Writing Web Applications with WAI

Netscape Enterprise Server/FastTrack Server

Version 3.0/3.01

.


Recycled and Recyclable Paper

The Team:
Engineering: Chris Apple, Mike Barbarino, Mike Belshe, Jim Black, Fred Cox, George Dong, Alex Feygin, Alan Freier, Andy Hakim, Warren Harris, John K. Ho, Ari Luotonen, Mike McCool, Rob McCool, Chuck Neerdaels, Howard Palmer, Ben Polk, Aruna Victor
Marketing: Mike Blakely, Atri Chatterjee, Ben Horowitz, David Pann
Publications: Guy K. Haas
Quality Assurance: Saleem Baber, Roopa Cheluvaiah, Shvetal Desai, Noriko Hosoi, Teresa Hsiao, Pramod Khincha, Joy Lenz, Rajesh Menon, Jun Tong, Cathleen Wang, Carol Widra, Ayyaz Yousaf
Technical Support: John Benninghoff, Brian Kendig, Anthony Lee-Masis, Trevor Placker, Bill Reviea, Dan Yang

Netscape Enterprise Server/Netscape FastTrack Server Version 3.0/3.0.1

Printed in USA
97 96 10 9 8 7 6 5 4 3 2 1

Netscape Communications Corporation 501 East Middlefield Road, Mountain View, CA 94043

Contents

About This Guide

The manual *Writing Web Applications with WAI* documents the web application interface (WAI). You can use this interface to write your own web application services for the Netscape web servers. (For an explanation of web application services, see Chapter 1, "Understanding WAI".)

Who Should Read This Guide?

This guide is intended for use by C, C++, and Java programmers who want to write their own web application services in Netscape web servers.

This document assumes you are familiar with the use of the HyperText Transfer Protocol (HTTP), the Common Gateway Interface (CGI), and client-server architecture, as well as the tools involved in compiling, linking, and launching programs written in languages such as C, C++, and Java. This document builds on that knowledge to enable you to interface your application to the web server to enable client programs to access that application.

What's in This Guide?

This guide explains how to use the web application interface (WAI) in the Netscape web servers. The guide documents the C, C++, and Java interfaces in the WAI.

Table 1 describes each chapter in more detail.

Table P.1 Finding information In this manual

To do this:	See this chapter:
Learn more about WAI and the Netscape web servers	Chapter 1, "Understanding WAI"
Learn how the sample applications work	Chapter 2, "Quick Start: Running the Examples"
Learn how to use WAI to write your own application	Chapter 3, "Using WAI"
Find out how to write a WAI application in C	Chapter 4, "Writing a WAI Application in C"
Find out how to write a WAI application in C++	Chapter 5, "Writing a WAI Application in C++"
Find out how to write a WAI application in Java	Chapter 6, "Writing a WAI Application in Java"
Find out how to write an in-process server plug-in	Chapter 7, "Writing a WAI Server Plug-In"
Understand security issues with WAI	Chapter 8, "Security Guidelines for Using WAI"
Look up the description of an interface	Chapter 9, "WAI Reference"
Learn about C++ and Java naming services	Chapter 10, "Naming Services"
Troubleshoot problems with WAI applications	Chapter 11, "Troubleshooting Problems"

Conventions in This Book

Monospaced font — This typeface is used for sample code and code listings, API and language elements (such as function names and class names), filenames, pathnames, directory names, HTML tags, and any text that must be typed on the screen. (Monospaced italic font is used for placeholders embedded in code.)

Italics — Italic type is used for book titles, emphasis, variables and placeholders, and words used in the literal sense.

Boldface Boldface type is used for glossary terms and tutorial steps.

Sidebar text Notes and warnings in the sidebar mark important information. Make sure you read the information before continuing with a task. In the reference section of this manual, sidebar text is also used to label different sections of the documentation for a language component (such as a function or class).

| The vertical bar is used as a separator for user interface elements. For example, File | New means you should click the File menu and select New; Server Status | Log Preferences means you should click the Server Status button in the Server Manager and click the Log Preferences link.

Chapter

1

Understanding WAI

The Web Application Interface (WAI) is one of the programming interfaces that allow you to extend the functionality of Netscape web servers.

WAI is a CORBA-based programming interface that defines object interfaces to the HTTP request/response data and server information. Using WAI, you can write a web application in C, C++, or Java that accepts an HTTP request from a client, processes it, and returns a response to the client. You can also write your own server plug-ins for processing HTTP requests.

Understanding Version Differences

The process for setting up and running WAI applications differs between versions 3.0 and 3.01 of the Netscape web servers:

- In the 3.0 release of Netscape web servers, the web server depends on the `osagent` utility. This utility is used to help operate the object request broker (ORB).

  In order to run a 3.0 version of a web server, you need to run the `osagent` utility first. You can also use the `osfind` utility (provided with 3.0 servers) to troubleshoot problems.

  You can install a patch that fixes and improves the WAI programming interface to the Enterprise Server in the following ways:

— osagent is no longer required to be running.

— WAI server plug-ins are officially supported.

— You can use OAD to activate your WAI applications. (Note that OAD will start only out-of-process WAI applications in C/C++ only and is not supported on Windows NT.)

For more information on this patch and instructions on how to get it and install it, go to http://help.netscape.com/filelib.html#wai.

- In the 3.01 release of Netscape web servers, the web server no longer requires the osagent utility. You do not need to run this utility before starting a 3.01 version of the web server.

  The osagent and osfind utilities are no longer included with the 3.01 release of the web server, since the web server no longer requires these utilities to run.

In general, features or instructions specific to a release are noted in the manual.

Understanding CORBA

The Common Object Request Broker Architecture (CORBA) provides a distributed object infrastructure that supports interoperability across networks, languages, and operating systems.

A CORBA Object Request Broker (ORB) is a mechanism that allows client objects to make requests and receive responses transparently, regardless of the server object's location, operating system, or implementation language. (With an ORB, you can design your object interfaces in a neutral language called the Interface Definition Language, or IDL).

Netscape includes a CORBA ORB with the Netscape web servers. WAI was designed in IDL and includes Java, C++, or C "wrappers". You can call functions in these wrappers when writing your own CORBA-compliant applications that interact with the server via this ORB. (For more details, see the next sections, "Understanding IDL" on page 7 and "WAI Wrapper Classes" on page 7.)

The CORBA architecture is a standard developed by the Object Management Group, Inc. (OMG), an international consortium of more than 500 computer industry companies. For more information about CORBA, IDL, or OMG, see the OMG publication entitled The Common Object Request Broker: Architecture and Specification at http://www.omg.org.

Understanding IDL

Interface Definition Language (IDL) is a generic, descriptive language used to define interfaces between client objects and object implementations. An interface described in IDL can be implemented in any language.

WAI describes a set of objects and methods that let you access HTTP requests and server information as well as return results to a browser. The description of WAI is detailed in an Interface Description Language (IDL) specification. IDL is a language that allows you to describe an interface in a generic way and then allows you to compile that specification to a target language such as Java or C++.

Each interface definition specifies the operations that can be performed and the input and output parameters required. For example, the interface definition for an HTTP request describes how clients can access request headers and set response headers.

(The interfaces are defined in *.idl files, which are located in the *server_root*/wai/idl directory on UNIX and the *server_root*\wai\idl directory on Windows NT.)

Because the interfaces are described in a generic language rather than in a specific programming language, you can use the description of an interface to implement client/server applications in a variety of languages.

WAI Wrapper Classes

WAI includes wrapper classes (classes that implement the interfaces) for C++ and Java and a C interface. You can use C, C++, or Java to write your own applications that access HTTP request objects through the defined interface.

You can also write server plug-ins in C or C++ that use the functions and classes defined in WAI.

For example, one of the methods of the HTTP request interface describes how clients can add a header to the response sent to the client. This method is described in IDL:

Interface described in IDL:

```
HttpServerReturnType addResponseHeader(in string header,
    in string value);
```

WAI provides wrapper classes in Java and C++ (and a C interface) that implement this interface:

Function call in C:

```
NSAPI_PUBLIC WAIReturnType_t WAIaddResponseHeader(ServerSession_t p,
    const char *header, const char *value);
```

Method in C++:

```
WAIReturnType addResponseHeader(const char * header,
    const char * value);
```

Method in Java:

```
public abstract netscape.WAI.HttpServerReturnType addResponseHeader(java.lang.String
    header,java.lang.String value);
```

In your application or plug-in, you can call these methods to add the response header. The methods (in Java and C++) and C function implement the interface specified in IDL; they share the same parameters (except the C function, which has an additional argument for the server session object) and return the same type of value.

How Web Application Services Work

Using WAI, you can write a server plug-in or a web application service. For example, you can write a web application service that processes posted data from forms. These web application services work in the following way:

1. **You write a web application service with WAI.**

   In your application or server plug-in, you define a class derived from the WAIWebApplicationService base class provided with WAI.

2. **On startup, your application/server plug-in registers with a web server.**

When writing your application or server plug-in, you register it by calling the RegisterService method of the WAIWebApplicationService base class.

You register your application/server plug-in under a unique instance name. Netscape web servers include a built-in name service that keeps track of these instance names.

3. **End users access your web application service.**

   To access a web application service, end users visit URLs in the following format:

   http://*server_name*:*port_number*/iiop/*service_name*

   For example, if your server is named mooncheese, it is on port 80, and your application/server plug-in registers under the name MyWebApp, users can access your web application service by visiting the following URL:

   http://mooncheese:80/iiop/MyWebApp

4. **The web server runs the appropriate method in your web application service class.**

   The web server invokes the Run method of your web application service class. You write this method to process the incoming HTTP request, retrieve data from the request, and send a response back to the client.

The rest of this manual describes this process in more detail.

Chapter

2

Quick Start: Running the Examples

This chapter explains how to compile and run some of the sample WAI applications provided with your server.

- Running the Sample C Application (CIIOP)
- Running the Sample C++ Application (WASP)
- Running the Sample Java Application (WASP.Java)
- Running the FormHandler Sample

You can find these sample applications in the *server_root*/wai/examples directory on UNIX and in the *server_root*\wai\examples directory on Windows NT.

Note These examples assume that your server is running in non-secure mode.

For more detailed information on setting up, writing, and running WAI applications, see the rest of the chapters in this manual:

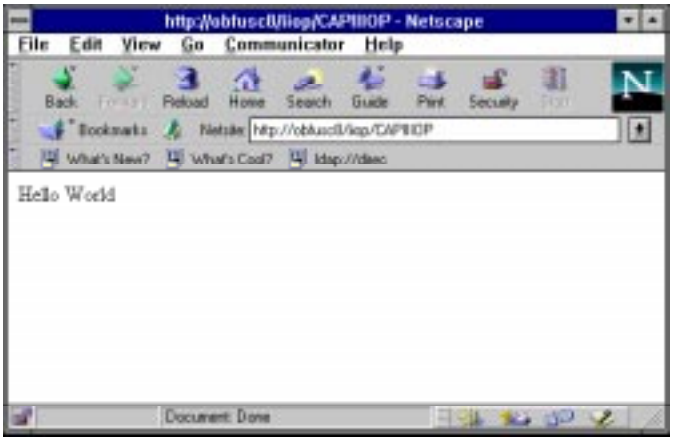
- To set up your server to run WAI applications, see Chapter 3, "Using WAI".
- To write a WAI application in C, C++, or Java, see Chapter 4, "Writing a WAI Application in C", Chapter 5, "Writing a WAI Application in C++", Chapter 6, "Writing a WAI Application in Java", and Chapter 7, "Writing a WAI Server Plug-In".

- For tips on troubleshooting problems with WAI applications, Chapter 11, “Troubleshooting Problems”.

Running the Sample C Application (CIIOP)

The sample C application provided with the web server is in the *server_root*/wai/examples/CIIOP directory. The source file for the example is CAPIIIOP.c.

This example sends a page containing the text Hello World back to the client, as shown in the following figure:



The rest of this section explains how to set up and use this example. You can use this as a guideline for setting up and running your own C examples.

To run the sample C application, follow these steps:

1. **(For 3.0 servers only)Start up osagent.**

   osagent is located under *server_root*/wai/bin in UNIX and *server_root*\wai\bin in Windows NT.

   Specify the -a flag to restrict osagent to the localhost IP address. For example:

   ```
   osagent -a 127.0.0.1
   ```

   For more information, see “Starting osagent (3.0 Servers Only)” on page 33.

2. **Enable WAI applications on the web server.**

From the Server Administration page in the administration server, click the button labelled with your server name. This displays the Server Manager for your server.

Click Programs | WAI Management to display the form for administering WAI on your server.

Under Enable WAI Services, select the Yes radio button and click OK. Save and apply your changes.

For more information, see "Setting the Option to Enable WAI" on page 34.

3. **In the wai/examples/CIIOP (in UNIX) or wai\examples\CIIOP (on Windows NT) directory, review the sample source file CAPIIIOP.c.**

   Basically, the code in this source file does the following (for a more complete explanation of these steps, see Chapter 4, "Writing a WAI Application in C"):

   - Accepts an argument that specifies the host and port where the web server is running. For example, you can use the following command argument to specify that your web server is running on port 80 of the server named mooncheese:

     ```
     CAPIIIOP mooncheese:80
     ```

   - Calls the WAIcreateWebAppService() function to create a new web application service named CAPIIIOP. Users will be able to access this web service through the following URL (if, for example, your web server is running on port 80 of the server named mooncheese):

     ```
     http://mooncheese.mydomain.com:80/iiop/CAPIIIOP
     ```

   - Calls the WAIregisterService() function to register the web application with the web server running on the host and port number specified on the command-line.

   - Calls the WAIimplIsReady() function to indicate to the web server that it is ready to receive requests.

   When the application receives a request, it does the following:

   - Calls the WAIsetResponseContentLength() function to specify the content length of the page returned to the client.

   - Calls the WAIStartResponse() function to start sending the response to the client.

- Calls the WAIWriteClient() function to send the text "Hello World" to the client.

4. **Compile and link the sample application.**

   The sample application includes a Makefile (for example, Makefile.SOLARIS or Makefile.WINNT) that you can use to compile and link the application.

   For more information on compiling and linking your application, see "Compiling C/C++ Applications" on page 39.

5. **After compiling and linking the application, run the application by entering the following command:**

   CAPIIIOP *hostname*:*port*

   where *hostname* and *port* identify the name of the machine that the web server runs on and the port number that the server listens to. For example:

   CAPIIIOP *myserver*:80

   This registers the application with the web server. The web server should be able to find the CAIIIOP WAI application.

6. **In a web browser, go to the following URL:**

   http://*hostname*:*port*/iiop/CAPIIIOP

   where *hostname* and *port* identify the name of the machine that the web server runs on and the port number that the server listens to. For example:

   http://*myserver*:80/iiop/CAPIIIOP

   The web server processes the request. While processing the request, the server parses the URL, retrieves the name of the service you want to access (CAPIIIOP), and contacts your application.

   Your application receives the request and returns the Hello World string. The web server returns this to the web browser.

Effectively, the web browser has requested a service, and your WAI application has delivered back results through the web server.

Running the Sample C++ Application (WASP)

The sample C++ application provided with the web server is in the *server_root*/wai/examples/WASP directory. The source file for the example is WASP.cpp.

This example does the following:

- Sends a cookie to the browser, if the browser does not already have a cookie set
- Gets information about the web server, including the host name and server ID of the web server
- Gets information from the request headers in the request sent by the browser
- Gets information about the request, including the types of information accessible through CGI 1.1 environment variables
- Sends this information back to the client in an HTML page

The following screenshot illustrates the results of this service.

Note that this example can be compiled and linked as a standalone application that runs outside the web server's process and as a server plug-in that runs within the web server's process.

The rest of this section explains how to set up and use the standalone application in this example. (For an example of writing a server plug-in, see Chapter 7, "Writing a WAI Server Plug-In".)

You can use this example as a guideline for setting up and running your own C++ examples.

To run the sample C++ application, follow these steps:

1. **(For 3.0 servers only) Start up osagent.**

   osagent is located under *server_root*/wai/bin in UNIX and *server_root*\wai\bin in Windows NT.

   Specify the -a flag to restrict osagent to the localhost IP address. For example:

   osagent -a 127.0.0.1

   For more information, see "Starting osagent (3.0 Servers Only)" on page 33.

2. **Enable WAI applications on the web server.**

   From the Server Administration page in the administration server, click the button labelled with your server name. This displays the Server Manager for your server.

   Click Programs | WAI Management to display the form for administering WAI on your server.

   Under Enable WAI Services, select the Yes radio button and click OK. Save and apply your changes.

   For more information, see "Setting the Option to Enable WAI" on page 34.

3. **In the wai/examples/WASP (in UNIX) or wai\examples\WASP (on Windows NT) directory, review the sample source file WASP.cpp.**

   Basically, the code in this source file does the following (for a more complete explanation of these steps, see Chapter 5, "Writing a WAI Application in C++"):

   - Accepts an argument that specifies the host and port where the web server is running. For example, you can use the following command argument to specify that your web server is running on port 80 of the server named mooncheese:

     ```
     WASP mooncheese:80
     ```

- Creates a new web application service named WASP. Users will be able to access this web service through the following URL (if, for example, your web server is running on port 80 of the server named mooncheese):

  http://mooncheese.mydomain.com:80/iiop/WASP

- Calls the RegisterService method to register the web application with the web server running on the host and port number specified on the command-line.

When the application receives a request, it does the following:

- Calls the getCookie and setCookie methods to demostrate how to get and set cookies in the client.
- Calls several different methods to illustrate the kinds of data you can get from the session's context and the client's request. For example, to get information from the request, the sample application calls the getRequestInfo method.
- Calls the setResponseContentLength method to specify the length of the content to be delivered to the client.
- Calls the StartResponse method to start sending the HTTP response back to the client.
- Calls the WriteClient method to send data back to the client.

4. **Compile and link the sample application.**

   The sample application includes a sample Makefile (for example, Makefile.SOLARIS or Makefile.WINNT) that you can use to compile and link the application.

   For more information on compiling and linking your application, see "Compiling C/C++ Applications" on page 39.

5. **After compiling and linking the application, run the application by entering the following command:**

   WASP *hostname*:*port*

   where *hostname* and *port* identify the name of the machine that the web server runs on and the port number that the server listens to. For example:

   WASP *myserver*:80

This registers the application with the web server. The web server should be able to find the WASP WAI application.

6. **In a web browser, go to the following URL:**

   http://*hostname*:*port*/iiop/WASP

   where *hostname* and *port* identify the name of the machine that the web server runs on and the port number that the server listens to. For example:

   http://*myserver*:80/iiop/WASP

   The web server processes the request. While processing the request, the server parses the URL, retrieves the name of the service you want to access (WASP), and contacts your application.

   Your application receives the request and retrieving information from the request and the web server. The web server returns this information to the web browser in an HTML page.

Effectively, the web browser has requested a service, and your WAI application has delivered back results through the web server.

Running the Sample Java Application (WASP.Java)

The sample Java application provided with the web server is in the *server_root*/wai/examples/WASP directory. The source file for the example is WASP.java.

This example does the following:

- Sends a cookie to the browser, if the browser does not already have a cookie set
- Gets information about the web server, including the hostname and serverID of the web server
- Gets information from the request headers in the request sent by the browser
- Gets information about the request, including the types of information accessible through CGI 1.1 environment variables
- Sends this information back to the client in an HTML page

The following screenshot illustrates the results of this service.

The rest of this section explains how to set up and use this example. You can use this as a guideline for setting up and running your own Java examples.

To run the sample Java application, follow these steps:

1. **(For 3.0 servers only) Start up osagent.**

   osagent is located under *server_root*/wai/bin in UNIX and *server_root*\wai\bin in Windows NT.

   Specify the -a flag to restrict osagent to the localhost IP address. For example:

   ```
   osagent -a 127.0.0.1
   ```

   For more information, see "Starting osagent (3.0 Servers Only)" on page 33.

2. **Enable WAI applications on the web server.**

   From the Server Administration page in the administration server, click the button labelled with your server name. This displays the Server Manager for your server.

   Click Programs | WAI Management to display the form for administering WAI on your server.

   Under Enable WAI Services, select the Yes radio button and click OK. Save and apply your changes.

   For more information, see "Setting the Option to Enable WAI" on page 34.

3. **In the wai/examples/WASP (in UNIX) or wai\examples\WASP (on Windows NT) directory, compile the sample application.**

   Make sure to include the following files in your CLASSPATH environment variable:

   - *server_root*/wai/java/nisb.zip
   - *server_root*/wai/java/WAI.zip

4. **After compiling the application, run the application.**

   If you are running a 3.0 version of a Netscape web server, run the following command:

   java -DDISABLE_ORB_LOCATOR WASP *hostname*:*port*

   The -DDISABLE_ORB_LOCATOR option specifies that osagent should not be used to find the ORB in the Netscape web server.

   If you are running a 3.0.1 version of a Netscape web server, run the following command:

   java WASP *hostname*:*port*

   This registers the application with the web server. The web server should be able to find the JavaWASP WAI application.

5. **In a web browser, go to the following URL:**

   http://*hostname*:*port*/iiop/JavaWASP

   where *hostname* and *port* identify the name of the machine that the web server runs on and the port number that the server listens to. For example:

   http://*myserver*:80/iiop/JavaWASP

   The web server processes the request. While processing the request, the server parses the URL, retrieves the name of the service you want to access (JavaWASP), and contacts your application.

   (Note that the name used to register the server -- JavaWASP -- does not necessarily need to be the same as the name of the class -- WASP.)

   Your application receives the request and retrieving information from the request and the web server. The web server returns this information to the web browser in an HTML page.

Effectively, the web browser has requested a service, and your WAI application has delivered back results through the web server.

Running the FormHandler Sample

The classes used for writing WAI applications include a class for handling submissions through HTML forms. Using the FormHandler class, you can write a WAI application that receives and interprets data submitted through an HTML form.

To read in and parse posted form data (where the client used the HTTP POST method to submit the form), create an instance of the FormHandler class. The constructor for this class reads in the data and parses it.

To read in and parse form data submitted through the HTTP GET method, create an instance of the FormHandler class and call the ParseQueryString method.

Depending on the language you are using, you can access the parsed data in different ways:

- In C++, you can call the Get method to get the value of a specific name-value pair, or you can call the InitIterator method and the Next method to iterate through all name-value pairs in the parsed data.

  You can also call the Add method to add a new name-value pair to the parsed form data and the Delete method to remove a name-value pair from the parsed form data.

- In Java, you can call the GetHashTable method to get a Java hash table containing the parsed data. Then, you can call methods of the java.util.Hashtable class to access the data.

  The names serve as keys in the hashtable. The values are stored as Java vectors (for details, see your Java documentation on java.util.Vector).

  The values are implemented as Java vectors because a given name may be associated with multiple values. For example, if the form contains multiple-selection input, the submitted form data can contain several name-value pairs with the same name but different values.

About the FormHandler Class Example

The FormHandler samples provided with the web server are in the *server_root*/wai/examples/forms directory. This directory contains C++ and Java examples of using the WAI FormHandler class. You can use this class to process data submitted through an HTML form.

This directory contains the following files:

- TestDriver.java (Java example)
- form.cpp (C++ example)
- Makefile.SOLARIS (makefile for C++ example on Solaris) or Makefile.WINNT (makefile for C++ example on WIndows NT)
- form.html (HTML form for testing the example)

The C++ example is written as an in-process server plug-in. The Java example is written as a stand-alone application (running out of process). Both examples process and display data submitted through the form.html form.

Running the C++ FormHandler Sample

The FormHandler sample provided with the web server is in the *server_root*/wai/examples/forms directory. The source file for the example is formHandler.cpp.

This example is written as an in-process server plug-in that performs the following tasks:

- It forms pairs of names and values using the NVPair class.
- It gets and parses form data submitted through an HTTP GET method by calling the Add method to add a new name-value pair to the parsed form data. Then it calls the Delete method to remove a name-value pair from the parsed form data.
- It calls the InitIterator method and the Next method to iterate through all name-value pairs in the parsed data.
- It checks whether the name-value pair is valid.
- It puts valid information into a hash table.

This example can be compiled and linked as a stand-alone application that runs outside the web server's process and as a server plug-in that runs within the web server's process.

The rest of this section explains how to set up and use the server plug-in that runs within the web server's process. (For an example of writing a server plug-in, see Chapter 7, "Writing a WAI Server Plug-In".)

You can use this example as a guideline for setting up and running your own C++ forms.

1. **Compile the example using the makefile provided.**

   For example:

   ```
   nmake -f Makefile.WINNT
   ```

   or

   ```
   make -f Makefile.SOLARIS
   ```

2. **Open the obj.conf file (located in the *server-root*/*server-id*/config directory) in a text editor.**

3. **Add an Init directive to specify the intialization function (FormInit) for this server plug-in (form.dll or form.so).**

   For example:

   ```
   Init funcs="FormInit" shlib="server-root/wai/examples/forms/form.dll" fm="load-modules"
   Init LateInit="yes" fn="FormInit"
   ```

   or

   ```
   Init funcs="FormInit" shlib="server_root/wai/examples/forms/form.so" fn="load-modules"
   Init LateInit="yes" fn="FormInit"
   ```

   When you specify the Init directive make sure to set LateInit to "yes".

