration of an existing site, go to the **Manage Sites** page, click on the site's **configure** link.

To create a new site, use the form provided in the middle of the **Manage Sites** page. The name you provide when creating a new site is used by the Application Server to uniquely distinguish the site object.

The name is also used to attract requests to the site. For example, the site named **toyzinc** would serve these requests:

```
http://localhost:8008/toyzinc
http://localhost:8008/toyzinc/catalog.ssp
```

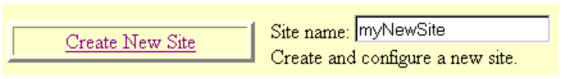
In the first example, since no file is specified, the Application Server directs the request to the site's home page.

Requests that do not include an explicit site name in their path are routed to the **default** site. There are a variety of other ways to attract requests to a site; for details, see "Resolving Web Requests" on page 10-3.

Configuring a Site

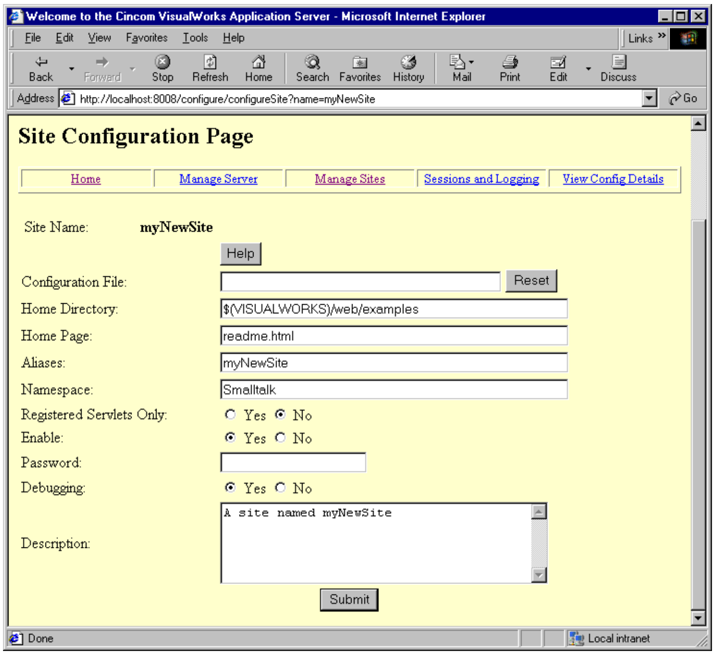
To configure a simple site for viewing the Web Toolkit examples:

1 Starting from the **Manage Sites** page, enter the new site name in the input field and click on the **Create New Site** button.



Site name: myNewSite
Create and configure a new site.

A Site Configuration page appears:



Welcome to the Cincom VisualWorks Application Server - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Links
Back Forward Stop Refresh Home Search Favorites History Mail Print Edit Discuss
Address http://localhost:8008/configure/configureSite?name=myNewSite
Go
Home
Manage Server
Manage Sites
Sessions and Logging
View Config Details
Help
Reset
Submit
Done
Local intranet

Site Configuration Page

Site Name:	myNewSite
Configuration File:	
Home Directory:	$(VISUALWORKS)/web/examples
Home Page:	readme.html
Aliases:	myNewSite
Namespace:	Smalltalk
Registered Servlets Only:	Yes [✓ No]
Enable:	[✓ Yes] No
Password:	
Debugging:	[✓ Yes] No
Description:	A site named myNewSite

2 Since you have not yet created a configuration file for your site, leave the form's **Configuration File** field blank. See “Deployment” on page 11-1 for details on using configuration files.

3 Enter the site’s **Home Directory**. This specifies the location of any server pages or static HTML pages used by your application. For example:

    $(VISUALWORKS)/web/examples

Note: The **Home Directory** is the only attribute that is absolutely required during configuration.

For details on each attribute of the site configuration, you may click on the **Help** button shown on the Site Configuration page, or see “Specifying Site Attributes” on page 5-8.

4 Enter a **Home Page** (e.g.: **readme.html**). This optional file is shown when clients access the site without specifying any particular page. For example, when no other home page is specified, the default site’s home page is redirected to the configuration site.

5 Enter one or more **Aliases** used to attract requests to the site (optional). An alias specifies the first path component of the request URL, much like a virtual directory.

6 Leave the **Namespace** attribute in its default state: Smalltalk.

7 Disable **Registered Servlets Only.**

8 **Enable** the site.

9 You may specify a **Password** for your site, but leave this field blank for the moment.

10 Enable **Debugging**.

11 Finally, enter a short **Description** of your site.

12 To configure the site using the specified attributes, click on **Submit**.

The site configuration is now complete.

With a configured site, you may use the administrator’s Web interface to **visit** any of the server pages located in the site’s home directory. For convenience, the Site Visit page includes a list of links to these pages on the lower part of the configuration results display.

The sample site we’ve created uses **`$(VISUALWORKS)\web\examples`** as its home directory. Thus, all files within or below the **`\web\examples`** directory are shown on the Site Visit page.

If the configuration you are using includes a **default** site, and that default site has no home page, then you can reach the configuration page by opening the basic host:port URL. If you are using only "named" sites (i.e., there is no **default** site), then you must use the following URL to get to the configuration page:

```
http://localhost:8008/configure
```

For a detailed discussion of the options available for resolving incoming requests to specific sites, see "Resolving Web Requests" on page 10-3.

Removing a Site

To delete a user-defined site, start from the **Manage Sites** page and click on the corresponding **remove** link.

Specifying Site Attributes

The following site attributes can be set from the Site Configuration page:

Site Name

A site name is used internally to uniquely identify the site for administrative purposes. The name you provide when creating a new site is also the name used to attract requests to the site. If no site name is specified, incoming requests are routed to the **default** site.

The site name need not correspond to the base directory in the site's URL, although this might often be the case. The physical location of the files served is relative to the site's home directory.

If you do not specify an **Alias** for the site (see discussion below), the site name will be used to attract requests in the same manner. For example, if the site named `mySite` does not contain an alias, then requests to the URL

```
http://www.myCorp.com/mySite/index.html
```

are directed to `mySite`. If you want both the site name and some other aliases, you must include your site name in the list of aliases. For details on working with aliases, see "Creating a Site Alias" on page 10-4.

Configuration File

The attributes of each site may optionally be set using a INI-formatted configuration file. Configuration files are used in conjunction with the advanced features of the Server such as content management and logical naming. They may also be used to automatically configure a headless image at startup time.

The INI file may be specified with either an absolute or relative path. If a relative filename is used, it is resolved relative to the directory containing the global configuration file — *not* the site's home directory.

By default, the global configuration file is **`webtools.ini`**. The Server expects to find this in the VisualWorks working directory (if it exists); otherwise, it is **`$(VISUALWORKS)/web/webtools.ini`**. This can be changed using the Administrator's Web interface.

Use the **Reset** button on the Site Configuration page to restore all the attributes of a site to those specified in the named configuration file.

For details about modifying the content of INI files and where they should be located, see "Deployment" on page 11-1.

Home Directory

Specifies the physical directory used as a base directory when serving static pages or Smalltalk server pages located in files on the server. The home directory indicates where these files are to be found, with the remainder of the URL being treated as a path underneath the home directory.

For example, if the home directory is **`c:\sites\myApp`** then the URL:

```
http://mycorp.com/sales/welcome.ssp
```

would serve the page **`c:\sites\myApp\sales\welcome.ssp`**.

If the Home Directory is specified using a relative path, then it is resolved relative to the current VisualWorks working directory.

Home Page

Defines the (optional) home page for a site. If defined, any requests directed to the site which do not resolve to a page are directed to the home page.

Aliases

One or more aliases may be used to attract requests to the site. If no alias is specified, the site name (see above) is used as the default site alias. If you want both the site name and some other aliases, you must include your site name in the list of aliases.

Aliases may be used to create the equivalent of a virtual directory on the server. A site serves any request if the first path component of the request URL is one of the site's aliases.

For example, if the site named red defines aliases red and rouge, then any request of the form:

```
http://myCorp.com/rouge/index.ssp
```

will be directed to the site named red. Requests which are not attracted to any site through an alias are redirected to the default site.

To specify multiple aliases for a site, enter a list of names separated by semi-colons, e.g.:

```
aliases = oneAlias; twoAlias; threeAlias
```

For details on defining site aliases or using them as virtual directories, see "Creating a Site Alias" on page 10-4.

Namespace

Defines the environment in which server page script is evaluated. A namespace should be specified when using server pages that access classes in your own namespaces.

To use more than one namespace, create a new namespace that imports all the other namespaces you want to use.

The Application Server's own classes belong to the VisualWave.* namespace, which is distinct from the Smalltalk.* namespace. If your application uses a private namespace, and you use any parts of the Application Server (e.g., servlets or custom tag handlers), you will need to either import VisualWave.* into your namespace or fully qualify all the VisualWave class names you reference

Registered Servlets Only

Enable or disable direct access to servlets using the standard servlet URL (e.g. /servlet/ServletClass). When disabled, servlets may be launched using URLs that include /servlet/ServletClass as a path component.

During deployment, **Registered Servlets Only** should be enabled as a security measure, so that servlets may only be accessed using logical names. For details, see "Enabling Use of Registered Servlets" on page 11-8.

Enable

Enable or disable a site without losing its configuration. When a site is disabled, it will not accept requests.

Password

Set to password protect a site.

Note: Before distributing a production image you should either disable or change the password on the **configure** site so that clients cannot change your site configuration.

Debugging

Activate debugging features on the site. For details on debugging options, see “Setting Site Debugging Options” on page 5-12.

Event Callbacks

Define methods which the Application Server calls when specific session- or application- level events occur. If you define an event callback in the global configuration file, it applies to all web sites. Event callbacks can only be defined in the configuration file.

In addition to the application and session events, a special configuration event is also available (for details, see “Application Events” on page 7-13).

Saving a Site Configuration

Any changes made using the administrator’s Web interface only affect WebSite objects in the serving image. Thus, site configurations are saved by saving the VisualWorks image.

In release 7.6 of VisualWorks Application Server, WebSite configurations may be read from an initialization file, but any changes you make are not saved to the file. This limitation will be removed in a subsequent release.

To make changes that do not reside in the image, you must use the server configuration files. For details, see “Deployment” on page 11-1.

Setting Site Debugging Options

When developing a Web application, it is often useful to allow exceptions to produce walkbacks rather than having them trapped and redirected as an error message to the client. The Application Server can be set to trap exceptions that may occur during the compilation of a server page, as well as those that occur during page execution. When deploying a Web application, however, you will normally want to trap all exceptions.

The site debugging feature is most useful when developing server pages. In addition to this feature, the Application Server defines two other mechanisms for exception handling. The three different mechanisms are as follows:

Site Debugging

The **Debugging** setting on the site configuration page enables you to trap exceptions raised when compiling server pages. When this setting is **False**, pages are cached for maximum performance, and exceptions are directed to the client as **HTTP 5xx** (Internal Server) errors. When set to **True**, compiled server pages are not cached, and exceptions open a notifier window on the server.

Server Debugging

The **Trap all Errors** setting in the Server Console may be used to capture exceptions that occur during the execution of a server page, servlet, or a VisualWave application. In the case of a server page, the exception may possibly occur outside of the code contained within the page. When **Trap all Errors** is set, these exceptions will be directed to the client as **HTTP 5xx** errors.

Development Debugging

A global flag isDevelopingOverride is provided that will set the **Trap all Errors** flag true for every defined server. To set this flag, evaluate ProcessEnvironment isDevelopingOverride: true/false.

The following table summarizes the recommended settings:

Activity	**Site Configuration**	**Server Configuration**
Development	Debugging: True	Allow Error Notifiers
Deployment	Debugging: False	Trap All Errors

By default, these flags are set appropriately for development, and the packaging process automatically sets them for deployment use.

Managing Server Attributes

The VisualWorks Application Server administrator’s Web interface also provides a means to browse and change various server attributes that apply to all defined sites. The **Server Management** page may be accessed from the server’s Welcome page (click on **Manage Server**).

The following functions are provided:

Configure Server

Set the name of the global configuration file. By default, this is **`webtools.ini`** in the VisualWorks working directory (if it exists), otherwise, it is **`$(VISUALWORKS)/web/webtools.ini`**. Use the input field to enter a new file name.

Relative file paths are resolved relative to the VisualWorks working directory.

Clear Caches

Clear all cached data, including compiled server pages and sessions. This should be used when server pages contained in files have been updated. The session you are currently using is not released.

Reset Configuration

Clear all cached data, and re-read the server configuration files. Both the global server configuration file (**`webtools.ini`**, or the file you have specified) and any per-site files are re-read.

Reset Server

Clear all cached data, reload all parcels specified on the command line, and re-read the server configuration files.

This is intended for use in a headless or **`runtime.im`** installation.

Exit Server

Terminate the server (quit the object engine).

Managing Server Logging and Sessions

The administrator's Web interface may be used to browse and change logging of server traffic. Access the **Sessions And Logging** page from the server's Welcome page.

The following functions are provided:

Start/Stop Logging
Toggle logging behavior.

Configure Logging
Set the name of the global log file (by default, **`webserve.log`** in the VisualWorks working directory). Use the input field to enter a new file name.

View Sessions
Show a list of all client sessions.

View Error Log
Show the contents of the error log file **`vwave.log`**.

View Web Log
Show the contents of the global log file (**`webserve.log`**, or the file you've specified).

6

Servlets

Servlets are lightweight server-side application components that accept HTTP requests and deliver an HTTP response. Servlets generally manage the application's presentation logic, keeping it separate from the business logic. Servlets are similar to traditional CGI scripts, but start more quickly, provide additional supporting code, and allow for more sophisticated interactions.

Servlets run inside a *servlet container*, which may hold many different servlets. Requests come to the container, and are parsed and routed to the appropriate servlet instance. Servlet classes include message protocol for accessing request information, emitting response data, and communicating with other servlets or server pages.

A servlet can be as small as a single, one-method class, but it may also be combined with other servlets to provide a larger unit of functionality. When applications are partitioned into a group of servlets, the servlet instances communicate by sharing and forwarding request information before sending a response.

The VisualWorks implementation of Smalltalk servlets closely follows version 2.2 of the Java Servlets API. Accordingly, the discussion in this chapter presupposes some familiarity with the servlets specification.

This chapter presents:

Overview

Servlets provide a language, protocol, and platform independent model for creating server-side application components. A servlet employs a request-response programming model that harmonizes with the structure of Web transactions. Although the servlet model is general, it is typically used to structure HTTP-based Web applications.

VisualWorks Application Server provides support for Smalltalk servlets. Under VisualWorks Application Server, a servlet is represented by a single Smalltalk class, and new servlets are typically built by subclassing class HttpServlet.

HTTP Servlets provide:

- Simple, request/response programming model
- Session tracking
- Dispatching/forwarding
- Security features

When to Use Servlets

When choosing between server pages and servlets, the following points may help to clarify the best design for your application.

- HTML should be kept in server pages, not hard-coded into servlet methods. Any reasonable quantity of Smalltalk code should be in an object, either a servlet or a component invoked from a server page.
- As a general rule of thumb, you should consider using servlets in the parts of your application where significant processing is required to handle requests; in particular, when your application needs to extensively manipulate request, response, and session object state.
- During development, servlets have the distinct advantage of being much easier to manage than server pages. All standard tools in the VisualWorks IDE are available to simplify the coding, debugging, and maintaining the servlet code.
- When control over page presentation is an issue, servlets are best used in conjunction with server pages.

Servlets, Containers, Contexts

The VisualWorks Application Server acts as the *servlet container*. The servlet container is responsible for mapping requests to servlets, loading and activating a particular servlet class on demand, and then managing the servlet through its entire life-cycle.

The Servlet interface accommodates two different execution models: single- and multi-threaded. Note that these names are somewhat misleading, because in fact both support multi-threading. The distinction has more to do with the flow of control and the container's memory allocation strategy.

Although the servlet specification recommends not to use the single-threaded model, if properly implemented a single-threaded servlet can perform equally well and be more convenient to use (for details, see "Multithreading Servlets" on page 6-8).

Once a servlet has been loaded and initialized by the container, it is ready to service requests. A request is processed when the servlet container passes a request and a response object to the servlet. Requests are processed until the container unloads the servlet.

Each Web application is also associated with a single *servlet context*. Since a single application may be composed of a number of different specialized servlets, the servlet context provides a way for them to share data and access resources. The context provides the servlet with a view on its local environment, although in a distributed environment, servlet contexts are *not* distributed between hosts. For details, see "Servlet Context" on page 6-10.

Servlet Basics

Let’s examine a simple example servlet to see how it works.

HTTP servlets are defined as subclasses of HttpServlet. Instance variables should not be declared in these classes. For example:

```
Smalltalk.VisualWave defineClass: #VeryBasicServlet
    superclass: #{VisualWave.HttpServlet}
    indexedType: #none
    private: false
    instanceVariableNames: ''
    classInstanceVariableNames: ''
    imports: ''
```

This class, along with a number of other servlet examples, is predefined in the VisualWorks Web Toolkit. To view the example servlet classes, open a Hierarchy Browser on class HttpServlet.

Any servlet class loaded in the VisualWorks image is ready for activation. When the servlet container receives a request that can be mapped to a servlet class, it either transmits the request to an existing servlet instance, or it creates and initializes a new servlet instance to field the request, and then transmits the request to the new instance.

Typically, the servlet container spawns a separate thread (in this case, a Smalltalk process) for each HTTP request. The thread executes a single *service method* that handles the request, returns an HTTP response, and then terminates. Servlets must define an appropriate service method by implementing either a doGet:response: or doPost:response: method.

Each type of HTTP message is handled by a separate method. A servlet that responds to an HTTP **GET** request, for example, must implement the service method doGet:response:, while a servlet that responds to an HTTP **POST** request must implement the service method doPost:response:.

Thus, the example VeryBasicServlet declares the following:

```
doGet: aRequest response: aResponse
  "write output to the response object"
    aResponse write: '<HTML><BODY>Hello world</BODY></HTML>'.
```

This method ignores the content of the request object, and simply writes a string of formatted HTML to the response object.

When this service method finishes, the servlet container flushes the response object to the client.

Testing the VeryBasicServlet

To test this servlet, open a Server Console to start a Smalltalk Server on localhost:8008 (for details, see "Using a Web Server" on page 4-2).

Open a Web browser and enter the following (case-sensitive) URL:

```
http://localhost:8008/servlet/VeryBasicServlet
```

The text "Hello world" should appear in the browser.