4. **Save your changes and exit from the text editor.**

5. **In the Administration Server, click the Apply Changes button in the top frame and restart the Enterprise Server.**

6. **Copy form.html to the documentation root directory of your Enterprise Server (for example, *server-root*/docs).**

7. **Open the form.html file in a text editor and verify that the action of the form is set to "/iiop/FORMip".**

For example:

```
<FORM name="submitform" method="POST" ACTION="/iiop/FORMip">
```

FORMip is the name with which this WAI server plug-in registers.

8. **Go to the following URL:**

   http://*server-name*:*port-number*/form.html

9. **Fill in the fields and click Send to submit the form.**

   The WAI server plug-in should send a generated HTML page back to your browser. The page should display some of the data you have submitted.

Running the Java FormHandler Sample

The Java example is written as a stand-alone application, running out of process. It processes and displays data submitted through the form.html form in the /wai/examples/forms directory.

1. **Compile the TestDriver.java example.**

   ```
   javac TestDriver.java
   ```

2. **Run the TestDriver Java application.**

   Specify the server name and port number of your Enterprise Server as follows:

   java TestDriver *server-name*:*port-number*

3. **Copy form.html to the documentation root directory for your Enterprise Server (for example, *server-root*/docs).**

4. **Open the form.html file in a text editor and change the action of the form to "/iiop/JavaForm".**

   For example:

   ```
   <FORM name="submitform" method="POST" ACTION="/iiop/JavaForm">
   ```

   JavaForm is the name with which this WAI application registers.

5. **Go to the following URL:**

http://*server-name*:*port-number*/form.html

6. **Fill in the fields and click Send to submit the form.**

   The WAI application should send a generated HTML page back to your browser. The page should display some of the data you have submitted.

Chapter

3

Using WAI

This chapter provides an overview for writing WAI applications. Read this chapter for general information on using WAI, including:

- System Requirements
- Overview
- Before You Use WAI
- Converting CGI Applications to WAI
- Setting Up the Web Server
- Compiling Applications and Server Plug-Ins
- Running Applications

To see working examples of WAI applications and to get a better understanding of how the material in this chapter applies to WAI, read Chapter 2, "Quick Start: Running the Examples".

System Requirements

C++ Requirements: If you are writing a C++ application in WAI, you must use the following:

- For Windows NT, Microsoft Visual C++ version 4.2
- For Solaris 2.5.x, the SparcWorks C++ compiler version 3.0.1
- For IRIX 6.2, the C++ compiler version 7.1

Java Requirements: If you are writing a Java application in WAI, you must use the following:

- The Javasoft Java Development Kit 1.1.x.

You can also use Java development tools that are compliant with the JDK 1.1.x.

Overview

You can use WAI to write a web application service in C, C++, or Java that receives a request from a client, processes the request, and returns data back to the client. You can:

- Access data from the headers in the HTTP request
- Access information about the web server
- Read data from the client (such as data in an HTML form sent through the HTTP POST method)
- Set the headers in the response that will be sent to the client
- Set the status of the response that will be sent to the client
- Redirect the client to another location
- Write data back to the client (such as an HTML page)

You can use WAI to write, compile, and run the following:

- An application that runs outside the web server's process. You can write this in C, C++, or Java. For details, see the following chapters:
  - Chapter 4, "Writing a WAI Application in C"
  - Chapter 5, "Writing a WAI Application in C++"
  - Chapter 6, "Writing a WAI Application in Java"

Note that by default, the web server configuration assumes that you will run these applications on the same machine as the web server. You can reconfigure the web server to interact with applications running on remote machines, but you need to be aware of the security issues involved with this configuration. For details, see Chapter 8, "Security Guidelines for Using WAI".

- A server plug-in that runs within the web server's process. A server plug-in is a shared library or dynamic link library that the web server loads and initializes during startup. You can write this in C or C++. For details, see the following chapter:
  - Chapter 7, "Writing a WAI Server Plug-In"

Before You Use WAI

Before you begin to set up your server to use WAI, you should read through the following sections.

Understanding Security Issues

Before you begin implementing WAI applications at your site, you should read the discussion on security-related issues in Chapter 8, "Security Guidelines for Using WAI".

In general, Netscape recommends that you restrict WAI applications to run only on the local host machine (where the web server runs). You should also restrict login access to this machine to prevent unauthorized users from executing WAI applications.

Read the material in Chapter 8, "Security Guidelines for Using WAI" for a complete explanation of these recommendations.

Understanding Version Differences

The process for setting up and running WAI applications differs between versions 3.0 and 3.01 of the Netscape web servers:

- In the 3.0 release of Netscape web servers, the web server depends on the osagent utility. This utility is used to help operate the object request broker (ORB).

In order to run a 3.0 version of a web server, you need to run the osagent utility first. You can also use the osfind utility (provided with 3.0 servers) to troubleshoot problems.

You can install a patch that fixes and improves the WAI programming interface to the Enterprise Server in the following ways:

— osagent is no longer required to be running.

— WAI server plug-ins are officially supported.

— You can use OAD to activate your WAI applications. (Note that OAD will start only out-of-process WAI applications in C/C++ only and is not supported on Windows NT.)

For more information on this patch and instructions on how to get it and install it, go to http://help.netscape.com/filelib.html#wai.

- In the 3.01 release of Netscape web servers, the web server no longer requires the osagent utility. You do not need to run this utility before starting a 3.01 version of the web server.

  The osagent and osfind utilities are no longer included with the 3.01 release of the web server, since the web server no longer requires these utilities to run.

In general, features or instructions specific to a release are noted in the manual.

Converting CGI Applications to WAI

If you have existing programs or modules in CGI, convert them to WAI modules or services to improve performance. CGI starts a new session every time you access it, increasing performance times. Because WAI modules (or WAI services) are persistent, they reduce performance times. You have the option of running applications externally or calling functions from an internal library.

A fundamental difference between CGI and WAI is that CGI programs are written to exist while WAI modules persist. Additionally, WAI modules are inherently multi-threaded so creating additional processes is unnecessary.

Table 3.1 describes the structure of a CGI program alongside the structure of a WAI service:

Table 3.1 Comparison of CGI program structure to WAI program structure

CGI Structure	WAI Structure
Read data from POST data input stream.	Collect data using the methods of the netscape::WAI::HttpServerRequest and netscape::WAI::HttpServerContext objects.
Process, using CGI variables as necessary.	Process using the methods in the WAIWebApplicationService class.
Writes HTML output to the browser.	Sends response back to the client using the methods in the netscape::WAI::HttpServerRequest object.

Table 3.2 lists the getRequestInfo variables with CGI equivalents.

Table 3.2 WAI getRequestInfo variables with corresponding CGI functions

WAI variable name	Description
CLIENT_CERT	Authenticationschemefortherequest(foundfromtheauth-scheme token in the request).
HOST	Name of the client's host machine
HTTPS	Specifies whether or not SSL is "ON" or "OFF".
HTTPS_KEYSIZE	Number of bits in the sesion key used to encrypt the session (if SSL is enabled).
HTTPS_SECRETKEYSIZE	Number of bits used to generate the server;s private key (if SSL is enabled).
URI	URI requested by the client
URL	Complete URL requested by the client.

Most of the CGI variables are the same as the getRequestInfo variables in WAI. The other CGI variables are retrieved out of the netscape::WAI::HttpServerContext object. Table 3.3 lists the CGI variables that correspond to the netscape::WAI::HttpServerContext variables:

Table 3.3 WAIServerContext methods with corresponding CGI functions

HttpServerContext method	Description
getName	SERVER_NAME. The name for the server, as used in the *host* part of the script URI. Either a fully qualified domain name or an IP address.
getPort	SERVER_PORT. The port on which this request was received, as used in the *port* part of the script URI.
getServerSoftware	SERVER_SOFTWARE. The name and version of the information server software answereing the request and running the gateway.

The CGI functions in Table 3.4 lists the CGI functions that have no equivalent in WAI.

Table 3.4 CGI variables that do not correspond to getRequestInfo or WAIServerContext variables

CGI variable name	Description
GATEWAY_INTERFACE	The version of the CGI specification to which the server complies.
REMOTE_IDENT	The identity information reported about the connection by an RFC 931[10] request to the remote agent, if available.

Setting Up the Web Server

In order to enable the web server to use applications written in WAI, you need to do the following:

1. **(For 3.0 servers only) Start osagent.**

   osagent is used to help operate the object request broker (ORB). See "Starting osagent (3.0 Servers Only)" on page 33 for details. If you are running a 3.01 version of a web server, you can ignore this step.

2. **(For 3.0 servers only) Install the patch that allows you to run the 3.01 version of WAI.**

   This patch release fixes and improves the WAI programming interface to the Enterprise Server in the following ways:

   - osagent is no longer required to be running.
   - WAI server plug-ins are officially supported.
   - You can use OAD to activate your WAI applications. (Note that OAD will start only out-of-process WAI applications in C/C++ only and is not supported on Windows NT.)

   For more information on this patch and instructions on how to get it and install it, go to http://help.netscape.com/filelib.html#wai.

3. **From the administration server, set the option to enable WAI applications to run on your server.**

   See "Setting the Option to Enable WAI" on page 34 for details.

4. **Optionally, you can change any of the default settings for the web server's ORB.**

5. **Optionally, you can configure the web server to log WAI status messages.**

   Some of the WAI messages, such as the startup message, are only logged if the server is configured to log messages at the "verbose" level.

   For more information about logging WAI status messages, read "Logging Status Messages".

6. **If you are running an in-process server plug-in, edit the server's configuration files to specify your shared library or shared object and the function that you want to invoke.**

Starting osagent (3.0 Servers Only)

osagent, which is provided with 3.0 versions of Netscape web servers, is used to help operate the object request broker (ORB).

Note osagent is not required for 3.01 versions of Netscape web servers and is no longer packaged with those versions of the server.

osagent is located in the *server_root*/wai/bin directory on UNIX and in the *server_root*\wai\bin directory on Windows NT. To run osagent, enter the following command:

```
osagent -a 127.0.0.1
```

The -a flag specifies the address that osagent binds to. You should specify the localhost address (127.0.0.1) for security reasons. For details on these reasons, see Chapter 8, "Security Guidelines for Using WAI".

On Windows NT, you can create a shortcut or program item that runs this command. If you have the Windows NT Resource Kit, you can use the SrvAny command to create a service for osagent. You can set up this service to automatically when your machine starts up. For details, consult the documentation in the Windows NT Resource Kit.

Setting the Option to Enable WAI

You need to configure the web server to interact with WAI applications and server plug-ins.

Configuring the Server

1. **In your web browser, go to the URL for the administration server**.

   When prompted, enter the username and password of the server administrator.

2. **On the Server Selector page, click the button labelled with your server name.**

   This brings you to the Server Manager page for your server.

3. **In the menu of categories in the top frame, click Programs.**

4. **Under Programs in the left frame, click the WAI Management link.**

5. **Under Enable WAI Services, select Yes, then click OK.**

6. **Click Save and Apply to save your changes.**

What Happens When You Enable WAI

When you enable WAI, the following changes are made to your obj.conf file:

- Adds an Init directive that loads the functions IIOPinit, IIOPexec, and IIOPNameService from the shared library libONEiiop.so.10 (filename extension may differ, depending on your UNIX platform) or the dynamic link library ONEiiop10.dll (on Windows NT).
- Adds an Init directive that executes the function IIOPinit on server startup. This function initializes the object request broker (ORB), the basic object adapter (BOA), and the built-in name service.
- Adds a NameTrans directive to associate requests for any resources matching /NameService* with the IIOPnameservice object. The stop parameter in this directive causes the server to skip over the other NameTrans directives (effectively, it returns a REQ_PROCEED to indicate that the server should proceed with the next step in processing the request).
- Adds an IIOPnameservice object, which represents the name service. The IIOPNameService service function associated with this object provides access to the built-in name service for WAI applications.
- Adds a NameTrans directive to translate requests for resources beginning with the /iiop prefix to the iiopexec object. URIs in this form typically use the format /iiop/instance_name, where instance_name is the name of the web service that the client wants to access. The dir parameter is used to help parse the /iiop prefix out of URI to get the instance name of the web service that needs to be accessed.
- Adds an object named iiopexec, which interprets a URI into a request for a web service. The IIOPexec function associated with this object passes the request on to the appropriate WAI application.

Configuring the Web Server's ORB

In most cases, you can run the web server without specifying any additional configuration parameters for the server's object request broker (ORB). In certain situations, however, you might need to override the default configuration.

Changing the ORB Configuration Information

To change the web server's ORB configuration information, you need to edit the obj.conf file for your server (which is located in the *server_root*/*server_id*/config directory of your server).

In the Init directive that executes the IIOPinit function, add configuration parameters to specify changes to the ORB configuration.

After editing the obj.conf file, you need to stop and start your server so that the server can read in the updated file.

Note Before changing the configuration, you should be aware of the security issues involved with running WAI applications on other machines. See Chapter 8, "Security Guidelines for Using WAI" for details.

Listing of Configurable Parameters

You can add any of the parameters listed in Table 3.5 to the Init directive for the IIOPinit function.

The following table lists the parameters that you can specify in the Init directive for the IIOPinit function

Table 3.5 IIOPInit Parameters

Parameter Name	Description
ORBagentaddr	**(For 3.0 servers only)** Specifies the IP address where osagent is running. The ORB uses this setting to find osagent. If this parameter is not set, the ORB uses the localhost IP address (127.0.0.1) by default. If you have configured osagent to use a different IP address than localhost, you need to include this parameter in the Init directive.
ORBagentport	**(For 3.0 servers only)** Specifies the port number used by osagent. The ORB uses this setting to find osagent. If you have configured osagent to use a port number other than the default port, you need to include this parameter in the Init directive.
ORBsendbufsize	Specifies the size of the send buffer to be used by the network transport mechanism. If not specified, an appropriate default size will be used.

Table 3.5 IIOPInit Parameters

Parameter Name	Description
ORBrcvbufsize	Specifies the size of the receive buffer to be used by the network transport mechanism. If not specified, an appropriate default size will be used.
ORBmbufsize	Specifies the size of the intermediate buffer used by the ORB. If not specified, the ORB will maintain a pointer to the argument and will not make an intermediate copy. Using this parameter incorrectly can seriously affect performance.
ORBshmsize	Specifies the size of the shared memory buffer used by the ORB. If this is not specified, an appropriate size will be used.
OAipaddr	Specifies the IP address to be used for this BOA. If this parameter is not set, the ORB uses the localhost IP address (127.0.0.1) by default.
OAport	Specifies the port number to use for this BOA. If not specified, an unused port number is used.
OAshm	Enables the use of shared memory.
OAnoshm	Disables the use of shared memory for sending and receiving messages when the client and object implementation are located on the same host.
OAsendbufsize	Specifies the size in bytes of the network transport's send buffer. If this option is not specified, an appropriate buffer size is used.
OArcvbufsize	Specifies the size in bytes of the network transport's receive buffer. If this option is not specified, an appropriate buffer size is used.

Example of Configuring the ORB

For example, in a 3.0 version of a web server, suppose you are running the osagent from IP address 205.217.229.39 on port 15001. By default, the web server expects the osagent utility to run on the localhost IP address (127.0.0.1) under the default port.

In the obj.conf file, change the Init directive for the IIOPinit function from:

```
Init LateInit="yes" fn="IIOPinit"
```

to:

```
Init LateInit="yes" fn="IIOPinit" ORBagentaddr="205.217.229.39" ORBagentport="15001"
```

In your WAI application, you also need to specify this argument when initializing the ORB and BOA. For example:

```
int bargc = 0;
char **bargv = new char *[3];
bargv[bargc++] = "-OAipaddr";
bargv[bargc++] = "204.200.215.98";
bargv[bargc] = 0;
// Initialize the ORB.
ORB orb = org.omg.CORBA.ORB.init(bargc, bargv);
// Initialize the BOA.
BOA boa = orb.BOA_init(bargc, bargv);
```

Logging Status Messages

Some of the status messages (such as the WAI initialization messages) are logged to the server's error log only if the server is running with the LogVerbose option turned on. These are messages that are logged with the severity level LOG_VERBOSE.

If you want these types of messages logged, edit the magnus.conf file and add the following directive:

```
LogVerbose on
```

The verbose log information is stored in *server-root*/https-*serverID*/logs/errors and *server-root*/https-*serverID*/logs/access.

After editing the magnus.conf file, you need to stop and start your server so that the server can read in the updated file. You can find the mangus.conf file in *server-root*/https-*serverID*/logs/config.

Compiling Applications and Server Plug-Ins

When compiling and linking your application or server plug-in, follow the tips in this section. (You can also look at the makefiles provided with the sample applications.)

Compiling C/C++ Applications

Follow these guidelines for compiling and linking C/C++ applications.

Include Directories

Add the following include directories to your makefile:

- *server_root*/include (UNIX) or *server_root*\include (Windows NT)
- *server_root*/wai/include (UNIX) or *server_root*\wai\include (Windows NT)

Libraries

On UNIX, you can add the following library directories to your linker command. Specify that libraries should be searched for shared object during runtime to resolve symbols (on Solaris, use the -R flag; on IRIX, use the -rpath flag):

- *server_root*/lib
- *server_root*/wai/lib
- *server_root*/bin/https

The following table lists the additional libraries that you need to link to:

Table 3.6 Libraries That You Need to Link to

Platform	Libraries
Solaris	lib/libldap10.so lib/liblcache10.so wai/lib/libONEiiop.so wai/lib/liborb_r.so bin/https/ns-httpd.so libthread.so libposix4.so libresolv.so libnsl.so lib/libnspr.so wai/lib/libIIOPsec.a
Windows NT (in addition to the standard Windows libraries)	wai\lib\ONEiiop10.lib WSOCK32.lib

Table 3.6 Libraries That You Need to Link to

Platform	Libraries
IRIX	lib/libldap10.so lib/liblcache10.so wai/lib/libONEiiop.so wai/lib/liborb_r.so bin/https/ns-httpd.so wai/lib/libIIOPsec.a
HP-UX	dce.sl wai/lib/orb_r.sl wai/lib/ONEiiop.sl bin/https/nshttpd.sl wai/lib/IIOPsec.sl
AIX	wai/lib/ONEiiop_shr wai/lib/IIOPsec bin/https/nshttpd_shr lib/nspr_shr wai/lib/orb_r dcepthreads C_r
Digital UNIX	lib/ldap10.so lib/lcache10.so wai/lib/ONEiiop.so wai/lib/orb_r.so bin/https/ns-httpd.so wai/lib/IIOPsec.so

Compile Flags

The following table lists the flags and defines that you need to use:

Table 3.7 Compile Flags

Platform	Flags/Defines
Solaris	-DXP_UNIX -D_REENTRANT -KPIC
Windows NT	-DXP_WIN32 -DWIN32 /MD
IRIX	-o32 -exceptions -DXP_UNIX -KPIC
HP-UX	-DXP_UNIX -D_REENTRANT -DHPUX

Table 3.7 Compile Flags

Platform	Flags/Defines
AIX	-DXP_UNIX -D_REENTRANT -DAIX $(DEBUG)
Digital UNIX	-DXP_UNIX -KPIC

Compiling C/C++ Server Plug-Ins

In addition to the tips above, follow these tips when compiling server plug-ins (which are shared libraries or dynamic link libraries):

- Specify the appropriate compile options for building shared objects or shared libraries.
- On UNIX, if you are specifying a relative path to the other libraries (using the -R flag on Solaris or the -rpath flag on IRIX), make sure to specify the paths relative to the ns-httpd executable (which is in the *server_root*/bin/https/ directory).

Compiling Java Applications

If you are compiling a Java application, make sure to include *server_root*/wai/java/nisb.zip and *server_root*/wai/java/WAI.zip in your CLASSPATH environment variable.

Running Applications

Start your application on the host machine that runs the web server. Make sure that when your application registers, you specify the host name and port of the web server.

Note that it is possible (but not recommended) to run WAI applications on other machines in the local network. For a complete explanation of the security concerns and instructions for configuring the server to recognize WAI applications on other machines, see Chapter 8, "Security Guidelines for Using WAI".

Setting Up Your Application with OAD

You can set up your WAI application with the Netscape Internet Service Broker's object activation daemon (OAD), a process which automatically starts up your application if it is not running.

For example, you may want to ensure that your application is always running and does not need to be started manually.

To set up your application with the OAD, follow these steps:

1. **Make sure to specify a name for your object in the** WAIWebApplicationService **constructor.**

2. **Set the second argument (activateObject) to WAI_FALSE.**

   At a point in your application where you are ready to launch your object, call the ActivateWAS method of WAIWebApplicationService.

   Compile and run your application at least once, in order to register your application with the web server's naming service.

   You need to register your application before setting it up with OAD. OAD expects your application to be registered with the web server.

3. **Set the following environment variables in the shells where the web server and OAD run:**

   - NS_SERVER_ROOT - set this to the location of your server root directory (for example, /usr/netscape/suitespot or C:\Netscape\SuiteSpot)
   - NS_SERVER_ID - set this to your server identifier (for example, https-myhost)
   - ORBELINE_IMPL_NAME - set this to name of the file created by the OAD; the OAD creates this file to keep track of object implementations. For example, if you want this file to be named myfile, set ORBELINE_IMPL_NAME to myfile.
   - ORBELINE_IMPL_PATH - set this to the path to an existing directory where you want the OAD to generate the file specified by the ORBELINE_IMPL_NAME environment variable. For example, if you want the file created under the /usr/tmp directory, set ORBELINE_IMPL_PATH to /usr/tmp.

   You also need to set the LD_LIBRARY_PATH (or SHLIB_PATH on HP-UX) environment variable to the paths that include all shared libraries linked to by your object server.

For example, in C shell, you might enter the following commands before starting OAD and your webserver:

```
setenv NS_SERVER_ID https-gromit

setenv NS_SERVER_ROOT /usr/netscape/suitespot

setenv LD_LIBRARY_PATH /usr/netscape/suitespot/wai/lib:
            /usr/netscape/suitespot/bin/https:
            /usr/netscape/suitespot/lib:
            /usr/local/java/lib

setenv ORBELINE_IMPL_NAME myfile

setenv ORBELINE_IMPL_PATH /usr/tmp
```

If you start OAD after setting these variables, the OAD will generate the file /usr/tmp/myfile to keep track of the object implementations.

4. **After starting your web server, start the OAD manually.**

   For instructions on starting OAD, see the Netscape Internet Service Broker Reference Guide for C++ or the Netscape Internet Service Broker Reference Guide for Java.

5. **Run regobj to register your service with the OAD.**

   regobj is located in the *server_root*/wai/bin directory. For details on the syntax for this command, see the Netscape Internet Service Broker Reference Guide for C++. You need to specify "*" as the interface name. You can pass arguments to the object server using the -a option.

   For example, to start up the object named WASP implemented by the WAI application /usr/local/ns-home/wai/bin/WASP, use the following command:

```
regobj -o "*,WASP" -f /usr/local/ns-home/wai/bin/WASP
    -a httpServerName=bar:80
```

   The example above assumes that the web server is running on port 80 of the machine named bar.

Using osagent with Java (3.0 Only)

In the 3.0 version of the web server, if you are running a Java application written with WAI, you should specify the -DDISABLE_ORB_LOCATOR flag. This minimizes potential problems with the osagent utility.

For example, if you have written the Java class WASP.class with WAI, use the following command to run your Java application:

```
java -DDISABLE_ORB_LOCATOR WASP
```

Note that if you are specifying the DISABLE_ORB_LOCATOR option for osagent, you must force the web server's basic object adapter (BOA) to listen on a particular port. To do this, follow the instructions below.

1. **Edit the obj.conf file (located in the *server_root*/*server_id*/config directory on UNIX and the *server_root**server_id*\config directory on Windows NT), and change the following line:**

   ```
   Init LateInit="yes" fn="IIOPinit"
   ```

   to:

   ```
   Init LateInit="yes" fn="IIOPinit" OAport="21000"
   ```

   The OAport option specifies the port selected where the web server's BOA listens. The example above sets up the BOA to listen to port 21000.

2. **Delete the files *server_root*/wai/NameService/*server_id*.* on UNIX or *server_root*\wai\NameService*server_id*.* on Windows NT.**

   For example, delete https-myhost.IOR, https-myhost.sav, and https-myhost.bak. These files are name service files for your currently registered objects.

3. **Register your objects with the web server again.**

   For example, start any WAS object servers. You must complete this step. If you do not, you might not be able to register objects with the web server.

Running Applications on Remote Machines

You can configure your WAI applications to run on separate machines other than the machine hosting the web server. Read through the information about security issues in Chapter 8, "Security Guidelines for Using WAI", for more information.

Chapter

4

Writing a WAI Application in C

WAI provides a set of C API functions that you can use to write a WAI application. Your C application should:

- Define a function for processing the incoming HTTP request. (For details, see "Defining a Function to Process Requests" on page 46.)
- Create and register a new web service to the web server. This step includes assigning an instance name to the service, and associating the service with the function you defined in the previous step. (For information, see "Registering Your Web Application Service" on page 52.)

After you write and compile your application, see the section "Running Your Web Service" on page 55 for instructions on setting up and running your web service.

For a summary of the C functions available in WAI, see the section "Summary of C Functions" on page 55

Before continuing on, note the following points:

- You must include the ONEiiop.h header file when writing a WAI application in C:

  ```
  #include "ONEiiop.h"
  ```

  This header file declares the C functions available in WAI.

- The web server includes a sample C application that demonstrates how you can use WAI to write a web application service. The example is located in the *server_root*/wai/examples/CIIOP directory on UNIX and the *server_root*\wai\examples\CIIOP directory on Windows NT.

  You can follow this example as a guideline for writing and compiling your application.

The rest of this chapter explains how to write a WAI application in C.

Defining a Function to Process Requests

The function that processes incoming HTTP requests (not all requests, just the requests directed specifically at your service) must comply with the following type definition:

```
typedef long (*WAIRunFunction)(ServerSession_t obj);
```

obj represents the HTTP request to be processed. You pass this argument to other WAI functions in order to get data from the client request, set data in the response, and send the response to the client.

The rest of this section explains how you can call WAI functions to process the request. WAI functions enable you to do the following tasks:

- Getting Data from the Request
- Sending the Response Back to the Client

Getting Data from the Request

WAI provides functions for getting data from the client's HTTP request. You can call functions to accomplish the following tasks:

- Getting Headers from the HTTP Request
- Getting Information about the Server

Getting Headers from the HTTP Request

To get headers from the HTTP request, call the WAIgetRequestHeader() function. For example, the following section of code gets and prints the user-agent header from the incoming request:

```
long MyRunFunction(ServerSession_t obj)
{
    char *var = 0;
    ...
    if (WAIgetRequestHeader(obj, "user-agent", var) == WAISPISuccess){
        printf( "User agent: %s\n", var);
    }
    ...
}
```

In addition to HTTP headers, you can get other types of information (such as CGI 1.1 environment variables) from the HTTP request by calling the WAIgetRequestInfo() function.

The section "getRequestInfo" on page 122 lists the types of information you can retrieve from the request. Note that the CGI 1.1 environment variables that describe the server are accessible through the WAIgetInfo() function. See "Getting Information about the Server" on page 48 for details.

The following section of code gets and prints the value of the REMOTE_ADDR CGI 1.1 environment variable for the incoming request:

```
long MyRunFunction(ServerSession_t obj)
{
    char *var = 0;
    ...
    if (WAIgetRequestInfo(obj, "REMOTE_ADDR", var) == WAISPISuccess){
        printf( "Client IP Address: %s\n", var);
    }
    ...
}
```

Getting Information about the Server

WAI also provides C functions for getting information about the server, such as the server identifier or CGI 1.1 environment variables that describe the server (for example, SERVER_NAME or SERVER_PORT).

To get these types of information, you can call the WAIgetInfo() function and specify the type of information that you want to retrieve. For example, the following section of code gets the value of the SERVER_PORT CGI 1.1 environment variable:

```
long MyRunFunction(ServerSession_t obj)
{
    int port_num;
    ...
    if (WAIgetInfo(obj, "SERVER_PORT", port_num) == WAISPISuccess){
        printf( "Server Port: %d\n", port_num);
    }
    ...
}
```

For a list of the types of information you can retrieve from this method, see the section "getInfo" on page 146.

You can also call functions that specifically retrieve a certain type of information. For example, to get the port number that the server listens to, you can call the WAIgetPort() function:

```
long MyRunFunction(ServerSession_t obj)
{
    int port_num = 0;
    ...
    if ((port_num = WAIgetPort(obj)) != 0){
        printf( "Server Port: %d\n", port_num);
    }
    ...
}
```

For details on getting server information, see the section "netscape::WAI::HttpServerContext" on page 144.

Getting and Setting Cookies in the Client

Before a client accesses a URL, the client checks the domain name in the URL against the cookies that it has. If any cookies are from the same domain as the URL, the client includes a header in the HTTP request that contains the name/value pairs from the matching cookies.

The Cookie header has the following format:

Cookie: *name=value*; [*name1=value1*; *name2=value2* ...]

To get these name/value pairs from the HTTP request, call the WAIgetCookie() function. To set your own name/value pairs in a client, call the WAIsetCookie() function.

The following example illustrates how you can use these functions to get and set cookies in the client.

```
long MyRunFunction(ServerSession_t obj)
{
    ...
    char *cookiebuff = NULL;
    /* If no cookie has been set in the client, set a cookie. */
    if (WAIgetCookie(obj, cookiebuff)== WAISPIFailure)
        WAIsetCookie(obj, "A_NAME", "A Value", "", "", "/", WAI_FALSE);
    ...
}
```

Sending the Response Back to the Client

WAI functions also allow you to control the response sent back to the client. You can call these functions to accomplish the following tasks:

- Setting Headers in the Response
- Setting the Status of the Response
- Sending the Response
- Redirecting Users to Another Page

Setting Headers in the Response

WAI includes functions that you can use to set headers in the response that you want sent back to the client. You can call the WAIaddResponseHeader() function to set any header in the response. For example, the following section of code adds the Pragma header to the response:

```
...
WAIaddResponseHeader(obj, "Pragma", "no-cache");
...
```

You can also call functions that set specific types of headers. For example, you can call:

- WAIsetResponseContentType() to specify the content type of the response (the Content-type header)
- WAIsetResponseContentLength() to specify the length of the response in bytes (the Content-length header)

Setting the Status of the Response

To set the status of the response sent back to the client, call the WAIsetResponseStatus() function. For example, the following section of code sets the response status to a 404 status code ("File Not Found"):

```
...
WAIsetResponseStatus(obj, 404, "");
...
```

Sending the Response

After you have set up the response you want sent back to the client, you can start sending the response to the client. Call the WAIStartResponse() function to start sending the response.

To send the rest of the data to the client, call the WAIWriteData() function.

The following example sends the string Hello World back to the client:

```
long MyRunFunction(ServerSession_t obj)
{
    /* Specify the string that you want to send back to the client. */
```

```
    char *buffer = "Hello World\n";
    size_t bufflen = strlen(buffer);

    /* Specify the length of the data that you are about to send back. */
    WAIsetResponseContentLength(obj, bufflen);


    /* Start sending the response back to the client. */
    WAIStartResponse(obj);

    /* Write the string to the client. */
    WAIWriteClient(obj, (const unsigned char *)buffer, bufflen);
    return 0;
}
```

Redirecting Users to Another Page

In your WAI application, you can also redirect users to a different page than the requested page. You can either automatically redirect the user to a new page, or you can present the user with a link to click manually.

To automatically redirect the user to a different page, you can do the following:

1. **Call the WAIaddResponseHeader() function to add a Location header.**

   The Location header points to the new location.

2. **Call the WAIsetResponseStatus() function to set the response status.**

   Set the response status to 301 if the page has permanently moved or 302 if the page has temporarily moved.

3. **Call the WAIStartResponse() function to send the response back to the client.**

For example:

```
long
MyRunFunction(ServerSession_t obj)
{
```

```
    WAIaddResponseHeader(obj, "Location", "http://www.newsite.com/");
    WAIsetResponseStatus(obj, 302, "Moved temporarily to newsite.com");
    WAIStartResponse(obj);
    return 0;
}
```

To give the user the choice of going to the new location (rather than automatically redirecting the URL), you can call the WAIRespondRedirect() function:

```
long
MyRunFunction(ServerSession_t obj)
{
    WAIRespondRedirect(obj, "http://www.newsite.com/");
    WAIStartResponse(obj);
    return 0;
}
```

Calling this method will send the following page back to the client:

```
Moved Temporarily
This document has moved to a new location. Please update your documents and hotlists accordingly.
```

The word "location" on this page is a link pointing to the new location of the page.

Registering Your Web Application Service

After you define the function for processing HTTP reequests, you need to create and register your web service. You need to register your web service to the web server under an instance name. The instance name that you select for your web service can be an arbitrary name; it does not need to be the same name as your application. (For example, if your application is named MyApp or MyApp.exe, your instance name can be MyWebService. They do not need to have the same name.)

Note, however, that your instance name must be unique. No other registered WAI application can have the same name.

Registering With a Web Server

To create and register your web application service, follow these steps:

1. **Call the WAIcreateWebAppService() function to create the web service.**

   Specify the name of the service and the name of your function (that you defined in "Defining a Function to Process Requests" on page 46) as arguments.

   The instance name that you select for your web service can be an arbitrary name. It does not need to be the same name as your application.

   WAIcreateWebAppService() returns a pointer to an IIOPWebAppService structure, which represents the newly created web service.

2. **Call the WAIregisterService() function to register the service.**

   Pass the pointer to the IIOPWebAppService structure to this function. You also need to specify the hostname and port number of the web server in the form *hostname:portnumber*.

   Note that if your web server is running with SSL enabled, you need to specify a different value for this argument. For details, see "Registering With an SSL-Enabled Server" on page 54.

3. **Call the WAIimplIsReady() function to indicate that your service is prepared to receive incoming requests.**

Note that the WAIimplIsReady() function puts the application into an endless loop. Any statements that you insert after this function are not executed. So, for example, if you want to add a printf statement to indicate whether or not the application has registered successfully, add the statement before calling the WAIimplIsReady() function.

For example, the following section of code creates and registers a new web service with the instance name CAPIIIOP. Whenever this web service is accessed, the web server sends the HTTP request to the function named MyRunFunction.

```
...
IIOPWebAppService_t obj;
WAIReturnType_t rv;
...
/* Create the web service. */
```

```
obj = WAIcreateWebAppService("CAPIIIOP", MyRunFunction);

/* Register the web service. */
rv = WAIregisterService(obj, "myhost.netscape.com:81");
if (rv == WAI_FALSE) {
    printf("Failed to Register with %s\n", host);
    return 1;
} else {
    printf("Registered successfully with %s\n", host);
}

/* Indicate that the service is ready to receive requests. */
WAIimplIsReady();
return 0;
...
```

Registering With an SSL-Enabled Server

Typically, when you call the WAIregisterService function to register your web service, you pass the host name and port number of your web server as an argument.

The function constructs a URL to the web server's built-in naming service and gets the object reference for this naming service. This object reference is used to register your application.

If your web server has SSL enabled, the WAIregisterService function cannot get the naming service object reference in the manner described above. Instead, it needs to use the Interoperable Object Reference (IOR) file to get the object reference for the naming service.

To find the IOR file, the WAIregisterService function assembles a path to the file using the following information:

- The server root (for example, the default server root is /usr/netscape/suitespot or C:\netscape\suitespot)
- The server identifier (for example, the default server identifier is https-*hostname*)

If your web server does not use the default values for either of these, you must set environment variables to identify the correct values before running your WAI application:

- If your server is installed under a different directory than the default server root, you must set the NS_SERVER_ROOT environment variable to the location of your server root.

  For example, suppose that your server is installed under /export/netscape/suitespot. In a C shell, you need to set the following environment variable before running your WAI application:

  ```
  setenv NS_SERVER_ROOT /export/netscape/suitespot
  ```

- If you are not using the default server identifier, you must set the NS_SERVER_ID environment variable to the server identifier that you are using.

  For example, suppose that your server is running on the machine preston and your server identifier is https-webserver instead of https-preston. In C shell, you need to set the following environment variable before running your WAI application:

  ```
  setenv NS_SERVER_ID https-webserver
  ```

Running Your Web Service

After you write and compile your application, you can run your application to make your web service available. The web server should recognize your application, if you've registered it (see "Registering Your Web Application Service" on page 52).

End users can access your service by going to the URL:

http://*server_name*:*port_number*/iiop/*instance_name*

For example, you can access the CAPIIIOP example by going to the URL:

http://*server_name*:*port_number*/iiop/CAPIIIOP

Summary of C Functions

The following table summarizes the C functions available in WAI.

Table 4.1 C Functions in WAI

Function Name	Description	For More Information, See...
WAIaddResponseHeader()	Adds a header to the HTTP response to be sent back to the client.	"addResponseHeader" on page 111
WAIBuildURL()	Builds an absolute URL from the URI prefix and the URI suffix.	"BuildURL" on page 113
WAIcreateWebAppService()	Creates a new web application service, assigns it an instance name, and associates it with a function for processing HTTP requests.	"WAIWebApplicationService" on page 151
WAIdeleteService()	Deletes a web application service.	
WAIdelResponseHeader()	Removes a header from the HTTP response to be sent back to the client.	"delResponseHeader" on page 115
WAIgetConfigParameter()	Gets the value of a parameter of the **iiopexec** function in the **Service** directive of the **obj.conf** file.	"getConfigParameter" on page 116
WAIgetCookie()	Retrieves any cookies sent by the client.	"getCookie" on page 119
WAIgetHost()	Gets the hostname of the machine where the web server is running.	"getHost" on page 145
WAIgetInfo()	Retrieves information about the web server (such as the value of CGI 1.1 environment variables that describe the server).	"getInfo" on page 146
WAIgetName()	Gets the server ID (for example, https-myhost) of the web server.	"getName" on page 147
WAIgetPort()	Gets the port number that the web server listens to.	"getPort" on page 148
WAIgetRequestHeader()	Gets a header from the HTTP request sent by the client.	"getRequestHeader" on page 121
WAIgetRequestInfo()	Gets information about the client request (such as the value of a CGI 1.1 environment variable).	"getRequestInfo" on page 122
WAIgetResponseContentLength()	Gets the content length (the value of the **Content-length** header) of the response.	"getResponseContentLength" on page 125

Table 4.1 C Functions in WAI

Function Name	Description	For More Information, See...
WAIgetResponseHeader()	Gets a header from the HTTP response you plan to send to the client.	"getResponseHeader" on page 126
WAIgetServerSoftware()	Gets the type and version of the server software.	"getServerSoftware" on page 148
WAIimplIsReady()	Prepares your WAI application to receive requests.	"Registering Your Web Application Service" on page 52
WAIisSecure()	Specifies whether or not the server is run with SSL enabled.	"isSecure" on page 149
WAILogError()	Logs an entry to the server's error log file (*server_root*/*server_id*/logs/errors on UNIX and *server_root**server_id*\logs\errors on Windows NT).	"LogError" on page 128
WAIReadClient()	Reads data from the client (for example, for data sent through the HTTP POST method).	"ReadClient" on page 130
WAIregisterService()	Registers the WAI application with the web server.	"RegisterService" on page 153
WAIRespondRedirect()	Redirects the client to a different URL.	"RespondRedirect" on page 134
(*WAIRunFunction)()	Type definition for the function that processes HTTP requests.	"Run" on page 153
WAIsetCookie()	Sets a cookie in the response header to be sent to the client.	"setCookie" on page 135
WAIsetRequestInfo()	(This method has no functional use at this time.)	"setRequestInfo" on page 138
WAIsetResponseContentLength()	Sets the content length (the value of the Content-length header) of the response to be sent to the client.	"setResponseContentLength" on page 138
WAIsetResponseContentType()	Sets the content type (the value of the Content-type header) of the response to be sent to the client.	"setResponseContentType" on page 139

Table 4.1 C Functions in WAI

Function Name	Description	For More Information, See...
WAIsetResponseStatus()	Sets the HTTP response code (for example, 404 for "File Not Found") of the response to be sent to the client.	“setResponseStatus” on page 140
WAIStartResponse()	Starts sending the response back to the client.	“StartResponse” on page 141
WAIstringFree()	Frees a string from memory.	“StringDelete” on page 154
WAIWriteClient()	Writes data to the client.	“WriteClient” on page 142

Chapter

5

Writing a WAI Application in C++

WAI provides a set C++ classes and methods that you can use to write a WAI application. Your C++ application should:

- Declare a class that derives from the Netscape WAIWebApplicationService base class. See "Declaring a Class for Your Web Service" on page 63
- Define a Run method for processing the incoming HTTP request. See "Defining a Method to Process Requests" on page 64.
- Define a getServiceInfo method for returning information about the service and its version.
- Create an instance of your class and register your service to the web server's host machine. (For instructions, see "Registering Your Web Application Service" on page 73.)

After you write and compile your application, see the section "Running Your Web Service" on page 75 for instructions on setting up and running your web service.

Before continuing on, note the following points:

- You must include the ONESrvPI.hpp header file when writing a WAI application in C++:

```
#include "ONESrvPI.hpp"
```

This header file declares the C++ classes available in WAI.

- The web server includes a sample C++ application that demonstrates how you can use WAI to write a web application service. The example is located in the *server_root*/wai/examples/WASP directory on UNIX and the *server_root*\wai\examples\WASP directory on Windows NT.

  You can follow this example as a guideline for writing and compiling your application.

- If you are using Visual C++ you need follow the instructions in Setting up Microsoft Visual C++ for use with WAI (Windows NT only) to set up your Visual C++ environment specifically for WAI.

The rest of this chapter explains how to write a WAI application in C++.

Setting up Microsoft Visual C++ for use with WAI (Windows NT only)

Follow these steps when setting up your C++ project in Microsoft Visual C++. These steps are specific to Microsoft Visual C++ version 5.0.

1. **Specify the type of application you want to write.**

   Choose New from the File menu. Click the Projects tab and select the type of application you want to write from this list:

   - Console application
   - Windows application
   - DLL application

2. **Fill in the Project Name field.**

   Type the name of the project in the Project Name field and click OK.

3. **Add the project files.**

From the Project menu, choose Add to Project and then choose Files. Use the file browser to add the files you want to include in your project.

4. **Specify that the code be generated using the multi-threaded dll run-time library.**

   From the Project menu, select Settings. Click the C/C++ tab and choose Code Generation from the pull-down menu next to the Category option (see Figure 5.1).

   Choose Multithreaded DLL from pull-down menu next to the "Use run-time library" option.

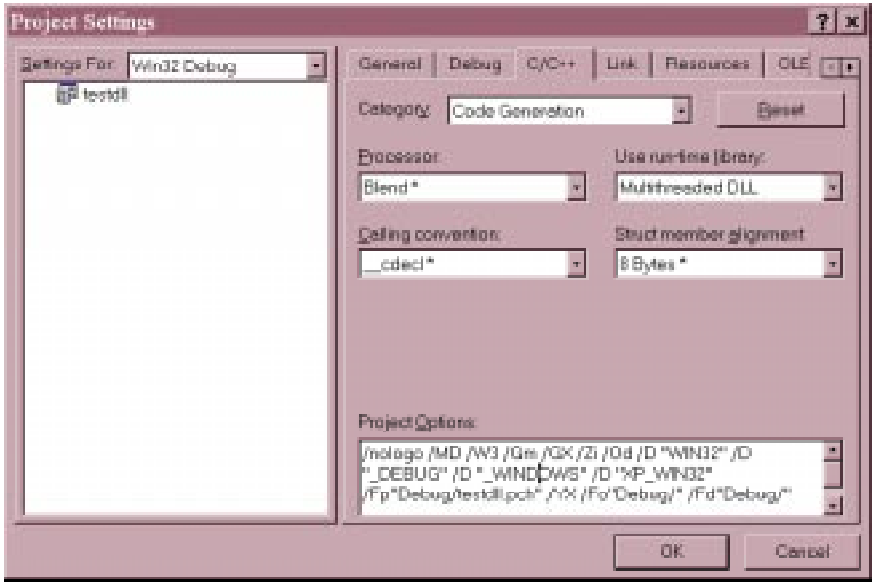


Figure 5.1 Project Settings, C/C++ Code Generation

5. **Specify XP_WIN32 as the macro definition.**

Click Settings from the Project menu. Click the C/C++ tab and choose Preprocessor from the Category option menu (see Figure 5.2).

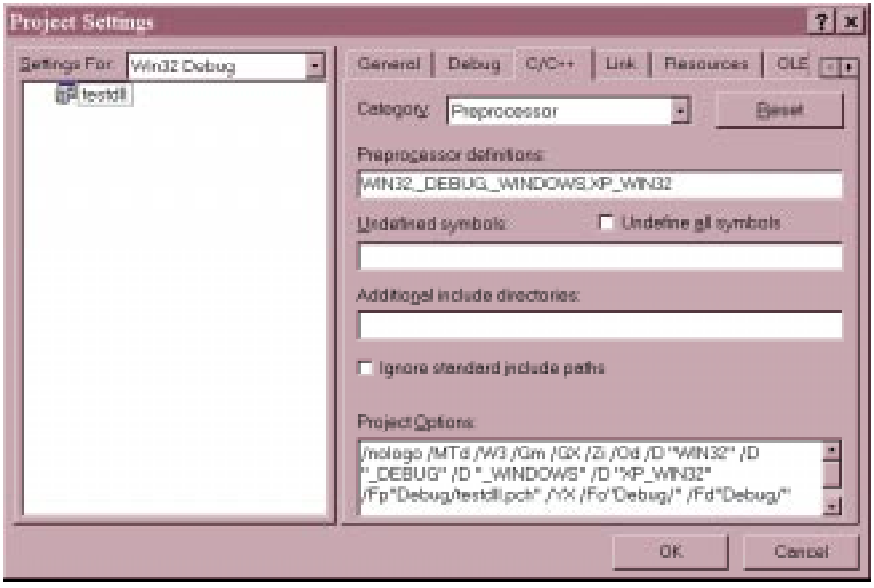


Figure 5.2 Project Settings, C/C++ Preprocessor

Add XP_WIN32 to the Preprocessor Definitions field.

6. **In the field labeled "Additional include directories," type the names of any additional include directories.**

   Add the include file directories (../../include,..\..\include)

   Alternatively, you can add the include file directories by choosing Options from the Tools menu and clicking the Directories tab. Choose "Include files" from the "Show directories for" field, then add the include directories to the list.

7. **Add any additional libraries to list of libraries.**

Choose Settings from the Project menu. Click the Link tab in the Project Settings dialogbox. Choose General from the pull-down menu next to the Category option. In the "Object/library modules" field, type the names of additional libraries. See Figure 5.3.

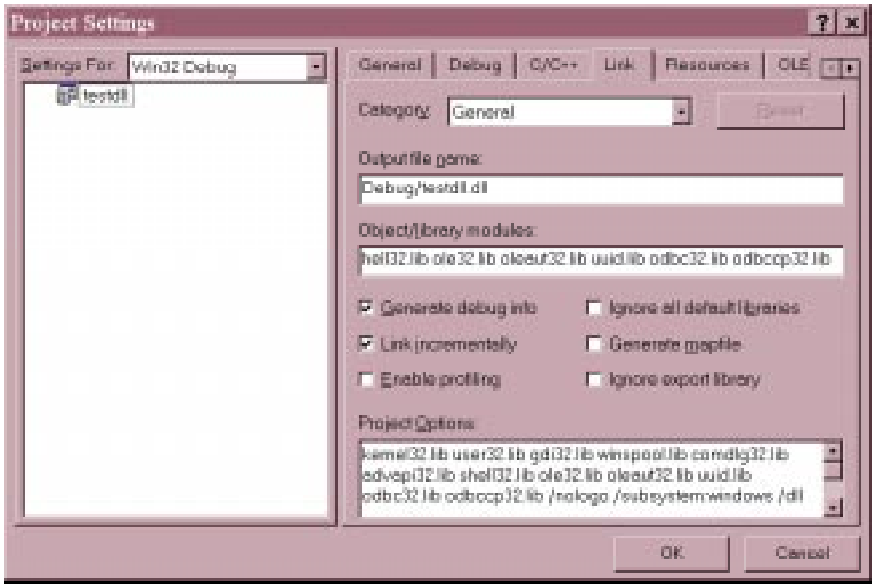


Figure 5.3 Project Settings Dialog, Link Options

If you are using the Visual C++ Debug, do not use the ALLOC and FREE executables. They conflict with the WAI API functions and can cause unpredictable results.

Declaring a Class for Your Web Service

The first step in developing a WAI application in C++ is to declare a class that derives from the Netscape WAIWebApplicationService base class. (This class represents a web application service.)

For example, the WASP example provided with the web server declares a WebApplicationServicePrototype class, which is derived from the WAIWebApplicationService base class:

```
//
// Declare a WAS class deriving from Netscape base class
```

```
//
class WebApplicationServicePrototype: public WAIWebApplicationService
{
public:
    WebApplicationServicePrototype(const char *object_name = (const char *)NULL)
;
    long Run(WAIServerRequest_ptr session);
    char *getServiceInfo();
};
WebApplicationServicePrototype::WebApplicationServicePrototype(const char
*object_name):WAIWebApplicationService(object_name)
{
}
...
```

The class that you define represents your web service. You need to define the following methods for your class; these methods are virtual methods in the WAIWebApplicationService base class:

- Run

  This method is called by the web server to process HTTP requests for this service. For details on defining this method, see "Defining a Method to Process Requests" on page 64.

- getServiceInfo

  This method returns information about your web service (such as version information). For details on defining this method, see "Providing Information About the Service" on page 72.

Defining a Method to Process Requests

The method that processes incoming HTTP requests (not all requests, just the requests directed specifically at your service) should use the following syntax:

```
long Run(WAIServerRequest_ptr session);
```

session represents the HTTP request to be processed. You can call the methods of this object to get data from the request, set data in the response headers, and send the response back to the client.