Although implemented with only a single method, this is a completely functional servlet.

The Redirect Servlet

A slightly more complicated example can be found in class Redirect.

This servlet, like the VeryBasicServlet, declares only one service method:

```
doGet: aRequest response: aResponse
    | url |
    url := aRequest getParameter: 'url'.
    url isNil ifTrue: [aResponse write: '<HTML><BODY>Cannot redirect, no
  url provided</BODY><HTML>'. ^self].

    aResponse redirectTo: url.
```

When the Redirect servlet is invoked, it expects to find a single parameter named "url" in the query string, e.g.:

```
http://localhost:8008/servlet/Redirect?url=http://www.cincom.com/smalltalk/
```

The servlet performs a client-side redirection by first obtaining an URL instance from the named query parameter (query parameters are separated from the base URL using the `?` character). If no parameter is available, an error message is returned.

The actual redirection is achieved by sending an HTTP redirect message to the client. The redirection URL is placed in the message header of the response, and flushed to the client. The client's browser then performs the redirection by opening a new HTTP connection to the target URL.

Mapping Requests to Servlets

During development you may launch servlets using URLs of the form /servlet/className, e.g.:

```
http://localhost:8008/servlet/VeryBasicServlet
```

In this case, /servlet is a fixed string recognized by the Application Server, and /className is the case-sensitive name of the servlet class. The /servlet path component may be preceded by other sections of the URL, e.g.:

```
http://localhost:8008/examples/servlet/VeryBasicServlet
```

When the Web application is deployed, you should restrict client access to servlet classes. Web sites include a registeredServlets attribute that can be changed to disallow use of the /servlet path and permit access only to servlets which have pre-defined logical names. For details, see “Enabling Use of Registered Servlets” on page 11-8.

During deployment, it is preferable to configure a special mapping to the application’s servlet classes using logical names. In general, the Application Server resolves each URI to a servlet by following two steps:

1. Resolve the request to a WebSite instance.

   A single server may host several distinct sites, each being described by a separate WebSite object. Requests can be mapped either on the basis of the domain name or the first path element. Site aliases may be used to parse the first path component of the request URL. These are similar to the virtual directory facility available on IIS and Apache servers.

2. Resolve any logical names in the URI.

   Logical names may be used to substitute part of the request URI. These names are defined in one of two special configuration files, the first being associated with the server (global to all sites hosted by the server), the second being associated with each site.

More information about URI mapping strategies can be found in the discussion of “Content Management” on page 10-1.

Note: The namespace of the Web site you define for servlets *must* include the namespace in which your servlets and support classes are defined. For details on setting this namespace in the site object, see “Creating and Configuring a Site” on page 5-5.

Servlets Implementation

VisualWorks Application Server includes functionality equivalent to version 2.2 of the Java Servlets specification.

Web applications built around the servlets model make use of protocol in the following classes:

- HttpServlet
- ServletContext
- Request
- Response
- HttpSession
- ServletConfig
- RequestDispatcher

HttpServlet

The abstract superclass HttpServlet provides the basic service framework for initializing servlet instances and processing requests.

Servlet Initialization

After loading and instantiating a servlet, the container allows it to perform one-time initialization (using the methods init or initialize). This is the opportunity for the servlet to read any persistent state information, initialize database connects, etc. During initialization, the servlet is provided with appropriate instances of ServletConfig and ServletContext.

Handling Requests

Once a servlet has been initialized, requests can be processed using the service:response: method. This method dispatches on the HTTP message type, passing the request to the corresponding service method (e.g., doGet:response: for an HTTP **GET** message; doPost:response: for an HTTP **POST** message, etc).

Servlets implement their functionality by overriding the doGet:response: and/or doPost:response: messages implemented in the superclass HttpServlet. It is not recommended to override the service:response: method.

In the case of a single-threaded model servlet, the flow of execution is slightly different. Instead of implementing doGet:response: and doPost:response:, single-threaded model servlets provide doGet and doPost.

Multithreading Servlets

Servlets can be designed to handle threading with one of two models, called the *single thread* model or the *multi-thread* model. By default, the servlet container uses the multi-threaded model.

If a servlet subclasses from SingleThreadModelServlet, or if it implements the class method singleThreadModel, then it will be run under the single-threaded model. The method singleThreadModel must return true.

In the multi-threaded model, the servlet container creates only one instance of the servlet class. For each request, a separate thread is spawned, but all threads run the same method against the same servlet instance. For this reason it is not advisable to hold any state in the instance variables of a multi-threaded servlet. Many threads may be running the same method at once, all attempting to use the same variable.

If a servlet uses the single-thread model, the container creates an instance of the servlet for each request. A single thread is spawned for each request, so each thread will have its own servlet instance. Note that the new servlet instances are created by copying the reference servlet instance, so information like the servlet context and configuration will be shared. Since only one thread will access a particular servlet instance, state may be stored using instance variables.

Single-threaded servlets do not need to accept the request and response objects as message parameters, but instead implement two parameterless service methods doGet and doPut. Class SingleThreadModelServlet defines instance variables for request, response, and session, and automatically populates them before invoking the servlet.

The single-thread model requires slightly more memory allocation than the multi-thread model, but it is more convenient. Given the high-quality garbage collection provided by VisualWorks, the additional overhead from using the single-threaded model is minimal.

Note: The implementations of the request and response objects are not guaranteed to be thread safe. Consequently, the request and response objects should not be passed to other threads that may attempt modifications. Only the thread containing the service method should handle these objects.

Ending Service

The servlet container unloads servlets when an administrator issues a server reset command. When the container unloads a servlet, it allows any existing threads to finish, and then sends the `destroy` message.

When the servlet receives `destroy`, it should release any external resources that may have been allocated during initialization. Once `destroy` has been called, the container ceases to pass requests to the servlet instance.

Servlet Context

Every servlet belongs to a *servlet context* that is provided by the container. The context object gives each servlet a view of the application in its entirety. In a sense, the servlet context is similar to the application object provided by server pages. It is used to share attributes of application scope, for accessing resources on the Web server (e.g., files, graphics), and for reading application-specific initialization information.

A servlet obtains its context using the convenience method servletContext defined in class HttpServlet.

Although a Web application implemented using servlets can be distributed across a number of hosts, the servlet context only resides on a single machine and is not designed for distributed object storage.

Initialization Parameters

Context initialization parameters are available to set up site-specific variables associated with the Web application. For example, these variables might be used to set the name of a database instance, or the login name and password required by the database.

Use initParameter: to add a String value to the context's parameters Dictionary, and initParameterNames to retrieve a collection of all keys in the Dictionary.

Note: Although these parameter values are actually stored in the ServletConfig object, they can only be set by using these access methods.

Setting Context Attributes

Attributes provide a mechanism for all servlets in a single Web application to share small amounts of String data. Class ServletContext provides protocol for setting and reading these attributes. In general, attributes should be used instead of variables of larger scope.

Use attribute: or attribute:ifAbsent: to read the value of an attribute, e.g.:

```
currentUser := self servletContext attribute: 'user_name'.
```

A collection of all attributes defined in the servet context may be obtained using the attributeNames method.

Use setAttribute:to: or attributeAt:put: to create an attribute or change its value, and removeAttribute: or removeAttribute:ifAbsent: to remove one, e.g.:

```
self servletContext setAttribute: 'user_name' to: currentUser.

self servletContext removeAttribute: productCategory.
```

Note that in applications that are implemented using both servlets and server pages, you may share these attributes. The attributes associated with the servlet context may be manipulated as application-scoped variables in your server pages (and vice versa).

Accessing Resources

Class ServletContext provides an interface for resolving the location of any static external resources (HTML and XML documents, GIF or JPG image files) that belong to the Web application. Typically, servlets refer to static content using relative paths, which the servlet context resolves to an absolute path. These resource-access methods are appropriate for accessing static, rather than dynamic content (see "Dispatching" on page 26 for details about accessing dynamic content).

Use getResource: to obtain an URL instance that is mapped to the path that contains the resource:

```
jpgURL := self servletContext getResource: '/insetGraphic.jpg'.
```

This method resolves the URL to the named resource, not its actual content. The result of getResource: is an instance of a subclass of Net.URI. To obtain a ReadStream on the content of the resource:

```
stream := context getResourceAsStream: '/includes/includeme.html'.
result := stream contents.
```

Resources are most commonly documents stored on the Web server's local file system, but they may be located anywhere.

Request

Instances of class Request are used to represent HTTP request messages. The servlet container creates a separate instance for each incoming client request.

Class Request provides protocol for accessing all information in an HTTP request, including the parameter data (passed either in a form or in the URL itself), cookies, x.509 certificate fields, as well as attributes that other servlets may have attached to the request. For more details on HTTP request messages, see "HTTP Request" on page 2-3.

Accessing Parameters

Parameters in an HTTP request are passed either as a part of the URL (in the case of a **GET** message) or in the body of the request (in the case of a **POST** message). A request object may contain different collections of parameter data (query and form data), though each is organized as a collection of name/value pairs.

It is important to note that a single name may contain several different parameter values. If this is the case, the values will be ordered such that those passed in the query string appear first, followed by those passed as part of the form data. See also the discussion of "Multi-Part Forms" on page 7-6.

To access parameter data stored in a request:

```
userName := self getParameter: 'user_name'.
allParams := self getParameterNames.
```

In the case of a single named parameter with multiple values, use getParameterValues:. This method returns an Array containing all the String values associated with the name. Using getParameter: to access a pair containing multiple values will return the first value in the Array.

Using Request Attributes

Attributes are used to attach information to a request object, especially when passing the request from one servlet to another. Attribute data is reserved for use by your application and the servlet container.

Use getAttribute: and setAttribute:value: to manipulate request attributes:

```
"set attribute as flag for servlet B"
    self setAttribute: 'validate' value: true.
...
"check if servlet A requested validation"
    (self getAttribute: 'validate')
```

```
        ifTrue: [self newuserValidation].
```

Each attribute name can have only one value.

Retrieving and Translating the Request Path

The *request path* is the portion of the request URL that identifies the context and active servlet on the VisualWorks Application Server. This path corresponds to the portion of the URL that follows the DNS name of the Web server:

```
URL:            http://mycorp.com/catalog/books/index.ssp
Request URI:    /catalog/books/index.ssp
```

The request path may be further decomposed into three parts:

Servlet Path
: Directly corresponds to the mapping which activates the servlet for this request; begins with **/** (forward slash) character.

Context Path
: Represents the path prefix associated with this servlet's context.

Path Info
: Represents the remainder of the path that belongs neither to the servlet path nor the context path.

Mapping between the URI and the servlet happens in two general ways. For details, see "Mapping Requests to Servlets" on page 6-6.

Thus, the URL shown above might be disassembled as follows:

```
URL:            http://mycorp.com/catalog/books/index.ssp

Context Path:   /catalog
Servlet Path:   /books
Path Info:      index.ssp
```

Class Request provides methods for accessing these three path elements: contextPath, servletPath, and pathInfo. Each returns a String value. Note that depending upon your use of a ServletContext, the context path may be empty, e.g.:

```
URL:            http://search.mycorp.com/search.ssp
```

Here, the context would simply default to the base URL of the Web server. In this case, the String returned by sending contextPath is empty.

A convenience method is also available for translating the request path into a corresponding file system path. To obtain a String representation of the local filename to which the URI path corresponds:

```
localName := self pathTranslated.
```

This method resolves the “path info” (see above) into a local file name.

Accessing Cookies

Web applications use *cookies* to maintain persistent state between individual transactions or across Web sessions. Each HTTP request object includes cookie data cached by the client machine. Only the cookies associated with the particular URL are sent with the request.

Cookies may be simple name/value pairs, or they may be dictionaries. Although you can send cookies to obtain the entire cookie Dictionary associated with the request, you probably want to access a particular cookie by name:

```
lastVisit := self cookieValueAt: 'last_visit_date'.
```

Use allCookieValuesAt: to access a cookie that has a number of values associated with a single key:

```
userDates := self allCookieValuesAt: 'user_dates'.
```

VisualWorks Application Server provides several interfaces for working with cookies. See “Cookies” on page A-1 for additional discussion.

Retrieving the Client’s Locale

A client may optionally specify language and locale preferences that are passed to the Web server via the Accept-Language header or other means available with HTTP/1.1.

A request object may be queried for locale preferences as follows:

```
userLocale := aRequest locale.
acceptedLocales := aRequest locales.
```

If the client has specified at least one locale, the locale method returns an instance of Locale, with priority being given to the first language/locale selected by the client. The locales method returns a collection of Locale objects, ordered from most preferred to least preferred as specified by the client. If no locale is indicated in the request, locale and locales will return the default Locale object assigned to the servlet container.

Clients may also indicate a character set used for encoding FORM data. For details, see “Encoding Form Data” on page 7-7.

For more details about working with Locales, refer to the *VisualWorks Internationalization Guide*.

Secure Sockets Layer

An X.509 client certificate may be associated with the request to implement a secure transfer protocol such as HTTPS. Your application can query for the presence of this certificate using the isSecure method.

Version 7.6 of VisualWorks Application Server can use the SSL support of a front-end Web server, but does not provide its own SSL implementation. Native SSL support will be added in a future release.

Accessing Headers

Request objects provide direct access to the headers in the HTTP request message. These headers are organized as name/value pairs (see RFC-822 for a full discussion) that may be repeated. To retrieve header values, use the access methods in Request.

Use headerNames to obtain a collection of all header names in the request, and header: for a specific value. For example:

```
allHeaders := self headerNames.
(allHeaders includes: 'Content-Type')
    ifTrue: [contentType := self header: 'Content-Type'].
```

Use the convenience method intHeader: to retrieve the value of a header with an integer value:

```
length := self intHeader: 'Content-Length'.
```

Note that if multiple headers with the same name are found in the request, header: and intHeader: will retrieve the first one.

Response

Instances of class Response are used to encapsulate all information associated with an HTTP response message. When a servlet is invoked by the container, it is invoked with both request and response objects. Typically, a servlet begins by setting any header information (data type, cookies, etc.), and then writing the content (the body of the response).

As content is written to the response, it may be sent to the client, depending on the buffering scheme in use. When the response is complete, output buffers are flushed and the complete response sent to the client.

Writing and Buffering Responses

As a servlet assembles a response, the Application Server can either store it in a buffer, or else send each string of HTML to the client as it is passed to the HTTP output stream. Since output buffering can improve server performance, the servlet container enables buffering by default.

Regardless of whether buffering is enabled or disabled, your application should normally use the write: method to send HTML to the client:

```
aResponse write: 'The time is: ' , Time now.
```

The write: message takes a String or ByteArray for its parameter, possibly including HTML tags, e.g.:

```
aResponse write: '<B>The time is: ' , Time now , '</B>'.
```

By default, output buffering is enabled. You may disable it as follows:

```
aResponse buffer: false.
```

When the response is unbuffered, each invocation of write: appends the string parameter directly to the output stream. In practice, this means the HTTP connection remains open between calls to write: and the client's browser reads the unbuffered HTML stream as the servlet writes it.

```
"query the status of output buffering, then enable it"
    [aResponse buffer]
       ifFalse: [aResponse buffer: true].
```

With buffering enabled, the response can be explicitly flushed to the client by sending flushBuffer, or else discarded by sending reset. Use isCommitted to obtain a boolean indicating whether any portion of the output stream has been sent to the client. Once the response has been committed, subsequent attempts to reset the output stream will raise an exception.

If your application requires stream protocol for output, you may also access the response's output stream directly. For example:

```
"send a collection of dates with HTML emphasis"
   responseStream := aResponse outputStream.
   responseStream nextPutAll: 'Important dates: '.
   datesCollection do:
         [:date |
         responseStream nextPutAll: '<B>' , date printString , '</B>, '].
   responseStream nextPutAll: '.<BR>'.
```

In a localized Web application, the output stream will be an initialized instance of EncodedStream that accepts Unicode text.

By default, the Content-Type for the response is 'text/html'. Use the contentType: method to specify a different type/subtype, e.g.:

```
aResponse contentType: 'image/jpeg'.
```

For details on content types, refer to RFC 2231.

If output buffering is disabled (it is enabled by default), contentType: must be used *before* any output is sent to the client using write:.

Passing Cookies to the Client

Both request and response objects may include cookies (see "Accessing Cookies" on page 6-14 for details about reading cookie values).

Cookie data in a response is recorded on the client's machine. When the client's browser receives a response message containing cookies, it will either create new cookies or else update existing ones to reflect the values of the cookies passed in the response.

To set the value of a cookie on a client machine, use cookieAt:put:.

```
response cookieAt: 'user_name' put: 'Xavier'.
```

Note: Although two types of cookies are generally available under HTTP, the VisualWorks Application Server currently provides support only for single value cookies. This limitation may be resolved in a future release.

To examine the OrderedCollection of cookies in the response:

```
cookieCollection := response cookies.
```

The domain, expiration date, path and secure properties of the cookie will be set automatically by the Application Server. By default, the domain of the cookie will be the domain of the server, and the cookie will be set to expire at the end of the client's session. The path property will default to the virtual path of the Web application in use (e.g., /MyApp/Home/), and the secure setting will be false.

Internally, cookies are represented using instances of class HTTPCookie (for a more complete discussion, see "Cookies" on page A-1).
To set any of the cookie properties other than name and value, create an instance of HTTPCookie and use the addCookie: method:

```
"create a cookie, enable the use of SSL, then add it to the response header"
   cookie := HTTPCookie named: 'user_password' value: 'secret'.
   cookie domain: (request serverVariableAt: 'SERVER_NAME').
   cookie secure: true.
   response addCookie: cookie.
```

Note that if the response output is unbuffered, cookies must be created *before* any output is sent to the client (because cookies are passed in the header of the HTTP response, using a **SET-COOKIE** header).

Note: Certain browsers (e.g., Netscape Navigator) are case sensitive in their treatment of URLs and filenames. To avoid problems when resolving the names of cookies, it is good practice to use names spelled in a consistent case (all lower case, for example).

Redirection

Clients may be redirected to another URL either via a client-side redirect, or by transferring control on the server side, inside the Web application.

Use the convenience method sendRedirect: or redirectTo: to request that the client's browser redirect to another URL:

```
aResponse sendRedirect: 'http://mycorp.com/index.html'.

aResponse redirectTo: 'http://noBiz.org/authenticate.html'.
```

The redirection parameter may be either a full, or a relative URL. If a relative URL is used (e.g., /index.html), the servlet container translates it into a fully qualified URL, raising an exception if it cannot.