The rest of this section explains how you can use these methods and objects to process the request. WAI functions enable you to do the following tasks:

- Getting Data from the Request
- Sending the Response Back to the Client

Getting Data from the Request

Using an object of the WAIServerRequest class (see the section "netscape::WAI::HttpServerRequest" on page 110 for details), you can get data from the client's HTTP request. You can call functions accomplish the following tasks:

- Getting Headers from the HTTP Request
- Getting Information about the Server

Getting Headers from the HTTP Request

Given an object of the WAIServerRequest class, you can get headers from the corresponding HTTP request by calling the getRequestHeader method. For example, the following section of code gets the user-agent HTTP request header from the incoming request:

```
long
WebApplicationServicePrototype::Run(WAIServerRequest_ptr session)
{
    char *var = 0;
    ostrstream outstr;
    ...
    if (session->getRequestHeader("user-agent", var) == WAISPISuccess){
        outstr << "User Agent: " << var;
        StringDelete(var);
    }
```

```
    outstr << endl;
    ...
}
```

In addition to HTTP headers, you can get other types of information (such as CGI 1.1 environment variables) from the HTTP request by calling the getRequestInfo method of the WAIServerRequest class.

The section "getRequestInfo" on page 122 lists the types of information you can retrieve from the request. Note that the CGI 1.1 environment variables that describe the server are accessible through the getInfo method. See "Getting Information about the Server" on page 66 for details.

The following section of code gets and prints the value of the REMOTE_ADDR CGI 1.1 environment variable for the incoming request:

```
long
WebApplicationServicePrototype::Run(WAIServerRequest_ptr session)
{
    char *var = 0;
    ostrstream outstr;
    ...
    if (session->getRequestInfo("REMOTE_ADDR", var) == WAISPISuccess){
        outstr << "Client IP Address: " << var;
        StringDelete(var);
    }
    outstr << endl;
    ...
}
```

Getting Information about the Server

WAI also provides methods for getting information about the server, such as the server identifier or CGI 1.1 environment variables that describe the server (for example, SERVER_NAME or SERVER_PORT).

These methods are available as part of the WAIServerContext class (for more information, see the section "netscape::WAI::HttpServerContext" on page 144). You can get an object of this class by using the getContext method of the WAIServerRequest class.

For example, the following section of code gets an WAIServerContext object:

```
long
WebApplicationServicePrototype::Run(WAIServerRequest_ptr session)
{
    ...
    WAIServerContext_ptr context = session->getContext();
    ...
}
```

To get information about the server, you can call the getInfo method of the WAIServerContext object and specify the type of information that you want to retrieve. For example, the following section of code gets the value of the SERVER_PORT CGI 1.1 environment variable:

```
long
WebApplicationServicePrototype::Run(WAIServerRequest_ptr session)
{
    int port_num;
    ostrstream outstr;
    WAIServerContext_ptr context = session->getContext();
    ...
    if (context->getInfo("SERVER_PORT", port_num) == WAISPISuccess){
        outstr << "Port Number: " << var;
        StringDelete(var);
    }
    outstr << endl;
    }
    ...
}
```

For a list of the types of information you can retrieve from this method, see the section "getInfo" on page 146.

You can also use methods that specifically retrieve a certain type of information. For example, to get the port number that the server listens to, you can call the getPort method:

```
long
WebApplicationServicePrototype::Run(WAIServerRequest_ptr session)
{
    int port_num = 0;
    ostrstream outstr;
    WAIServerContext_ptr context = session->getContext();
    ...
    if ((port_num = context->getPort()) != 0){
        outstr << "Port Number: " << var;
        StringDelete(var);
    }
    outstr << endl;
    }
    ...
}
```

For details on getting server information, see the section "netscape::WAI::HttpServerContext" on page 144.

Getting and Setting Cookies in the Client

Before a client accesses a URL, the client checks the domain name in the URL against the cookies that it has. If any cookies are from the same domain as the URL, the client includes a header in the HTTP request that contains the name/value pairs from the matching cookies.

The Cookie header has the following format:

Cookie: *name=value*; [*name1=value1*; *name2=value2* ...]

To get these name/value pairs from the HTTP request, call the getCookie method. To set your own name/value pairs in a client, call the setCookie method.

Thefollowingexampleillustrateshowyoucanusethesemethodstogetandsetcookies in the client.

```
long
WebApplicationServicePrototype::Run(WAIServerRequest_ptr session)
{
    ...
    char *cookiebuff = NULL;
    /* If no cookie has been set in the client, set a cookie. */
    if (session->getCookie(cookiebuff)== WAISPIFailure)
        session->setCookie("MY_NAME", "My Value", "", "", "/", WAI_FALSE);
    ...
}
```

Sending the Response Back to the Client

Methods of the HttpServerRequest class also allow you to control the response sent back to the client. You can call these functions to accomplish the following tasks:

- Setting Headers in the Response
- Setting the Status of the Response
- Sending the Response
- Redirecting Users to Another Page

Setting Headers in the Response

WAI includes functions that you can use to set headers in the response that you want sent back to the client. You can call the addResponseHeader method to set any header in the response. For example, the following section of code adds the Pragma header to the response:

```
long
```

```
WebApplicationServicePrototype::Run(WAIServerRequest_ptr session)
{
    ...
    session->addResponseHeader("Pragma", "no-cache");
    ...
}
```

You can also call functions that set specific types of headers. For example, you can call:

- setResponseContentType to specify the content type of the response (the Content-type header)
- setResponseContentLength to specify the length of the response in bytes (the Content-length header)

Setting the Status of the Response

To set the status of the response sent back to the client, call the setResponseStatus method. For example, the following section of code sets the response code to a 404 status code ("File Not Found"):

```
long
WebApplicationServicePrototype::Run(WAIServerRequest_ptr session)
{
    ...
    session->setResponseStatus(404, "");
    ...
}
```

Sending the Response

After you have specified the length of the content you want sent back to the client, you can start sending the response to the client. Call the StartResponse method to start sending the response.

To send the rest of the data to the client, call the WriteClient method.

The following example sends the string Hello World back to the client:

```
long
```

```
WebApplicationServicePrototype::Run(WAIServerRequest_ptr session)
{
    ...
    /* Specify the string that you want to send back to the client. */
    char *buffer = "Hello World\n";
    size_t bufflen = strlen(buffer);

    /* Specify the length of the data that you are about to send back. */
    session->setResponseContentLength(bufflen);

    /* Start sending the response back to the client. */
    session->StartResponse();

    /* Write the string to the client. */
    session->WriteClient((const unsigned char *)buffer, bufflen);
    ...
}
```

Redirecting Users to Another Page

In your WAI application, you can also redirect users to a different page than the requested page. You can either automatically redirect the user to a new page, or you can present the user with a link to click on manually.

To automatically redirect the user to a different page, you can do the following:

1. **Call the** addResponseHeader **method to add a** Location **header, which points to the new location.**
2. **Call the** setResponseStatus **method to set the response status to 301 (if the page has permanently moved) or 302 (if the page has temporarily moved).**
3. **Call the** StartResponse **method to send the response back to the client.**

For example:

```
long
```

```
WebApplicationServicePrototype::Run(WAIServerRequest_ptr session)
{
    session->addResponseHeader("Location", "http://www.newsite.com/");
    session->setResponseStatus(301, "Moved permanently to newsite!");
    session->StartResponse();
    return 0;
}
```

To give the user the choice of going to the new location (rather than automatically redirecting the URL), you can call the RespondRedirect method:

```
long
WebApplicationServicePrototype::Run(WAIServerRequest_ptr session)
{
    session->RespondRedirect("http://www.newsite.com/");
    session->StartResponse();
    return 0;
}
```

Calling this method will send the following page back to the client:

```
Moved Temporarily
This document has moved to a new location. Please update your documents and hotlists accordingly.
```

The word "location" on this page is a link pointing to the new location of the page.

Providing Information About the Service

Part of the WAIWebApplicationService base class is the virtual getServiceInfo method. When you write your web application class (which is derived from the base class), you need to include a definition of this method.

The getServiceInfo method should provide information about the web service, such as the name of the author, the version of the service, and so on.

The following section of code defines the getServiceInfo method for a web service class WebApplicationServicePrototype. The example uses the StringDup method to allocate memory for the returned string.

```
...
char *
WebApplicationServicePrototype::getServiceInfo(void)
{
    return StringDup("My Test Web Service. Version 1.0\nCopyright Netscape Communications
Corporation\nAuthor: Mozilla\n");
}
...
```

Registering Your Web Application Service

Next, you need to create an instance of your class and assign an instance name to the object. You need to register your web service to the web server under this instance name. The instance name that you select for your web service can be an arbitrary name; it does not need to be the same name as your application. (For example, if your application is named MyApp or MyApp.exe, your instance name can be MyWebService. They do not need to have the same name.)

Note, however, that your instance name must be unique. No other registered WAI application can have the same name.

Registering With a Web Server

To register your application with the web server's built-in name service, call the RegisterService method. Pass the name of the web server's hostname and port number as an argument (in the form *hostname:portnumber*) to this method.

Note that if your web server is running with SSL enabled, you need to specify a different value for this argument. For details, see "Registering With a Web Server" on page 73.

The following section of code creates the web service ExeFoo from the web service class WebApplicationServicePrototype. The example registers this object to the web server under the instance name MyService.

```
...
WAIBool rv;
```

```
char *host = "myhost.mydomain.com:81";
char *instanceName = "MyService";
...
/* Create the web service. */
WebApplicationServicePrototype ExeFoo(instanceName);

/* Register the web service. */
rv = ExeFoo.RegisterService(host);

/* Provide feedback on the result of the registration attempt. */
if (rv == WAI_FALSE) {
    printf("Failed to register with %s\n", host);
} else {
    printf("Successfully registered with %s\n", host);
    }
...
```

Registering With an SSL-Enabled Server

Typically, when you call the RegisterService or the WAIregisterService function to register your web service, you pass the host name and port number of your web server as an argument.

The function constructs a URL to the web server's built-in naming service and gets the object reference for this naming service. This object reference is used to register your application.

If your web server has SSL enabled, the RegisterService or WAIregisterService function cannot get the naming service object reference in the manner described above. Instead, it needs to use the Interoperable Object Reference (IOR) file to get the object reference for the naming service.

To find the IOR file, the RegisterService function assembles a path to the file using the following information:

- The server root (for example, the default server root is /usr/netscape/suitespot or C:\netscape\suitespot)
- The server identifier (for example, the default server identifier is https-*hostname*)

If your web server does not use the default values for either of these, you must set environment variables to identify the correct values before running your WAI application:

- If your server is installed under a different directory than the default server root, you must set the NS_SERVER_ROOT environment variable to the location of your server root.

  For example, suppose that your server is installed under /export/netscape/suitespot. In a C shell, you need to set the following environment variable before running your WAI application:

  ```
  setenv NS_SERVER_ROOT /export/netscape/suitespot
  ```

- If you are not using the default server identifier, you must set the NS_SERVER_ID environment variable to the server identifier that you are using.

  For example, suppose that your server is running on the machine preston and your server identifier is https-webserver instead of https-preston. In C shell, you need to set the following environment variable before running your WAI application:

  ```
  setenv NS_SERVER_ID https-webserver
  ```

Running Your Web Service

After you write and compile your application, you can run your application to make your web service available. The web server should recognize your application, if you've registered it (see "Registering Your Web Application Service" on page 73).

End users can access your service by going to the URL:

http://*server_name*:*port_number*/iiop/*instance_name*

For example, you can access the C++ WASP example by going to the URL:

http://*server_name*:*port_number*/iiop/WASP

Chapter

6

Writing a WAI Application in Java

WAI provides a set of Java classes and methods that you can use to write a WAI application. Your Java application should:

- Declare a class that derives from the Netscape WAIWebApplicationService base class.
- Define a Run method for processing the incoming HTTP request. (For details, see "Defining a Method to Process Requests" on page 80.)
- Define a getServiceInfo method for returning information about the service and its version.
- Create an instance of your class and register your service to the web server's host machine. (For instructions, see "Registering Your Web Application Service" on page 89.)

After you write and compile your application, see the section "Running Your Web Service" on page 92 for instructions on setting up and running your web service.

Before continuing on, note the following points:

- You must import the class files under netscape.WAI.*, org.omg.CORBA.*, and org.omg.CosNaming.*:

```
import org.omg.CORBA.*;
import org.omg.CosNaming.*;
```

```
import netscape.WAI.*;
```

- You must include the files nisb.zip and WAI.zip in your CLASSPATH environment variable. These files are located in the *server_root*/wai/java directory in UNIX and in the *server_root*\wai\java directory on Windows NT.

  For example, in C shell on UNIX, enter the following command (if your server is installed under /usr/netscape/suitespot):

```
setenv CLASSPATH "$CLASSPATH":/usr/netscape/suitespot/wai/java/nisb.zip:/usr/netscape/suitespot/
wai/java/WAI.zip
```

  On Windows NT, open the System Control Panel, and add these zip files to your CLASSPATH environment variable listed there.

- The web server includes a sample Java application that demonstrates how you can use WAI to write a web application service. The example is located in the *server_root*/wai/examples/WASP directory on UNIX and the *server_root*\wai\examples\WASP directory on Windows NT.

  You can follow this example as a guideline for writing and compiling your application.

The rest of this chapter explains how to write a WAI application in Java.

Declaring a Class for Your Web Service

The first step in developing a WAI application in Java is to declare a class that derives from the Netscape WAIWebApplicationService base class. (This class represents a web application service.)

For example, the WASP example provided with the web server declares a MyWebApplicationService class, which is derived from the WAIWebApplicationService base class:

```
import java.applet.*;

import java.io.*;

import java.awt.*;

import java.net.*;

import java.util.*;

import java.lang.*;
```

```
/* Make sure to import these classes. */
import org.omg.CORBA.*;
import org.omg.CosNaming.*;
import netscape.WAI.*;
...
/*
* Implementation class for A WAS.
* Extends wrapper class for WAI CORBA object
*/
class MyWebApplicationService extends WAIWebApplicationService {
    String instanceName;
    MyWebApplicationService(java.lang.String name) throws
    org.omg.CosNaming.NamingContextPackage.CannotProceed,
    org.omg.CosNaming.NamingContextPackage.InvalidName,
    org.omg.CosNaming.NamingContextPackage.AlreadyBound,
    org.omg.CORBA.SystemException{
        super(name);
        instanceName = name;
    }
...
```

The class that you define represents your web service. You need to define the following methods for your class; these methods are virtual methods in the WAIWebApplicationService base class:

- Run

  This method is called by the web server to process HTTP requests for this service. For details on defining this method, see "Defining a Method to Process Requests" on page 80.

- getServiceInfo

  This method returns information about your web service (such as version information). For details on defining this method, see "Providing Information About the Service" on page 88.

Defining a Method to Process Requests

The method that processes incoming HTTP requests (not all requests, just the requests directed specifically at your service) should use the following syntax:

```
public int Run(netscape.WAI.HttpServerRequest request);
```

request represents the HTTP request to be processed. You can call the methods of this object to get data from the request, set data in the response headers, and send the response back to the client.

The rest of this section explains how you can use these methods and objects to process the request. WAI functions enable you to do the following tasks:

- Getting Data from the Request
- Sending the Response Back to the Client

Getting Data from the Request

Using an object of the netscape.WAI.HttpServerRequest class (see the section "netscape::WAI::HttpServerRequest" on page 110 for details), you can get data from the client's HTTP request. You can call functions accomplish the following tasks:

- Getting Headers from the HTTP Request
- Getting Information about the Server

Getting Headers from the HTTP Request

Given an object of the netscape.WAI.HttpServerRequest class, you can get headers from the corresponding HTTP request by calling the getRequestHeader method. For example, the following section of code gets the user-agent HTTP request header from the incoming request:

```
public int Run(netscape.WAI.HttpServerRequest request) {
    ...
    /* Prepare an output stream to send data back to the client. */
    ByteArrayOutputStream streamBuf = new ByteArrayOutputStream();
    PrintStream content = new PrintStream(streamBuf);
```

```
    ...

    /* Get the value of the user-agent header. */
    org.omg.CORBA.StringHolder value = new org.omg.CORBA.StringHolder();
    if (request.getRequestHeader("user-agent", value) == HttpServerReturnType.Success){
        content.print("User agent: " + value.value);
    }
    ...
}
```

In addition to HTTP headers, you can get other types of information (such as CGI 1.1 environment variables) from the HTTP request by calling the getRequestInfo method of the netscape.WAI.HttpServerRequest class.

The section "getRequestInfo" on page 122 lists the types of information you can retrieve from the request. Note that the CGI 1.1 environment variables that describe the server are accessible through the getInfo method. See "Getting Information about the Server" on page 82 for details.

The following section of code gets and prints the value of the REMOTE_ADDR CGI 1.1 environment variable from the incoming request:

```
public int Run(netscape.WAI.HttpServerRequest request) {
    ...
    /* Prepare an output stream to send data back to the client. */
    ByteArrayOutputStream streamBuf = new ByteArrayOutputStream();
    PrintStream content = new PrintStream(streamBuf);
    ...
    /* Get the client's IP address. */
    org.omg.CORBA.StringHolder value = new org.omg.CORBA.StringHolder();
    if (request.getRequestInfo("REMOTE_ADDR", value) == HttpServerReturnType.Success){
        content.print("Client addr: " + value.value);
    }
    ...
}
```

Getting Information about the Server

WAI also provides methods for getting information about the server, such as the server identifier or CGI 1.1 environment variables that describe the server (for example, SERVER_NAME or SERVER_PORT).

These methods are available as part of the netscape.WAI.HttpServerContext class (for more information, see the section "netscape::WAI::HttpServerContext" on page 144). You can get an object of this class by using the getContext method of the netscape.WAI.HttpServerRequest class.

For example, the following section of code gets an netscape.WAI.HttpServerContext object:

```
public int Run(netscape.WAI.HttpServerRequest request) {

    ...

    /* Get the HttpServerContext object describing this web server. */

    HttpServerContext context = request.getContext();

    ...

}
```

To get information about the server, you can call the getInfo method of the netscape.WAI.HttpServerContext object and specify the type of information that you want to retrieve. For example, the following section of code gets the value of the SERVER_PORT CGI 1.1 environment variable:

```
public int Run(netscape.WAI.HttpServerRequest request) {

    ...

    /* Prepare an output stream to send data back to the client. */

    ByteArrayOutputStream streamBuf = new ByteArrayOutputStream();

    PrintStream content = new PrintStream(streamBuf);


    /* Get the HttpServerContext object for this web server. */

    HttpServerContext context = request.getContext();

    ...

    /* Get the port number that the web server listens to. */

    org.omg.CORBA.StringHolder svar;

    if (context.getInfo("SERVER_PORT", svar) == HttpReturnType.Success){
```

```
        content.print("Web Server port number: " + svar);
    }
    ...
}
```

For a list of the types of information you can retrieve from this method, see the section “getInfo” on page 146.

You can also use methods that specifically retrieve a certain type of information. For example, to get the port number that the server listens to, you can call the getPort method:

```
public int Run(netscape.WAI.HttpServerRequest request) {
    ...
    /* Prepare an output stream to send data back to the client. */
    ByteArrayOutputStream streamBuf = new ByteArrayOutputStream();
    PrintStream content = new PrintStream(streamBuf);

    /* Get the HttpServerContext object for this web server. */
    HttpServerContext context = request.getContext();
    ...
    /* Get the port number that the web server listens to. */
    int portNum = 0;
    if ((portNum = context.getPort()) != 0){
        content.print("Web Server port number: " + portNum);
    }
    ...
}
```

For details on getting server information, see the section “netscape::WAI::HttpServerContext” on page 144.

Getting and Setting Cookies in the Client

Before a client accesses a URL, the client checks the domain name in the URL against the cookies that it has. If any cookies are from the same domain as the URL, the client includes a header in the HTTP request that contains the name/value pairs from the matching cookies.

The Cookie header has the following format:

Cookie: *name=value*; [*name1=value1*; *name2=value2* ...]

To get these name/value pairs from the HTTP request, call the getCookie method. To set your own name/value pairs in a client, call the setCookie method.

The following example illustrates how you can use these methods to get and set cookies in the client.

```
public int Run(netscape.WAI.HttpServerRequest request)
{
    ...
    org.omg.CORBA.StringHolder
    cookiebuff = new org.omg.CORBA.StringHolder();

    /* If no cookie has been set in the client, set a cookie. */
    if (request.getCookie(cookiebuff)== HttpServerReturnType.Failure)
        request.setCookie("MY_NAME", “My Value”, "", "", "/", false);
    ...
}
```

Sending the Response Back to the Client

Methods of the HttpServerRequest class also allow you to control the response sent back to the client. You can call these functions to accomplish the following tasks:

- Setting Headers in the Response
- Setting the Status of the Response
- Sending the Response

- Redirecting Users to Another Page

Setting Headers in the Response

WAI includes functions that you can use to set headers in the response that you want sent back to the client. You can call the addResponseHeader method to set any header in the response. For example, the following section of code adds the Pragma header to the response:

```
public int Run(netscape.WAI.HttpServerRequest request)
{
    ...
    request.addResponseHeader("Pragma", "no-cache");
    ...
}
```

You can also call functions that set specific types of headers. For example, you can call:

- setResponseContentType to specify the content type of the response (the Content-type header)
- setResponseContentLength to specify the length of the response in bytes (the Content-length header)

Setting the Status of the Response

To set the status of the response sent back to the client, call the setResponseStatus method. For example, the following section of code sets the response code to a 404 status code ("File Not Found"):

```
public int Run(netscape.WAI.HttpServerRequest request)
{
    ...
    request.setResponseStatus(404, "");
    ...
}
```

Sending the Response

After you have specified the length of the content you want sent back to the client, you can start sending the response to the client. Call the StartResponse method to start sending the response.

To send the rest of the data to the client, call the WriteClient method.

The following example sends the string Hello World back to the client:

```
public int Run(netscape.WAI.HttpServerRequest request)
{
    ...
    /* Prepare an output stream to send data back to the client. */
    ByteArrayOutputStream streamBuf = new ByteArrayOutputStream();
    PrintStream content = new PrintStream(streamBuf);
    ...

    /* Send “Hello World” to the print stream. */
    String buffer = "Hello World\n";
    content.print(buffer);

    /* Convert the string to a byte array for WriteClient(). */
    HttpServerReturnType rc;
    byte[] outbuff = streamBuf.toByteArray();
    try {

        /* Specify the length of the data you will send. */
        rc = request.setResponseContentLength(outbuff.length);

        /* Start sending your response. */
        request.StartResponse();
    }
    catch(org.omg.CORBA.SystemException e){
    }
```

```
    /* Write data back to the client. */
    int write_cnt = request.WriteClient(outbuff);
    ...
}
```

Redirecting Users to Another Page

In your WAI application, you can also redirect users to a different page than the requested page. You can either automatically redirect the user to a new page, or you can present the user with a link to click on manually.

To automatically redirect the user to a different page, do the following:

1. **Call the** addResponseHeader **method to add a** Location **header, which points to the new location.**
2. **Call the** setResponseStatus **method to set the response status to 301 (if the page has permanently moved) or 302 (if the page has temporarily moved).**
3. **Call the** StartResponse **method to send the response back to the client.**

For example:

```
public int Run(HttpServerRequest request){
try {
    request.addResponseHeader("Location", "http://www.newsite.com/");
    request.setResponseStatus(301, "Moved permanently to newsite.com!");
    request.StartResponse();
}
catch(org.omg.CORBA.SystemException e){
}
catch(java.lang.Exception e) {
    System.err.println(e);
}
return 0;
}
```

To give the user the choice of going to the new location (rather than automatically redirecting the URL), you can call the RespondRedirect method:

```
public int Run(HttpServerRequest request){
request.RespondRedirect("http://www.newsite.com/");
try {
    request.StartResponse();
}
catch(org.omg.CORBA.SystemException e){
}
catch(java.lang.Exception e) {
    System.err.println(e);
}
return 0;
}
```

Calling this method will send the following page back to the client:

```
Moved Temporarily
This document has moved to a new location. Please update your documents and hotlists accordingly.
```

The word "location" on this page is a link pointing to the new location of the page.

Providing Information About the Service

Part of the WAIWebApplicationService base class is the virtual getServiceInfo method. When you write your web application class (which is derived from the base class), you need to include a definition of this method.

The getServiceInfo method should provide information about the web service, such as the name of the author, the version of the service, and so on.

The following sections of code defines the getServiceInfo method for a web service class WebApplicationServicePrototype.

```
...
public java.lang.String getServiceInfo(){
    return "Java Test Web Application Service V1.0\nCopyright Netscape Communications
```

```
Corporation\nAuthor: Mozilla\n";
}
...
```

Registering Your Web Application Service

1. **Initialize the object request broker (ORB) and the basic object adaptor (BOA):**

   - Call the org.omg.CORBA.ORB.init() method to initialize the ORB. This method returns an ORB object.
   - Call that ORB object's BOA_init() method to initialize the BOA. This method returns a BOA object.

   For example:

   ```
   /* Initialize the object request broker (ORB). */
   ORB orb = org.omg.CORBA.ORB.init();
   /* Initialize the basic object adapter (BOA). */
   BOA boa = orb.BOA_init();
   ```

   For more information on these objects and methods, see the *Netscape Internet Service Broker for Java Reference Guide*.

2. **Create an instance of your class and assign an instance name to the object.**

   You need to register your web service to the web server under this instance name. The instance name that you select for your web service can be an arbitrary name; it does not need to be the same name as your application. (For example, if your application is named MyApp.class, your instance name can be MyWebService. They do not need to have the same name.)

   Note, however, that your instance name must be unique. No other registered WAI application can have the same name.

Registering With a Web Server

To register your application with the web server's built-in name service:

1. **Call the** RegisterService **method.**

   Pass the name of the web server's hostname and port number as an argument (in the form *hostname:portnumber*) to this method.

   Note that if your web server is running with SSL enabled, you need to specify a different value for this argument. For details, see "Registering With an SSL Enabled Server" on page 91.

2. **After you register the service, call the impl_is_ready() method of the BOA object to indicate that your service prepared to receive incoming requests.**

Registering With a Web Server

The following section of code creates the web service mpi from the web service class MyWebApplicationService. The example registers this object to the web server under the instance name MyJavaService.

```
...
String host = "myhost.mydomain.com:81";
String instanceName = "MyJavaService";


try {
    /* Initialize the object request broker (ORB). */
    ORB orb = org.omg.CORBA.ORB.init();


    /* Initialize the basic object adapter (BOA). */
    BOA boa = orb.BOA_init();


    /* Create the web service. */
    try {
        MyWebApplicationService
        mpi = new MyWebApplicationService(instanceName);
        System.out.println(mpi + " is ready.");


        /* Register the web service. */
        mpi.RegisterService(host);
```

```
            /* Wait for incoming requests */
            boa.impl_is_ready();
        }
catch(java.lang.Exception e){
            System.out.println("WAS failed to initialize.");
            System.err.println(e);
}
...
```

Registering With an SSL Enabled Server

If your web server has SSL enabled, you need to use the following format specifying the argument to RegisterService. (In the case of an SSL-enabled server, the method gets the object reference from the Interoperable Object Reference (IOR) file.)

file:*path_to_IOR_file*

This file is located in the wai/NameService directory under your server root directory. The file uses the following naming convention:

server_id.IOR

For example, on the machine named preston, the IOR might be named https-preston.IOR.

Suppose your web server is running on the host machine named feathers on port number 8080. Suppose that the server is installed under the server root directory /usr/netscape/suitespot with the server identifier https-feathers. If SSL is enabled, you register your WAI application in Java by calling:

```
RegisterService("file:/usr/netscape/suitespot/wai/NameService/https-feathers.IOR" );
```

The RegisterService method uses the Interoperable Object Reference (IOR) file to get the object reference for the naming service. This object reference is used to register your application.

Running Your Web Service

After you write and compile your application, you can run your application to make your web service available. The web server should recognize your application, if you've registered it (see "Registering Your Web Application Service" on page 89).

End users can access your service by going to the URL:

http://*server_name*:*port_number*/iiop/*instance_name*

For example, you can access the JavaWASP example by going to the URL:

http://*server_name*:*port_number*/iiop/JavaWASP

Chapter

7

Writing a WAI Server Plug-In

Using WAI, you can write server plug-ins that run within the web server's process (as opposed to standalone applications that run in their own processes). A server plug-in is a shared library or dynamic link library that is loaded and initialized when the server starts up.

Most of the instructions in the previous chapters apply to writing server plug-ins as well as applications. (For details on writing applications with WAI, see Chapter 4, "Writing a WAI Application in C" and Chapter 5, "Writing a WAI Application in C++".)

Typically, when you are writing a standalone application, you register your web application service when your application starts up. If you are writing a server plug-in instead of an application, you need to register your web application service when the server starts up. To do this, you need to:

- Write an initialization function to register your service (see "Writing an Initialization Function" on page 94 for details)
- Configure the web server to run your function during startup (see "Configuring Your Web Server" on page 96 for details)

Writing an Initialization Function

If you are writing a server plug-in, you need to write an initialization function to register your web application service. You can set up this initialization function to get invoked when the web server starts up.

In general, you call the same functions and methods to register a web application service in a server plug-in as you do to register the service in an application. The difference is that you call these functions and methods within an initialization function.

The next section, “Initialization in C” on page 94, explains how to write your initialization functions.

Initialization in C

The initialization function must have the following prototype:

```
myfunc(pblock *pb, Session *sn, Request *rq)
```

In the initialization function, you create a new web application service and register the service. As is the case with standalone applications, you call the WAIcreateWebAppService() function to create the service and WAIregisterService() to register the service. For example:

```
...
// Declare the global variable obj as the web service
IIOPWebAppService_t obj;
...
// Create a new web application service
obj = WAIcreateWebAppService("MyServiceName", MyRunFunction, 0, 0);


// Register the web application service
WAIregisterService(obj, "");
...
```

Unlike standalone applications, you do not need to specify host and port information as arguments to the WAIcreateWebAppService() function. Because your service runs within the web server process, the host and port information is not necessary.

The following example registers a web application service under the instance name CIIOPip. The service is defined in a server plug-in, which provides the initialization function CIIOPinit() for registering the service.

```
...
// Define your Run function
long
MyRunFunction(ServerSession_t obj)
{
...
}
...
// Declare the global variable anObject as a web service instance
IIOPWebAppService_t obj;
...
// Specify the right type for compiling on Windows NT
#if defined(WIN32)
#define DLLEXPORT __declspec(dllexport)
#else
#define DLLEXPORT
#endif
...
// Make the initialization function available
extern "C" {
    DLLEXPORT int CIIOPinit(pblock *pb, Session *sn, Request *rq);
}
...
// Your initialization function (called at server startup)
int
```

```
CIIOPinit(pblock *pb, Session *sn, Request *rq)
{
// Create a new web application service
    obj = WAIcreateWebAppService("CIIOPip", MyRunFunction, 0, 0);

// Register the web application service
    WAIregisterService(obj, "");
    return 0;
}
...
```

Configuring Your Web Server

Next, you need to configure the web server to run your initialization function when the server starts up.

Add the following Init directives to your obj.conf file (which is located under *server_root*/*server_id*/config in UNIX and *server_root**server_id*\config in Windows NT.

```
Init funcs="init_function" fn="load-modules" shlib="shared_lib"
Init fn="init_function"
```

For example, suppose you define an initialization function myinit() in a shared/dynamic library /usr/netscape/suitespot/wai/lib/mylib.so. You need to add the following directives to your obj.conf file:

```
Init funcs="myinit" fn="load-modules" shlib="/usr/netscape/suitespot/wai/lib/mylib.so"
Init fn="myinit"
```

When a WAI plugin needs to be run in-process to the http server, the load-modules and Init directives for this should occur after those corresponding to the load-modules and Init directives libONEiiop.so (or .dll).

Chapter

8

Security Guidelines for Using WAI

Using WAI, you can write and compile an application that runs as its own process (outside the web server's process). When a client accesses your web service, the web server uses a built-in name service to find your application process and execute the Run method (or, in C programs, the corresponding C function of the type WAIRunFunction) in your web service application class.

This section discusses some of the potential security concerns that may arise from the way in which the web server finds your application process. Before you enable WAI on your server, make sure to read this chapter thoroughly.

How the Server Finds Your Application

When you start up your WAI application for the first time, your application registers with the web server's built-in name service. The web server saves the information with the name service.

In order to access your service, end users enter a URL (or click on a link) that contains the name of your service. When this URL is requested, the web server uses its built-in name service to find the registered WAI application with the same name. The server then invokes the Run method in your web application service class.

For example, when you start the WASP example (which is provided with the web server) for the first time, the example registers itself to the web server with the name WASP (for the C++ example) or JavaWASP (for the Java example). End users can access the service through the URL http://*hostname*:*port*/iiop/WASP (or JavaWASP).

By default, the basic object adapter (BOA) in the web server is set to listen only to the local host (the loopback address, 127.0.0.1), not to a network IP address. This configuration assumes that you plan to run your web application services on the same machine as your web server.

Although it is possible to enable the web server's BOA to accept requests from remote machines, you should be aware of the potential security issues surrounding this configuration before choosing to set up your web server in this way. The rest of this chapter explains these potential security concerns.

Potential Security Concerns

When running WAI applications with your web server, the following scenarios could occur:

- **Someone could replace a web service by running another program that registers under the same name.** Potentially, a user could write a program that registers itself under the same name as an existing web service. If the original application that provides the service stops running (for example, if it crashes), another application registered under the same name can take its place.

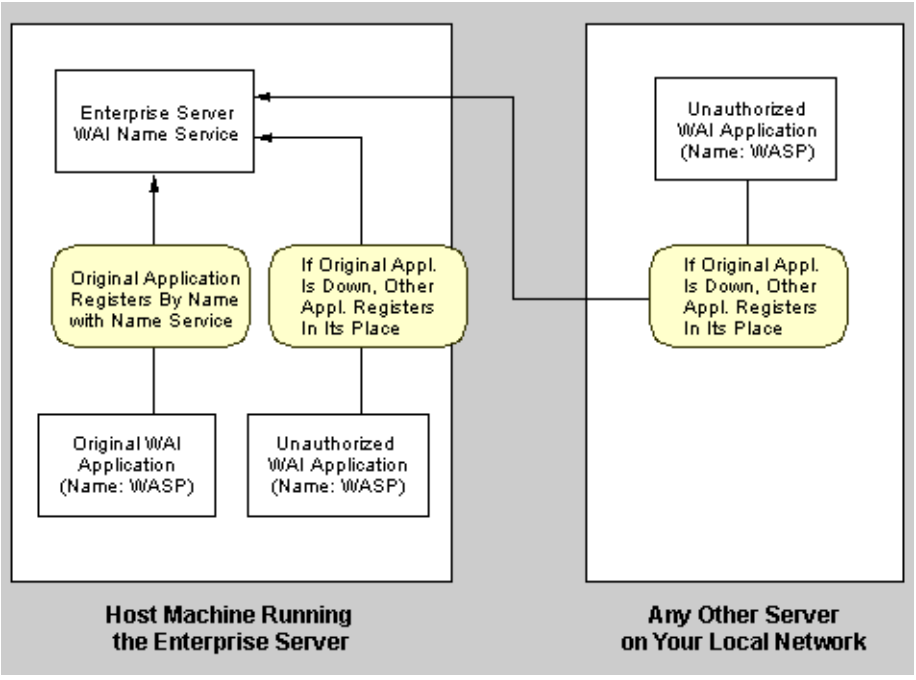
  For example, suppose you are running the WASP example. Someone else could write a program that registered itself under the same name (WASP) and run the program on the web server's host machine. If the original WASP application terminates, the web server's name service will find the other service registered as WASP, and the web server will use that service.

- **Someone could replace a web service or add a new service by uploading a file to the server.** A user with permission to the directory containing your plug-ins or programs could conceivably overwrite those files. For example, if you are running the WASP example, someone else could write a program with the same filename (WASP) and copy that file over your original file.

- **Someone could run a program on a separate machine and register the program with your web server.** If you configure your web server to allow IIOP connections from other machines, programs running on other machines can register with your web server.

  (Note that by default, your web server is configured to listen for IIOP connections from only the local host address 127.0.0.1.)

The following figure illustrates the potential security concerns with enabling the web server to run WAI applications.



Recommended Guidelines

In order to reduce the possibility that security problems might occur, Netscape recommends that you follow these guidelines:

- **Restrict login access to the web server's host machine.** If possible, do not allow guest logins to the machine. Anyone with the ability to execute a program has the potential to register it as a WAI service to your web server.

- **Make sure that write permissions are adequately set on web server's host machine.** Verify that write permissions are restricted to directories and files on the web server. In particular, make sure that server plug-ins loaded by the server or programs started automatically by your machine are write-protected.

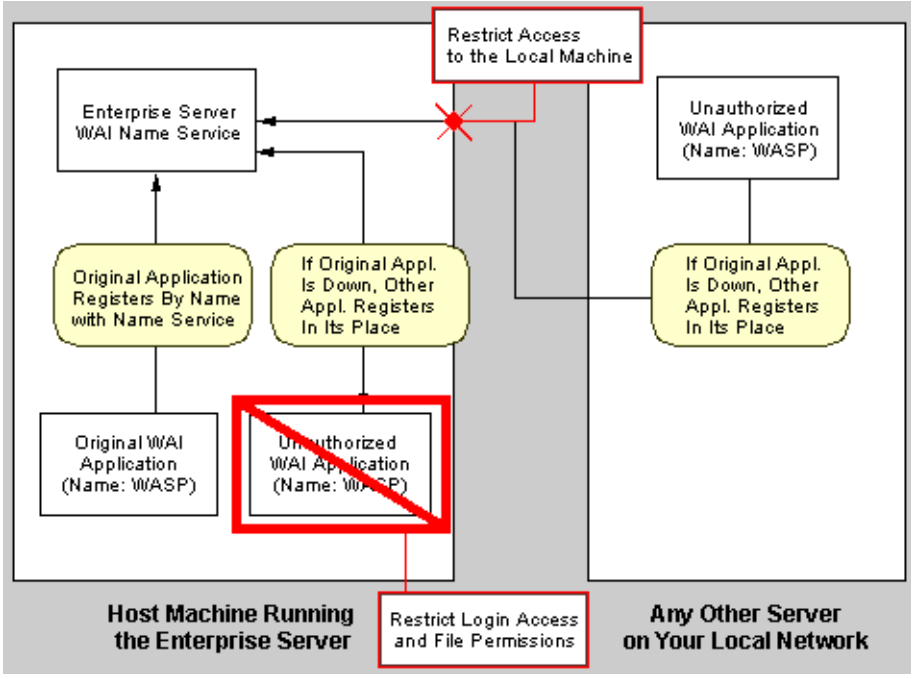
- **Run WAI applications on the local machine only (the machine on which the web server runs).** Although you can set up the web server to access WAI applications running on other machines, configuring the server this way increases the risk of potential security problems. Anyone with the ability to run a program on any machine will have the potential to register the program as a WAI service.

- **(For 3.0 servers only) Restrict osagent so that it only accepts connections from the local host.** Although the web server primarily uses its built-in name service to register WAI applications, osagent can also register WAI applications if the name service is down.

  To configure osagent to accept connections only from the local host machine, specify the -a option with the argument 127.0.0.1 (localhost):

  ```
  osagent -a 127.0.0.1
  ```

The following figure illustrates the recommended guidelines for dealing potential security concerns.



Enabling IIOP Connections from Other Machines

Although Netscape recommends running WAI applications only on the web server's host machine, it is possible to run WAI applications on other machines and have CORBA object implementations on other machines interact with the web servers.

Configuring Your Web Server

To enable the web server to register and find WAI applications running on other machines, you need to configure the web server to use its network IP address instead of the localhost IP address (127.0.0.1).

In the obj.conf file for your server, find the Init directive that calls the IIOPinit function. Use the OAipaddr parameter to specify the IP address that the BOA uses. For example, if you want the BOA set up to use the IP address 204.200.215.98 instead of the local hosr, use the following syntax:

```
Init LateInit="yes" fn="IIOPinit" OAipaddr="204.200.215.98"
```

For more information, see "Configuring the Web Server's ORB" on page 35.

(3.0 only) Running osagent

If you are not restricting the ORB to the local host machine only, you do not need to specify the -a flag when running the osagent utility.

This flag restricts osagent to finding WAI applications on the local host machine only. Without this flag specified, osagent will be able to find applications running on any machine in your local network.

Chapter

9

WAI Reference

This section discusses the signatures of the methods of the three WAI interfaces. According to the CORBA specification, a **signature** describes the legitimate values of request parameters and returned results.

The following table summarizes the signatures, classes, and methods available.

Methods of the HTTPServerRequest Interface

addResponseHeader	Adds a header to the response to be sent back to the client.
BuildURL	Builds a URL from the prefix of a URI and the suffix of a URI.
delResponseHeader	Deletes a header from the response to be sent to the client.
getConfigParameter	Gets the value of a parameter of the iiopexec function in the Service directive of the obj.conf file.
getContext	Gets the HTTPServerContext object for the server.
getCookie	Gets a cookie from the request headers sent by the client.
getRequestHeader	Gets a specified header from the client's request.
getRequestInfo	Gets information about the client request (such as the value of a CGI 1.1 environment variable).
getResponseContentLength	Gets the value of the Content-length header from the response to be sent to the client.
getResponseHeader	Gets the specified header from the response to be sent to the client.

LogError	Logs an entry to the server's error log file (*server_root*/*server_id*/logs/errors on UNIX and *server_root**server_id*\logs\errors on Windows NT).
ReadClient	Reads data from the client (for example, for data sent through the HTTP POST method).
RespondRedirect	Redirects the client to a specified URL.
setCookie	Sets a cookie in the response header to be sent to the client.
setRequestInfo	(This method has no functional use at this time.)
setResponseContentLength	Sets the content length (the value of the Content-length header) of the response to be sent to the client.
setResponseContentType	Sets the content type (the value of the Content-type header) of the response to be sent to the client.
setResponseStatus	Sets the HTTP response code (for example, 404 for "File Not Found") of the response to be sent back to the client.
StartResponse	Starts to send the response to the client.
WriteClient	Writes data to the client.

Methods of the HTTPServerContext Interface

getHost	Retrieves the host name of the machine running the web server.
getInfo	Retrieves information about the web server (such as the value of CGI 1.1 environment variables that describe the server).
getName	Retrieves the server ID (for example, https-myhost).
getPort	Retrieves the port number that the server listens to.
getServerSoftware	Retrieves the product name and version of the web server (for example, Netscape Enterprise/3.0).
isSecure	Specifies whether or not SSL is enabled on the server.

Constructor of the WAIWebApplicationService Base Class

WAIWebApplicationService	Creates an instance of this class.

Methods of the WAIWebApplicationService Base Class

ActivateWAS	Activates the object (if the object has not already been activated by the constructor).
getServiceInfo	(This is a method that you need to implement.) Provides information about the author, version, and copyright of the web application service that you are writing.
RegisterService	Registers your WAI application with the web server running on the specified host.
Run	(This is a method that you need to implement.) Executes your web application service (this is called whenever the server receives an HTTP request for your service).
StringAlloc	Allocates memory for a string.
StringDelete	Frees a string from memory.
StringDup	Copies a string into a newly allocated buffer in memory.

Constructor of the FormHandler Class

FormHandler	Creates an instance of this class.

Methods of the FormHandler Base Class

IsValid	Specifies whether or not the submitted data was successfully parsed by the FormHandler class.
GetQueryString	Gets the query part of the URI (the name-value pairs after the question mark) from the request.
ParseQueryString	Parses the query part of the URI (the name-value pairs after the question mark) from the request.
Get (C++ only)	Gets the value of a specified name-value pair from the parsed form data.
Add (C++ only)	Adds a name-value pair to the parsed form data.

Delete (C++ only)	Removes a name-value pair from the parsed form data.
InitIterator (C++ only)	Sets up a pointer to the beginning of the list of name-value pairs in the parsed form data so that the Next method gets the first name-value pair in the list.
Next (C++ only)	Gets the next name-value pair from the parsed form data.
GetHashTable (Java only)	Returns a hashtable containing the parsed form data.

The following table summarizes the C functions available in WAI.

Table 9.1 C Functions in WAI

Function Name	Description	For More Information, See...
WAIaddResponseHeader()	Adds a header to the HTTP response to be sent back to the client.	"addResponseHeader" on page 111
WAIBuildURL()	Builds an absolute URL from the	"BuildURL" on page 113
WAIcreateWebAppService()	Creates a new web application service, assigns it an instance name, and associates it with a function for processing HTTP requests.	"WAIWebApplicationService" on page 151
WAIdeleteService()	Deletes a web application service.	
WAIdelResponseHeader()	Removes a header from the HTTP response to be sent back to the client.	"delResponseHeader" on page 115
WAIgetConfigParameter()	Gets the value of a parameter of the iiopexec function in the Service directive of the obj.conf file.	"getConfigParameter" on page 116
WAIgetCookie()	Retrieves any cookies sent by the client.	"getCookie" on page 119
WAIgetHost()	Gets the hostname of the machine where the web server is running.	"getHost" on page 145
WAIgetInfo()	Retrieves information about the web server (such as the value of CGI 1.1 environment variables that describe the server).	"getInfo" on page 146
WAIgetName()	Gets the server ID (for example, https-myhost) of the web server.	"getName" on page 147

Table 9.1 C Functions in WAI

Function Name	Description	For More Information, See...
WAIgetPort()	Gets the port number that the web server listens to.	"getPort" on page 148
WAIgetRequestHeader()	Gets a header from the HTTP request sent by the client.	"getRequestHeader" on page 121
WAIgetRequestInfo()	Gets information about the client request (such as the value of a CGI 1.1 environment variable).	"getRequestInfo" on page 122
WAIgetResponseContentLength()	Gets the content length (the value of the Content-length header) of the response.	"getResponseContentLength" on page 125
WAIgetResponseHeader()	Gets a header from the HTTP response you plan to send to the client.	"getResponseHeader" on page 126
WAIgetServerSoftware()	Gets the type and version of the server software.	"getServerSoftware" on page 148
WAIimplIsReady()	Prepares your WAI application to receive requests.	"Registering Your Web Application Service" on page 52
WAIisSecure()	Specifies whether or not the server is run with SSL enabled.	"isSecure" on page 149
WAILogError()	Logs an entry to the server's error log file (*server_root*/*server_id*/logs/errors on UNIX and *server_root**server_id*\logs\errors on Windows NT).	"LogError" on page 128
WAIReadClient()	Reads data from the client (for example, for data sent through the HTTP POST method).	"ReadClient" on page 130
WAIregisterService()	Registers the WAI application with the web server.	"RegisterService" on page 153
WAIRespondRedirect()	Redirects the client to a different URL.	"RespondRedirect" on page 134
(*WAIRunFunction)()	Type definition for the function that processes HTTP requests.	"Run" on page 153

Table 9.1 C Functions in WAI

Function Name	Description	For More Information, See...
WAIsetCookie()	Sets a cookie in the response header to be sent to the client.	"setCookie" on page 135
WAIsetRequestInfo()	(This method has no functional use at this time.)	"setRequestInfo" on page 138
WAIsetResponseContentLength()	Sets the content length (the value of the Content-length header) of the response to be sent to the client.	"setResponseContentLength" on page 138
WAIsetResponseContentType()	Sets the content type (the value of the Content-type header) of the response to be sent to the client.	"setResponseContentType" on page 139
WAIsetResponseStatus()	Sets the HTTP response code (for example, 404 for "File Not Found") of the response to be sent to the client.	"setResponseStatus" on page 140
WAIStartResponse()	Starts sending the response back to the client.	"StartResponse" on page 141
WAIstringFree()	Frees a string from memory.	"StringDelete" on page 154
WAIWriteClient()	Writes data to the client.	"WriteClient" on page 142

How to Use This Reference

The methods in this section are documented in Interface Definition Language, or IDL. The C, C++, and Java syntax for each method is listed under the IDL syntax for the method.

The following section is an example of the documentation for a WAI method. The syntax for the interface is described first. Next, the prototypes for the methods that implement this operation are documented.

...

Syntax HttpServerReturnType addResponseHeader(in string header,
in string value);

C Prototype:

```
NSAPI_PUBLIC WAIReturnType_t WAIaddResponseHeader(ServerSession_t p, const char
*header,
    const char *value);
```

C++ Prototype:

```
WAIReturnType addResponseHeader(const char *header,
    const char *value);
```

Java Prototype:

```
public netscape.WAI.HttpServerReturnType addResponseHeader(java.lang.String header,
    java.lang.String value);
```

...

Use the prototype for the language that you are using to write your application. Note that the parameters may differ between languages. For example, the C functions have an extra parameter (of the type ServerSession_t) that represents the HTTP request object.

Interfaces

The methods for the interfaces in this section are described in terms of their signatures. The interfaces described in WAI are:

- netscape::WAI::HttpServerRequest

  Provides access to the data in an HTTP request sent from the client to your server.

- netscape::WAI::HttpServerContext

  Provides access to data about the web server, such as the server's hostname and port number.

- netscape::WAI::WebApplicationService and netscape::WAI::WebApplicationBasicService

  Represent the web service that you want to write. Typically, you do not need to deal with these two interfaces; instead, you work directly from the WAIWebApplicationService base class, which implements these interfaces.

WAI also includes the following base class:

- WAIWebApplicationService

Base class from which you derive your web service that processes HTTP requests.

The rest of this chapter documents these interfaces and classes. Note that although in C, there is no concept of classes, the C API functions are documented here among the interfaces and classes for convenience.

netscape::WAI::HttpServerRequest

The HttpServerRequest interface declares methods for processing HTTP requests. It provides access to the data in an HTTP request sent from the client to your server.

This interface is implemented by the following classes:

- WAIServerRequest (in C++)
- netscape.WAI.HttpServerRequest (in Java)

When you write your own WAI class (which should derive from the Netscape base class WAIWebApplicationService; for details, see "WAIWebApplicationService" on page 150), you pass in a reference to an WAIServerRequest object (in C++) or an HTTPServerRequest object (in Java) as an argument to the Run method.

Using methods in these classes, you can get HTTP headers from a client request, set HTTP headers in a response to the request, get and set cookies in the client, write entries to the server's error log, and read and write data to the client.