If the redirection is to another page in the same application, you may also forward control to another servlet using a dispatcher. This takes place entirely on the server (see "Dispatching" on page 6-26 for details).

Error Notification

Two convenience methods are provided for communicating runtime error conditions to the client. These methods require the three-digit status Integer that gets sent to the client's browser, e.g.:

```
"tell client we've encountered a server error"
   aResponse sendError: 500.
```

Optionally, String data may be sent in the content body of the response.

```
"send standard 404 -- not found error"
   aResponse sendError: 404
                        message: '<B>This item could not be found.</B>'
```

Any content that is buffered for output may be sent, but attempting to write: after using sendError: or sendError:message: will raise an exception.

Specifying Character Sets and Content Type

Use charSet: to specify the character set for the content of the HTTP response. For example:

```
aResponse charSet: 'ISO-8859-1'
```

Specifying a character set by setting the attribute in a response object overrides the whatever character set is associated with the session.

If no character set is specified, the client uses its platform native encoding. PC and compatible clients assume ISO-LATIN-1, while MacOS clients default to the Mac Roman encoding. Your application should specify a character set appropriate for its textual content.

Note: When using encoded streams or other features of the VisualWorks Internationalization package, note that character sets are specified using instances of Symbol, not String.

By default, the Content-Type for the response is 'text/html'. Use the contentType: method to specify a different type/subtype, e.g.:

```
aResponse contentType: 'image/jpeg'.
```

The String parameter must be in a type/subtype format. The default value is 'text/html'. For details on content types, refer to RFC 2231.

If output buffering is disabled, charSet: and contentType: should be used *before* any output is sent to the client using write:.

When used in conjunction with contentType:, the response object may be used to send graphics, database objects, or other media types to a client. Generally, servlets are better for sending custom media types.

For example, the servlet shown below may be used to send a GIF image to a client. Note: this code requires loading the GIFEncoder package into your development image.

```
doGet: aRequest response: aResponse
    | pix gc gifImage |
    pix := Pixmap extent: 500 @ 400.
    gc := pix graphicsContext.
    gc paint: ColorValue blue.
    gc
        displayWedgeBoundedBy: (20 asPoint extent: pix extent - 20)
        startAngle: 20
        sweepAngle: 320.
    gc paint: ColorValue red.
    gc displayString: Time now printString at: 5 @ 20.
    gifImage := pix asImage asGIFNonTransparent.
    aResponse
        contentType: 'image/gif';
        write: gifImage gifBytes
```

This servlet creates a simple Pixmap, draws into it, and then converts it to a GIFImage object, which is written to the response stream.

Setting Language or Locale Attributes

If a client specifies preferred language or locale information, a servlet can set the language/locale attributes of a response object. These attributes are generally sent as HTTP message headers (additional mechanisms are available in HTTP/1.1).

Use locale: in conjunction with an instance of Locale to set the various locale attributes of a response:

```
"create a locale and use it to set the locale of a response"
    locale := Locale new.
    locale
        currencyPolicy: (NumberPrintPolicy newFor: #fr);
        numberPolicy: (NumberPrintPolicy newFor: #fr);
        timePolicy: (TimestampPrintPolicy newFor: #fr).
    locale name: #'Français'.
  aReponse locale: locale.
```

For example, to set the locale of a response to correspond to the client’s browser's characteristics:

```
aResponse locale: request locale
```

Each session object also contains a locale, which is used by default when writing output to a response object (for details on locales and sessions, see "Specifying a Locale" on page 6-24).

If output buffering is disabled (it is enabled by default), locale: should be used *before* any output is sent to the client using write:.

Note: by default, the web locale is the ISO-8859-1 character set since all clients must accept this encoding (for details, see: RFC 2068, section 3.7.1 or 14.2). However, if you set the locale using locale:, either on the response or session object, this takes precedence over the default.

For more details about working with Locales, refer to the *VisualWorks Internationalization Guide*.

Accessing Headers

Servlets can directly access the HTTP headers of a response object using the following protocol. These headers are used to pass cookies, character encoding, and similar parameters. Although specialized protocol is defined for setting the most common response properties, you may want to add custom headers directly to the response object.

Use setHeader:value: to add a new header:

```
"set the 'pragma' attribute to disable browser caching"
   aResponse setHeader: 'Pragma' value: 'no-cache'.
```

The convenience methods addDateHeader:value: and addIntHeader:value: are also provided for setting Date and Integer values in a header.

Note: When using custom headers, it is best not to choose names with embedded underscores, as these might collide with predefined header values used by HTTP, producing unpredictable results.

If output buffering is disabled (it is enabled by default), these messages should be used before *any* output is sent to the client using write:.

Session

Servlets support the notion of a *session object* that may be used to track the client's progress through the different stages of a Web application. Since the HTTP messaging protocol is by nature stateless, client sessions are managed by the servlet container. By establishing a distinct session for each new client, the application can give meaningful continuity to all subsequent transactions with the same client.

Sessions allow the application to track user-identification and user-specific data across a number of discrete page transactions between client and server. All variables of session scope may be shared by all servlets belonging to the same servlet context.

A session object is created when a user first requests a page, and it exists until either the client abandons the session or the server concludes it. Sessions can remain inactive until a specified timeout, at which time they will be invalidated by the server.

A number of different mechanisms are used for session tracking, though your application generally need not concern itself with these details.

Establishing a Session

Sending the session message to a request object will return the appropriate session object associated with that request. If no session object exists, one is created and associated with the client.

Your application can use isNew to test whether a session has just been established.

```
mySession := self request session.
   mySession isNew
      ifTrue:   ["new session -- redirect user to starting page"
                 aResponse sendRedirect: 'http://mycorp.com/start.html'].
```

Tracking Sessions

Sessions are tracked using a cookie named sessionKey that is passed transparently by the server. It is possible that the client may open several different windows to view your application. Although each window can show a separate view, the browser will nevertheless associate the same sessionKey cookie with each window. The developer should design the Web application to anticipate this situation.

If the client's browser does not accept cookies, the servlet container will detect the condition and the session ID will be encoded into the URL as a query string. This technique is known as “URL rewriting”.

Note: URL rewriting for sessions is currently unimplemented; thus, clients *must* have cookies enabled to for session tracking to function properly with release 7.6 of VisualWorks Application Server. This limitation will be removed in a subsequent release.

Each session is identified using a unique ID. In general, your application should not need to access the session ID directly, though if you do make use of this ID, it should *never* be cached or otherwise used as a mechanism to identify a particular client beyond the duration of a session.

The request object may be queried to determine whether the session ID has been established using cookies or using URL rewriting.

```
aRequest isRequestedSessionIdFromCookie]
     ifTrue: [self weCanUseCookies].

aRequest isRequestedSessionIdFromURL
   ifTrue: [self weUseURLrewriting].
```

Binding Session Attributes

Servlets can associate name/value attributes with their assigned session object. The scope of session attributes allows access to all servlets belonging to the same servlet context. Use the following methods to manipulate the dictionary of session attributes:

```
"retrieve the value of an attribute"
   userName := mySession attribute: 'user_name'.

"set the value of an attribute"
   mySession setAttribute: 'user_name' to: userName.

"remove an attribute from the Dictionary"
   mySession removeAttribute: 'preferred_category'.

"obtain a Set of all attribute names in the session"
   sessionVars := mySession attributeNames.
```

As a convenience, you may also use some Dictionary protocol directly on the session object, including at: and at:put:. You can also ask for the session contents, which returns a dictionary containing all session attributes.

Note that in applications that are implemented using both servlets and server pages, you may share these attributes. The attributes associated with a session may be manipulated as session-scoped variables in your server pages (and vice versa).

Setting the Session Timeout

Class HttpSession provides a timeout mechanism for expiring inactive sessions. The default value is defined by the servlet container, but may be changed by your application.

```
"set the session timeout to at least 20 minutes (20 * 60 seconds)"
   mySession := self request session.
   mySession maxInactiveInterval < 1200
     ifTrue: [mySession maxInactiveInterval: 1200].
```

The timeout value is set in seconds. If user sessions tend to be short, you should consider lowering the timeout value to conserve memory resources on the server machine.

Note: Though it is not advised, specifying a value of -1 indicates that the session should never expire.

Ending a Session

In applications where session-based transactions must be secure, it is often desirable to allow the user to log out of the session. This feature is often used in banking service and internet e-mail systems. When explicitly terminating a session, all session-scoped variables and associated memory resources are released. The next request from the client will prompt the servlet container to create a new session.

Use the methods abandon or invalidate to end a session, e.g.:

```
"terminate the session"
   mySession := self request session.
   mySession invalidate.
```

In general, it is a good idea to end the client's session using abandon or invalidate, otherwise the servlet container preserves the client's session state until a timeout occurs.

Accessing the session object after it has been invalidated will raise an exception.

Specifying a Locale

Each session object carries a locale attribute that corresponds to a standard international country code. This attribute is used when creating responses and is used to indicate how information like dates and currency should be formatted in the output stream. In North America, for example, dates are formatted as month/day/year, whereas in Europe they are formatted day/month/year.

Locale information is held in the session object, and persists as long as the session exists. By default, the Application server sets the locale and character set automatically when the session is created, based upon the **`Accept-Language`** header of the request object (however, this can be turned off in the settings pages). Your application can also specify a different locale by explicitly setting the attribute in a response object.

Use `locale:` with an instance of `Locale` to change the locale ID that will be received by the client's browser. For example, to set the session to the current locale used by the server:

```
self request session locale: Locale current.
```

Clients may optionally specify their language and locale preferences using the **`Accept-Language`** header attached to an HTTP request.

For details about working with `Locale`, refer to the *VisualWorks Internationalization Guide*.

Sessions and Character Sets

Each session object includes an attribute that specifies the encoding (character set) used by default when writing output to response objects. Encoding information is held in the session object, and persists as long as the session exists. Note that encoding and locale attributes are treated by the Application server as distinct values.

By default, the Application server selects the encoding automatically when the session is created, based upon the **`Accept-Charset`** header of the request object (however, this can be turned off from the **Character Sets and Locales** settings page). Your application can also specify a different character set by explicitly setting the attribute in a response object.

Clients may also indicate a character set used for encoding FORM data. For details, see "Encoding Form Data" on page 7-7.

Dispatching

It is often desirable to factor a Web application such that one servlet dispatches a request for additional processing by other servlets. By chaining different servlets together, specialized behavior can be localized and reused in a smaller number of components.

To dispatch a request, the destination path must be wrapped using an instance of RequestDispatcher. The destination path can point to either a servlet, a static page, a dynamic page, or another type of resource.

Creating a Request Dispatcher

The servlet context provides methods to delegate requests to other servlets or server pages using a request dispatcher object. To obtain a dispatcher object:

```
dispatcher := self servletContext getRequestDispatcher: '/catalog/Search'.
```

The String parameter to getRequestDispatcher: must describe a relative path within the scope of the servlet context.

It is also possible to query the request object using an analogous method (of the same name) in class Request. Using the request object to resolve the path has the advantage that a full path relative to the base of the servlet context is not mandatory.

For example, let’s say we want to resolve the path to the Search servlet in the context of /bookCatalog.

To resolve a path using the context, we must use '/bookCatalog/Search' as the path parameter, but if we use the request object the path does not need to be relative to the servlet context:

```
dispatcher := request getRequestDispatcher: '/Search'.
```

Using a Request Dispatcher

Requests may be dispatched in one of two ways: the caller can either *forward* the request to the called servlet, or the caller can *include* the response of the called servlet.

For example, to forward a request from inside a service method:

```
doGet: aRequest response: aResponse
   ...
   "wrap a dispatcher"
   dispatcher := self servletContext getRequestDispatcher: '/servlet/Search'.
   dispatcher forward: aRequest response: aResponse.
```

Using forward:response: will pass control to the /Search servlet. The thread of execution is transferred entirely to /Search, and the remainder of the caller's service routine will be ignored.

A request may be forwarded at any time. If content has been written to the response, the headers of the final response will be whatever headers the original servlet had written, but the content will come from both the original servlet and the servlet that received the forwarded request.

To dispatch so that the target servlet will eventually return control, use include:response: instead of forward:response:, e.g.:

```
doGet: aRequest response: aResponse
    ...
    "wrap a dispatcher"
    dispatcher := self servletContext
                            getRequestDispatcher: '/servlet/NewsUpdate'.
    dispatcher include: aRequest response: aResponse.
```

The path for the new request may be an absolute path, or a relative path (in which case it is relative to the directory of the current request, rather than to the home directory of the current web site).

The "included" servlet is granted unrestricted access to the request object, though query parameters in the request object are not forwarded.

Including Query Strings in Dispatcher Paths

By default, query parameters of the current request are not automatically included in a request that is dispatched. However, query strings may optionally be included in the path resolved for a dispatcher.

To pass query parameters, you may extract the queryString from the current request and append it to the path of the new request when constructing the RequestDispatcher object.

For example, on a server page, or in a servlet:

```
dispatcher := self servletContext
                        getRequestDispatcher:
                            '/servlet/Search?', (self request queryString).
```

7

Server Page Applications

Server pages are a language-independent technology for server-side scripting. VisualWorks Application Server includes a scripting interface and application models to support Smalltalk server pages (SSP). The scripting interface uses a templating mechanism to combine static HTML pages with Smalltalk script.

The VisualWorks Web Toolkit supports the development of applications that are constructed following either an ASP or a JSP model. In addition to using the standard APIs for each model, your application can access components from the entire Smalltalk class library.

Web applications built using server pages can maintain persistent state both during and between client sessions. Persistant state may easily be shared between servlets and server pages. Several mechanisms are provided in the application framework to manage client state information.

Server pages can also use XML-style tags and include custom tag libraries to define special-purpose scripting actions. For details, see "Predefined Scripting Actions" on page 8-7.

Understanding Server Pages

The VisualWorks Application Server resolves incoming requests to files, server pages, servlets, or Smalltalk methods. If the request is resolved to a server page, the scripting engine evaluates the page, generating a new document that is returned to the client. The Application Server treats any requested file with an extension of **.ssp** (preferred), **.jsp** or **.asp** as a Smalltalk Server Page.

Server pages follow the general model of a typical HTTP transaction. Roughly speaking, the script may be viewed as a single method that is evaluated within a special execution context. The script takes its input from a request object and writes its output to a response object. For a general overview of this request/response mechanism used by HTTP, see “Web Transactions” on page 2-2.

The scripting elements of a server page can reference a number of *implicit objects* that represent the state of the transaction and the application.

In the following expression, for example, the method cookieValueAt: is used to fetch a cookie value from the implicit variable that represents a request:

```
userName := request cookieValueAt: 'user_ID'.
```

Five implicit objects are available at any time: request, response, session, application, and out. The first four of these objects are used to store attributes, session or application variables. Class protocol for each is described in “VisualWorks Implementation” on page 7-4.

The implicit objects that are accessible as variables belong to one of four possible scopes: page, request, session, application:

Variable	Scope	Description
request	request	Request objects may be forwarded from one page to another; thus, each request has a distinct scope.
response	page	Response objects belong to a single page.
out	page	Response output stream.
local	page	Any variables declared on the page are treated like method variables.
session	session	Variables and attributes are accessible by all pages that belong to the same client session.
application	application	Variables and attributes shared between all sessions of a single Web application.

Example: Server Pages

A Smalltalk server page begins life as a document formatted in HTML. The scripting elements that are evaluated when a client requests the page are embedded in the HTML using the special delimiters <% ... %>.

To illustrate how server pages work, consider the following examples:

```
<%@ language="SScript" %>
<html>
<head>
<title>Smalltalk Server Page Example</title>
</head>
<body>
<% theHour := Time now hours. %>
<%= theHour > 18
     ifTrue: ['Good evening.']
        ifFalse: [theHour > 12
                    ifTrue:['Good afternoon.']
                    ifFalse: ['Good morning.']]. %>
</body>
</html>
```

Here, the first scripting action stores the hour in a local variable theHour. The second tests theHour and uses an output expression <%= ... %> to insert the appropriate string (e.g., 'Good morning') into the template.

Testing the Example

To test this server page:

1 Save the script file in the home directory.

Be sure to use the **.ssp** file extension when saving the script file, e.g., **C:\visualworks7\web\example1.ssp**.

2 Set up a Web site (for details, see "Creating and Configuring a Site" on page 5-5).

3 Open a Server Console to start a Smalltalk Server on localhost:8008 (for details, see "Using a Web Server" on page 4-2).

4 Launch a Web browser and enter the following URL:

http://localhost:8008/example1.ssp

The text "Good morning.", "Good afternoon." or "Good evening." should appear in the browser.

Additional server page examples can be found in the **\web\examples** directory of the VisualWorks installation.

VisualWorks Implementation

The VisualWorks Application Server provides class protocol analogous to the developer's interface to ASP/JSP. When using the server page application model, implicit variables may be referenced by scriptlets and expressions. The implicit variable names, and the classes which provide their protocol are summarized in the following table:

Variable	Class	Description
request	Request	Instances represent the contents and all parameters received by the Web application with incoming HTTP requests. Corresponds to the ASP or JSP Request object.
response	Response	Used to compose and send HTTP response messages to the client. Provides protocol for sending individual responses, creating cookies, and sending general information about the outgoing content.
out	Response	Provides direct access to the response stream.
session	HttpSession	Maintains variables that are specific to a particular client session. Instantiated for each active session of the Web application. Corresponds to the ASP Session object and the Servlet HttpSession class.
application	HttpApplication	Maintains variables shared between all sessions of a single application. Instantiated once when the application is started, and released when it is shut down. Corresponds to the ASP Application or JSP ServletContext object.

Aspects of the server environment are controlled using the class protocol provided by ASPServer. This corresponds to the ASP server object.

Support for ASP-style scripting elements is described in the following section. For details on support for JSP-style scripting elements, see "Server Page Extensions" on page 9-1.

Request

Web applications are built around a message-based request/response model, with the requests always being sent by the client. An incoming request may belong to an active session, or it may initiate a new one. For every HTTP request received by the server, a distinct request object is instantiated. Within the context of a particular Smalltalk Server Page, this object is accessed as a variable named request.

Class Request provides protocol for accessing all information in an incoming HTTP request, including the parameter data (passed either in a form or in the URL itself), cookies, x.509 certificate fields, or gateway environment variables (also called *server variables*) that come with each HTTP request.

Accessing Parameters

To retrieve the parameters passed to an HTTP request, your application may use three different access protocols to class Request.

Generally, requests that use forms employ the **POST** method. All parameter data accompanying a **POST** message is passed in the body of the HTTP request.