Member Summary

The netscape::WAI::HttpServerRequest interface describes the following members:

Methods

addResponseHeader	Adds a header to the response to be sent back to the client.
BuildURL	Builds a URL from the prefix of a URI and the suffix of a URI.
delResponseHeader	Deletes a header from the response to be sent to the client.
getConfigParameter	Gets the value of a parameter of the iiopexec function in the Service directive of the obj.conf file.
getContext	Gets the HTTPServerContext object for the server.
getCookie	Gets a cookie from the request headers sent by the client.
getRequestHeader	Gets a specified header from the client's request.
getRequestInfo	Gets information about the client request (such as the value of a CGI 1.1 environment variable).

getResponseContentLength	Gets the value of the Content-length header from the response to be sent to the client.
getResponseHeader	Gets the specified header from the response to be sent to the client.
LogError	Logs an entry to the server's error log file (*server_root*/*server_id*/logs/errors on UNIX and *server_root**server_id*\logs\errors on Windows NT).
ReadClient	Reads data from the client (for example, for data sent through the HTTP POST method).
RespondRedirect	Redirects the client to a specified URL.
setCookie	Sets a cookie in the response header to be sent to the client.
setRequestInfo	(This method has no functional use at this time.)
setResponseContentLength	Sets the content length (the value of the Content-length header) of the response to be sent to the client.
setResponseContentType	Sets the content type (the value of the Content-type header) of the response to be sent to the client.
setResponseStatus	Sets the HTTP response code (for example, 404 for "File Not Found") of the response to be sent back to the client.
StartResponse	Starts to send the response to the client.
WriteClient	Writes data to the client.

Methods

addResponseHeader

Adds a specified header to the response to be sent to the client.

Syntax

```
HttpServerReturnType addResponseHeader(in string header,
    in string value);
```

C Prototype:

```
NSAPI_PUBLIC WAIReturnType_t WAIaddResponseHeader(ServerSession_t p, const char
*header,
    const char *value);
```

C++ Prototype:

```
WAIReturnType addResponseHeader(const char *header,
    const char *value);
```

Java Prototype:

```
public netscape.WAI.HttpServerReturnType addResponseHeader(java.lang.String header,
    java.lang.String value);
```

Parameters This method has the following parameters:

p	**(C only)** Handle to the server session object, which is passed as an argument to your callback function.
header	Name of the header to add.
value	Content of the header.

Returns HttpServerReturnType::Success if the header was successfully added. The actual return value differs, depending on the language you are using:

- WAISPISuccess in C/C++
- netscape.WAI.HTTPServerReturnType.Success in Java

HttpServerReturnType::Failure if the header could not be added. The actual return value differs, depending on the language you are using:

- WAISPIFailure in C/C++
- netscape.WAI.HTTPServerReturnType.Failure in Java

Example The following example in Java adds a Pragma:no-cache header to the response sent to the client.

```
...
/* Define a class for your service. */
class MyWebApplicationService extends WAIWebApplicationService {
...
/* Define the Run method, which is called whenever the client requests your service. */
    public int Run(HttpServerRequest request){

/* Create an output stream for the content that you are delivering to the client. */
        ByteArrayOutputStream streamBuf = new ByteArrayOutputStream();
```

```
            PrintStream content = new PrintStream(streamBuf);
            HttpServerReturnType rc;
...
/* Insert code to write the content to the stream. */
...
/* Prepare to send the content back to the client.*/
            byte[] outbuff = streamBuf.toByteArray();
            try {

/* Add the Pragma: no-cache header to the response. */
                rc = request.addResponseHeader("Pragma", "no-cache");

/* Specify the length of the data to be sent.*/
                rc = request.setResponseContentLength(outbuff.length);

/* Start sending the response. */
                request.StartResponse();
            }
                catch(org.omg.CORBA.SystemException e){
            }
...
      }
...
}
...
```

See Also delResponseHeader, getResponseHeader.

BuildURL

Using a specified URI prefix and URI suffix, creates a full URL of the form http://*server*:*port prefix suffix*.

If you do not want to specify a prefix or a suffix, use the empty string ("") instead of a NULL pointer.

Syntax string BuildURL(in string prefix, in string suffix);

C Prototype:

```
NSAPI_PUBLIC char *WAIBuildURL(ServerSession_t p,
    const char *prefix, const char *suffix);
```

C++ Prototype:

```
char *BuildURL(const char *prefix, const char *suffix);
```

Java Prototype:

```
public java.lang.String
BuildURL(java.lang.String prefix, java.lang.String suffix);
```

Parameters This method has the following parameters:

p	**(C only)** Handle to the server session object, which is passed as an argument to your callback function.
prefix	URI prefix that you want to use in the URL.
suffix	URI suffix that you want to use in the URL.

Returns The full URL containing the prefix and suffix.

Example The following example in C++ uses the suffix /index.html to build the URL http://*server_name*:*port_number*/index.html.

```
...
/* Define a class for your service. */
long
WebApplicationServicePrototype::Run(WAIServerRequest_ptr session)
{
...
    char *url;

/* Build the complete URL from the specified suffix. */
    url = session-BuildURL("", "/index.html");
...
}
```

...

delResponseHeader

Deletes a specified header from the response to be sent to the client. You use this method to remove a header that added when calling the addResponseHeader method.

Syntax HttpServerReturnType delResponseHeader(in string header);

C Prototype:
NSAPI_PUBLIC WAIReturnType_t WAIdelResponseHeader(ServerSession_t p, const char *header);

C++ Prototype:
WAIReturnType delResponseHeader(const char *header);

Java Prototype:
public netscape.WAI.HttpServerReturnType delResponseHeader(java.lang.String header);

Parameters This method has the following parameters:

p	**(C only)** Handle to the server session object, which is passed as an argument to your callback function.
header	Name of the header that you want to delete.

Returns HttpServerReturnType::Success if header was successfully deleted. The actual return value differs, depending on the language you are using:

- WAISPISuccess in C/C++
- netscape.WAI.HTTPServerReturnType.Success in Java

HttpServerReturnType::Failure if header could not be deleted. The actual return value differs, depending on the language you are using:

- WAISPIFailure in C/C++
- netscape.WAI.HTTPServerReturnType.Failure in Java

Example The following example in Java removes a header added through the addResponseHeader method.

...

```
/* Add the Pragma: no-cache header to the response. */
rc = request.addResponseHeader("Pragma", "no-cache");
...
/* Remove the Pragma: no-cache header.*/
rc = request.deleteResponseHeader("Pragma");
...
/* Start sending the response. */
request.StartResponse();
...
```

See Also addResponseHeader, getResponseHeader.

getConfigParameter

Obtains the current value of a parameter in the web service's object in the obj.conf file.

For example, if you specify the name-value pair Flavor=Peach in the web service's object:

```
<Object name="iiopexec">
Service fn="IIOPexec" Flavor=”Peach”
</Object>
```

you can get the value Peach by specifying the name Flavor as an argument to this method.

Syntax

```
HttpServerReturnType getConfigParameter(in string name,
    out string value);
```

C Prototype:

```
NSAPI_PUBLIC WAIReturnType_t WAIgetConfigParameter(ServerSession_t p, const char
*name,
    char ** value);
```

C++ Prototype:

```
WAIReturnType getConfigParameter(const char *name,
    char *& value);
```

Java Prototype:

```
public netscape.WAI.HttpServerReturnType getConfigParameter(java.lang.String name,
    org.omg.CORBA.StringHolder value);
```

Parameters This method has the following parameters:

p	**(C only)** Handle to the server session object, which is passed as an argument to your callback function.
name	Name of the parameter to retrieve.
value	Value retrieved by this method. **Note for Java Programmers:** StringHolder is a class in the org.omg.CORBA package. Holder classes support the passing of out and inout parameters associated with operation requests. For details on this and other Holder classes, see the *Netscape ISB for Java Reference Guide*.

Returns HttpServerReturnType::Success if the variable exists and is accessible. The actual return value differs, depending on the language you are using:

- WAISPISuccess in C/C++
- netscape.WAI.HTTPServerReturnType.Success in Java

HttpServerReturnType::Failure if the variable cannot be found or is not accessible. The actual return value differs, depending on the language you are using:

- WAISPIFailure in C/C++
- netscape.WAI.HTTPServerReturnType.Failure in Java

Example The following example in Java gets the value of the Flavor parameter in the iiopexec object in the obj.conf file.

```
...
/* Define a class for your service. */
class MyWebApplicationService extends WAIWebApplicationService {
...
/* Define the Run method, which is called whenever the client requests your service. */
    public int Run(HttpServerRequest request){
    ...
        /* Get the Flavor parameter from the iiopexec object. */
        if (request.getConfigParameter("Flavor", value) ==
            HttpServerReturnType.Success) {
            System.out.println("Flavor: " + value.value + "\n");
        }
```

```
    ...
    }
...
}
```

getContext

Retrieves the WAIServerContext object (in C++) or the HTTPServerContext object (in Java) for the server. (For details on this object, see "netscape::WAI::HttpServerContext" on page 144.) This object holds server information, such as the server's hostname and port number.

Call this function if you want to get information about the server (for example, if you want to get the name and version of the server software, or if you want to determine if the server is running SSL).

Syntax HttpServerContext getContext();

C Prototype:
N/A (you don't need to get the object to call the functions/methods associated with the object)

C++ Prototype:
WAIServerContext_ptr getContext();

Java Prototype:
public netscape.WAI.HttpServerContext getContext();

Returns The HttpServerContext object for the server.

Example The following example in C++ gets the WAIServerContext object for the web server and uses that object to get the server's version information.

```
long
WebApplicationServicePrototype::Run(WAIServerRequest_ptr session)
{
    ...
    /* Get the WAIServerContext object for the web server. */
    WAIServerContext_ptr context = session->getContext();
    ...
```

```
    /* Use WAIServerContext to get info on the web server version. */
    char *var;
    if ((var = context->getServerSoftware()) && *var){
        printf("Web Server software: %s", var);

        /* Free the string from memory when done. */
        StringDelete(var);
    }
    ...
}
```

See Also netscape::WAI::HttpServerContext.

getCookie

Retrieves the cookie from the request headers sent by the client.

Syntax HttpServerReturnType getCookie(out string cookie);

C Prototype:
NSAPI_PUBLIC WAIReturnType_t
WAIgetCookie(ServerSession_t p, char ** cookie);

C++ Prototype:
WAIReturnType getCookie(char *& cookie);

Java Prototype:
public netscape.WAI.HttpServerReturnType getCookie(org.omg.CORBA.StringHolder cookie);

Parameters This method has the following parameters:

p	**(C only)** Handle to the server session object, which is passed as an argument to your callback function.
cookie	Value of the cookie. **Note for Java Programmers:** StringHolder is a class in the org.omg.CORBA package. Holder classes support the passing of out and inout parameters associated with operation requests. For details on this and other Holder classes, see the *Netscape ISB for Java Reference Guide*..

Returns HttpServerReturnType::Success if the cookie was retrieved successfully. The actual return value differs, depending on the language you are using:

- WAISPISuccess in C/C++
- netscape.WAI.HTTPServerReturnType.Success in Java

HttpServerReturnType::Failure if the cookie could not be retrieved. The actual return value differs, depending on the language you are using:

- WAISPIFailure in C/C++
- netscape.WAI.HTTPServerReturnType.Failure in Java

Description When requesting a URL from an HTTP server, the client matches the URL against all cookies it has. If the client has cookies from the same domain as the URL, the client includes a line containing the name/value pairs of all matching cookies in the HTTP request headers. The format of that line is as follows:

Cookie: *name1=string1*; *name2=string2*...

For more information on cookies, see "setCookie" on page 135, the preliminary Netscape cookie specification at http://home.netscape.com/newsref/std/cookie_spec.html, and RFC 2109 ("HTTP State Management Mechanism") at http://www.internic.net/rfc/rfc2109.txt.

Example The following example in Java checks to see if a cookie is already set on a client before setting a new cookie on the client.

```
public int Run(HttpServerRequest request){

    ...

    org.omg.CORBA.StringHolder
    cookiebuff = new org.omg.CORBA.StringHolder();


    /* Check to see if the client is returning any cookies. */

    if (request.getCookie(cookiebuff)== HttpServerReturnType.Failure)


        /* If no cookies have been returned, set a new cookie. */

        request.setCookie("MY_NAME", "MY_VALUE", "", "", "/iiop", false);

    ...

}
```

See Also setCookie.

getRequestHeader

Retrieves a specified header from the client request.

Syntax

```
HttpServerReturnType getRequestHeader(in string header,
    out string value);
```

C Prototype:

```
NSAPI_PUBLIC WAIReturnType_t WAIgetRequestHeader(ServerSession_t p, const char
*name,
    char ** value);
```

C++ Prototype:

```
WAIReturnType getRequestHeader(const char *header,
    char *& value);
```

Java Prototype:

```
public netscape.WAI.HttpServerReturnType getResponseHeader(java.lang.String header,
    org.omg.CORBA.StringHolder value);
```

Parameters This method has the following parameters:

p	**(C only)** Handle to the server session object, which is passed as an argument to your callback function.
header	Name of the header to retrieve.
value	The current content of the header retrieved by this method.

Returns HttpServerReturnType::Success if the header was successfully retrieved. The actual return value differs, depending on the language you are using:

- WAISPISuccess in C/C++
- netscape.WAI.HTTPServerReturnType.Success in Java

HttpServerReturnType::Failure if the header could not be retrieved. The actual return value differs, depending on the language you are using:

- WAISPIFailure in C/C++
- netscape.WAI.HTTPServerReturnType.Failure in Java

Example The following example in C++ gets the value of the user-agent header in a client's request.

```
long
WebApplicationServicePrototype::Run(WAIServerRequest_ptr session)
{
    ...
    char *var;
    /* Get the value of the user-agent header. */
    if (session->getRequestHeader("user-agent", var) == WAISPISuccess){
        printf("User agent: %s", var);
        /* Free the string from memory when done. */
        StringDelete(var);
    }
    ...
}
```

getRequestInfo

Accesses information about the server and a specific HTTP request.

Syntax

```
HttpServerReturnType getRequestInfo(in string name,
    out string value);
```

C Prototype:

```
NSAPI_PUBLIC WAIReturnType_t
WAIgetRequestInfo(ServerSession_t p, const char *name,
    char ** value);
```

C++ Prototype:

```
WAIReturnType getRequestInfo(const char *name,
    char *& value);
```

Java Prototype:

```
public netscape.WAI.HttpServerReturnType getRequestInfo(java.lang.String name,
    org.omg.CORBA.StringHolder value);
```

Parameters This method has the following parameters:

p	**(C only)** Handle to the server session object, which is passed as an argument to your callback function.
name	Name of the variable to retrieve.
value	The current value of the variable. **Note for Java Programmers:** StringHolder is a class in the org.omg.CORBA package. Holder classes support the passing of out and inout parameters associated with operation requests. For details on this and other Holder classes, see the *Netscape ISB for Java Reference Guide*.

The following table lists the names of the variables that you can specify for the name argument.

Table 9.2 getRequestInfo variables and the types of information they represent

Variable Name	Description
AUTH_TYPE	Authentication scheme for the request (found from the auth-scheme token in the request).
CLIENT_CERT	Base-64 DER-encoded certificate received from the client if the PathCheck built-in function get-client-cert is called. (See the *NSAPI Programmer's Guide* for details on this function.)
CONTENT_LENGTH	Length of the content of the client request.
CONTENT_TYPE	MIME type of the content of the client request.
HOST	Name of the client's host machine.
HTTPS	Specifies whether or not SSL is "ON" or "OFF".
HTTPS_KEYSIZE	Number of bits in the session key used to encrypt the session (if SSL is enabled).
HTTPS_SECRETKEYSIZE	Number of bits used to generate the server's private key (if SSL is enabled).
HTTP_*	Value of the specified HTTP_* header (headers with names that begin with the prefix HTTP_).
PATH_INFO	Trailing part of the URI that follows the SCRIPT_NAME part of the path.
PATH_TRANSLATED	The filesystem path to the file requested by the URI.
QUERY_STRING, QUERY	The query part of the URI (the name-value pairs following the question mark).

Table 9.2 getRequestInfo variables and the types of information they represent

Variable Name	Description
REMOTE_ADDR	IP address of the client sending the request.
REMOTE_HOST	Fully qualified domain name of the client sending the request.
REMOTE_USER	If the client is using the basic authentication scheme, the user ID sent by the client for authentication.
REQUEST_METHOD	Method in which the request was made (for example, GET or POST or HEAD).
SCRIPT_NAME	Part of the URI that identifies the script being executed.
SERVER_PROTOCOL	Name and revision number of the information protocol of the incoming request.
URI	URI requested by the client.

Returns HttpServerReturnType::Success if the information exists and is accessible. The actual return value differs, depending on the language you are using:

- WAISPISuccess in C/C++
- netscape.WAI.HTTPServerReturnType.Success in Java

HttpServerReturnType::Failure if the information does not exist or is not accessible. The actual return value differs, depending on the language you are using:

- WAISPIFailure in C/C++
- netscape.WAI.HTTPServerReturnType.Failure in Java

Example The following example in Java gets the IP address of the client that sent the request.

```
public int Run(HttpServerRequest request){
    ...
    org.omg.CORBA.StringHolder value = new org.omg.CORBA.StringHolder();
    /* Get the value of the client's IP address. */
    if (request.getRequestInfo("REMOTE_ADDR", value) ==
        HttpServerReturnType.Success){
        System.out.println("Client addr: %s", value.value + "\n");
    }
```

```
    ...
}
```

Note The C function, WAIgetRequestInfo, internally allocates memory for the value string. To free the memory, call WAIstringFree (see StringDelete).

See Also setRequestInfo

getResponseContentLength

Retrieves the content length of the response to be sent to the client. You use this method to get the value that you set when calling the setResponseContentLength method.

Syntax

```
HttpServerReturnType getResponseContentLength(
    out unsigned long Length);
```

C Prototype:

```
NSAPI_PUBLIC WAIReturnType_t WAIgetResponseContentLength(ServerSession_t p,
    unsigned long *Length);
```

C++ Prototype:

```
WAIReturnType
getResponseContentLength(unsigned long& Length);
```

Java Prototype:

```
public netscape.WAI.HttpServerReturnType
getResponseContentLength(org.omg.CORBA.IntHolder Length);
```

Parameters This method has the following parameters:

p	**(C only)** Handle to the server session object, which is passed as an argument to your callback function.
Length	Content length of the response. **Note to Java Programmers:** IntHolder is a class in the org.omg.CORBA package. Holder classes support the passing of out and inout parameters associated with operation requests. For details on this and other Holder classes, see the *Netscape ISB for Java Reference Guide*.

Returns HttpServerReturnType::Success if the content length was successfully fetched. The actual return value differs, depending on the language you are using:

- WAISPISuccess in C/C++

- netscape.WAI.HTTPServerReturnType.Success in Java

HttpServerReturnType::Failure if the content length could not be determined. The actual return value differs, depending on the language you are using:

- WAISPIFailure in C/C++
- netscape.WAI.HTTPServerReturnType.Failure in Java

Example The following example in C gets the value of the content length set through the setResponseContentLength method.

```
long
MyRunFunction(ServerSession_t obj)
{
long *length;
...
/* Specify the content to send back to the client. */
char *buffer = "Hello World\n";
size_t bufflen = strlen(buffer);

/* Set the length of this content in the content-length header. */
WAIsetResponseContentLength(obj, bufflen);
...
/* Get the content-length. */
WAIgetResponseContentLength(obj, &length);
...
}
```

See Also setResponseContentLength.

getResponseHeader

Gets a specific header from the response to be sent to the client. You use this method to get the value of a header that added when calling the addResponseHeader method.

Syntax HttpServerReturnType getResponseHeader(in string header,

```
    out string value);
```

C Prototype:

```
NSAPI_PUBLIC WAIReturnType_t WAIgetResponseHeader(ServerSession_t p, const char
*header,
    char ** value);
```

C++ Prototype:

```
WAIReturnType getResponseHeader(const char *header,
    char *& value);
```

Java Prototype:

```
public netscape.WAI.HttpServerReturnType getResponseHeader(java.lang.String header,
    org.omg.CORBA.StringHolder value);
```

Parameters This method has the following parameters:

p	**(C only)** Handle to the server session object, which is passed as an argument to your callback function.
header	Name of the header that you want to retrieve.
value	The current value of the header. **Note for Java Programmers:** StringHolder is a class in the org.omg.CORBA package. Holder classes support the passing of out and inout parameters associated with operation requests. For details on this and other Holder classes, see the *Netscape ISB for Java Reference Guide*.

Returns HttpServerReturnType::Success if the header was successfully retrieved. The actual return value differs, depending on the language you are using:

- WAISPISuccess in C/C++
- netscape.WAI.HTTPServerReturnType.Success in Java

HttpServerReturnType::Failure if the header could not be retrieved. The actual return value differs, depending on the language you are using:

- WAISPIFailure in C/C++
- netscape.WAI.HTTPServerReturnType.Failure in Java

Example The following example in Java gets the value of a header added through the addResponseHeader method.

```
...
```

```
/* Add the Pragma: no-cache header to the response. */
rc = request.addResponseHeader("Pragma", "no-cache");
...
/* Get the value of the Pragma header.*/
org.omg.CORBA.StringHolder value = new org.omg.CORBA.StringHolder();
rc = request.getResponseHeader("Pragma", value);
...
/* Start sending the response. */
request.StartResponse();
...
```

See Also addResponseHeader, delResponseHeader.

LogError

Logs messages to the server error log (*server_root*/https-*server_id*/logs/errors).

Syntax

```
HttpServerReturnType LogError(in long degree, in string func,
    in string msg, in boolean clientinfo);
```

C Prototype:

```
NSAPI_PUBLIC WAIReturnType_t WAILogError(ServerSession_t p,
    long degree, const char *func, const char *msg,
    WAIBool clientinfo);
```

C++ Prototype:

```
WAIReturnType LogError(long degree, const char *func,
    const char *msg, WAIBool clientinfo);
```

Java Prototype:

```
public
netscape.WAI.HttpServerReturnType LogError(int degree,
    java.lang.String func, java.lang.String msg,
    boolean clientinfo);
```

Parameters This method has the following parameters:

p	**(C only)** Handle to the server session object, which is passed as an argument to your callback function.
degree	Degree of severity of the error. (This is included in the log entry.) The degree of severity can be one of the following values: • 0 (warning message) • 1 (misconfiguration error; for example, if there is a syntax error or permission violation in a configuration file) • 2 (security error; for example, if authentication fails or if the client is forbidden to access the resource) • 3 (failure; for example, if an internal problem prevents the request from being fulfilled) • 4 (catastrophe; for example, a fatal server error such as running out of memory) • 5 (informational message) • 6 (internal message; messages will only appear if the magnus.conf file contains the LogVerbose On setting) If you are writing a C/C++ application, you can include the nsapi.h header file and use the defined values for the degree of severity.
func	Name of the function reporting the error. (This function name is included in the log entry. You can use this to help identify which function caused the log entry to be written.)
msg	Message that you want logged.
clientinfo	If true, information about the session (such as the IP address of the client) and request (such as the requested URI) are included in the log entry.

Returns HttpServerReturnType::Success if the message was successfully logged. The actual return value differs, depending on the language you are using:

- WAISPISuccess in C/C++
- netscape.WAI.HTTPServerReturnType.Success in Java

HttpServerReturnType::Failure if the message could not be logged. The actual return value differs, depending on the language you are using:

- WAISPIFailure in C/C++
- netscape.WAI.HTTPServerReturnType.Failure in Java

Example The following lines of code log informational and warning messages.

```
public int myMethod(HttpServerRequest request){
...
request.LogError(5, "myMethod()", "An informational message.\n", true);
request.LogError(0, "myMethod()", "A warning message.\n", false);
...
```

These lines of code generate the following messages in the server's error log:

```
[15/May/1997:07:53:49] info: for host 198.95.249.43 trying to GET /iiop/JavaWASP, myMethod() reports:
An informational message.
[15/May/1997:07:53:49] warning: myMethod() reports: A warning message.
```

Note that in the first entry, the IP address of the client, the method used to access the resource, and the URI of the resource are logged to the entry because LogError is called with the clientinfo argument set to true.

ReadClient

Reads data from the client.

Syntax long ReadClient(inout HttpServerBuffer buffer);

C Prototype:

```
NSAPI_PUBLIC long WAIReadClient(ServerSession_t p,
    unsigned char *buffer, unsigned buffsize);
```

C++ Prototype:

```
long ReadClient(unsigned char *buffer,
    unsigned buffsize);
```

Java Prototype:

```
public int
ReadClient(netscape.WAI.HttpServerBufferHolder buffer);
```

Parameters This method has the following parameters:

p	**(C only)** Handle to the server session object, which is passed as an argument to your callback function.
buffer	Buffer to receive data from the client. **Note for Java Programmers:** HttpServerBufferHolder is a class in the netscape.WAI package. When you construct an object of this class, you need to pass a byte array to the constructor (see the example below).
buffsize	**(C/C++ only)** Size of the buffer of data.

Returns Number of bytes read.