For example, the following excerpt of HTML may be used to submit a book search request using **POST**:

```
<FORM ACTION="http://mycompany.com/search.ssp" METHOD="POST">
    Author: <INPUT TYPE="text" NAME="author" SIZE=40><BR>
    Title: <INPUT TYPE="text" NAME="title" SIZE=40><BR>
    <INPUT TYPE="hidden" NAME="details" VALUE="no">
    <INPUT TYPE="search" VALUE="Search">
    <INPUT TYPE="cancel" VALUE="Cancel">
    </FORM>
```

The Web application then uses script elements to access this parameter data from an instance of Request:

```
showDetails := request anyFormValueAt: 'details'.
bookName := request anyFormValueAt: 'author'.
bookTitle := request anyFormValueAt: 'title'.
```

These statements save the parameters as local variables on the server page, so that their values may be passed to the component that actually performs the book search.

Alternately, HTTP requests may use the **GET** method (sometimes called a *query*) to pass parameter data in the URL. Here, a list of name/value pairs are appended to the URL following a **?** (question mark) character and separated by **&** (ampersand).

In the example shown above, submitting the form to the book searching application might result in a new page containing list of books found, each of which is presented to the client as a hyperlink.

A typical link might be as follows:

```
http://mycompany.com/search.ssp?details=yes&author=Plato&title=Republic
```

This URL, when selected, sends an HTTP **GET** message with three parameters: one to specify that the client wants details; one for author and one for title. When the server handling SSPs receives this HTTP request, a Request object will be created that holds the three parameters in its queryString collection.

The Web application can access the elements in the queryString collection using the following expressions:

```
showDetails := request anyQueryValueAt: 'details'.
bookName := request anyQueryValueAt: 'author'.
bookTitle := request anyQueryValueAt: 'title'.
```

Finally, class Request also provides a single message that may be used to access both query and form data:

```
showDetails := request anyParameterValueAt: 'details'.
bookName := request anyParameterValueAt: 'author'.
bookTitle := request anyParameterValueAt: 'title'.
```

The access methods anyParameterValueAt: and allParameterValuesAt: will search through the queryString collection, then the form collection, and lastly the cookies collection.

Multi-Part Forms

Multi-part forms may be used when capturing a large amount of form input from a client, or transferring data from client to server (uploading). They offer several advantages over a simple **POST** operation. The Application Server supports multi-part forms, using instances of class MimeEntity to represent the form data.

No additional protocol is provided for accessing multi-part form data. Sending anyParameterValueAt: to the request object normally returns a String, but will instead return an instance of MimeEntity when multi-part forms are used. To obtain the value of the MimeEntity, you must send it the #value message twice, e.g.:

```
uploadFileName := (request anyFormValueAt: 'fileName') value value.
```

Two pieces of example code are provided to demonstrate the use of multi-part forms: class SimpleFileUploadServlet and the server page **$(VISUALWORKS)/web/examples/fileuploadtest.ssp**.

Encoding Form Data

When a web request is sent, any form data containing non-ASCII characters is encoded by the client's browser. The URL-encoding scheme is employed, and out-of-range characters are represented using %HH notation. However, the particular encoding is not specified in the request, so the server application must know how to decode these characters.

Encoding can be a problem even within a particular locale, because different clients may be using different character sets, and they may differ only in a few characters. For example, in the US-English locale, the ISO-8859-1 character set (common on Unix) mostly overlaps with the Microsoft Code Page 1252, but a few characters are different. The Microsoft quote characters, for instance, are not included and can pose problems if the ISO-8859-1 encoding is used.

There are several distinct steps that must be taken to ensure that form data is encoded and decoded predictably.

First, when creating forms that may contain non-ASCII characters, the META tag should be used to specify the character set. For example:

```
<META http-equiv="Content-Type" content="text/html; charset=utf-8" />
```

When in doubt, UTF-8 is a safe choice because it captures all Wetsern and Eastern characters.

Note that when using the META tag in this way, the page must also be served in the specified encoding. By default, the VisualWorks Application Server uses UTF-8. To change this, use the setting **Charset for Form Data** on the **Character Sets and Locales** settings page (you may also wish to disable **Set Session Charset from Initial Request**), or WebToolkitSettings>>formEncoding.

Second, the server page should instruct the browser to use a known character set by including an **`Accept-charset`** parameter in the form definition. For example:

```
<FORM Accept-charset="utf-8">
```

Normally, the browser uses this parameter to include **`Accept-charset`** among the HTTP headers for the request sent to the server.

Caution: Several browsers, including Internet Explorer 6.0, do not generate any **`Accept-Charset`** headers, though they do try to follow the encoding specified in the META tag.

Due to known limitations with several browsers (e.g., Internet Explorer), your application should emply both the META tag for each page and the **`Accept-charset`** parameter for each FORM definition.

Cookies

Cookies enable Web developers to maintain client persistent state across Web sessions. Request objects may contain cookie information that can be accessed within a server page. The Application Server provides several different interfaces for working with cookies (for a more complete discussion, see "Cookies" on page A-1).

Cookies may be simple name/value pairs, or they may be dictionaries. Starting with an instance of Request, your application can access a particular cookie by name using the following expression:

```
lastVisit := request cookieValueAt: 'last_visit_date'.
```

If several values belong to a single name, cookieValueAt: will return the first; if none, it will return nil. To obtain all cookies associated with a particular key:

```
userDates := request allCookieValuesAt: 'user_dates'.
```

Server Variables

Request objects provide access to a collection of predefined environment variables containing specialized information about the client's HTTP request. For example:

```
clientAddress := request serverVariableAt: 'REMOTE_HOST'.
```

This returns the name of the remote machine hosting the client, as it would appear on a DNS lookup. For a complete description of the environment variables available in an request object, see "Server Variables" on page 2-4.

Retrieving the Client's Locale

A client may optionally specify language and locale preferences that are passed to the Web server via the Accept-Language header.

Query the request object for locale preferences as follows:

```
userLocale := request locale.
acceptedLocales := request locales.
```

If the client has specified at least one locale, the locale method returns an instance of Locale, with priority being given to the first language/locale selected by the client. The locales method returns a collection of Locale objects, ordered from most to least preferred as specified by the client. If no locale is indicated in the request, locale and locales will return the default Locale object assigned by the Application server.

For more details about working with Locales, refer to the *VisualWorks Internationalization Guide*.

Response

A *response object* represents an HTTP response sent from the Web application to the client. Within a Smalltalk Server page, this object may be referenced as a local variable named response. Class Response provides protocol for writing HTML information to the outgoing HTTP response, including the content that will be sent to the client (the response body), header information specifying the content's data type, and cookies. The response object also provides protocol for controlling the buffering of the actual HTTP message sent to the client.

You may use the response object to set the expiration time of a given page, to specify whether or not the page may be cached by a proxy server, and to query whether or not the client is still connected to the Web application.

As a general strategy, you should set any special header values in the response *before* beginning to write HTML content to it. Failure to observe this rule may cause header information to be lost en route to the client.

Writing and Buffering Responses

Generally, text is sent to the HTTP stream using an output expression. For example, the following expression sends the current time to the output stream:

```
The time is: <%= Time now printString %>
```

There are occasions when you may wish to pass text to the output stream using an explicit message send within Smalltalk code:

```
<% response write: 'The time is: ' , Time now printString. %>
```

This yields the same result as the output expression, using <%= ... %>.

The write: message takes a String or ByteArray for its parameter. You can also pass HTML tags directly to the output stream using write:.
For example:

```
<% response write: '<B>The time is: ' , Time now printString , '</B>'. %>
```

When writing to the response object, applications written using server pages use essentially the same message protocol as those implemented using servlets. For details, and a discussion of response buffering, see "Writing and Buffering Responses" on page 6-16.

Setting Character Sets and Content Type

You may set both the character set and MIME content type attributes of the response object. As a general rule, servlets are better for sending custom media types to a client's browser. For details, and an illustration of how to send non-textual media types, see "Specifying Character Sets and Content Type" on page 6-19.

Setting Language or Locale Attributes

If a client specifies preferred language or locale information, you may wish to set the language/locale attributes of a response object. These attributes are generally sent as HTTP message headers (additional mechanisms are available in HTTP/1.1).

When manipulating language and locale settings, applications written using server pages use essentially the same message protocol as those implemented using servlets. For details, see "Setting Language or Locale Attributes" on page 6-20.

Setting the Expiration Time

Use expires: and expiresAbsolute: to set the length of time that the client machine should cache the response page. If the client returns to view the page before the specified expiration time, the Web browser will display the copy stored in its local cache. For example:

```
"expires in one hour"
    response expires: 60.
```

The browser can be directed to never cache the page by sending expires: with a value of zero. This will cause the browser to reload the page from the server on every re-viewing.

Use expiresAbsolute: to specify a date and/or time (using an instance of class Timestamp) when the browser should invalidate the cached page:

```
response expiresAbsolute: expirationTimestamp.
```

Alternately, you may omit either the date or time. If no date is specified, the browser will invalidate the cached page at midnight of the current day; if no time is specified, the browser uses midnight of the specified date.

Since the client and server are very often in different time zones, GMT time is used so that there is no confusion about the expiration time. The Application Server automatically converts this expiration time to GMT before sending the response to the client.

If output buffering is disabled (it is enabled by default), expires: and expiresAbsolute: should be used *before* any output is sent to the client. For additional details, see "Controlling Caching" on page 7-12.

Creating and Updating Cookies

Cookies are passed to the client via the response object. When working with cookies, applications written using server pages use essentially the same message protocol as those implemented using servlets.

For details, see "Passing Cookies to the Client" on page 6-17.

Testing the Client's Connection

During lengthy script operations, it may be desirable to discontinue script execution if the client has disconnected or moved to another page. Use isClientConnected to test the status of the connection. For example:

```
response isClientConnected
    ifFalse: [response end].
```

The method isClientConnected returns true as long as the HTTP response stream to the client is open.

Response Status

Use status: to specify the three-digit status Integer that is passed to the client's browser with the HTTP response.

For example:

```
response status: 200.
...
response status: 404
            reasonString: 'The requested page could not be found.'.
```

Generally, status: is used to indicate unavailable resources or internal error conditions. For a complete description of available status categories, see "Response Status" on page 2-6.

Redirection

Use redirectTo: to request the client's browser to redirect to another URL:

```
response redirectTo: '/index.html'.
```

The redirectTo: message takes a parameter which is either a relative URL (e.g., /catalog/search.ssp), or a full one (e.g., http://mycorp.com/index.html).

If output buffering is disabled (it is enabled by default), the message redirectTo: must be used *before* any content is written to the response.

If the redirection is to another page in the same application, you may also use the ASPServer method transfer:for:, which performs the redirection on the server, rather than the client side. For details on server-side redirection, see "Dispatching and Transferring Execution of a Script" on page 7-18.

Controlling Caching

Use cacheControl: to indicate that proxy servers can cache the page. By default, pages are not cached, but for large pages with static content, clients may receive faster responses from a local proxy server.

Note: The cacheControl: option has no effect over local client caching, only the behavior of proxy servers.

By default, this setting is 'Private'; to disable caching. To enable caching:

```
response cacheControl: 'Public'.
```

For additional details, see "Setting the Expiration Time" on page 7-10.

Accessing the Response Header

Each response object contains a collection of HTTP message headers used to pass cookies, character encoding, and similar parameters. Although class Response provides some access protocol, under certain circumstances you may want to add headers directly to the response object.

Use addHeader:value: to append a message header:

```
response addHeader: 'My-Header' value: 'SomeToken'.
```

If you add a header with a previously defined name, the new value is included in the message headers collection, and all headers will be sent with the response.

To examine message headers sent by the client, including custom headers, send the serverVariableAt: message to the request object.

Note: When defining custom headers, it is best not to use names with embedded underscores, as these might collide with predefined header values used by HTTP, producing unpredictable results.

If output buffering is disabled (it is enabled by default), addHeader:value: should be used *before* any output is sent to the client.

Logging

Use appendToLog: to append a String to the server's W3C-format log file.

Note: The appendToLog: method is unimplemented in version 7.6 of VisualWorks Application Server.

Application

The Application object provides a way to share state between all users of an application. It has one instance variable, contents, which is a Dictionary used to hold variables of application scope. When a Web application is first started, a single instance of class HttpApplication is created. It exists until the application is unloaded or the server is shut down.

Class HttpApplication should not be subclassed, and in general its instances should be used sparingly. Typically, the contents Dictionary is initialized at startup to contain constants that are global to the application. For example, we might place the following expression within a server-side include:

```
application at: 'Catalog' put: 'Books'.
```

Subsequently, during the execution of the application, we can access the value of 'Catalog' using this expression:

```
application at: 'Catalog'.
```

Application Events

The application object provides a simple event mechanism that may be used to initialize application variables during startup or to save them at shutdown. Presently, two events are defined: ApplicationStartup and ApplicationShutdown (these correspond to the ASP-style events named onStart and onEnd).

The ApplicationStartup event is triggered once when the first incoming request is received to launch a Web application. The ApplicationShutdown event is triggered when the Web application is about to be shut down, typically when an administrator stops it using the Server Console. Like the ApplicationStartup event, ApplicationShutdown is only triggered once. Since there may be active Web sessions when the shutdown event is triggered, changes to some application variables may be lost.

Events are handled by *callback methods*. Each must be a class method, with a single parameter for the object for which the event is triggered (i.e., the application object). To receive event notification from the Application Server, these callback methods must be registered in an initialization file.

For details, see “Event Callbacks” on page 5-11.

It should be noted that the Application Server does not configure itself (i.e. read and install the configuration from the designated configuration files) until it receives the first request through a listening server.

For applications which may wish to perform their own configuration, a private, internal event is sent to signal the completion of Application Server configuration.

Note: This is a private event defined by the VisualWorks Application Server, and is not the same as the ApplicationStartup event, which occurs for each web site as it is configured.

To catch this configuration-completion event and pass control to your own application configuration logic, you may use a piece of code like the following in a class-side initialize method or in the postLoad action for your application's parcel:

```
WebConfigurationManager
    when: #finishedServerConfiguration
    send: #configureMyApplication
    to: MyApplicationConfigurator.
```

Parallelism

VisualWorks Application Server supports parallel processing within an application. By default, each incoming Web request spawns a separate Smalltalk process. In general, care should be taken to ensure that all code which modifies application or session variables is thread safe.

Note: It is the developer's responsibility to protect modifications to application variables using critical sections.

Multiple pages can also be chained together within the same thread using the execute:for: and transfer:for: methods provided by the server object (for details, see "Dispatching and Transferring Execution of a Script" on page 7-18).

Session

Since the basic HTTP messaging protocol is stateless, the VisualWorks Application Server provides a transparent “session” interface that makes it possible to associate state with each active client. Session objects are used to track user-identification and user-specific data across a number of discrete transactions between the client and Web application.

When a client first requests a server page in a Web application, the server creates an instance of class HttpSession and initializes it for that client. This session object exists until the site visit is concluded, either because the client has abandoned the session, or because the server has concluded it. A session can remain inactive until a specified timeout (20 minutes by default), at which point it is terminated by the server.

Session objects are typically used as a place to store data that can be accessed across a number of different pages. They may also be used to hold the client’s movement through a series of pages (especially when entering form data).

Accessing Session Variables

Each session object holds a Dictionary of *session variables* which may be shared by all pages in the application. Use at:put: to add a key to this Dictionary, e.g.:

```
session at: 'userName' put: 'Marie'.
```

To access the value of a variable during the session:

```
name := session at: 'userName'.
name := session at: 'userName' ifAbsent: [self getNewUsername].
```

To remove the variable from the Dictionary:

```
session removeKey: 'userName'.
```

To obtain a Set of all variable names in the session:

```
sessionVars := session keys.
```

Specifying a Locale

Each response object carries a locale attribute that corresponds to a standard international country code. This attribute indicates how information like dates and currency should be formatted by the browser. In North America, for example, dates are formatted as month/day/year, whereas in Europe they are formatted day/month/year.

By default, the Application server sets the locale and character set automatically when the session is created, based upon the **`Accept-Language`** header of the request object (however, this can be turned off in the settings pages). Your application can also specify a different locale by explicitly setting the attribute in a response object. Locale information is held in the session object, and persists as long as the session exists.

Use locale: with an instance of Locale to change the locale ID that will be received by the client's browser. For example, to set the session to the current locale used by the server:

```
session locale: Locale current.
```

Clients may optionally specify their language and locale preferences using the **`Accept-Language`** header attached to an HTTP request.

For details about working with Locale, refer to the *VisualWorks Internationalization Guide*.

Setting the Session Timeout

By default, the Application Server will expire all sessions that have been inactive for 20 minutes. Use timeout: to change this interval:

```
"set session timeout to 30 minutes"
   session timeout: 30.
```

If user sessions tend to be short, you should consider lowering the timeout value to conserve memory resources on the server machine. Increasing the timeout value increases the amount of idle session state the server must maintain.

Abandoning a Session

When a user session reaches its end, you may use the abandon method to release all session-scoped variables and associated memory resources. The next server page requested by the client will prompt the server to instantiate a new session object.

In general, it is a good idea to end the client's session using abandon, otherwise the server continues to maintain the client's session state until a timeout occurs.

Note that the abandon method immediately releases all session-scoped variables. Any references to session variables between the abandon and the end of the page containing its invocation may raise an exception.

Obtaining the Session ID

During a client session, a unique ID is assigned by the Application Server to each active client. The server automatically maintains this ID using a cookie on each client's machine. During each subsequent transaction, the server identifies the client's identity by matching the cookie sent with each HTTP request message. This mechanism enables the server to maintain session state for each active client.

Clients who have disabled cookies cannot use an application with this particular session logic. In this case, the application may either (a) maintain session state using variables in a queryString or, (b) maintain session state using variables in an HTML form hidden in each page.

Send the message id to obtain the unique ID assigned by the server:

```
myID := session id.
```

The session ID cookie is maintained on the client's machine until either the client restarts the browser, the cookie expires on the client's machine, or the server ends the session. Note that the expiration of the cookie holding the session ID and the timeout value used by the server to expire sessions are two distinct intervals.

Caution: A session ID should never be cached in a database or used for the purposes of identifying a user across visits to the Web site.

In effect, the session ID is only semi-unique. For example, if the user's session ends normally, or if the server abandons the session, and the client subsequently returns to the Web application, the server may re-issue the same session ID.

The session ID for any given client is guaranteed to change only when client's browser and server application have both been restarted. Thus, your application should not assume the value persists between sessions.

Session Events

Session objects include a simple event mechanism that may be used to initialize variables during startup or to save them at shutdown. Presently, two events are defined: SessionStartup and SessionShutdown (these correspond to the ASP-style events named onStart and onEnd).

The SessionStartup event is triggered once when the first incoming request is received to create a session object. The event occurs before any code in the server page is executed. The SessionShutdown event is triggered when the session times out or is ended explicitly using the abandon method. Like the SessionStartup event, SessionShutdown is only triggered once.

Events are handled by *callback methods*. Each must be a class method, with a single parameter for the object for which the event is triggered (i.e., the session object). To receive event notification from the Application Server, these callback methods must be registered in an initialization file.

For details, see "Application Events" on page 7-13.

Server

The server object provides protocol for controlling the flow of page execution, and accessing various aspects of the server environment. The server object is essentially a collection of miscellaneous functionality that belongs to the Web application server itself. It is most frequently used when your application needs to redirect the thread of execution from one page to another.

Setting the Script Timeout

Use the scriptTimeout: method to specify the maximum amount of time the server will devote to executing the current page:

```
"evaluate the current script for no longer than 30 seconds"
   ASPServer scriptTimeout: 30.
```

By default, a script will be evaluated for a maximum of 90 seconds. If the timeout is reached, the server sends a timeout error to the client.

Note: The scriptTimeout: method is unimplemented in version 7.6 of VisualWorks Application Server.

Dispatching and Transferring Execution of a Script

During the execution of a script, you may dispatch control to another script using execute:for:, or transfer control entirely using transfer:for:. For example:

```
ASPServer execute: '/MyApp/CreateProfile.ssp' for: aPageModel.
```

Each method takes the same parameters: a path string and an instance of PageModel. The sole difference between the two is that whereas execute:for: returns control to the original page once execution is complete, transfer:for: does not.

The execute:for: and transfer:for: methods are generally used to assist when breaking a more complicated application into smaller elements. In cases where the application needs to redirect the client to another URL on the same server, transfer:for: may be used for faster redirection that is possible with a client-side redirect.

Since local variables defined within the context of a page are confined to the name scope of that page, local variables on the page of the sender of execute:for: are distinct from similarly-named local variables on the page of the receiver. If you need to share objects between pages, use the session object or request attributes (for details, see "Session" on page 7-15).

Unlike variables local to the page, the contents of the request object are maintained when evaluating execute:for: and transfer:for:. Thus, incoming form data or cookie objects that are accessible in the page that sends transfer:for: are likewise accessible in the target page.

Converting a Virtual Path

Use mapPath: to convert a virtual path to the physical path used by the VisualWorks Application Server:

```
ASPServer mapPath: '/MyApp/CreateProfile.ssp'
```

If the parameter to mapPath: begins with a slash character (**/** or ****), a mapping from a virtual to a physical path is performed, otherwise it is assumed that the parameter contains a path relative to the location of the current script's home directory on the Web server. In either case, the result is a String containing the physical path, formatted following the conventions of the host file system.

For details on URL mapping, see "Content Management" on page 10-1.

Encoding URLs for Queries

Characters such as **?**, **&**, and **+** all have specific meanings in a URL, and must be translated into equivalent URL entities before they can be sent as queries. Use the urlEncode: method to perform the proper translation.

For example, consider the following URL in plain text:

```
http://mycorp.com/search.ssp?title=The Olive Book
```

To pass the following String as a valid URL, it must be converted like this:

```
http://mycorp.com/search.ssp?title=The+Olive+Book
```

Thus, if we want to generate this URL in a server page and then include it as a hyperlink, we must first convert the URL:

```
"translate URL entities"
   <% anURL := 'http://mycorp.com/search.ssp?title=The Olive Book'. %>
   ...
   <A HREF="<%= server urlEncode: anURL %>The Olive Book</A>
```

Any URL that contains a query string should be converted before being included in a server page.

All characters with an ANSI value greater than 126 decimal must be converted into % character followed by the ANSI value represented in hex. Additionally, the characters listed below must also be converted:

Character	URL Entity	Character	URL Entity
space	+	\	%5C
'	%27	]	%5D
!	%21	^	%5E
#	%23	'	%60
$	%24	{	%7B
%	%25	|	%7C
&	%26	}	%7D
(	%28	+	%2B
)	%29	<	%3C
/	%2F	=	%3D
:	%3A	>	%3E
;	%3B	[	%5B

Encoding HTML for Page Display

Use the htmlEncode: method to convert characters which have reserved meaning in HTML into their quoted equivalents. For example:

```
"explain simple text emphasis in HTML"
    unencodedText := 'The <B> tag is used to indicate bold text.'.
    encodedText := server htmlEncode: unencodedText.

    <%= encodedText. %>
```

This method should be used anytime you wish to display HTML source or otherwise prevent the client's browser from interpreting reserved HTML characters.

Error Handling

Errors during the execution of a server page are of two general types:

- Errors that occur during the compilation of a page
- Errors that occur during the evaluation of a page

The first type of error includes syntax errors and exceptions raised when attempting to resolve references to bindings at compilation time. Ordinarily, these will prevent any page processing and return a response of type 500, indicating an Internal Server Error.

Errors that occur when evaluating scripting elements on a page raise an exception. Your application code should handle these exceptions (most commonly "message not understood"). Any unhandled exceptions will cause page processing to abort and return an error page to the client.

Handling Exceptions in Server Pages

Applications written using server pages may raise exceptions during the course of normal execution. Generally, the application developer will want to catch these exceptions, often redirecting control to another page.

During exception handling, it is often desirable to pass an error string or other parameters to the code that handles the exception. In the case of an application implemented using server pages, this may be done using session variables.

In the following code sample, the session variable serverError is used to hold the exception's error string while control is transferred to the page showServerError.ssp:

```
[doSomething]
   on:  Error
   do:  [:ex | session at: 'serverError' put: ex description.
        ASPServer transfer: '/showServerError.ssp' for: self].
```

The page showServerError.ssp might report the error using code like this:

```
Error: <%= session at: 'serverError' ifAbsent: ['Unknown error']. %>
```

In a slightly more elaborate situation, code that validates a user name against a profile stored in a database might redirect to a special page called loginError.ssp that handles login errors, e.g.:

```
[userProfile := databaseBroker findUserName: userName]
   on:  Error
   do:  [:ex | session at: 'loginError' put: 'Unknown user -- try again'.
        session at: 'unvalidatedUserName' put: userName.
        ASPServer transfer: '/loginError.ssp'  for: self].
```

8

Server Page Syntax

Smalltalk Server Pages follow the general conventions for server-side scripting with Microsoft's Active Server Pages (TM) and Sun's Java Server Pages (TM). This chapter presents an overview of server page conventions and explains the relationship between the code that appears in a server page and that of a standard Smalltalk method.

A single server page may contain elements of both HTML and Smalltalk code, as well as a client-side scripting language such as JavaScript. Special tags separate the server- and client-side code, identifying which parts of the page will be processed by the application server, and which parts by the client's browser.

When the client's browser requests a page with the **`.ssp`**, **`.jsp`**, or **`.asp`** extension, the application server loads the page and evaluates any script contained within. The results of this evaluation are then sent to the client as a new HTML document. From the client's perspective, the transformation is invisible.

Smalltalk Server Pages may also be written in an alternate syntax using XML-format scripting expressions. These predefined tags make it possible to author pages using standard XML-editing tools, and you can extend the standard action types by defining your own tag libraries.

Syntax

Server pages are HTML documents composed of a *template* (the static HTML) and *scripting elements*. Scripting elements are generally identified in a server page using the following tag:

```
<% ... %>
```

Alternately, scripting elements may be placed in XML format, including a tag followed by some number of attributes (JSP-tag style), e.g.:

```
<jsp:scriptlet>
...
</jsp:scriptlet>
```

Smalltalk server pages may contain three types of scripting element: *directives*, *actions*, and *output expressions*. Actions are sometimes referred to as *scriplets*, and output expressions simply as *expressions*.

A *directive* is a scripting element that is interpreted by the scripting engine as a command. It is not treated as a Smalltalk expression, and has no direct effect on the HTML page output. Directives are used to indicate the scripting language, set page buffering modes, and so forth (for details, see "Directives" on page 5).

An *action* or *scriptlet* is a Smalltalk expression or group of expressions that will be evaluated as part of a single method when the page is requested by the client. The result of the evaluation is merged with the HTML sent to the client.

For example, the following Smalltalk server page code:

```
<HTML>
<TITLE>Time Example</TITLE>
<% response write: 'The time is now ' , Time now printString. %>
</HTML>
```

causes the following HTML to be sent to the client:

```
<HTML>
<TITLE>Time Example</TITLE>
The time is now 3:29:18 pm
</HTML>
```

Since the results should appear in a textual form, we send the message printString to the instance of Time.

Note: Although Smalltalk syntax does not require a period at the end of a method, a scriptlet should always end with a period character.

Alternatively, an action may use an *output expression* tag <%= ... %> to generate a textual result. To do this, we could rewrite the middle line of the example like this:

```
The time is <%= Time now %>
```

When using an output expression, the period that marks the end of a Smalltalk statement is not required.

To produce more complex structures, Smalltalk scripting elements and HTML can be mixed together, particularly by placing one type of tag within the constructs of the other. In this way, Smalltalk expressions may be used to generate HTML code.

For example, to create a list of customer names for an HTML drop-down menu, we might use the following:

```
<% response write: '<select name="developerNames" size="10" multiple>'.
customerNames do: [:name |
    response write: '<option value="'.
    response write: name.
    response write: '">'.
    response write: name.
    response write: '</option>'].
response write: '</select>'. %>
```

In HTML, the select tag is used in the definition of a FORM. Here we can use a loop written in Smalltalk to generate a list of customer names that will populate the list of items to “select” from.

The Smalltalk code alone looks like this:

```
customerNames do: [:name | name].
```

Inside the select tag, we begin a loop over the list of customer names. To define the select options, the Smalltalk block variable name is embedded in HTML using two output expressions <%= ... %>. Since the loop hasn’t been closed, the option tag will be repeated for each customer name, expanding into a list of names. Within the option tag, we embed the name of the customer, which varies according to the loop. Finally, a new Smalltalk code fragment is used to terminate the loop with "].".

Although it is not necessary to place all server-side script within the <HTML></HTML> tags, you must be sure to place it within these tags when using scripting elements to generate HTML code.

Capitalization

The capitalization conventions for Smalltalk scripting elements follow those of VisualWorks Smalltalk. In general, these are more lenient than the conventions used by server-scripting languages like ASP and JSP, though all tags and names are case sensitive.

In Smalltalk, only class, name space, and shared variable names must use an initial capital. By convention, all other names (methods, instance, class instance, and temporary variables) begin with a lower-case letter.

Variables

Smalltalk server pages may reference predefined, persistent variables in the application model, as well as declare their own temporary variables to use during the execution of a page.

Classes and shared variables within the scope of page's application model may be referred to by name. For details on the scope of reference within a page, see "Server Page Applications" on page 7-1.

When the code on a server page is evaluated, it is transformed into a method. Temporary variables in this method may be defined explicitly using vertical bars | ... | in standard Smalltalk syntax, or they may be defined implicitly by the compiler at execution time. The latter is a feature particular to Smalltalk server pages.

For example, the following code defines a temporary variable:

```
<% | customer |%>
<HTML>
<HEAD><TITLE>DynamicForm.asp</TITLE></HEAD>
...
```

Alternately, this variable may be definied implicitly by the compiler. If we include the following tags in the body or the page:

```
<HTML>
<HEAD><TITLE>DynamicForm.asp</TITLE></HEAD>
...
<% customer := database findCustomerWithId: 12. %>
```

In the Smalltalk programming environment, we would need to declare `customer` as a temporary variable at the beginning of a method. In the case of a server page, though, the compilation process is such that any variable name used within that page that is not otherwise defined is assumed to be a temporary variable.

Note: Although some scripting languages use a special notation for declaring variables, Smalltalk does not require explicit declarations. Thus, the JSP convention of declaring variables with <%! ... %> is not necessary on Smalltalk Server Pages. In fact, it is not allowed.

Finally, the fragments of HTML on the server page are stored as variables and made available as temporaries named htmlChunk1, htmlChunk2, etc. These should only be used for debugging purposes.

Scripting Variables

Server pages include the notion of *scripting variables*, which are declared at page-execution time and accessible only within the scope of the page.

Actions may reference these scripting variables by name, just like other variables. The only difference is in their declaration. For example, the following tag defines a scripting variable named user that is assigned a new instance of UserProfile, available in the context of the enclosing page:

```
<jsp:useBean id="user" class="UserProfile" />
```

The scripting variable “user” is always available within the context of the enclosing page, and the useBean tag allows it to be defined within other scopes as well (for details, see “Standard Actions” on page 8-8).

The Application Server provides a number of *implicit scripting variables* (sometimes called *implicit objects*). These objects are part of the SSP application model, and may be accessed within scriptlets using their reserved names: request, response, out, handler, application, and session. For details on their protocol, refer to “Server Page Applications” on page 7-1.

Comments

Within a scripting element, regular Smalltalk comments may be used, delimited by double quotes. Anywhere on the page, it is possible to use a server page comment, using the notation:

```
<%-- comments --%>
```

Directives

Directives are indicated using special delimiters:

```
<%@ ... %>
```

Directives do not have a direct effect of the page that contains them, but are treated as commands to the scripting engine. Generally, a directive is an attribute/value pair.

The following directives are defined in VisualWorks Application Server 7.6:

Language

Smalltalk server pages are identified by a special LANGUAGE tag that appears at the beginning of each page of code processed by the server:

```
<%@ LANGUAGE="SScript" %>
```

This directive identifies the scripting engine to be used by the server when processing the page. The value SScript sets Smalltalk as the scripting language. Normally, a single server technology is specified for the site as a whole.

Note that it is possible and often convenient to combine client-side script in the same page by using the <SCRIPT> tag. For example, client-side widgets may be implemented using JavaScript using the following tag:

```
<SCRIPT LANGUAGE="JScript" >
```

Script that is embedded in this way will be ignored by the server.

When using commercial Web design applications, you may need to change the preferences to include this tag in the server pages generated for your site. When using Macromedia Dreamweaver, for example, you must specify Smalltalk server pages as the server technology in order to generate the correct LANGUAGE tag.

Note: In version 7.6 of VisualWorks Application Server, the LANGUAGE tag is optional. The Application Server assumes that all files containing server-side script will be in Smalltalk. This may be changed in a future release.

Taglib

Specifies a tag library to be used when evaluating a server page that uses JSP tags. The tag library is contained is a separate file that may be named by either an absolute or a relative URI, e.g.:

```
<%@ taglib uri="file:cincomtags.tld" prefix="cincom" %>
```

The tag library file is an XML document containing tag definitions for use in the page. If the file is specified using a relative URI, it is interpreted as being relative to the directory containing the server page. A prefix must also be specified to distinguish the actions that use the library's tags.

For details on the use of tag libraries and custom tags, see "Server Page Extensions" on page 9-1.

Predefined Scripting Actions

Smalltalk Server pages can also be written using special XML-format actions. These follow the JSP specification and can be combined with scripting elements or used in their place.

The advantages of using these actions are that you can entirely eliminate the use of code from your server pages, helping to separate the roles of Web designers and developers, as well as making it simpler to manipulate the pages using HTML and XML tools.

The VisualWorks Application Server provides a predefinied set of tags, and also supports mechanisms that let you define your own tags.

Smalltalk server pages can include the standard JSP actions specified using scripting element tags. Note that these tags are case sensitive and often mixed case.

Tag Attributes

Scripting elements in XML format can include named attributes. The value of a named attribute must be quoted.

There are two general tag attributes which may be applied to most scripting elements:

id

You may use the `id="name"` attribute to declare a name for the element that may be used within the scope of the current server page. I.e., if the action instantiates a new object, this object will be associated with `name` for the duration of the page.

scope

Use the `scope="page|request|session|application"` attribute to assign a specific scope to the value of an action. When associating the object with session or application scope, the object is assigned as an attribute of the session/application.

Standard Actions

The following tags are supported in VisualWorks Application Server 7.6:

useBean

Create a new instance of the class, assigning it the scripting variable name id within the specified scope. Note that a scripting variable is always declared for id within the scope of the page, regardless of the value specified for scope.

The action may optionally include a body. If a body is included, it will generally contain scriptlets or one or several setProperty tags.

The following attributes are defined:

Attribute	Description
id	Name used to identify the object within the specified scope, as well as the scripting (local) variable name used during page evaluation. The id is case sensitive.
scope	Scope of the newly named instance — can be page | request | session | application. Defaults to page scope if not specified.
class	Fully qualified name of the class used to instantiate the object. The class name is case sensitive.
beanName	Not supported.
type	Not relevant in Smalltalk.

Example:

```
<jsp:useBean id="clock" class="MyNameSpace.ClockClass" scope="session" />
```

setProperty

Set the value of properties in a scripting variable. Use the name attribute to identify the variable.

The following attributes are defined:

Attribute	Description
name	Name of the instance created with a useBean action.
property	Name of the property to be set. If the property attribute is set to "*", the setProperty tag sets every named property in the specified scripting variable with the corresponding values in the current Request object.

Attribute	Description
param	Name of the request parameter to be assigned to an instance — usually from a Web form.
value	The value to assign to the named property.

Examples:

```
<jsp:setProperty name="request" property="*" />
```

```
<jsp:setProperty name="customer" property="fullName" param="username"/>
```

getProperty

In the scripting variable specified using name, obtain the instance variable specified using property, and return its String value. The getProperty tag functions as a scripting expression.

The following attributes are defined:

Attribute	Description
name	Name of the scripting variable to be referenced.
property	Name of the instance variable to retrieve.

Examples:

```
<jsp:getProperty name="userInfo" property="name" />
```

include

Include the named resource in the current page. The resource is named using a relative URL that is translated using the active Web site object.

The following attributes are defined:

Attribute	Description
page	Relative URL of resource to include.
flush	Boolean (mandatory) indicating whether or not to flush the page, i.e., “true” will cause the page to be flushed.

Example:

```
<jsp:include page="/MyApp/legalPrologue.html" flush="true" />
```

forward

Dispatch the current request to the named page. The target can be a resource (static HTML), a server page, or a servlet class name. Since the forward tag dispatches without returning, the remainder of the current page will not be executed.

The following attributes are defined:

Attribute	Description
page	Relative URL of the page that will assume control.

Example:

```
<jsp:forward page="/MyApp/servlet/ClientValidate" />
```

scriptlet

Evaluate the element as a Smalltalk expression.