Example The following example in C++ gets data posted from the client (through the HTTP POST method and displays the posted data back to the client in its raw form (in other words, as an unparsed string of name/value pairs).

```
long
WebApplicationServicePrototype::Run(WAIServerRequest_ptr session)
{
    ostrstream outstr;
    char *var = NULL;
    unsigned contentLength;
    long status;
    char *myBuffer = NULL;

    outstr << "<P><FONT SIZE=+3>Resulting Posted Data</FONT></P>";

    /* Get the value of the content-length header.*/
    if (session->getRequestHeader("content-length", var) ==
        WAISPIFailure){
        return 1;
    }

    /* Use the content length to allocate memory for the data. */
    contentLength = atoi(var);
    StringDelete(var);
```

```
    /* Allocate memory for the content plus one byte for the trailing 0. */
    myBuffer = StringAlloc(contentLength+1);
    if (myBuffer==NULL) {
        return 1;
    }
    myBuffer[contentLength] = '\0';

    /* Read the posted data from the client.*/
    status = session->ReadClient((unsigned char*)myBuffer, contentLength);

    /* Print the raw posted data back to the client. */
    outstr << "\n<PRE>\n<B>Output of the Form:</B>\n\n" << (const char*)myBuffer << "\n</
PRE>\n<P>";
    StringDelete(myBuffer);
    outstr << endl;
    session->setResponseContentLength(outstr.pcount());
    session->StartResponse();
    session->WriteClient((const unsigned char *)outstr.str(), outstr.pcount());
    outstr.rdbuf()->freeze(0);
    return 0;
}
```

The following example in Java gets data posted from the client (through the HTTP POST method and displays the posted data back to the client in its raw form (in other words, as an unparsed string of name/value pairs).

```
public int Run(HttpServerRequest request){
/* Set up an output stream to send data back to the client. */
    org.omg.CORBA.StringHolder value = new org.omg.CORBA.StringHolder();
    request.getRequestHeader("content-length", value);
    ByteArrayOutputStream contentStream = new ByteArrayOutputStream();
    /* Create the buffer holder and initialize it the number of bytes to receive.*/
    netscape.WAI.HttpServerBufferHolder inbuff = new netscape.WAI.HttpServerBufferHolder(new
```

```
byte[1024]);
    Integer content_length = new Integer(value.value);
    int cnt;
    int data_left;

    /* Read the posted data into the buffer holder. */
    for (data_left = content_length.intValue(); data_left > 0;
        data_left -= cnt){
        cnt = request.ReadClient(inbuff);
        if (cnt == 0)
            data_left = 0;
        else
            contentStream.write(inbuff.value, 0, cnt);
        }
    HttpServerReturnType rc;
    byte[] outbuff = contentStream.toByteArray();
    try {
        rc = request.setResponseContentLength(outbuff.length);
        request.StartResponse();
    }
    catch(org.omg.CORBA.SystemException e){
    }
    catch(java.lang.Exception e) {
        System.err.println(e);
    }
    int write_cnt = request.WriteClient(outbuff);
    return 0;
}
```

See Also WriteClient.

RespondRedirect

Sends a page back to the client to notify the client that the page has moved.

Syntax HttpServerReturnType RespondRedirect (in string *url*);

C Prototype:
NSAPI_PUBLIC WAIReturnType_t WAIRespondRedirect(ServerSession_t p, const char *url);

C++ Prototype:
WAIReturnType RespondRedirect(const char *url);

Java Prototype:
public netscape.WAI.HttpServerReturnType RespondRedirect(java.lang.String url);

Parameters This method has the following parameters:

p	**(C only)** Handle to the server session object, which is passed as an argument to your callback function.
url	URL to redirect the client to.

Returns HttpServerReturnType::Success if redirect was successful. The actual return value differs, depending on the language you are using:

- WAISPISuccess in C/C++
- netscape.WAI.HTTPServerReturnType.Success in Java

HttpServerReturnType::Failure if the response failed to redirect the client. The actual return value differs, depending on the language you are using:

- WAISPIFailure in C/C++
- netscape.WAI.HTTPServerReturnType.Failure in Java

Description When you call this method (followed by StartResponse), the server returns the following page to the client:

Moved Temporarily

This document has moved to a new location. Please update your documents and hotlists accordingly.

The word "location" on this page is a link pointing to the new location of the page. The user can choose to click on this link to go to the new location.

If instead you want the client to be automatically redirected to the new location, call addResponseHeader to add the Location header, call setResponseStatus to set a response code of 301 or 302, then call StartResponse to send the response back to the client. For an example of this scenario, see the following sections:

- "Redirecting Users to Another Page" on page 51 in "Writing a WAI Application in C" on page 45
- "Redirecting Users to Another Page" on page 71 in "Writing a WAI Application in C++" on page 59
- "Redirecting Users to Another Page" on page 87 in "Writing a WAI Application in Java" on page 77

setCookie

Creates a cookie and sends it to the client.

Syntax

```
HttpServerReturnType setCookie(in string name, in string value,
    in string expires, in string domain, in string path,
    in boolean secure);
```

C Prototype:

```
NSAPI_PUBLIC WAIReturnType_t WAIsetCookie(ServerSession_t p,
    const char *name, const char *value, const char *expires,
    const char *domain, const char *path, WAIBool secure);
```

C++ Prototype:

```
WAIReturnType setCookie(const char *name, const char *value,
    const char *expires, const char *domain, const char *path,
    WAIBool secure);
```

Java Prototype:

```
public netscape.WAI.HttpServerReturnType setCookie(java.lang.String name, java.lang.String
value,
    java.lang.String expires, java.lang.String domain,
    java.lang.String path, boolean secure);
```

Parameters This method has the following parameters:

p	**(C only)** Handle to the server session object, which is passed as an argument to your callback function.
name	A sequence of characters excluding semicolon, comma, and white space. If there is a need to place such data in the name, some encoding method such as URL-style %XX encoding is recommended, though no encoding is defined or required.
value	A sequence of characters excluding semicolon, comma, and white space. If there is a need to place such data in the value, some encoding method such as URL-style %XX encoding is recommended, though no encoding is defined or required. This is the only required attribute of the Set-Cookie header.

expires	Specifies a date string that defines the valid life time of the cookie. Once the expiration date has been reached, the cookie will no longer be stored or given out. The date string is formatted as: Wdy, DD-Mon-YYYY HH:MM:SS GMT. This is based on RFC 822, RFC 850, RFC 1036, and RFC 1123, with the variations that the only legal time zone is GMT and the separators between the elements of the date must be dashes. expires is an optional attribute. If expires is not specified, the cookie expires when the user's session ends.
domain	Specifies a domain from which cookies can be set. When searching the cookie list for valid cookies, a comparison of the domain attributes of the cookie is made with the Internet domain name of the host from which the URL will be fetched. If there is a tail match, then the cookie will go through path matching to see if it should be sent. Tail matching means that domain attribute is matched against the tail of the fully qualified domain name of the host. A domain attribute of acme.com would match host names anvil.acme.com as well as shipping.crate.acme.com. Only hosts within the specified domain can set a cookie for a domain, and domains must have at least two or three periods in them to prevent domains of the form: .com, .edu, and va.us. Any domain that fails within one of seven special top level domains only requires two periods. Any other domain requires at least three. The seven special top level domains are: com, edu, net, org, gov, mil, and int. The default value of domain is the host name of the server that generated the cookie response.
path	Specifies the subset of URLs in a domain for which the cookie is valid. If a cookie has already passed domain matching, then the pathname component of the URL is compared with the path attribute, and if there is a match, the cookie is considered valid and is sent along with the URL request. The path /sales would match /sales west and /sales/west.html. The path "/" is the most general path. If you don't specify a value for path, setCookie uses the path described by the header that contains the cookie.
secure	If secure is set to True, the cookie is transmitted only if the communications channel with the host is a secure one. Currently, this means that secure cookies are sent only to HTTPS (HTTP over SSL) servers. If secure is False, a cookie is considered safe to send in the clear over unsecured channels.

Returns HttpServerReturnType::Success if cookie was set successfully. The actual return value differs, depending on the language you are using:

- WAISPISuccess in C/C++

- netscape.WAI.HTTPServerReturnType.Success in Java

HttpServerReturnType::Failure if the cookie could not be set. The actual return value differs, depending on the language you are using:

- WAISPIFailure in C/C++
- netscape.WAI.HTTPServerReturnType.Failure in Java

Examples See .

See Also getCookie.

setRequestInfo

This method has no functional use at this time.

setResponseContentLength

Sets the length of the response content.

Syntax
```
HttpServerReturnType setResponseContentLength(
    in unsigned long Length);
```

C Prototype:
```
NSAPI_PUBLIC WAIReturnType_t WAIsetResponseContentLength(ServerSession_t p,
    unsigned long Length);
```

C++ Prototype:
```
WAIReturnType setResponseContentLength(unsigned long Length);
```

Java Prototype:
```
public netscape.WAI.HttpServerReturnType setResponseContentLength(int Length);
```

Parameters This method has the following parameters:

p	**(C only)** Handle to the server session object, which is passed as an argument to your callback function.
Length	Content length that you want to set for the response.

Returns HttpServerReturnType::Success if the content length was successfully set. The actual return value differs, depending on the language you are using:

- WAISPISuccess in C/C++
- netscape.WAI.HTTPServerReturnType.Success in Java

HttpServerReturnType::Failure if the content length could not be set. The actual return value differs, depending on the language you are using:

- WAISPIFailure in C/C++
- netscape.WAI.HTTPServerReturnType.Failure in Java

Example The following example in C sets the content-length header for a response before sending the response back to the client.

```
long
MyRunFunction(ServerSession_t obj)
{
...
/* Specify the content to send back to the client. */
char *buffer = "Hello World\n";
size_t bufflen = strlen(buffer);

/* Set the length of this content in the content-length header. */
WAIsetResponseContentLength(obj, bufflen);
...
}
```

See Also getResponseContentLength.

setResponseContentType

Adds a header for the content type for the response. The default content type is text/html.

Syntax
```
HttpServerReturnType setResponseContentType(
    in string ContentType);
```

C Prototype:
```
NSAPI_PUBLIC WAIReturnType_t WAIsetResponseContentType(ServerSession_t p,
    const char *ContentType);
```

C++ Prototype:
WAIReturnType setResponseContentType(const char *ContentType);

Java Prototype:
public netscape.WAI.HttpServerReturnType setResponseContentType(java.lang.String ContentType);

Parameters This method has the following parameters:

p	**(C only)** Handle to the server session object, which is passed as an argument to your callback function.
Length	Content type that you want to assign to the response.

Returns HttpServerReturnType::Success if the content type was successfully set. The actual return value differs, depending on the language you are using:

- WAISPISuccess in C/C++
- netscape.WAI.HTTPServerReturnType.Success in Java

HttpServerReturnType::Failure if the content type could not be set. The actual return value differs, depending on the language you are using:

- WAISPIFailure in C/C++
- netscape.WAI.HTTPServerReturnType.Failure in Java

setResponseStatus

Sets status to the request status code.

Syntax HttpServerReturnType setResponseStatus(in long status,
in string reason);

C Prototype:
NSAPI_PUBLIC WAIReturnType_t WAIsetResponseStatus(ServerSession_t p, long status,
const char *reason);

C++ Prototype:
WAIReturnType setResponseStatus(long status,
const char * reason);

Java Prototype:
public netscape.WAI.HttpServerReturnType
setResponseStatus(int status, java.lang.String reason);

Parameters This method has the following parameters:

p	**(C only)** Handle to the server session object, which is passed as an argument to your callback function.
status	Status that you want to assign to the response.
reason	Message that you want associated with the status that you've set. If this argument is NULL, the server attempts to find the standard message for the status code (for example, "File Not Found" for the status code 404). If no message is found for the status code, the message "Unknown Reason" is used.

Returns HttpServerReturnType::Success if the status was successfully set. The actual return value differs, depending on the language you are using:

- WAISPISuccess in C/C++
- netscape.WAI.HTTPServerReturnType.Success in Java

HttpServerReturnType::Failure if the status could not be set. The actual return value differs, depending on the language you are using:

- WAISPIFailure in C/C++
- netscape.WAI.HTTPServerReturnType.Failure in Java

StartResponse

Starts the HTTP response.

If the incoming request specifies that it follows the HTTP 0.9 standard (which does not specify that headers can be included in requests and responses), StartResponse does nothing.

If the request specifies that it follows the HTTP 1.0 (or later) standard (which allows headers in requests and responses), StartResponse sends a header.

Syntax long StartResponse();

C Prototype:
NSAPI_PUBLIC long WAIStartResponse(ServerSession_t p);

C++ Prototype:
long StartResponse();

Java Prototype:
public int StartResponse();

Parameters This method has the following parameters:

p	**(C only)** Handle to the server session object, which is passed as an argument to your callback function.

Returns REQ_NOACTION if the request used the HEAD method (meaning that the body of the resource should not be sent).

REQ_PROCEED otherwise.

Example The following example in C starts sending a response back to the client after setting the content-length header in the response.

```
long
MyRunFunction(ServerSession_t obj)
{
    ...
    /* Specify the length of the content you want to send. */
    WAIsetResponseContentLength(obj, contentLength);

    /* Start sending the response. */
    WAIStartResponse(obj);
    ...
}
```

WriteClient

Writes data to the client.

Syntax long WriteClient(in HttpServerBuffer buffer);

C Prototype:

```
NSAPI_PUBLIC long WAIWriteClient(ServerSession_t p,
    const unsigned char *buffer, unsigned buffsize);
```

C++ Prototype:

```
long WriteClient(const unsigned char *buffer,
    unsigned buffsize);
```

Java Prototype:

```
public int WriteClient(byte [] buffer);
```

Parameters This method has the following parameters:

p	**(C only)** Handle to the server session object, which is passed as an argument to your callback function.
buffer	Buffer of data to write to the client.
buffsize	**(C/C++ only)** Size of the buffer of data.

Returns 1 if successful or -1 if an error occurs.

Example The following example in C writes an HTML page containing the words "Hello World" back to the client.

```
long
MyRunFunction(ServerSession_t obj)
{
    /* Specify the content to be written. */
    char *buffer = "Hello World\n";
    size_t bufflen = strlen(buffer);

    /* Set the content-length header in the response to be sent to the client.*/
    WAIsetResponseContentLength(obj, bufflen);

    /* Start sending the response. */
    WAIStartResponse(obj);

    /* Write the data to the client. */
    WAIWriteClient(obj, (const unsigned char *)buffer, bufflen);
```

```
    return 0;
}
```

See Also ReadClient.

netscape::WAI::HttpServerContext

The HttpServerContext interface provides access to information about the web server.

This interface is implemented as the following classes:

- WAIServerContext (in C++)
- netscape.WAI.HttpServerContext (in Java)

In C++, you can get access to an WAIServerContext object by calling the getContext method of a WAIServerRequest object. In Java, you can get access to an HTTPServerContext object by calling the getContext method of a HTTPServerRequest object. (See the section "netscape::WAI::HttpServerRequest" on page 110 for details on these objects.)

You can use the methods of these classes to get the following information on the web server:

- The hostname of the machine where the server is running
- The port number that the server listens to
- The server identifier (for example, https-myhost)
- The product name and version of the server software
- The version of CGI supported by the server (for example, CGI 1.1)
- Whether or not the server is running with SSL enabled

Member Summary

The netscape::WAI::HttpServerContext interface describes the following members:

Methods

getHost	Retrieves the host name of the machine running the web server.
getInfo	Retrieves information about the web server (such as the value of CGI 1.1 environment variables that describe the server).
getName	Retrieves the server ID (for example, https-myhost).
getPort	Retrieves the port number that the server listens to.
getServerSoftware	Retrieves the product name and version of the web server (for example, Netscape Enterprise/3.0).
isSecure	Specifies whether or not SSL is enabled on the server.

Methods

getHost

Retrieves the hostname of the machine where the web server is running.

Syntax

```
string getHost();
```

C Prototype:

```
NSAPI_PUBLIC char *WAIgetHost(ServerSession_t p);
```

C++ Prototype:

```
char *getHost();
```

Java Prototype:

```
public java.lang.String getHost();
```

Parameters

This method has the following parameters:

p	**(C only)** Handle to the server session object, which is passed as an argument to your callback function.

Returns

The name of the machine where the web server is running.

getInfo

Retrieves information about the server, such as the server's ID or the value of CGI 1.1 environment variables that describe the server (for example, SERVER_NAME and SERVER_PORT).

Syntax HttpServerReturnType getInfo(in string *name*, out string *value*);

C Prototype:

```
WAIBool WAIgetInfo(ServerSession_t p, const char *name,
    char **value);
```

C++ Prototype:

```
WAIReturnType getInfo(const char *name, char *&value);
```

Java Prototype:

```
public netscape.WAI.HttpServerReturnType getInfo(java.lang.String name,
    org.omg.CORBA.StringHolder value);
```

Parameters This method has the following parameters:

p	**(C only)** Handle to the server session object, which is passed as an argument to your callback function.
name	Name of the variable to retrieve.
value	The current value of the variable. **Note for Java Programmers:** StringHolder is a class in the org.omg.CORBA package. Holder classes support the passing of out and inout parameters associated with operation requests. For details on this and other Holder classes, see the *Netscape ISB for Java Reference Guide*.

The following table lists the names of the variables that you can specify for the name argument.

Table 9.3 getInfo variables and the types of information they represent

Variable Name	Description
GATEWAY_INTERFACE	CGI version supported by the web server (for example, CGI/1.1).
HTTPS	Specifies whether or not SSL is enabled on the server. • If SSL is enabled, the value of this variable is "ON". • If SSL is disabled, the value of this variable is "OFF".
SERVER_ID	Server identifier (for example, https-myhost). Currently, this only works on Windows NT.
SERVER_NAME	Name of the machine running the web server.
SERVER_PORT	Port number that the server listens to.
SERVER_SOFTWARE	Type and version of web server software (for example, Netscape-Enterprise/3.0).

Returns HttpServerReturnType::Success if the information exists and is accessible. The actual return value differs, depending on the language you are using:

- WAISPISuccess in C/C++
- netscape.WAI.HTTPServerReturnType.Success in Java

HttpServerReturnType::Failure if the information does not exist or is not accessible. The actual return value differs, depending on the language you are using:

- WAISPIFailure in C/C++
- netscape.WAI.HTTPServerReturnType.Failure in Java

getName

Retrieves the server ID (for example, https-myhost).

Syntax string getName();

C Prototype:
NSAPI_PUBLIC char *WAIgetName(ServerSession_t p);

C++ Prototype:
char *getName();

Java Prototype:
public java.lang.String getName();

Parameters This method has the following parameters:

p	**(Conly)** Handle to the server session object, which is passed as an argument to your callback function.

Returns The server ID, or an empty string if the information is not accessible.

getPort

Retrieves the number of the port the server listens to.

Syntax long getPort();

C Prototype:
NSAPI_PUBLIC long WAIgetPort(ServerSession_t p);

C++ Prototype:
long getPort();

Java Prototype:
public int getPort();

Parameters This method has the following parameters:

p	**(Conly)** Handle to the server session object, which is passed as an argument to your callback function.

Returns Port number that the web server listens to.

getServerSoftware

Retrieves the server type and version number (for example, Netscape-Enterprise/3.0).

Syntax string getServerSoftware();

C Prototype:
NSAPI_PUBLIC char *WAIgetServerSoftware(ServerSession_t p);

C++ Prototype:
char *getServerSoftware();

Java Prototype:
public java.lang.String getServerSoftware();

Parameters This method has the following parameters:

p	**(Conly)** Handle to the server session object, which is passed as an argument to your callback function.

Returns A string containing the server type and version number.

isSecure

Specifies whether or not SSL is enabled on the server.

Syntax boolean isSecure();

C Prototype:
NSAPI_PUBLIC WAIBool WAIisSecure(ServerSession_t p);

C++ Prototype:
int isSecure();

Java Prototype:
public boolean isSecure();

Parameters This method has the following parameters:

p	**(Conly)** Handle to the server session object, which is passed as an argument to your callback function.

Returns True if this server has SSL enabled.

netscape::WAI::WebApplicationService

WebApplicationService is one of the interfaces that represent web services.

Typically, you do not need to use this interface; instead, you work directly with the WAIWebApplicationService base class, which implements netscape::WAI::WebApplicationBasicService interface.

netscape::WAI::WebApplicationBasicService

WebApplicationBasicService is one of the interfaces that represent web services.

Typically, you do not need to use this interface; instead, you work directly with the WAIWebApplicationService base class, which implements this interface.

WebApplicationBasicService is derived from the netscape::WAI::WebApplicationService interface.

WAIWebApplicationService

TheWAIWebApplicationServicebaseclassrepresentsawebservice.Youderiveyourown web service class from this base class.

Member Summary

The WAIWebApplicationService base class contains the following members:

Constructor	
WAIWebApplicationService	Creates an instance of this class.
Methods	
ActivateWAS	Activates the object (if the object has not already been activated by the constructor).
getServiceInfo	(This is a method that you need to implement.) Provides informationabouttheauthor,version,andcopyrightofthe web application service that you are writing.
RegisterService	RegistersyourWAIapplicationwiththewebserverrunning on the specified host.

Run	(This is a method that you need to implement.) Executes your web application service (this is called whenever the server receives an HTTP request for your service).
StringAlloc	Allocates memory for a string.
StringDelete	Frees a string from memory.
StringDup	Copies a string into a newly allocated buffer in memory.

Constructor

WAIWebApplicationService

Creates an instance of the WAIWebApplicationService class. Note that in the 3.01 version of the server, the C++ constructor has an additional parameter to allow you to specify whether or not the object is activated when constructed.

If you want to activate the object at a later time, you can call the ActivateWAS method.

Syntax

C Prototype:

```
WAIcreateWebAppService(const char *name, WAIRunFunction func,
    int argc, char **argv);
```

C++ Prototype (3.0 version of the server):

```
WAIWebApplicationService(const char *name);
WAIWebApplicationService(const char *name, int argc,
    char **argv);
```

C++ Prototype (3.01 version of the server):

```
WAIWebApplicationService(const char *name);
WAIWebApplicationService(const char *name, WAIBool activateObj);
WAIWebApplicationService(const char *name, int argc,
    char **argv);
WAIWebApplicationService(const char *name, int argc,
    char **argv, WAIBool activateObj);
```

Java Prototype:

```
public WAIWebApplicationService(java.lang.String name);
```

Parameters This constructor has the following parameters:

name	Name of the instance of the service that you want to create.
WAIRunFunction	**(Conly)** Callback function invoked when an HTTP request for your service is received. This is the function that you define for processing the HTTP request. For details, see "Run" on page 153.
argc, argv	**(C/C++ only)** Allows you to pass command-line arguments into your application. argc is the number of command-line arguments and argv is an array of the arguments.
activateObj	**(C++ only)** If WAI_TRUE, specifies that the object should be immediately activated upon creation. If WAI_FALSE, you need to activate the object by calling the ActivateWAS method.

Methods

ActivateWAS

(3.01 servers only) Allows you to activate the web application service object at some later point in time after the object is constructed. In your application, you can call this method when you are ready to activate the object.

Syntax

C Prototype:
No equivalent function.

C++ Prototype:
```
void ActivateWAS();
```

Java Prototype:
No equivalent method.

getServiceInfo

Provides information about the author, version, and copyright of the web application service that you are writing.

This is a virtual/abstract method. You need to define this method when deriving your own class from the WAIWebApplicationService base class.

Syntax

C Prototype:
No equivalent function.

C++ Prototype:
virtual char *getServiceInfo();

Java Prototype:
public abstract java.lang.String getServiceInfo();

Returns A string containing author, version, and copyright. For example, you might define this function to return the string My Web Application Service v1.0.

RegisterService

Registers your WAI application with the web server running on the specified host.

Syntax

C Prototype:
NSAPI_PUBLIC WAIBool WAIregisterService(IIOPWebAppService_t p,
const char *host);

C++ Prototype:
WAIBool RegisterService(const char *host);

Java Prototype:
public boolean RegisterService(java.lang.String host);

Parameters This method has the following parameters:

p	**(C only)** Handle to the IIOP web application service structure.
host	Name of the host machine where the web server is running. Your WAI application will be registered as a web service on this server.

Returns WAI_True if your application was successfully registered to the web server.

WAI_False if your application could not be registered to the web server.

Run

Executes the web application service. This method is called by the server when an HTTP request for your service is received.

This is a virtual/abstract method. You need to define this method when deriving your own class from the WAIWebApplicationService base class.

Syntax

C Prototype:
typedef long (*WAIRunFunction)(ServerSession_t session);

C++ Prototype:
virtual long Run(WAIServerRequest_ptr session);

Java Prototype:
public abstract int Run(netscape.WAI.HttpServerRequest session);

Parameters This method has the following parameters:

session	Reference to the HTTPServerRequest object representing the client's HTTP request (see "netscape::WAI::HttpServerRequest" on page 110).

Returns: Status code representing the result of processing the HTTP request.

StringAlloc

Allocates memory for a string.

Syntax

C Prototype:
N/A

C++ Prototype:
char *StringAlloc(size_t size);

Java Prototype:
N/A

Parameters This method has the following parameters:

size	Size of the string that you want to allocate memory for.

Returns A buffer for the specified size of string.

StringDelete

Frees a string from memory.

Syntax

C Prototype:
NSAPI_PUBLIC void WAIstringFree(char *s);

C++ Prototype:
void *StringDelete(char *s);

Java Prototype:
N/A

Parameters This method has the following parameters:

s	String that you want to free from memory.

StringDup

Copies a string into a newly allocated buffer in memory.