Equivalent to the tag <% ... %>.

Example:

```
<jsp:scriptlet>response write: Date today printString</jsp:scriptlet>
```

expression

Evaluate the element as a Smalltalk expression, adding the result directly to the response object.

Equivalent to the tag <%= ... %>.

Example:

```
<jsp:expression>Date today printString</jsp:expression>
```

An Example using JSP-style Script

Server pages are generally authored to follow either an ASP or a JSP pattern. Although the two models are similar at the level of semantics, their syntax, scripting elements, and scripting protocol are rather different.

This can be seen in the following simple JSP-style page:

```
<jsp:useBean id="customer" scope="request" class="CustomerInfo"/>
<html>
<head>
<title>Customer Registration</title>
</head>
<h1>Welcome,<jsp:getProperty name="customer" property="name"/>.</h1>
</br>
<p>Your current profile:</p>
<ul>
<li>
<h2>Name: <jsp:getProperty name="customer" property="fullName"/></h2>
<h2>Address: <jsp:getProperty name="customer" property="address"/></h2>
<h2>Phone: <jsp:getProperty name="customer" property="phone"/></h2>
<h2>Account: <jsp:getProperty name="customer" property="account"/></h2>
</li>
</ul>
</br>
<a href="./updateProfile.ssp">Update your profile.</a>
</body>
</html>
```

This example page displays the values captured in a CustomerInfo object. To obtain a named instance of the object, and then display its contents, two of the standard JSP actions are used: useBean and getProperty.

The jsp:useBean action creates a new instance of CustomerInfo, or else uses an instance associated with the client's current Web session. By setting the "id" attribute, the new instance is identified by the name 'customer' for the scope of the page.

The remainder of the page uses the jsp:getProperty scripting action to obtain various properties of the CustomerInfo instance (fullName, address, phone, account), displaying them in an unordered list.

9

Server Page Extensions

Smalltalk server pages are written using scripting actions in either ASP or JSP syntax. When using JSP-style pages, developers may extend the standard action types by creating *tag libraries*.

A tag library defines a set of *custom actions* for use in a server page. Developers can use tag libraries to abstract the functionality of server pages into a more natural and portable format. Tag libraries are a means to provide content developers with predefined sets of functionality.

The VisualWorks Application Server supports all both the predefined JSP 1.1 scripting actions, as well as custom actions. This chapter describes the use of tag libraries and custom tags (for details, see “Predefined Scripting Actions” on page 8-7).

Overview

Custom scripting actions are invoked by using *custom tags* in a server page. To define custom tags, you must create a *tag library*.
Tag libraries facilitate another level of componentization, and are designed to be used independently of the scripting language.

For example, an application written using server pages and tag libraries could be ported from one scripting environment to another simply by changing the tag library. The server pages could use business logic written either in Java or Smalltalk, while server pages remain unchanged.

Each custom tag defines a set of scripting actions (much like a method body) which may access all variables and implicit objects contained within the server page that invokes the tag. Custom tags may also be nested. The body of a custom tag can, for example, contain other custom tags that modify the response object which belongs to the calling page.

When to Use Tag Libraries

When considering the use of tag libraries, the following points may help to choose the best design for your application:

- As a general rule, server pages should not be used to perform extensive processing of requests. Instead, they should defer processing to code components in the application's business logic.
- Tag libraries declare a stable interface to code components. They can make it easier to work with authoring tools, and to encapsulate code such that it is more portable between different Application Servers.
- Custom tags and their handlers may be used to modify the content of a server page. Server-side components invoked from server pages cannot modify the server pages.
- The use of tag libraries requires some additional effort on the part of the developer, but generally only in the early phases of the project.

How Tag Libraries Work

Tag libraries are defined using two parts: (1) a set of tag handler classes, one class for each custom tag and, (2) a tag library descriptor file (TLD). The tag handler classes exist as application code that extends the functionality of the Application Server.

The tag handler classes are associated with the application's business logic. They establish an interface to components in the Web application that may be accessed by code in the server page. The actual interface definition is declared in the tag library descriptor file. This allows the server pages and the TLD to be written in a fashion that is portable and independent of the scripting language.

To use a tag library, each server page needs to explicitly declare an interest in that tag library's descriptor file. This is achieved by including a **`taglib`** directive at the head of the server page.

Tag Library Descriptor File

The *tag library descriptor* file (TLD) is an XML document that describes a tag library. It is essentially an interface description. The TLD contains a short header, and a series of definitions, one for each custom tag in the library. A tag definition specifies the name of the tag, its corresponding handler class, whether it can support nested tags, and so forth.

Within the TLD, a typical tag definition looks like this:

```
<tag>
   <name>myTag</name>
   <tagclass>MyTag</tagclass>
   <info>Simple tag that provides my functionality</info>
   <bodycontent>empty</bodycontent>
</tag>
```

The descriptor file is used by the Application Server when processing custom tags, and may also be used by authoring tools when creating server pages that include custom tags.

For details, see "Creating a Tag Library Descriptor File" on page 9-5.

Tag Handlers

A *tag handler* is a class that defines how to evaluate a tag during the execution of a server page. For each custom tag used in a server application, a corresponding tag handler class must be declared. The tag handler defines the tag's semantics, and serves as a point of contact between the server page and components in the server application.

When a server page that includes custom tags is evaluated, the Application Server instantiates a tag handler object for each occurrence of a custom action. In effect, the instances of the tag handler class serve as runtime representations for the custom tags.

Tag handlers have two main interfaces, found in class Tag and BodyTag. Custom tag handlers are created by sub-classing either class Tag or BodyTag. For details, see “Creating a Tag Handler” on page 9-7.

Custom Tags in Server Pages

Server pages that use custom tags must first include a **taglib** directive that refers to the TLD, e.g.:

```
<%@ taglib uri=”file:myLib_1_2.tld” prefix=”myLib” %>
```

This directive must be placed at the beginning of the server page, before any scripting expressions that use custom tags declared in the library.

The **taglib** directive contains two attributes: the first, uri, specifies the TLD to be used by the page. This URI may be either absolute or relative.

The second attribute, prefix, associates a *tag prefix* with the actions in the library. This prefix must be used when referencing the library actions from a server page.

Custom tags may appear anywhere in the body of the server page. Typically, a custom tag includes attributes. It may or may not include a tag body.

For example, the following tag would be handled by an instance of the tag handler class MyAction, as specified in the tag library called “myLib”:

```
<myLib:myAction att="myAttribute" />
```

This tag may be described as “simple” because it contains no body.

Alternately, a tag may also include a body, e.g.:

```
<myLib:myActionBody>
   Arbitrary Text
</myLib:myActionBody>
```

The body of a tag may be raw HTML, JSP, or other (nested) custom tags. If the tag includes a body, the tag handler processes it before passing the result as part of the page generated by the Application Server.

For details on the various types of tags and how to implement handler classes for them, see “Creating a Tag Handler” on page 9-7.

Creating Tag Libraries

To create a tag library, you must:

1 Create a tag library descriptor file (TLD)

2 Define appropriate tag handler classes

The tag library descriptor file resides on the server along with the application's server pages.

The tag handler classes are associated with the application logic. Generally, these are part of the Web application, though you may choose to package them as a separate component.

The following sections describe each of these steps in more detail.

Creating a Tag Library Descriptor File

The tag library descriptor file is an XML format file that is generally located in the same directory as the server pages that reference it. By convention, the file's suffix is **tld** and it is common to include the version of the TLD as part of the file name, e.g.:

myLib_1_0.tld

The official DTD for a tag library descriptor file is described at:

http://java.sun.com/j2ee/dtds/web-jsptaglibrary_1_1.dtd

A sample tag library descriptor file might look as follows:

```
<?xml version="1.0" encoding="ISO-8859-1" ?>

<taglib>
    <tlibversion>1.0</tlibversion>
    <jspversion>1.1</jspversion>
    <shortname>cincom</shortname>
    <uri>http://mySite.com/taglibs/myLib_1_0.tld</uri>
    <info>An example tag library, provided by Cincom</info>

    <tag>
        <name>myTag</name>
        <tagclass>VisualWave.MyTag</tagclass>
        <info>Simple example: do something</info>
        <bodycontent>empty</bodycontent>
    </tag>
    <!-- Additional tag definitions follow -->

</taglib>
```

The (required) taglib element is the root of the entire document. Within this element, the TLD has two parts: a set of header elements, which are generally fixed, followed by a series of tag definitions.

The following sub-elements constitute the header for the tag library descriptor file:

Element	Description
tlibversion	Version number of this tag library.
jspversion	Version of the JSP implementation required by this tag library; default is version 1.1.
shortname	A short token for use by authoring tools when creating pages; may be used as a prefix to identify tags associated with this library. Should not include white space or begin with an underscore character.
uri	The public URI that identifies this version of the tag library.
info	A short text string that describes this tag library.

The TLD header is followed by a series of tag definitions.

Each tag definition specifies the tag's name, its handler class, details about the tag's body content (e.g., whether it is always empty), and documentation on the tag's function. Each valid attribute accepted by the tag must be explicitly declared, with an indication of whether or not it is required, and whether it accepts run-time expressions.

Tag definitions may contain the following sub-elements:

Element	Description
name	Base name of the action, i.e., as it appears in the server page (required).
tagclass	Fully-qualified name of the tag handler class (required). Typically, tag handlers are located in the VisualWave name space. The tag handler must be a subclass of Tag or BodyTag.
teiclass	N/A
bodycontent	Type of the tag's body content. May be either tagdependent, JSP, or empty: **tagdependent** Indicates that the body of the action is interpreted by the tag handler itself. The body may be empty. **JSP** Indicates that the body of the action contains JSP elements. The body may be empty. This is the default value for the bodycontent element. **empty** Indicates that the body must be empty.

info	A short text string describing this tag.	
attribute	Defines an attribute of the custom action. Attributes may define the following sub-elements:	
	name	Name of this attribute (required).
	required	This attribute is required (optional). May be true, false, yes, or no.
	rtexprvalue	The attribute's value may be dynamically generated at runtime by a scripting expression (optional). May be true, false, yes, or no.

The URI of the tag library descriptor file must be declared in each server page using the **`taglib`** directive (for details, see page 4).

Creating a Tag Handler

Tag handlers for custom actions make use of two main interfaces, found in the classes Tag and BodyTag. A new custom tag handler is created by subclassing either of these two, depending upon the desired behavior of the scripting action.

Recall that there are two basic types of actions:

- Simple actions
- Actions with a body

Simple actions are defined as those which do not include a body. Tag handlers for these actions only need the protocol defined by class Tag, which provides basic support for initialization of the handler object, and evaluation of attributes associated with the tag.

If the scripting action includes a body, then the handler may need to be defined as a subclass of BodyTag. Class BodyTag is a subclass of Tag that provides additional protocol for accessing and evaluating the tag's body.

Your custom handler should only subclass BodyTag if it needs to perform processing on the tag body (i.e., parse the body content and transform it programmatically). For handlers that just pass the body as text or JSP, it is sufficient to sub-class Tag.

Actions may also be described as *cooperating*, i.e., two or more actions which are nested, and share some variable data locally.

Defining a Simple Tag Handler

A handler class for a simple tag that includes no body and uses no attributes might be defined as follows:

```
Smalltalk.VisualWave defineClass: #MyTag
    superclass: #{VisualWave.Tag}
    indexedType: #none
    private: false
    instanceVariableNames: ''
    classInstanceVariableNames: ''
    imports: ''
    category: 'MyApp-JSP'
```

To process a simple tag, the new handler class MyTag must define the method doStartTag. The logic for handling the tag is located primarily in this method. Since it is defined in the abstract class Tag, the method doStartTag is thus overridden by the custom handler class. The message doStartTag is always sent to the handler object after it has been initialized.

For example, consider a simple tag that returns the current date:

```
<myLib:dateToday />
```

The handler method for this could be implemented as follows:

```
doStartTag
    pageContext response write: (Date today printString).
    ^#SKIP_BODY.
```

This method obtains the current date, and writes it to the response stream. Handlers do not have direct access to the request and response objects, but may fetch them from the pageContext. As a return value, the method returns a special Symbol that is interpreted by the Application Server. Since the tag we've defined doesn't contain a body, #SKIP_BODY is passed as a return value. This instructs the Application Server to ignore any text between the start and end tags.

For tags without body or attributes, the custom handler class can place most of its logic in one method: doStartTag.

Handler Properties

Each tag handler contains two properties (instance variables) that are set by the Application Server: pageContext and parent. These are initialized automatically once when the tag handler object is created, and may be used by the methods in your custom tag handlers (accessor methods are provided).

The pageContext property holds the object that represents the server page containing the active tag. This property may be used to access objects associated with the page model, e.g., the request and response objects.

The parent property holds the tag handler for the enclosing action. This property is generally used by a tag that is nested within another tag.

Note: The pageContext property should always be set *before* parent.

Handling Tag Attributes

Tags may optionally contain named attributes, e.g. the tag myAction might use two attributes called name and value:

```
<myLib:myAction name="first" value="second" />
```

Tag handler classes use a simple protocol for processing these attributes: for each named attribute, there must be a corresponding access method. For example, the handler class for a tag that includes name="first" would need to declare a method named setName: that accepts a String value.

When the tag handler object is created during page processing, the Application Server uses these access methods to initialize the handler object. This takes place before the tag's body is processed.

Typically, the access method simply stores the attribute's value in an instance variable of the tag handler, so that it may be used during subsequent processing of the tag or its body.

For the example mentioned above, the method might look like this:

```
setName: aString
    "Set the name attribute to aString"
    aName := aString
```

Although by convention the named attributes in tags begin with a lower-case letter, note that the attribute name in the access method is expected to use an upper-case letter (i.e., name vs. setName:).

When a tag includes attributes, they must be explicitly declared in the TLD file. The tag declaration should include an attribute element for each named attribute, defining the following sub-elements:

Sub-element	Description
name	A case-sensitive attribute name (required).
required	A flag that indicates whether or not this attribute is required in the tag (required). This element may be true, false, yes, or no.
rtexprvalue	A flag indicating whether the attribute can be a JSP expression that is evaluated at run-time (optional). If this element is included and set to true, the attribute may have a value like <%= self doSomething %>. This element may be true, false, yes, or no.

To illustrate, a TLD definition for the tag myAction (mentioned above) would declare the two attribute elements as follows:

```
<tag>
    <name>myAction</name>
    <tagclass>VisualWave.MyAction</tagclass>
    <info>Simple example: do something</info>
    <bodycontent>empty</bodycontent>
    <attribute>
        <name>name</name>
        <required>true</required>
    </attribute>
    <attribute>
        <name>value</name>
        <required>true</required>
    </attribute>
</tag>
```

Including the Tag Body

As mentioned previously, tags may also include a body, e.g.:

```
<myLib:myActionInclude>Body</myLib:myActionInclude>
```

The body may contain plain text, HTML, or scripting expressions, and the tag handler can optionally include this body content in the response object. Since the body may contain scripting expressions, this processing occurs when the server page is first translated into executable code.

To include the body content, a custom tag handler should be a subclass of Tag and, as in the case of a handler for simple tags (see above), it must define a doStartTag method. However, in order for the Application Server to evaluate and include the tag body, the doStartTag method must return the Symbol #EVAL_BODY_INCLUDE (not #SKIP_BODY, as in the case of a handler for simple tags).

In addition to doStartTag, tag handlers that include the body content must define a doEndTag method. This method is invoked after the body has been processed, and it returns a Symbol that instructs the Application Server whether or not to continue processing the server page.

Normally, doEndTag should return #EVAL_PAGE to indicate that page processing should continue. If the tag handler wants to stop processing the page, this method should return #SKIP_PAGE, in which case the response object is considered to be complete.

To illustrate the use of these methods, consider the following example:

```
<myLib:applyColor color=”green”>Plants and Trees</myLib:applyColor>
```

To implement a simple tag handler (ApplyColor) that uses the color attribute to set the emphasis of the body text, we can use the following methods:

```
setColor: aString
    “Set the color attribute to aString”
    colorAttribute := aString

doStartTag
    “If a color attribute exists, use it to emphasize the body content”
    colorAttribute ~= nil
      ifTrue: [pageContext response write: ‘<font color=’ , colorAttribute, ‘>’].
      ^#EVAL_BODY_INCLUDE.

doEndTag
    colorAttribute ~= nil
      ifTrue: [pageContext response write: ‘</font>’].
      ^#EVAL_PAGE.
```

The first method, setColor:, receives the color attribute (if any) and saves it in an instance variable. If a color attribute is included in the tag, the doStartTag and doEndTag methods use it to include HTML formatting around the body content.

The TLD definition for tags that include body content (plain text, HTML, or scriptlets) must specify “JSP” for the bodycontent element, e.g.:

```
<tag>
  <name>applyColor</name>
  <tagclass>VisualWave.ApplyColor</tagclass>
  <info>Apply color to the tag body</info>
  <bodycontent>JSP</bodycontent>
  <attribute>
    <name>color</name>
    <required>true</required>
  </attribute>
</tag>
```

Note that the tag handler may be designed to *optionally* include the tag body depending upon some request-time condition. In such a case, doStartTag would return either #EVAL_BODY_INCLUDE or #SKIP_BODY.

When implementing a custom tag handler in this manner, care should be taken to write both doStartTag and doEndTag such that each method accommodates both cases (i.e., to include body content, or not).

Processing the Tag Body

For some applications, it is necessary to manipulate or transform the tag body before it is passed to the response stream. Tag handlers that extend class Tag may ignore the tag body or include it, but cannot manipulate it. Custom tag handlers that need to perform more elaborate processing of the tag body should subclass BodyTag.

Class BodyTag includes additional protocol for stream-based manipulation of the body content. The handler methods obtain a BodyContent object, and the custom handler class assumes responsibility for passing its contents to the response object.

The semantics of the methods doStartTag and doEndTag are generally the same as those in class Tag (see preceding sections for details). To process the body, the doStartTag method should return #EVAL_BODY_TAG, and #SKIP_TAG to ignore it. The doStartTag in a subclass of BodyTag should never return #EVAL_BODY_INCLUDE.

Class BodyTag provides additional three additional methods to support processing of body content: bodyContent, doBeforeBody, and doAfterBody.

The bodyContent method may be used by a tag handler to obtain an object holding an object that buffers the body using an internal stream. The contents of the stream may be extracted using the message string.

For example, to take the contents of the body and simply write it to the response object, a custom tag handler could include the following:

```
pageContext response write: self bodyContent string.
```

The message doBeforeBody is sent once to the handler object, before the body content is processed in any way (unless doStartTag returns #SKIP_BODY, in which case doBeforeBody and doAfterBody are not sent). This method is typically used to initialize variables used by the handler when processing the bodyContent object.

Similarly, the message doAfterBody is sent after the body has been processed. Typically, this method contains the logic for handling the bodyContent object. If the tag handler is intended to process the body in a single pass, doAfterBody should return #SKIP_BODY, thereby signalling the Application Server that no further processing is desired. Alternately, if doAfterBody is used in an iterative manner, it returns #EVAL_BODY_TAG until finishing.

For example, a tag handler could be used to iterate over its body a variable number of times, e.g.:

```
<myLib:repeat count="6">
   <%= Date today printString %>
   <br>
</myLib:repeat>
```

The attribute count indicates the number of iterations. This can be stored in the tag handler class as an instance variable. The tag handler (Repeat) would contain the following three methods:

```
setCount: aString
   "Set the count attribute, converting the string to an integer"
     count := aString asNumber

doStartTag
     "Always evaluate the body content"
   ^self class EVAL_BODY_TAG

doAfterBody
   "Write the body content to the response object until the variable count is
   equal to zero"
   pageContext response write: self bodyContent string.
   count := count - 1.
   count == 0 ifTrue: [^self class SKIP_BODY].
   ^self class EVAL_BODY_TAG.
```

The method setCount: captures the value of the count attribute, storing it in an instance variable of the handler object.

The iteration over the tag body is controlled by the method soAfterBody, which passes the bodyContent to the response object, and returns #EVAL_BODY_TAG until the variable count reaches zero.

In general, the execution of a BodyTag handler follows this pattern:

1. Instantiate handler object
2. Set pageContext
3. Set parent
4. Set attributes
5. Send doStartTag
6. Set bodyContent
7. Send doBeforeBody
8. Send doAfterBody
9. Send doEndTag
10. Send release

Using Nested Tags

Custom tags may be nested in order to share locally scoped variables. Tags that are intended to nested in this way are often referred to as *cooperating tags*. Nested tags are useful when writing handlers that behave differently depending upon context. Tags that are nested may be based upon either Tag or BodyTag.

For example, consider the following expressions:

```
<myLib:outerTag color=”red”>
    <myLib:innerTag />
</myLib:outerLib>
```

For these two tags, our application might define the handler classes OuterTag and InnerTag. The handler object for outerTag would typically hold the color attribute in an instance variable. When activated, the handler object for innerTag could then access this value as follows:

```
doStartTag
    | outerTagHandler |
    outerTagHandler := self findAncestorWithClass: OuterTag.
    pageContext response write: outerTagHandler color printString.
    ^self class SKIP_BODY
```

The method findAncestorWithClass: checks each enclosing handler until it finds one with the expected class (or returns nil).

Implementation of Tag and BodyTag

This section summarizes the public protocol of class Tag and BodyTag.

Note: The JSP 1.1 specification names class TagSupport and BodyTagSupport as the base classes for defining custom tag handlers. These classes are not provided by VisualWorks Application Server. Instead, developers should subclass Tag and BodyTag.

Class Tag provides the following protocol for manipulating properties:

pageContext:
Set the handler's pageContext property. This property must be set before sending parent:.

parent
Get the handler's parent tag. Used with findAncestorWithClass:.

parent:
Set the handler's parent tag. This property must be set after the pageContext property.

Class Tag provides the following protocol for processing actions:

doStartTag
Called once the handler object has been initialized, but before evaluating the tag body (if any). Typically, the tag handler's logic is located in doStartTag. May return #EVAL_BODY_INCLUDE to indicate that the body content should be included, or #EVAL_BODY_TAG to indicate the body should be evaluated, or #SKIP_BODY, to ignore it.

doEndTag
Called after the tag and body (if present) have been processed. Returns #EVAL_PAGE to indicate that page evaluation should continue, and #SKIP_PAGE to ignore the remainder of the page and complete the response object.

release
Release any internal state held by the tag handler. After sending release, all properties of the tag have an unspecified value.

Class Tag provides one method for supporting nested tags:

findAncestorWithClass:
Returns the handler object for an enclosing tag of the specified class. Returns nil if the tag is not nested, or if no match is found.

Class BodyTag provides the following protocol for manipulating the body:

bodyContent

Return a BodyContent object, which is a special block used for buffering the body content generated during evaluation.

bodyContent:

Set the handler's bodyContent property. This property is set once per tag evaluation, before the message doStartTag is sent to the handler.

doBeforeBody

Sent by the Application Server after doStartTag but before the body content is processed (unless doStartTag returns #SKIP_BODY, in which case doBeforeBody is not sent). The doBeforeBody method is typically used to initialize variables used by the handler when processing the bodyContent object.

doAfterBody

Called after the body has been processed (unless doStartTag returns #SKIP_BODY, in which case doBeforeBody and doAfterBody are not sent). Typically, this method contains the logic for handling the bodyContent object. Returns #SKIP_BODY if the tag handler is intended to process the body in a single pass, or #EVAL_BODY_TAG to request that the Application server re-evaluate the body content.

10

Content Management

VisualWorks Application Server provides content management features to simplify the construction and administration of complex sites. Web applications that are composed of many heterogeneous elements (server pages, servlets, resources, static HTML insets) can be structured into a unified whole by using server-side includes and logical names.

When using a single VisualWorks server to host several different applications, or when building a Web application that serves a number of different IP domains, content management features may be used to simplify the configuration and management of site-specific parameters.

Content management also provides facilities for creating and managing visitor profiles, as well as for logging content serving statistics.

This chapter presents:

Overview

Content management provides a general framework for accessing and organizing the heterogeneous content of a Web application using site policies, logical names, and logical links.

When used in conjunction with the Application Server's site configuration tools, content management can be used to set up multiple domain mappings, aliases, and virtual directories. Content management features can also be used within server pages to rationalize the link structure of a Web application.

Sites and applications may use the following features:

Site Policies
Web site objects may be associated with policy objects. A site policy describes, among other things, the mapping of the site to domain names, and any URL aliases (virtual directories) assigned to the site. For details, see "Resolving Web Requests" on page 10-3.

Logical Names
Each Web site may define a dictionary of logical names used to translate URL path components, register servlets, and to translate links on server pages. Logical names are defined using a two-tier configuration mechanism. For details, see "Logical Names" on page 10-6.

Server-Side Includes
VisualWorks adds logical linking to standard server-side includes, providing a way to include both static and dynamically generated content. For details, see "Server-Side Includes" on page 10-8.

Smalltalk Links
When using logical names, Web requests can be translated directly into message sends. Two varieties are available: SSP methods, for use with server pages, and servlet methods, for use in place of generic servlets. For details, see "Smalltalk Links" on page 10-7.

Both incoming client requests and page content can be manipulated using a single set of logical names. Incoming requests can be manipulated using site policies and logical links, and static or dynamic includes can be generated by using logical includes or Smalltalk links.

Note: Many of the features discussed in this chapter require the use of configuration files. For details, see "Deployment" on page 11-1.

Resolving Web Requests

At a high level, each Web request received from a client is resolved to a particular application using a two-step procedure: first, the request is resolved to an active Web site, then the site object is used to resolve the request to a particular application or resource that belongs to that site. VisualWorks Application Server provides configuration options to create rules for governing both steps of request resolution.

In the first step of request resolution (identifying the appropriate Web site object), each Web site instance uses an associated *policy* to determine the domain(s) it serves and to apply any URL aliases it understands.

Depending upon the site policy, a single Application Server may be configured to host a one or number of distinct sites. Each site object may be associated with a single domain name, or with several, as well as an URL alias (or aliases) at that domain name.

By default, the Application Server listens at a specified port, as specified using the Server Console (see "Using a Web Server" on page 4-2). Requests to each particular host:port pair are resolved to a site object, and the site's default resolution policy is to map its home directory to the base of the URL path of the server.

A policy for selecting a Web site object must be specified when multiple sites are configured for use by a single server, or when the application requires that a site be registered with a specific domain and/or URL alias (for a discussion of the latter, see "Creating a Site Alias" on page 10-4).

For a site object to receive Web requests, it must include either a domain definition or an alias. Any requests which do not correspond to a recognized domain and/or alias are dispatched to the default site.

Associating a Site with a Domain Name

Ordinarily, a server is registered with a physical IP address, and the latter has a single assigned domain name. For cases in which the server may be identified with several domain names, a distinct Web site object may be associated with each name.

For example, a company might register the domain names `http://support.foo.com/`, `http://sales.foo.com/`, and `http://employeesonly.foo.com/` such that all three resolve to the same physical server (i.e., the domain names all share the same IP address). However, the application designer may wish to have each domain name provide different content by creating three distinct site objects within the server.

To associate a site object with a specific domain:

1. Add an entry to specify a DNS name in the **`[configuration]`** section of the site's configuration file, e.g.:

   `domain = newyork.cityplan.com`.

   To associate the site with multiple domains, add them to the same entry, separating each domain name with the **`;`** character, e.g.:

   ```
   domain = brooklyn.ny.org; coney-island.ny.org
   ```

2. Save your changes to the configuration file.

3. Use the administrator's Web interface to select the Site Configuration page, and **Submit** the existing configuration.

The changes made to the site configuration file are used to update the Web site object. For details on the site configuration file and using the site administrator's toolset, see "Creating and Configuring a Site" on page 5-5.

Creating a Site Alias

A site alias may be used to associate a single WebSite instance with a particular base URL. Aliasing is also used as an equivalent to the *virtual directory* feature available on most commercial Web servers.

Aliasing begins with the first path component of the URL (to alias any other portion of the path, you must use a logical name). For example, by defining a site alias for /support, the URLs http://mycorp.com/index.html and http://mycorp.com/support/index.html can reside on the same machine while being configured as two distinct sites. The first path component (in this case /support) determines the correct site instance.

Note: For sites other than the **default** site, if you wish to use the site name as well as other site aliases, you must include the site name in the list of aliases.

To create a site alias:

1. Add an entry to specify a relative directory name in the **`[configuration]`** section of the site's configuration file, e.g.:

   ```
       alias = support
   ```

   To associate the site with multiple aliases, add them to the same entry, separating each alias with the **`;`** character, e.g.:

```
colors = red; green; blue
```

2. Save your changes to the configuration file.
3. Open the administrator's Web interface, navigate to the site's configuration page, and **Reset** the existing configuration.

The changes made to the site configuration file will update the site object.

Creating a Virtual Directory

On IIS and Apache servers, *virtual directories* can be created to establish a mapping between a URL path and a physical path on the server. The first component of the URL path can be specified as a "virtual" directory. Any URL that includes the virtual directory path element is translated by the server into its physical counterpart.

For example, if the virtual directory **`/MyApp`** were defined and given the value of **`C:\WebApps\MyApp`**, then a request to **`/MyApp/index.html`** would be directed to **`C:\WebApps\MyApp\index.html`**.

VisualWorks Application Server provides web sites as an equivalent to virtual directories. Each virtual directory is in effect a separate site object.

To configure a WebSite as a virtual directory, follow the same steps to create a web site, setting the site's home directory to be the target directory. Any request URLs whose first path component matches the alias will be resolved to using the web site.

Alternatively you may also add an alias to an existing web site whose home directory references the desired virtual directory location.

When using the Smalltalk HTTP Server to test a VisualWorks Web application running behind IIS or Apache, it is often useful to create a web site for use as a virtual directory.

Since the Smalltalk server does not have access to the internal definitions used by Apache or IIS, the VisualWorks Application Server cannot resolve pages that assume the existence of a virtual directory defined by the front-end server, unless you define a web site for this purpose.

Resolving Requests to Applications

The second step in resolving a client request is to identify the target page on the server. Once a client request has been resolved to a particular Web site instance, the remainder of the path is translated using that site to identify the target page.

The exact resolution of the request to a target page is a function of the *site catalog*. Each site has an associated catalog of *logical names* that are used to translate portions of the request's URL. When all logical names have been translated, the path is used to select a *request handler*.

Requests may be handled by server pages, servlets, SSP or servlet methods, or a FileResponder (for serving images or static HTML files).

Logical Names

Logical names are mappings used to translate path components. They have two modalities, and may be used in several different ways.

The first and most general use of a logical name is for path translation.

For example, the site catalog might define:

```
current = 2001/july
```

in which case the URL

http://dailyblab.com/news/current/index.html

would be resolved to

http://dailyblab.com/news/2001/july/index.html

During the process of request resolution, the VisualWorks server will recursively translate any logical name in the URL path, until nothing is found in the catalog. Any element of the path that does not correspond to a logical name in the site catalog remains unchanged.

Since the entire local path of the URL is used for translation, any place the name occurs as a complete path entry is translated.

Logical names may be *single-* or *multi-pass*, depending upon the number of translations.

For example, a multi-pass name for mapping the name of a servlet could be defined like this:

```
setup = configure/signon
signon = servlet/SignOnServlet
```

Using these definitions,

```
http://support.myCorp.com/setup
```

would be resolved as:

```
http://support.myCorp.com/configure/servlet/SignOnServlet
```

Smalltalk Links

Generally, a logical name resolves a URL to a server page, a servlet, or a file, but they may also be resolved to directly to Smalltalk methods. This second type of logical name is called a *Smalltalk link*. For example:

```
books = x-ssp:BookStore$htmlDepartments
```

This link would cause the method htmlDepartments to be sent to the class BookStore. The class-side method *must* return a String formatted in HTML. These SSP methods are invoked in a manner analogous to server pages.

In general, the format of an SSP method link is:

name = x-ssp:MySmalltalkClassName$methodName

A second type of Smalltalk link resolves to a servlet-style protocol, e.g.:

```
toys = x-servlet:ToyStore$catalogReq:response:
```

Here, the request is resolved into a message sent to a newly-created instance of class ToyStore, to the method catalogReq:response:. The two parameters to the method are servlet request and response objects (for details on the protocol of these objects, see "Servlets Implementation" on page 7).

The format of a servlet method link is:

name = x-servlet:MySmalltalkClassName$methodNameReq:response:

The protocol **x-ssp** or **x-servlet** is used to distinguish SSP from servlet methods.

The server imposes a security restriction on Smalltalk links: they may not be included in HTML pages, rather, they may *only* be resolved using logical names defined in the site catalog (typically, the per-site INI file).

The behavior of SSP and servlet methods is summarized below:

Type	Method	Description
SSP method (x-ssp)	class-side method zero parameter	No parameters. Returns an HTML-formatted String.
servlet method (x-servlet)	instance-side two parameter	Returns result in response object (second parameter).

Using Logical Names and Logical Links

Logical names must be defined in the **`[logical-names]`** section of the site configuration file. These definitions are used to build the site catalog. It is also possible to define logical names with global scope by adding them to the **`[global]`** section of the **`webtools.ini`** file.

Generally, logical names are used to translate path elements in an URL, but they may also be referenced in server pages using the following scripting element:

```
<%= self linkNamed: 'linkName' %>
```

When the server page is compiled, this logical link is expanded into the full URL.

For example, given the definition:

```
siteIndex = http://myCorp.com/index.html
```

the logical link

```
HREF = "<%= self linkNamed: 'siteIndex' %>"
```

would be replaced with

```
HREF = "http://myCorp.com/index.html"
```

Logical names embedded in scripting elements using linkNamed: may resolve file URLs, HTTP URLs or Smalltalk links.

Server-Side Includes

Sometimes it is convenient to compose a server page by inserting the contents of another file. The Application server provides support for server-side includes equivalent to the services found in Apache and IIS.

Use the #include directive in a Smalltalk server page to achieve this:

```
<!-- #include path-type = "target.inc" -->
```

The path-type attribute can be either file, virtual, or logical. The target may have any file extension, though the **`.inc`** suffix is conventional, e.g.:

```
<!-- #include file = "copyright-footer.inc" -->
```

The keyword file specifies an include file with an absolute path, or else located relative to the directory containing the page that contains the include. In a relative path, you may use **`..\`** to move up the directory structure and access files in sibling directories.

Use the `virtual` keyword to specify a file located using a virtual directory. For example, if the file `includes.inc` were located in a path accessible from the virtual directory `/NewApp`, the following directive could be used:

```
<!-- #include virtual = "/NewApp/includes.inc" -->
```

The target in this case is an URL relative to the Web site's home directory. The first path component must be an alias defined by a Web site instance (e.g., `NewApp`). Platform-specific separator syntax may also be used, though an URL ensures greater portability. Since the target is a relative path, navigation using `..\` is specifically disallowed.

Use the `logical` keyword to specify a logical name:

```
<!-- #include logical = "boilerPlate" -->
```

All logical names in the target are resolved before the contents of the include file are inserted in the page. Logical names are defined in the site configuration file, and can resolve to either file paths or Smalltalk links (for details, see "Using Logical Names and Logical Links" on page 10-8).

A logical name may be either single- or multi-pass, the difference being that a single-pass name is resolved to a file path while a multi-pass name is resolved from one logical name to another, eventually being resolved to a file path or Smalltalk link.

If the logical name is resolved to a Smalltalk link, it must be a link to a SSP method that returns a string. "Servlet methods" cannot be used in server-side includes.

11 Deployment

The VisualWorks Application Server has been designed to simplify both the development and the deployment of Web applications. For ease of development, the Application Server provides an administrator's Web interface for setting and changing server and site configuration paramaters (for details, see "Web Sites" on page 5-1).

Since the administrator's Web interface is not ideal for setting site attributes in a production environment, an application may also be deployed by using configuration files. In this way, you may save all server- and site-specific parameters in a form which can easily be loaded into one or more servers. The server configuration files are also necessary if you intend to use any of the content management features.

Configuration files are used when deploying the headless runtime image provided with the VisualWorks release media. All site parameters defined in the configuration file are read at startup time by the headless image.

Additional information on configuring servers can be found in the *VisualWorks Web Server Configuration Guide*.

This chapter presents:

- Working with Configuration Files
- Specifying Server Attributes
- Changing the Server's Default Configuration

Working with Configuration Files

Site parameters are defined in two separate configuration files: one global, and one per-site. The global file contains sections for configuration, sites and global logical names.

By default, the global configuration file is named **`webtools.ini`**, though you may use the administrator's Web interface to change this.

During development, or whenever the Application Server cannot find **`webtools.ini`**, it uses **`$(VISUALWORKS)/web/webtools.ini`**. This is the pre-set configuration for the Web Toolkit examples.

The per-site files contain sections for configuration and local logical names. Each site is configured from a separate INI file. Site names and the names of their corresponding INI files must be defined in the **`[sites]`** section of the global INI file.