Syntax

C Prototype:
N/A

C++ Prototype:
char *StringDup(const char *s);

Java Prototype:
N/A

Parameters This method has the following parameters:

s	String that you want to copy.

Returns Copy of the specified string.

netscape::WAI::FormHandler

The FormHandler class handles WAI application submissions through HTML forms. Using the FormHandler class you can write a WAI applications that receives and interprets data submitted through an HTML form.

FormHandler

The FormHandler class defines methods for processing data submitted through HTML forms sent from clients to your server. This class in new in the 3.01 releases of Netscape web servers.

Member Summary

The FormHandler class contains the following members:

Constructor	
FormHandler	Creates an instance of this class.
Methods	
IsValid	Specifies whether or not the submitted data was successfully parsed by the FormHandler class.
GetQueryString	Gets the query part of the URI (the name-value pairs after the question mark) from the request.
ParseQueryString	Parses the query part of the URI (the name-value pairs after the question mark) from the request.
Get (C++ only)	Gets the value of a specified name-value pair from the parsed form data.
Add (C++ only)	Adds a name-value pair to the parsed form data.
Delete (C++ only)	Removes a name-value pair from the parsed form data.
InitIterator (C++ only)	Sets up a pointer to the beginning of the list of name-value pairs in the parsed form data so that the Next method gets the first name-value pair in the list.
Next (C++ only)	Gets the next name-value pair from the parsed form data.
GetHashTable (Java only)	Returns a hashtable containing the parsed form data.

FormHandler

Creates an instance of the FormHandler class. This constructor reads in and parses the posted form data from the specified request.

Syntax

C++ Prototype:

```
FormHandler::FormHandler(WAIServerRequest_ptr request);
```

Java Prototype:

```
public FormHandler(HttpServerRequest request);
```

Parameters

This constructor has the following parameters:

request	Reference to the HTTPServerRequest object representing the client's HTTP request.

IsValid

Specifies whether or not the posted data is in a valid format that the server can parse.

You can call this method after creating an instance of the FormHandler class to determine if the constructor sucessfully read and parsed the posted form data.

Syntax

C++ Prototype:

```
WAIBool IsValid();
```

Java Prototype:

```
public boolean IsValid();
```

Returns

The actual return value differs, depending on the language you are using:

- **C++:** WAI_True if the submitted data is in a valid format, or WAI_False if it is not in a valid format.
- **Java:** true if the submitted data is in a valid format, or false if it is not in a valid format.

GetQueryString

Gets the query part of the URI (the name-value pairs following the question mark) from an HTTP GET request.

Syntax

C++ Prototype:

```
char* GetQueryString();
```

Java Prototype:

```
public String GetQueryString();
```

Returns The query part of the URI (the name-value pairs following the question mark in the URI).

ParseQueryString

Parses the query part of the URI (the name-value pairs following the question mark) from an HTTP GET request. Note that this method does not directly return the parsed data. Depending on the language you are using, you can access the parsed data in different ways:

- In C++, you can call the Get method to get the value of a specific name-value pair, or you can call the InitIterator method and the Next method to iterate through all name-value pairs in the parsed data.

  You can also call the Add method to add a new name-value pair to the parsed form data and the Delete method to remove a name-value pair from the parsed form data.

- In Java, you can call the GetHashTable method to get a Java hash table containing the parsed data. Then, you can call methods of the java.util.Hashtable class to access the data.

  The names serve as keys in the hashtable. The values are stored as Java vectors (for details, see your Java documentation on java.util.Vector).

  The values are implemented as Java vectors because a given name may be associated with multiple values. For example, if the form contains multiple-selection input, the submitted form data can contain several name-value pairs with the same name but different values.

Syntax **C++ Prototype:**
WAIBool ParseQueryString();

Java Prototype:
public boolean ParseQueryString();

Returns The actual return value differs, depending on the language you are using:

- **C++:** WAI_True if the server successfully parsed the query part of the URI, or WAI_False if an error occurred.
- **Java:** true if the server successfully parsed the query part of the URI, or false if an error occurred.

Get

Gets the value associated with the specified name in the submitted form data. If a name is associated with multiple values, you can call this method in iterations until the method returns NULL.

Syntax **C++ Prototype:**
const char* Get(const char* name);

Java Prototype:
N/A

Parameters This method has the following parameters:

name	Name of the form input that you want to get the value of.

Returns The value of the specified form input, or NULL if no other values are associated with that input.

Add

Adds a new name-value pair to the parsed form data.

Syntax **C++ Prototype:**
WAIBool Add(const char* name, const char* value);

Java Prototype:
N/A

Parameters This method has the following parameters:

name	Name of the name-value pair that you want to add to the parsed form data.
value	Value of the name-value pair that you want to add to the parsed form data.

Returns WAI_True if the name-value pair was successfully added, or WAI_False if an error occurred.

Delete

Removes a name-value pair from the parsed form data.

Syntax **C++ Prototype:**
WAIBool Delete(const char* name);

Java Prototype:
N/A

Parameters This method has the following parameters:

name	Name of the name-value pair that you want to remove from the parsed form data.

Returns WAI_True if the name-value pair was successfully removed, or WAI_False if an error occurred.

InitIterator

Sets up a pointer to the beginning of the list of name-value pairs in the parsed form data so that the Next method gets the first name-value pair in the list.

If you want to iterate through each name-value pair in the parsed form data, call this method before iteratively calling the Next method.

Syntax

C++ Prototype:
WAIBool InitIterator();

Java Prototype:
N/A

Returns WAI_True if the pointer to the list is successfully set to the beginning of the list, or WAI_False if an error occurred.

Next

Returns the name and value of the next name-value pair in the parsed form data.

To start at the beginning of the list of name-value pairs, call the InitIterator method. To iterate through the entire list, call this method iteratively until it returns the value WAI_False.

Syntax

C++ Prototype:
WAIBool Next(const char* &name, const char* &value);

Java Prototype:
N/A

Parameters This method has the following parameters:

name	Name of the next name-value pair in the parsed form data.
value	Value of the next name-value pair in the parsed form data.

Returns WAI_True if the next name-value pair is successfully retrieved, or WAI_False if there are no more name-value pairs or if an error occurred.

GetHashTable

Returns the hashtable containing the parsed form data.

You can call the methods of the java.util.Hashtable class to get data from this hashtable.

Syntax

C++ Prototype:
N/A

Java Prototype:
```
public Hashtable GetHashTable();
```

Returns The hashtable containing the parsed form data.

Chapter

10

Naming Services

This chapter covers the functions, classes, and methods available for the naming services built into the web server.

- C++ Classes for Naming Services (3.01 only)
- Java Classes for Naming Services

C++ Classes for Naming Services (3.01 only)

Version 3.01 of Netscape web servers contain functions that allow you to access the naming services built into the web server. These naming services allow you to associate a URL with an object. Once the URL is associated with the object, clients of the web server can access the object reference through the URL.

The NameUtil.hpp header file (located in the *server_root*/wai/include directory on UNIX and the *server_root*\wai\include directory on Windows NT) declares functions for registering an object implementation (associating the object with a URL) and for resolving a URL into an object reference.

This header file declares the following functions:

Methods	
registerWAS	Registers an object implementation with a URL that has the following format: http://*host*:*port*/NameService/WAS/*object_name*
resolveWAS	Resolves an object name and returns the corresponding object reference.
resolveURI	Resolves a URL that has the following format: http://*host*:*port*/NameService/WAS/*object_name* and returns the corresponding object reference.
registerObject	Registers an object implementation with a URL of the form: http://*hostname*:*portnumber*/NameService/*object_name*.
putObject	Associates an object with a URL, effectively registering the object with the name service.
putContext	Associates a naming context with a URL. You can register an object under this naming context.

registerWAS

Registers an object implementation with a URL of the form http://*hostname*:*portnumber*/NameService/WAS/*object_name*.

Syntax

```
WAIBool DLLEXPORT
registerWAS(const char *host, const char *object_name,
CORBA::Object_ptr obj);
```

Parameters This method has the following parameters:

host	Hostname and port number of the web server's host machine where you want to register your object implementation. Use the following format: *hostname*:*portnumber* If the server has SSL enabled, do not specify the hostname and port number. Instead, specify the location of the Interoperable Object Reference (IOR) file: *IOR_filename*
object_name	Instance name with which you want to register your object.
obj	The object implementation that you want to register

Returns WAI_TRUE if the object implemented was registered with the URL successfully. WAI_FALSE if registration did not complete.

Description When you register your object, a URL of the following format is constructed (based on the arguments you pass to the registerWAS method) and is associated with your object:

http://*hostname*:*portnumber*/NameService/WAS/*object_name*

where *object_name* is a unique name that you want to assign to the object instance.

After you register an object implementation with a URL, you can retrieve the object reference by resolving the URL (call the resolveURI method).

To register an object that is not under the web application services section of the URL (NameService/WAS), call the registerObject function instead.

resolveWAS

Resolves an object name (a string value) and returns the corresponding object reference.

Syntax
```
CORBA::Object_ptr DLLEXPORT
resolveWAS(const char *object_name);
```

Parameters This method has the following parameters:

object_name	Name of the object (a string value)

Returns An object reference to the object associated with the name.

Description To register an object with a URL, call the registerWAS method.

resolveURI

Resolves a URL and returns the corresponding object reference.

Syntax
```
WAIReturnType_t DLLEXPORT resolveURI(const char *host, int port,
    const char *uri, CORBA::Object_ptr& obj);
```

Parameters This method has the following parameters:

host	Name of host machine. • If protocol is http, name of the host on which the web server is running. • If protocol is file, this can be an empty string (""").
port	Port number on which the server listens. • If protocol is http, the port number on which the web server is listening. • If protocol is file, this can be 0.
url	The URL that you want to resolve to an object reference.
obj	Object reference to the object associated with the URI.

Returns WAISPISucess if the object reference was retrieved successfully. WAISPIFailure if no object reference could be determined.

Description The URI is typically in the following format:

http://*hostname*:*portnumber*/NameService/WAS/*object_name*

where *object_name* is a name under which the object instance is registered.

To register an object with a URL, call the registerWAS method.

registerObject

Registers an object implementation with a URL of the form http://*hostname*:*portnumber*/NameService/*object_name*.

Syntax

```
WAIReturnType_t DLLEXPORT registerObject(const char *host,
    const char *url, CORBA::Object_ptr obj);
```

Parameters This method has the following parameters:

host	Hostname and port number of the web server's host machine where you want to register your object implementation. Use the following format: *hostname*:*portnumber* If the server has SSL enabled, do not specify the hostname and port number. Instead, specify the location of the Interoperable Object Reference (IOR) file: *IOR_filename*
object_name	Instance name with which you want to register your object.
obj	The object implementation that you want to register

Returns WAI_TRUE if the object implemented was registered with the URL successfully. WAI_FALSE if registration did not complete.

Description When you register your object, a URL of the following format is constructed (based on the arguments you pass to the registerWAS method) and is associated with your object:

http://*hostname*:*portnumber*/NameService/*object_name*

where *object_name* is a unique name that you want to assign to the object instance.

After you register an object implementation with a URL, you can retrieve the object reference by resolving the URL (call the resolveURI method).

To register an object under the web application services section of the URL (NameService/WAS), call the registerWAS function instead.

putObject

For internal use only.

Syntax
```
WAIReturnType_t DLLEXPORT putObject(const char *url,
    CORBA::Object_ptr obj,
    WAIBool create_intermediate_nodes=WAI_FALSE);
```

putContext

For internal use only.

Syntax
```
WAIReturnType_t DLLEXPORT
```

```
putContext(const char *url,
    WAIBool create_intermediate_nodes=WAI_FALSE);
```

Java Classes for Naming Services

Netscape Communicator 4.0 and version 3.0/3.01 of Netscape web servers contain naming services that allow you to associate a URL with an object. Once the URL is associated with the object, clients of the web server can access the object reference through the URL.

Netscape provides two Java classes for associating URLs with objects:

- netscape.WAI.Naming (available in Netscape Communicator and in Netscape web servers)
- netscape.WAI.NameUtil (available in Netscape web servers)

These classes are described in more detail in this chapter.

netscape.WAI.Naming

The netscape.WAI.Naming class provides methods for registering an object implementation (associating the object with a URL) and for resolving a URL into an object reference.

The netscape.WAI.Naming class is part of the iiop10.jar file in Netscape Communicator and is part of the nisb.zip file in Netscape web servers.

Member Summary

The Naming class defines the following members:

Constructors

Naming	Creates a new Naming object.
Methods	
register	Registers an object implementation with a URL that has the following format: http://*hostname*:*portnumber*/*path*/*object_name*
resolve	Resolves a URL that has the following format: http://*hostname*:*portnumber*/*path*/*object_name* and returns the corresponding object reference.

Methods

register

Registers an object implementation with a URL.

Syntax public static
void register(String url, org.omg.CORBA.Object obj);

Throws SystemException.

Parameters This method has the following parameters:

url	The URL that you want to register your object with
obj	The object implementation that you want to register

Description The URL must have the following format:

http://*hostname*:*portnumber*/*path*/*object_name*

where *object_name* is a unique name that you want to assign to the object instance.

After you register an object with a URL, you can retrieve the object reference by resolving the URL (call the resolve method).

resolve

Resolves a URL and returns the corresponding object reference.

Syntax public static org.omg.CORBA.Object resolve(String url);

Throws SystemException.

Parameters This method has the following parameters:

url	The URL that you want to resolve to an object reference.

Returns An object reference to the object associated with the URL.

Description The URL must have the following format:

http://*hostname*:*portnumber*/*path*/*object_name*

where *object_name* is a name under which the object instance is registered.

To register an object with a URL, call the register method.

netscape.WAI.NameUtil

The netscape.WAI.NameUtil class provides methods for registering an object implementation (associating the object with a URL) and for resolving a URL into an object reference.

The netscape.WAI.NameUtil class is part of the WAI.zip file in Netscape web servers.

Member Summary

The NameUtil class defines the following members:

Methods

getRootNaming	Gets the object reference of the NamingContext object for a web server.
NameFromString	Gets a list of name components for a given string.
registerObject	Registers an object implementation with a URL of the form: http://*hostname*:*portnumber*/NameService/*object_name*.
registerWAS	Registers an object implementation with a URL that has the following format: http://*host*:*port*/NameService/WAS/*object_name*
resolveURI	Resolves a URL that has the following format: http://*host*:*port*/NameService/WAS/*object_name* and returns the corresponding object reference.

Methods

getRootNaming

Gets the object reference of the NamingContext object for a web server, given the server's hostname and port number.

Syntax

```
public static
CosNaming.NamingContext getRootNaming(String host, int port);
```

Throws SystemException.

Parameters This method has the following parameters:

host	Hostname of the machine running the web server.
port	Port number that the web server listens to.

NameFromString

Gets a list of name components from a given string.

Syntax
```
public static
CosNaming.NameHolder NameFromString(String s, String sepchar);
```

Throws SystemException.

Parameters This method has the following parameters:

s	String that you want parsed into name component
sepchar	Character representing the separator between name components (for example, "/")

registerObject

Registers an object implementation with a URL of the form http://*hostname*:*portnumber*/NameService/*object_name*.

Syntax
```
public static
boolean registerObject(String host, String object_name, org.omg.CORBA.Object obj);
```

Throws CosNaming.NamingContextPackage.NotFound
CosNaming.NamingContextPackage.CannotProceed
CosNaming.NamingContextPackage.InvalidName, org.omg.CORBA.SystemException,
java.lang.Exception

Parameters This method has the following parameters:

host	Hostname and port number of the web server's host machine where you want to register your object implementation. Use the following format: *hostname*:*portnumber* If the server has SSL enabled, do not specify the hostname and port number. Instead, specify the location of the Interoperable Object Reference (IOR) file: *IOR_filename* Note: hostname should not be null or "".
object_name	Instance name with which you want to register your object. Note: object_name should include the /NameService prefix.
obj	The object implementation that you want to register

Example

```
try {

    // Initialize the ORB.
    org.omg.CORBA.ORB orb = org.omb.CORBA.ORM.init();

    // Initialize the BOA.
    org.omg.CORBA.BOA boa = orb.BOA_init();

    // Create the account manager object.
    AccountManager manager =
        new AccountManager("Netscape Bank");

    // Export the newly created object.
    boa.obj_is_ready(manager);

    // Register the object with a name service.
    netscape.WAI.NameUtil.registerObject (InetAddress.getLocalHost().getHostName(),
        "/NameService/NetscapeBank", manager);
    System.out.println(manager + " is ready.");

    // Wait for incoming requests.
    boa.impl_is_ready();
    }
    catch(CosNaming.NamingContextPackage.InvalidName e) {
        System.err.println(e);
    }
    catch(CosNaming.NamingContextPackage.NotFound e) {
        System.err.println(e);
    }
    catch(CosNaming.NamingContextPackage.CannotProceed e) {
        System.err.println(e);
    }
    catch(org.omg.CORBA.SystemException e) {
        System.err.println(e);
```

```
    }
    catch(java.lang.Exception e) {
        System.err.println(e);
    }
}
```

registerWAS

Registers an object implementation with a URL.

Syntax

```
public static
boolean registerWAS(String host, String object_name,
    org.omg.CORBA.Object obj);
```

Throws CosNaming.NamingContextPackage.NotFound, CosNaming.NamingContextPackage.CannotProceed, CosNaming.NamingContextPackage.InvalidName, org.omg.CORBA.SystemException.

Parameters This method has the following parameters:

host	Hostname and port number of the web server's host machine where you want to register your object implementation. Use the following format: *hostname*:*portnumber* If the server has SSL enabled, do not specify the hostname and port number. Instead, specify the location of the Interoperable Object Reference (IOR) file in the following format: file:*IOR_filename*
object_name	Instance name that you want to register your object as.
obj	The object implementation that you want to register

Returns true if the object implemented was registered with the URL successfully. false if registration did not complete.

Description When you register your object, a URL of the following format is constructed (based on the arguments you pass to the registerWAS method) and is associated with your object:

http://*hostname*:*portnumber*/NameService/WAS/*object_name*

where *object_name* is a unique name that you want to assign to the object instance.

After you register an object implementation with a URL, you can retrieve the object reference by resolving the URL (call the resolveURI method).

resolveURI

Resolves a URL and returns the corresponding object reference.

Syntax

```
public static
org.omg.CORBA.Object resolveURI(String protocol, String host,
    int port, String uri);
```

Throws SystemException.

Parameters This method has the following parameters:

protocol	Protocol used to find the naming service: • If SSL is not enabled, specify http. • If SSL is enabled, specify file.
host	Name of host machine. • If protocol is http, name of the host on which the web server is running. • If protocol is file, this can be an empty string ("").
port	Port number on which the server listens. • If protocol is http, the port number on which the web server is listening. • If protocol is file, this can be 0.
url	The URL that you want to resolve to an object reference.

Returns An object reference to the object associated with the URL.

Description The URI is typically in the following format:

http://*host*:*port*/NameService/WAS/*object_name*

where *object_name* is a name under which the object instance is registered.

To register an object with a URL, call the registerWAS method.

Chapter

11

Troubleshooting Problems

If you experience problems running WAI applications, consult this chapter for troubleshooting tips.

Error: WAI Application Not Found

Symptom: The web server cannot find your WAI application (for example, if it responds to an /iiop URI with a "Not Found" page).

Possible Explanation: Your WAI application is not properly registered with the web server. Try the following troubleshooting tips:

- Verify that the application successfully registered with the web server. Check the return value of the function or method that registers the service. (In C, check the WAIregisterService() function. In C++ and Java, check the RegisterService method.)

- Verify that you have passed the hostname and port of the web server to the WAIregisterService() function or the RegisterService method. The argument containing the hostname and port should specify this information in the following format:

  hostname:*port_number*

  For example, the sample WASP and CAIIIOP examples retrieve the web server's hostname and port number from the command line (these examples expect you to enter this information as an argument).

  Suppose you are running the web server on the machine named myhost on the port 80. To execute these applications, you enter the following commands:

  ```
  WASP myhost:80

  java -DDISABLE_ORB_LOCATOR WASP myhost:80

  CAIIIOP myhost:80
  ```

- Verify that the web server is actually running on the specified host name and port.

- If you are using a version 3.0 web server, run the osfind utility (under the *server_root*/wai/bin directory on UNIX and *server_root*\wai\bin directory on Windows NT) to see a list of the implementations running on your machine.

  If you have set up the osagent utility to run on a specific IP address (or localhost, 127.0.0.1), you need to specify this address as a command-line parameter to the osfind utility. Use the -ORBagentaddr flag to specify this address.

  For example, if the osagent utility is running on localhost (IP address 127.0.0.1), use this command to start osfind:

  ```
  osfind -ORBagentaddr 127.0.0.1
  ```

  osfind returns information about any instances of osagent, OAD (the object activation daemon), and WAI applications running.

```
osfind: Found one agent at port 14000

    HOST: localhost

osfind: There are no OADs running on in your domain.

osfind: There are no Object Implementations registered with OADs.

osfind: Following are the list of Implementations started manually.

    HOST: 204.222.222.22

        INTERFACE NAME: netscape::WAI::WebApplicationBasicService

            OBJECT NAME: JavaWASP
```

```
INTERFACE NAME: netscape::WAI::WebApplicationService
    OBJECT NAME: JavaWASP
INTERFACE NAME: IDL:netscape/WAI/WebApplicationBasicService:1.0
    OBJECT NAME: JavaWASP
INTERFACE NAME: IDL:netscape/WAI/WebApplicationService:1.0
    OBJECT NAME: JavaWASP
```

Verify that your object implementation appears in this list under the correct object name.

- Go to the following URL to verify that your web service is registered under the built-in name service:

  http://*hostname*:*port_number*/NameService/WAS/*service_name*

  If the server returns a page displaying the word IOR followed by some numbers, your service is registered.

  For example, the WASP example provided with the web server registers under the service name WASP (for the C++ version) or JavaWASP (for the Java version). To verify that these applications register correctly, run the applications and go to the following URL:

  http://*server*:*port*/NameService/WAS/WASP (for C++)

  http://*server*:*port*/NameService/WAS/JavaWASP (for Java)

  If the server returns a page containing the word IOR followed by a long string of numbers, your application has registered successfully to the web server.

  If instead the server returns a "File Not Found" error, your service is not registered correctly.

Error: Server Error

Symptom: When you run your WAI application, you get a server error.

Possible Explanation: Server errors can occur for a number of different reasons. See the list of possible explanations below.

- This type of problem may occur if you are running the object activation daemon (oad) while the web server's ORB is configured for localhost use only. (See the section "Configuring the Web Server's ORB" on page 35 and Chapter 8, "Security Guidelines for Using WAI" for details.) You cannot run oad if the web server's ORB is configured this way.
- Check the error log for messages. If a message similar to the following appears:

  ```
  [10/Aug/1997:22:52:51] failure: IIOPexec CORBA exception
  CORBA::NO_IMPLEMENT.  Minor code: 0 Completed: NO
  ```

  make sure that your WAI application is running.

  The error log is stored in *server-root*/https-*serverID*/logs/errors.

Error: Invalid Stringified Object Reference ”

Symptom: When you attempt to run your WAI application, your application exits with the following error message:

```
Invalid Stringified Object Reference ”

Failed to Register with hostname
```

Possible Explanation: This error message can appear for a number of different reasons. See the list of possible explanations below.

- If you are running one of the sample applications, make sure that you specify the hostname and port number as a command-line argument. For example:

  ```
  WASP myhost:80
  ```

Web Service Registration

The following two commands, unregobj and listimpl in the wai/bin directory are useful for troubleshooting whether you registered your web service properly.

listimpl

This command lets you list all ORB object implementations registered with the Object Activation Daemon (OAD).

Description

This command lists information in the OAD's implementation repository. The information for each object includes:

- Interface names of the ORB objects.
- Instance names of the object or objects offered by that implementation.
- Full pathname of the server implementation's executable.
- Activation policy of the ORB object (shared, unshared, or per-method).
- Reference data specified when the implementation was registered with the OAD.
- List of arguments to be passed to the server at activation time.
- List of environment variables to be passed to the server at activation time.

For UNIX, if interface_name is specified, only information for that ORB object is displayed, otherwise all ORB objects registered with the OAD and their information will be shown.

The implementation repository files are assumed to reside in the impl_dir subdirectory whose path is defined by the ORBELINE environment variable. A different directory name can be set using the ORBELINE_IMPL_NAME environment variable. The path to this directory can be changed using the ORBELINE_IMPL_PATH environment variable.

Example:

```
listimpl -i Library
```

unregobj

This command unregisters ORB objects registered with the Object Activation Daemon (OAD).

Description

This command unregisters one or more ORB objects with the Object Activation Daemon. Once an object is unregistered, it can no longer be activated automatically by the OAD when a client requests the object.

ORB objects being unregistered must have been previously registered using the regobj command.

If you specify only an interface name, all ORB object with that interface that are registered with the OAD will be unregistered. Alternatively, you may specificially identify an ORB object by its interface name and object name.

If an object implementation is started manually as a persistent server, it does not need to be registered with the OAD.

Example:

```
unregobj -o Library,Harvard
```

Example:

```
unregobj -i Library
```

Index

Numerics

A

B

C

D

E

F

G

H

I

J

L

M

N

O

P

Q

R

S

T

U

V

W

X