The Application Server does not configure itself (ie. read and install the configuration from the designated configuration files) until it receives the first request through a listening server.

The Web Toolkit configuration files use a format similar to that of Microsoft INI files. INI files are a simple text-based format in the default encoding for the server's platform.

File and directory references used in a configuration file may be in either URL format (using forward slashes), or in the format appropriate for the platform, or using environment variables such as **`$(VISUALWORKS)`**.

The file is divided into named sections, each section containing some number of keys. Keys may have a single value or an enumeration; e.g.:

```
[section-name]
   # comment
   key=value
   keymultiple=value; value; value
```

Enumerations are delimited using semicolons. Spaces may be used around the equals sign in key/value pairs, but they are not required.

Section names, as well as key names, are case sensitive. Values are also generally case sensitive, although the values representing file or directory names may not be, depending on the server platform's file system.

The Global Configuration File

The following sections are defined for the global Web Toolkit INI file:

`[configuration]`

Server configuration parameters, if any. (This section is required, even if it is empty. All parameters are optional.)

This section may use the following pre-defined keys:

Key	Description
name	Specify a logical name for the server configuration.
logfile	Specify the path for the Application Server's global log file.
callbacks	Name methods for the server to call at session or application events. When specified at this level, these callbacks apply to all sites.

`[sites]`

(This section is required.) Logical names used to locate the individual site INI files, where each key is the name of the site, and each assigned value is the name of the INI file for that site (relative to the directory containing this global configuration file).

`[global]`

Logical names that are global to all sites. Note that site logical names take precedence over global logical names. The global section may include user-specified keys.

When an Application Server is used in a production environment (i.e., deployed), none of the internal Web Toolkit sites are used by default.

To deploy the Application Server, you must define a configuration for each site, including the **configure** site, and the **default** site. Even if you plan to disable the **configure** and **default** sites, the Server expects to find a configuration file for each site.

Note: Many of the configuration parameters are optional, however it is strongly recommended that you add a password to restrict access to the **configure** site. For details, see "Securing an Application for Deployment" on page 11-7.

Site-Specific Configuration Files

The following sections are defined for the site-specific INI files:

[configuration]

Contains the site parameters, which are the same as those which can be set from the administrator's Web interface. (This section is required. All parameters are optional, except directory.)

This section may include the following pre-defined keys:

Key	Description
directory	Specify a file path, either relative or absolute, using platform-specific syntax or URL syntax. Relative file paths are resolved relative to the VisualWorks working directory.
environment	Specify a Smalltalk namespace. Note that Application Server classes belong to the VisualWave namespace.
description	Textual description of the Web site — not quoted.
enabled	Specify that the site is active — "true" or "false".
domains	Specify domain names. Use semicolons to delimit multiple names.
debuggable	Debugging options — "true" or "false".
password	Specify a per-site password.
home	Specify a file name for the home page.
registeredServlets	Allow only servlets with defined logical names — "true" or "false".
callbacks	Names of methods for the server to call at session or application events.

Details on a number of these configuration attributes are available in other sections of this guide.

For a discussion of debugging options, see "Setting Site Debugging Options" on page 5-12. For details on using registered servlets, see "Enabling Use of Registered Servlets" on page 11-8. For details on using callbacks, see "Specifying Event Callbacks" on page 11-9.

[logical-names]

Definitions for logical names representing directory paths, file names for HTML content, server-side includes, and Smalltalk URLs to be executed during page processing.

The logical-names section may include any user-specified keys.

Configuring a Site with an Initialization File

The following steps show how to configure a site for use with the test pages bundled with the Web Toolkit. You may use either the prebuilt image file **`runtime.im`** in the **`\web`** directory, or an Application Server development image.

In this example, we shall use the default configuration files provided on the release media, and create a new configuration file to add an additional site.

1 Create a new directory to hold the image and configuration files.

It is recommended that this directory be located at the same level in the file hierarchy as the **`\web`** directory. For example:

```
c:\visualworks7\myApp
```

It is further recommended that the image and configuration files be located in the same directory. As a rule, however, these Server configuration files should not be located alongside the site content.

2 Copy the three initialization files **`webtools.ini`**, **`default-site.ini`**, and **`configure-site.ini`** from the **`\web`** directory to the new directory of your choice (e.g. **`c:\visualworks7\myApp`**).

3 After copying the site-specific INI file **`default-site.ini`**, rename it for use with your new site.

For example, to configure a site named blue, copy and rename the per-site file (**`default-site.ini`**) as **`blue-site.ini`**.

4 In the **`[configuration]`** section of the **`blue-site.ini`** file, set the **`home`** key to the name of the file displayed as the home page:

```
home = Readme.html
```

5 Set the **`directory`** key to the default directory used for files associated with the site, e.g.:

```
directory = $(VISUALWORKS)/myApp
```

6 Change the **`description`** entry to text describing your new site.

7 Save your modifications to the **`blue-site.ini`** file.

8 Since you are adding a new site to your configuration, you must also edit the **`[sites]`** section in the **`webtools.ini`** file (which contains links to the names of the per-site files). Add the new site, e.g.:

```
[sites]
  blue = blue-site.ini
```

9 Save your modifications to the **`webtools.ini`** file.

10 Before starting the Application Server image, copy it to the new directory (e.g., **`c:\visualworks7\myApp`**). Generally, you will use either the prebuilt **`runtime.im`** from the **`/web`** directory, or a copy of your development image.

   The **`runtime`** image is normally invoked with a parcel containing your application. Your development image should contain the Web Toolkit and your Web application.

This completes the configuration. You may start the image and access the main **`Readme`** file using the following URL:

    http://localhost:8008/blue/

When starting the prebuilt **`runtime.im`**, the server will automatically configure itself from the local configuration files on startup. If you are using a development image, the server will configure itself when it receives the first request directed to one of its web sites.

Note: Placing server configuration files in the same directory that contains the HTML pages and script files in your application poses a security risk. As a rule, configuration files should be separated from the page and script content of your application.

Securing an Application for Deployment

When using your Web application in a production environment, you generally want to secure the Application Server. For example, under normal circumstances, clients should not be able to access the various Site Management pages. As a rule, access to these pages should be reserved only for authorized administrators.

The Application Server may be secured by changing its site configuration parameters. In a production environment, this must be done by editing the three files **`default-site.ini`**, **`configure-site.ini`**, and **`webtools.ini`**. Before deploying your Web application, you should make sure to review the configuration settings for the **default** and **configure** sites in these three files.

Note: Any changes you make using the administrator's Web interface are only temporary. To ensure that your settings are applied when the Application Server is restarted, you must modify the configuration files.

For details about building a runtime or headless image suitable for deployment, see the *VisualWorks Web Server Configuration Guide*.

The Application Server defines a special site called **configure** that is used for running the administrator's Web interface. To secure the Server, you may password protect the **configure** site, or you may simply disable it.

Password-Protecting the Server

You may password-protect the server's configuration pages by assigning a value to the **`password`** key in the **`[configuration]`** section of the file **`configure-site.ini`**.

Disabling the Server's Configuration Page

For maximum security, you may entirely disable this site.

To disable the Server's configure site, assign **`false`** to the **`enable`** key in the **`configuration`** section of the file **`configure-site.ini`**.

Setting the Server's Default Home Page

The home page for the Server's **default** site may be set by editing either the **`default-site.ini`** or the **`webtools.ini`** file. You may change all of the default settings in **`default-site.ini`**, or you may simply create a new configuration file and assign it to the **`default`** key in the **`webtools.ini`** file (located in the **`[sites]`** section of the file), e.g.:

```
default = blue-site.ini
```

Enabling Use of Registered Servlets

During deployment, the registeredServlets site attribute should be set to true as a security measure. This disables direct access to servlets using the standard servlet URL (e.g. /servlet/ServletClass), and only allows access to servlets that are "registered" using logical names. The servlet must be registered in the configuration file using a logical name with the syntax:

```
mySservlet = servlet/MyServletClass
```

Or:

```
myServlet = x-servlet:MyServletClass
```

For details on defining logical names, see "Logical Names" on page 10-6.

Configuration Errors

If the Application Server encounters a global configuration error, requests to the configuration pages return a plain-text error page. In the event of a site configuration error, any request for pages from that site will also be redirected to this error page.

To open the administrator's Web interface when there is a configuration error on the default site, use the configure site name in the path, e.g.:

```
http://localhost:8008/configure
```

In the administrator's Web interface, use the Configuration Details page, to make any configuration changes necessary to eliminate the error.

Errors usually result when the server is unable to find the configuration file in the specified location, or the file is incomplete in some way. As a rule, a global configuration file must have the **[configuration]** and **[sites]** sections defined properly, and a site configuration file must have a valid **[configuration]** section.

When the Application Server is used in a production environment, an internally generated password is used to protect the site in the event of a global configuration error.

By default, the password is: **vw7!WCMdCP**.

This password can be found in:

```
WebConfigurationManager class >> defaultConfigurePassword
```

Specifying Server Attributes

The following server-specific configuration parameters may be defined in the global configuration file. These parameters are applied to all sites.

To change these parameters using the administrator's Web interface, see “Managing Server Logging and Sessions” on page 5-14.

Setting the Name of the Logfile

Specify the (optional) path for the web logfile in the **`configuration`** section of the file. The syntax is:

```
logfile = <file-path>
```

If a relative file name is used it is resolved relative to the VisualWorks working directory.

The default log file is named **`webserve.log`**, and it is created in the VisualWorks working directory

Setting the Name of the Configuration

Sepcify the (optional) logical name for the configuration in the **`configuration`** section of the file. This name is displayed on the Configuration Details page in the administrator's Web interface.

The syntax is:

```
name = MyApp Configuration
```

Specifying Event Callbacks

You can define methods which the Application Server calls when specific session- or application-level events occur. If you define an event callback in the global configuration file, it applies to all web sites. Events may also be defined for individual sites. Note that these callbacks may only be defined in a configuration file.

The following events are supported:

Event Name	Description
applicationStartup	Occurs when the Server creates a web site's application instance during site initialization.
applicationShutdown	Occurs when the web site is released, thereby releasing its application instance.

Event Name	Description
sessionStartup	Occurs when the Server creates a session in which to process a client's request.
sessionShutdown	Occurs when the session expires. Neither the browser nor the server explicitly release the session.To force the release unexpired sessions, use the **Clear Caches** command on the Server Management page of the administrator's Web interface.

The syntax to define event callbacks is:

```
callbacks = event(MyClass$myMethod1:); event(MyClass$myMethod2:)
```

The methods must be class-side methods that accept a single parameter, the session or application object, depending on the type of event.

For example, the following line might appear in the **webtools.ini** file:

```
callbacks =
   applicationStartup(MyApp$onStart:); applicationShutdown(MyApp$onEnd:)
```

With this definition in place, the Application Server sends #onStart: to class MyApp when the server initializes the application object, and it sends #onEnd: to the class when the server is shut down.

When using callbacks in a deployed application, the Server image must be built with the name of the global configuration file. Otherwise, the Application Server defaults to look for a global configuration file named **webtools.ini** in the current working directory for the image.

For more details on using sessions to manage state, see "Session" on page 7-15. For details on application and configuration events, see "Application Events" on page 7-13.

Specifying User-Defined Parameters

You may also include your own application-specific parameters in the **configuration** section of the **webtools.ini** file. This feature is useful for specifying database connection parameters, etc in an INI file.

For example, if the **webtools.ini** file contained the line:

```
servers = myServerName
```

The following Smalltalk code may be used in the application to read the value associated with servers:

```
WebConfigurationManager configParameterNamed: 'servers'
```

The method configParameterNamed: answers the named global configuration parameter as read from the INI file or else an empty String (if no parameter has been defined).

A Dictionary containing all the information from the **`configuration`** section in the INI file may be retrieved using:

```
WebSite siteConfiguration configParameters.
```

It is also possible to define and fetch configuration parameters on a site by site basis. E.g.:

```
aSite := WebSite siteConfiguration siteNamed: 'smalltalk'.
siteParams := aSite configParameters.
```

Alternately, configuration parameters may be festched by name:

```
WebConfigurationManager
   configParameterNamed: 'servers' forSite: ‘mySite’
```

Changing the Server's Default Configuration

When configuring the Application Server using initialization files, the effect of the **Set Default Configuration** operation in the administrator's Web interface may be changed.

You should be aware that when restoring the Application Server to the default configuration, the server selects its configuration file, in this order:

1. **`webtools.ini`** from the working directory, if it exists, or
2. The WebToolkit default, **`$(VISUALWORKS)/web/webtools.ini`**

The second default is only available in a development environment.

Once configured, a development image will continue to start with that configuration until you change or reset the configuration. A headless or runtime image contains the name of its global configuration file. The default name is **`webtools.ini`**, but you may change this name.

The prebuilt **`runtime.im`** has been configured to use **`webtools.ini`** from the VisualWorks current directory. Therefore, regardless of where you copy this image, whenever you start **`runtime.im`**, the server will always configure itself from **`webtools.ini`** in the current directory, or launch with a global configuration error if it does not exist.

A

Cookies

HTTP Cookies are a mechanism used by Web applications to both store and retrieve information on the client (browser) side of the connection.

The following discussion presupposes that you are familiar with HTTP cookies. Information about HTTP cookies can be found in the following web sites:

- Persistent Client State — HTTP Cookies (http://www.netscape.com/newsref/std/cookie_spec.html) is the official specification from Netscape.
- Cookies (Client-side Persistent Information) and Their Use (http://home.netscape.com/assist/support/server/tn/cross-platform/20019.html) contains technical tips from Netscape.

Class HTTPCookie

Class HTTPCookie provides a general mechanism that Web applications can use to both store and retrieve information on the client (web browser) side of the connection.

Each cookie is essentially a **`name=value`** pair that may contain additional information. The server sends this information to the client's browser in the form of an HTTP header message and the client subsequently returns it in the same fashion — as an entity header.

An instance of HTTPCookie represents a single cookie.

Instance variable	HTML attribute	Description
name (String)	name=	Name of the cookie
value (String)	value	Value of the cookie
expires (Timestamp)	expires=*date*	When this time is reached, the browser may discard the cookie [optional]. Should be in local time.
domain (String)	domain=*domain Name*	The browser must be speaking to this domain before it sends the cookie [optional]
path (String)	path=*pathName*	This must be a prefix of the URL path before the browser sends the cookie [optional]
secure (Boolean)	secure	If this is true, the cookie will only be returned if we are using a secure server

Working with Cookies

To create a new cookie object:

```
myCookie := HTTPCookie named: 'CustomerProfile' value: 'anonymous'.
```

The name must be a String; the value can be anything (excluding commas, semi-colons, or white space), but it is saved as a string.

To create a cookie on the client's machine, it must be attached to an HTTP response object and then sent to the client. For applications implemented using VisualWave, the cookie may be attached like this:

```
aWebPage addCookie: myCookie.
```

For applications implemented using Smalltalk Server Pages or servlets:

```
response addCookie: myCookie.
```

Setting a Cookie's Expiration Time

Cookies include several attributes which may be (optionally) modified. Each cookie may be given an expiration value, which tells the client's browser when it should discard the cookie.

By default, cookies are set to expire at the end of the client's session. To specify a longer expiration time, or to save the cookie between sessions, a number of methods are provided:

expireAfterDays: *aNumber*
: Mark this cookie for expiration after *aNumber* of days from now.

expireAfterHours: *aNumber*
: Mark this cookie for expiration after *aNumber* of hours from now.

expireImmediately
: Mark this cookie for immediate expiration.

expireNever
: Mark this cookie to never expire.

expires: *aTimeStamp*
: Note: *aTimeStamp* should be in local time.

For example, to set a cookie to expire one year in the future:

```
memberIDCookie expireAfterDays: 365.
```

On the client's machine, the expiration time is saved in GMT. Applications sending `expires:` may use local time, and it will be converted appropriately by the Application Server.

Setting a Cookie's Path and Domain

When a Web browser sends a request to a server, the browser checks the server's domain and path to see if they match any associated with cookies that it is holding. If there is a match, the browser sends the appropriate cookie(s) with the request.

Your application may set both the domain and path attributes of cookies sent to the client.

Note: If your Web application is associated with more than one DNS name, or if it uses pages with different URL paths, it may be necessary to set the path and/or domain attributes explicitly.

For example, an application might use pages at both europe.travel.com and asia.travel.com with links between them. By default, if the application's pages at europe.travel.com create a cookie, it is only returned to the server with requests to europe.travel.com. To access the same cookie at asia.travel.com, you must set its domain attribute to travel.com. E.g.:

```
myCookie domain: 'travel.com'.
```

Similarly, if your application uses one cookie in a number of different pages, you may need to set the cookie's path attribute. By default, a cookie is only accessable to pages that reside in the same path. I.e., the path which contains the page which created the cookie.

For example, if a page located at europe.travel.com/germany/register.ssp creates a cookie, it will only be available to pages on the /germany path. To access the cookie from the homepage of europe.travel.com, you must set the path explicitly. E.g.:

```
myCookie path: '/'.
```

In general, setting the path to '/something' would match both '/something.ssp' as well as '/something/search.ssp'.

Using Cookies in Secure Communication

By default, a cookie's secure attribute is set to false. With the secure attribute set to true, the cookie will only be sent to the server if the communication channel is secure (currently, this means HTTPS).

Using Cookies with Server Pages

An application implemented with server pages may manipulate cookie data directly using scripting expressions.

For example, a portal application that maintains a database of registered members might save each member's ID or name using a cookie stored on the client machine. For maximum security, each member should be asked to verify a password, but the cookie may be used as a hint.

To store and then later retrieve a cookie, two different server pages may be used. First, to create the "hint" cookie, one server page might use the following code:

```
memberIDCookie := HTTPCookie named: 'member_ID' value: '4567'.
memberIDCookie expireAfterDays: 90.
response addCookie: memberIDCookie.
```

The cookie named member_ID is assigned a string that contains the ID (‘4567’). Then, it is set to expire after 90 days. To attach the cookie to the HTTP response, we use addResponse:. When the page is displayed in the client’s browser, a new cookie is created on the client’s machine.

When the user returns to the portal application at a later time, the client’s browser will include the saved cookie in the HTTP request. The ID may be retrieved from the request as follows:

```
    memberID := request cookieValueAt: 'member_ID'.
```

Note that cookieValueAt: does not return an instance of HTTPCookie, but rather the String value associated with the member_ID cookie (or nil, if no cookie by that name was included with the request).

Index